



개발자 안내서



Borland®
Delphi™ 6
Windows 용

볼랜드 코리아 주식회사
서울특별시 강남구 삼성동 159-1 ASEM 타워 30 층
연락처 : (02) 6001-3162
www.borlandkorea.co.kr



볼랜드와 볼랜드 코리아는 이 문서에 포함된 모든 내용에 대한 특허를 가지고 있습니다.
이 문서를 공급함에 있어 사용자에게 특허권에 대한 어떠한 라이선스도 주어지지 않습니다 .

COPYRIGHT © 1997, 2001 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Other product names are trademarks or registered trademarks of their respective holders.

제품 구입 문의: Tel:02-6001-3194, 02-6001-3191

Fax:02-6001-3199

볼랜드 코리아

서울시 강남구 삼성동 159-1 ASEM 타워 30 층

목 차

1 장

서문	1-1
설명서 내용	1-1
설명서 규칙	1-3
개발자 지원 서비스	1-3
인쇄된 설명서 주문	1-3

1 부

Delphi 프로그래밍

2 장

Delphi를 사용한 애플리케이션 개발	2-1
통합 개발 환경	2-1
애플리케이션 디자인	2-2
애플리케이션 개발	2-3
프로젝트 생성	2-3
코드 편집	2-4
애플리케이션 컴파일	2-4
애플리케이션 디버깅	2-5
애플리케이션 배포	2-5

3 장

컴포넌트 라이브러리 사용	3-1
컴포넌트 라이브러리 이해	3-1
속성, 메소드 및 이벤트	3-2
속성	3-2
메소드	3-3
이벤트	3-3
사용자 이벤트	3-3
시스템 이벤트	3-4
오브젝트 파스칼 및 클래스 라이브러리	3-4
객체 모델 사용	3-4
객체의 개념	3-5
Delphi 객체 살펴 보기	3-5
컴포넌트 이름 변경	3-7
객체로부터 데이터와 코드 상속	3-8
유효 범위(scope) 및 한정자	3-8
private, protected, public, published 선언	3-9
객체 변수 사용	3-10
객체 생성, 인스턴스화 및 소멸	3-11
컴포넌트와 소유권	3-12
객체, 컴포넌트 및 컨트롤	3-12

TObject 분기	3-14
TPersistent 분기	3-14
TComponent 분기	3-15
TControl 분기	3-16
TWinControl/TWidgetControl 분기	3-17
TControl의 공통 속성	3-18
액션 속성	3-18
위치, 크기 및 정렬 속성	3-19
표시 속성	3-19
부모 속성	3-19
탐색 속성	3-19
드래그 앤 드롭 속성	3-20
드래그 앤 도킹 속성(VCL 전용)	3-20
TControl의 표준 공통 이벤트	3-20
TWinControl 및 TWidgetControl의 공통 속성	3-21
일반 정보 속성	3-21
테두리 스타일 표시 속성	3-22
탐색 속성	3-22
드래그 앤 도킹 속성(VCL 전용)	3-22
TWinControl 및 TWidgetControl의 공통 이벤트	3-22
애플리케이션 사용자 인터페이스 만들기	3-23
Delphi 컴포넌트 사용	3-24
컴포넌트 속성 설정	3-24
Object Inspector 사용	3-25
속성 편집기 사용	3-25
런타임 시 속성 설정	3-25
메소드 호출	3-25
이벤트 및 이벤트 핸들러 작업	3-26
새 이벤트 핸들러 생성	3-26
컴포넌트의 기본 이벤트에 대한 핸들러 생성	3-26
이벤트 핸들러 찾기	3-27
이벤트를 기존 이벤트 핸들러에 연결	3-27
메뉴 이벤트를 이벤트 핸들러에 연결	3-28
이벤트 핸들러 삭제	3-29
VCL 및 CLX 컴포넌트	3-29
컴포넌트 팔레트에 사용자 지정 컴포넌트 추가	3-31
텍스트 컨트롤	3-31
텍스트 컨트롤 속성	3-32
메모 및 서식있는 텍스트 컨트롤의 속성	3-32
서식있는 텍스트 컨트롤(VCL 전용)	3-33
특수한 입력 컨트롤	3-33
스크롤 막대	3-33

트랙 표시줄	3-33
Up-down 컨트롤(VCL 전용)	3-34
스핀 편집 컨트롤(CDX 전용)	3-34
Hot key 컨트롤(VCL 전용)	3-34
스플리터 컨트롤	3-34
버튼 유형의 컨트롤	3-35
버튼 컨트롤	3-35
비트맵 버튼	3-36
스피드 버튼	3-36
체크 박스(Check box)	3-36
라디오 버튼	3-37
툴바	3-37
쿨바(VCL 전용)	3-37
목록 처리	3-38
리스트 박스(List box)와 체크 리스트	
박스(Check-list box)	3-38
콤보 박스(Combo box)	3-39
트리 뷰(Tree View)	3-39
리스트 뷰(List View)	3-40
날짜/시간 선택(Data-time picker)과	
달력(Month calendar) (VCL 전용)	3-40
컴포넌트 그룹화	3-40
그룹 상자와 라디오 그룹	3-41
패널	3-41
스크롤 상자	3-41
탭 컨트롤	3-42
페이지 컨트롤	3-42
헤더 컨트롤	3-42
비주얼 피드백 제공	3-43
레이블 및 정적 텍스트 컴포넌트	3-43
상태 표시줄(Status bar)	3-43
진행 표시줄(Progress bar)	3-44
도움말과 힌트 속성	3-44
그리드	3-44
그리기 그리드	3-44
문자열 그리드	3-45
값 목록 편집기(VCL 전용)	3-45
그래픽 표시	3-46
이미지	3-46
도형(Shape)	3-46
빗면(Bevel)	3-46
그리기 상자	3-47
애니메이션 컨트롤(VCL 전용)	3-47
대화 상자 개발	3-47
열린 대화 상자 사용	3-48
helper 객체 사용	3-48
목록 사용	3-49
문자열 목록 사용	3-49
문자열 목록의 로드 및 저장	3-50

새 문자열 목록 작성	3-50
목록에서 문자열 처리	3-52
문자열 목록에 객체 연결	3-54
Windows 레지스트리 및 INI	
파일(VCL 전용)	3-54
TIniFile 사용(VCL 전용)	3-55
TRegistry 사용	3-55
TRegIniFile 사용	3-56
드로잉 공간 만들기	3-56
인쇄	3-57
스트림 사용	3-57

4 장

일반적인 프로그래밍 작업	4-1
클래스 이해	4-1
클래스 정의	4-2
예외 처리	4-4
코드 블록 보호	4-4
예외에 대한 응답	4-5
예외와 제어 흐름	4-6
예외 응답 중첩	4-6
리소스 할당 보호	4-7
보호가 필요한 리소스 종류	4-7
리소스 보호 블록 만들기	4-8
RTL(Run-Time Library) 예외 처리	4-9
RTL 예외	4-9
예외 핸들러 작성	4-10
예외 처리 문장	4-10
예외 인스턴스 사용	4-11
예외 핸들러의 범위	4-12
기본 예외 핸들러 제공	4-12
예외 클래스 다루기	4-13
예외 재발생	4-13
컴포넌트 예외 처리	4-14
외부 소스를 사용한 예외 처리	4-14
예외 숨기기(Silent exceptions)	4-15
사용자 고유의 예외 정의	4-16
예외 객체 타입 선언	4-16
예외 발생	4-16
인터페이스 사용	4-17
랭귀지 기능의 인터페이스	4-17
클래스 간의 인터페이스 공유	4-18
프로시저와 함께 인터페이스 사용	4-19
IInterface 구현	4-20
TInterfacedObject	4-20
as 연산자 사용	4-21
코드 재사용 및 위임(Delegation)	4-22
위임을 위한 implements 사용	4-22

추상화(Aggregation)	4-23	파일 스트림 사용	4-54
인터페이스 객체의 메모리 관리	4-24	파일 생성 및 열기	4-55
참조 카운팅 사용	4-24	파일 핸들 사용	4-56
참조 카운팅을 사용하지 않음	4-25	파일 읽기 및 쓰기	4-56
분산 애플리케이션에서 인터페이스		문자열 읽기 및 쓰기	4-57
사용(VCL 전용)	4-26	파일 찾기	4-57
사용자 지정 가변 정의	4-27	파일 위치 및 크기	4-58
사용자 지정 가변 타입 데이터 저장	4-27	복사	4-58
사용자 지정 가변 타입을 활성화하기		측정 변환	4-59
위한 클래스 생성	4-28	변환 수행	4-59
타입 변환 활성화	4-28	간단한 변환 수행	4-59
이항 연산 구현	4-30	복잡한 변환 수행	4-59
비교 연산 구현	4-32	새 측정 타입 추가	4-60
단항 연산 구현	4-33	간단한 변환 패밀리 생성과 단위 추가	4-60
사용자 지정 가변의 복사 및 지우기	4-34	변환 함수 사용	4-61
사용자 지정 가변 값 로드 및 저장	4-35	클래스를 사용한 변환 관리	4-63
TCustomVariantType 자손 사용	4-36	데이터 타입 정의	4-66
사용자 지정 가변 타입을 사용하는			
유틸리티 작성	4-36		
사용자 지정 가변의 지원 속성 및 메소드	4-37		
TInvokeableVariantType 사용	4-37		
TPublishableVariantType 사용	4-39		
문자열 작업	4-39		
문자 타입(Character types)	4-39		
문자열 타입(String types)	4-40		
짧은 문자열	4-40		
긴 문자열	4-41		
WideString	4-41		
PChar 타입	4-42		
OpenString	4-42		
런타임 라이브러리 문자열 처리 루틴	4-42		
와이드 문자 루틴	4-43		
일반적으로 사용되는 긴 문자열 루틴	4-43		
문자열 선언 및 초기화	4-46		
문자열 타입의 혼합 및 변환	4-47		
문자열에서 PChar로 변환	4-47		
문자열 종속성	4-47		
PChar 지역 변수 반환	4-48		
지역 변수를 PChar로 전달	4-48		
문자열에 대한 컴파일러 지시어	4-49		
문자열과 문자: 관련 주제	4-50		
파일 작업	4-50		
파일 처리	4-51		
파일 삭제	4-51		
파일 찾기	4-51		
파일 이름 재지정	4-53		
파일 date-time 루틴	4-53		
파일 복사	4-53		
파일 I/O와 파일 형식	4-54		
		5 장	
		애플리케이션, 컴포넌트, 라이브러리	
		구축	5-1
		애플리케이션 생성	5-1
		GUI 애플리케이션	5-1
		사용자 인터페이스 모델	5-2
		SDI 애플리케이션	5-2
		MDI 애플리케이션	5-2
		IDE, 프로젝트 및 컴파일 옵션 설정	5-3
		프로그래밍 템플릿	5-3
		콘솔 애플리케이션	5-3
		서비스 애플리케이션	5-4
		서비스 스레드	5-6
		서비스 이름 속성	5-8
		서비스 디버깅	5-8
		패키지와 DLL 생성	5-9
		패키지와 DLL 사용하는 경우	5-9
		데이터베이스 애플리케이션 개발	5-10
		분산 데이터베이스 애플리케이션	5-11
		웹 서버 애플리케이션 만들기	5-11
		Web Broker 사용	5-11
		WebSnap 애플리케이션 만들기	5-13
		InternetExpress 사용	5-13
		Web Services 애플리케이션 만들기	5-13
		COM을 사용한 애플리케이션 작성	5-14
		COM과 DCOM 사용	5-14
		MTS 및 COM+ 사용	5-15
		데이터 모듈 사용	5-15
		표준 데이터 모듈 생성 및 편집	5-16
		데이터 모듈과 유닛 파일 이름 지정	5-16

컴포넌트 배치 및 이름 지정	5-17	폼 연결	6-2
데이터 모듈의 컴포넌트 속성 및 이벤트 사용	5-18	순환 유닛 참조 방지	6-2
데이터 모듈에서 비즈니스 룰 만들기	5-18	메인 폼 숨기기	6-3
폼에서 데이터 모듈 액세스	5-18	애플리케이션 레벨에서 작업	6-3
원격 데이터 모듈을 애플리케이션 서버 프로젝트에 추가	5-19	화면 처리	6-3
Object Repository 사용	5-20	레이아웃 관리	6-4
프로젝트 내에서 항목 공유	5-20	이벤트 공지에 대한 응답	6-5
항목 추가: Object Repository	5-20	폼 사용	6-6
덱 환경에서 객체 공유	5-20	메모리에 상주하는 폼의 제어	6-6
프로젝트에서 Object Repository 항목 사용	5-21	자동 생성된 폼 표시	6-6
항목 복사	5-21	동적으로 폼 생성	6-7
항목 상속	5-21	창 같은 모달리스 폼 생성	6-8
항목 사용	5-21	로컬 변수로 폼 인스턴스 생성	6-8
프로젝트 템플릿 사용	5-21	폼에 추가 인수 전달	6-8
공유 항목 수정	5-22	폼에서 데이터 검색	6-9
기본 프로젝트, 새 폼 및 메인 폼 지정	5-22	모달리스 폼에서 데이터 검색	6-9
애플리케이션에서 도움말 사용	5-22	모달 폼에서 데이터 검색	6-11
도움말 시스템 인터페이스	5-23	컴포넌트와 컴포넌트 그룹의 재사용	6-12
ICustomHelpViewer 구현	5-24	컴포넌트 템플릿 생성 및 사용	6-13
Help Manager와 통신	5-24	프레임 사용	6-14
Help Manager에 정보 요청	5-25	프레임 생성	6-14
키워드 방식 도움말 표시	5-25	컴포넌트 팔레트에 프레임 추가	6-14
목차 표시	5-26	프레임 사용 및 수정	6-15
IExtendedHelpViewer 구현	5-26	프레임 공유	6-16
IHelpSelector 구현	5-27	툴바 및 메뉴의 작업 구성	6-16
도움말 시스템 객체 등록	5-28	액션(action)의 개념	6-17
도움말 뷰어 등록	5-28	액션 밴드 설정	6-18
도움말 선택기 등록	5-28	툴바 및 메뉴 생성	6-19
VCL 애플리케이션에서 도움말 사용	5-28	메뉴, 버튼 및 툴바에 색상, 패턴 또는 그림 추가	6-20
TApplication이 VCL 도움말을 처리하는 방법	5-29	메뉴 및 툴바에 아이콘 추가	6-21
VCL 컨트롤의 도움말 처리 방법	5-29	사용자가 사용자 정의할 수 있는 툴바 및 메뉴 생성	6-21
CLX 애플리케이션에서 도움말 사용	5-30	작업 밴드의 사용되지 않는 항목 및 범주 숨기기	6-22
TApplication이 CLX 도움말을 처리하는 방법	5-30	액션 리스트 사용	6-23
CLX 컨트롤의 도움말 처리 방법	5-30	액션 리스트 설정	6-23
도움말 시스템 직접 호출	5-31	액션 실행 시 발생하는 이벤트	6-24
IHelpSystem 사용	5-31	이벤트에 응답	6-24
IDE 도움말 시스템 사용자 지정	5-32	액션이 해당 대상을 찾는 방법	6-26
6 장		액션 업데이트	6-26
애플리케이션 사용자 인터페이스 개발	6-1	이미 정의된 액션 클래스	6-27
애플리케이션 동작 제어	6-1	액션 컴포넌트 작성	6-28
메인 폼 사용	6-1	액션 등록	6-28
폼 추가	6-2	메뉴 생성 및 관리	6-29
		메뉴 디자이너 열기	6-30
		메뉴 작성	6-32
		메뉴 이름 지정	6-32
		메뉴 항목 이름 지정	6-32

메뉴 항목 추가, 삽입 및 삭제	6-33
구분자 표시줄 추가	6-34
가속키 및 단축키 지정	6-34
하위 메뉴 생성	6-34
기존 메뉴를 밑으로 내려 하위 메뉴 생성	6-35
메뉴 항목 이동	6-35
메뉴 항목에 이미지 추가	6-36
메뉴 보기	6-37
Object Inspector에서 메뉴 항목 편집	6-37
메뉴 디자이너 컨텍스트 메뉴 사용	6-37
컨텍스트 메뉴의 명령	6-38
디자인 타임 시 메뉴 간의 전환	6-38
메뉴 템플릿 사용	6-39
메뉴를 템플릿으로 저장	6-40
템플릿 메뉴 항목 및 이벤트 핸들러의 이름 지정 규칙	6-41
런타임 시 메뉴 항목 처리	6-41
메뉴 병합	6-41
활성 메뉴 지정: Menu 속성	6-42
병합된 메뉴 항목의 순서 결정: GroupIndex 속성	6-42
리소스 파일 import하기	6-42
툴바 및 쿨바 디자인	6-43
패널 컴포넌트를 사용하여 툴바 추가	6-44
패널에 스피드 버튼 추가	6-45
스피드 버튼의 glyph 할당	6-45
스피드 버튼의 초기 상태 설정	6-45
스피드 버튼 그룹 생성	6-46
버튼 토글 허용	6-46
툴바 컴포넌트를 사용하여 툴바 추가	6-46
툴 버튼 추가	6-47
툴 버튼에 이미지 할당	6-47
툴 버튼 모양 및 초기 상태 설정	6-47
툴 버튼 그룹 생성	6-48
툴 버튼 토글 허용	6-48
쿨바 컴포넌트 추가	6-48
쿨바 모양 설정	6-49
클릭 이벤트에 응답	6-49
툴 버튼에 메뉴 할당	6-49
숨겨진 툴바 추가	6-50
툴바 숨기기 및 표시	6-50
데모 프로그램	6-50

7 장

컨트롤 사용	7-1
컨트롤의 드래그 앤 드롭 구현	7-1
드래그 연산 시작	7-1

드래그된 항목 수용	7-2
항목 드롭	7-2
드래그 연산 끝내기	7-3
드래그 객체를 사용하여 드래그 앤 드롭 사용자 지정	7-3
드래그 마우스 포인터 변경	7-4
컨트롤에서 드래그 앤 드롭 구현	7-4
윈도우 컨트롤을 도킹 공간으로 만들기	7-4
컨트롤을 도킹 가능한 자식 컨트롤로 만들기	7-5
자식 컨트롤의 도킹 방법 제어	7-5
자식 컨트롤이 도킹 해제되는 방법 제어	7-6
자식 컨트롤이 드래그 앤 드롭 연산에 응답하는 방법 제어	7-6
컨트롤에서 텍스트 사용	7-7
텍스트 정렬 설정	7-7
런타임 시 스크롤 막대 추가	7-8
클립보드 객체 추가	7-8
텍스트 선택	7-9
모든 텍스트 선택	7-9
텍스트 잘라내기, 복사 및 붙여넣기	7-10
선택한 텍스트 삭제	7-10
메뉴 항목 비활성화	7-10
팝업 메뉴 제공	7-11
OnPopup 이벤트 처리	7-12
컨트롤에 그래픽 추가	7-12
컨트롤이 owner-raw 항목임을 표시	7-13
문자열 목록에 그래픽 객체 추가	7-13
애플리케이션에 이미지 추가	7-14
문자열 목록에 이미지 추가	7-14
Owner-draw 항목 그리기	7-15
Owner-draw 항목 크기 조정	7-15
Owner-draw 항목 그리기	7-16

8 장

그래픽 및 멀티미디어 작업	8-1
그래픽 프로그래밍 개요	8-1
화면 새로 고침	8-2
그래픽 객체 형식	8-3
캔버스의 일반적인 속성과 메소드	8-4
캔버스 객체의 속성 사용	8-5
펜 사용	8-5
브러시 사용	8-8
픽셀 읽기 및 설정	8-9
캔버스 메소드를 사용하여 그래픽 객체 그리기	8-10
선 및 다각선 그리기	8-10
도형 그리기	8-11

애플리케이션에서 여러 드로잉 객체 처리	8-12	다중 읽기 배타적 쓰기 동기화 장치 (<i>Multi-read exclusive-write synchronizer</i>)	
사용할 드로잉 툴 파악	8-12	사용	9-8
스피드 버튼으로 툴 변경	8-13	메모리 공유에 이용되는 그 밖의 기술	9-9
드로잉 툴 사용	8-14	다른 스레드 기다리기	9-9
그래픽 위에 그리기	8-16	스레드의 실행 완료 대기	9-9
스크롤 가능한 그래픽 만들기	8-17	작업 완료 대기	9-9
이미지 컨트롤 추가	8-17	스레드 객체 실행	9-11
그래픽 파일 로드 및 저장	8-19	기본 우선 순위 오버라이드	9-11
파일에서 그림 로드	8-19	스레드 시작 및 중지	9-11
그림을 파일에 저장	8-20	다중 스레드 애플리케이션 디버깅	9-12
그림 교체	8-20		
클립보드에서 그래픽 사용	8-21	10 장	
클립보드에 그래픽 복사	8-22	크로스 플랫폼 개발을 위한 CLX 사용	10-1
클립보드로 그림 잘라내기	8-22	크로스 플랫폼 애플리케이션 만들기	10-1
클립보드에서 그림 붙여넣기	8-23	VCL 애플리케이션을 CLX로 포팅	10-2
양단 묶음(Rubber banding) 예제	8-23	포팅 기법	10-3
마우스에 응답	8-24	플랫폼 특정 포팅	10-3
마우스 다운 동작에 응답	8-25	크로스 플랫폼 포팅	10-3
마우스 동작을 추적하기 위해 폼 객체에 필드 추가	8-26	Windows 에뮬레이션 포팅	10-3
선 그리기 다듬기	8-27	애플리케이션 포팅	10-4
멀티미디어 작업	8-29	CLX와 VCL 비교	10-5
애플리케이션에 소리가 없는 비디오 클립 추가	8-29	CLX가 다르게 하는 동작	10-6
소리가 없는 비디오 클립 추가 예제	8-30	룩앤필(Look and feel)	10-6
애플리케이션에 오디오 및/또는 비디오 클립 추가	8-31	스타일	10-7
오디오 및/또는 비디오 클립 추가 예제 (VCL 전용)	8-33	Variants	10-7
		Registry	10-7
		그 밖의 차이점	10-8
		CLX에 없는 내용	10-8
		포팅되지 않는 기능	10-9
		CLX와 VCL 유닛 비교	10-10
		CLX 객체 생성자의 차이점	10-13
9 장		Windows와 Linux 간의 소스 파일 공유	10-14
다중 스레드 애플리케이션 작성	9-1	Windows와 Linux 간의 환경적 차이점	10-14
스레드 객체 정의	9-1	Linux의 디렉토리 구조	10-16
스레드 초기화	9-2	포팅 가능한 코드 작성	10-17
기본 우선 순위 할당	9-2	조건 지시어 사용	10-18
스레드 해제 시기 표시	9-3	조건 지시어 종료	10-19
스레드 함수 작성	9-4	메시지 나타내기	10-20
메인 VCL/CLX 스레드 사용	9-4	인라인 어셈블러 코드 포함	10-21
스레드 로컬 변수 사용	9-5	메시지와 시스템 이벤트	10-21
다른 스레드로 종료 확인	9-6	Linux에서의 프로그래밍 차이점	10-23
스레드 함수에서의 예외 처리	9-6	크로스 플랫폼 데이터베이스 애플리케이션	10-23
지우기 코드 작성	9-6	dbExpress 차이점	10-24
스레드 조정	9-7	컴포넌트 수준 차이점	10-25
동시 액세스 피하기	9-7		
객체 잠금	9-7		
임계 구역(Critical sections) 사용	9-7		

사용자 인터페이스 수준 차이점	10-26
데이터베이스 애플리케이션을 Linux로 포팅	10-26
dbExpress 애플리케이션에서 데이터 업데이트	10-28
크로스 플랫폼 인터넷 애플리케이션	10-30
인터넷 애플리케이션을 Linux로 포팅	10-30

11 장

패키지와 컴포넌트 사용	11-1
패키지의 이점	11-2
패키지와 표준 DLL	11-2
런타임 패키지	11-2
애플리케이션에서 패키지 사용	11-3
패키지 동적 로딩	11-4
사용할 런타임 패키지 결정	11-4
사용자 지정 패키지	11-4
디자인 타임 패키지	11-5
컴포넌트 패키지 설치	11-5
패키지 생성 및 편집	11-6
패키지 생성	11-6
기존 패키지 편집	11-7
수동으로 패키지 소스 파일 편집	11-8
패키지 구조 이해	11-8
패키지 이름 지정	11-8
Requires 절	11-9
Contains 절	11-9
패키지 컴파일	11-10
패키지 특정 컴파일러 지시어	11-10
명령줄 컴파일러와 링커 사용	11-12
컴파일이 성공했을 때 생성되는 패키지 파일	11-12
패키지 배포	11-13
패키지를 사용하는 애플리케이션 배포	11-13
다른 개발자에 패키지 배포	11-13
패키지 모음 파일	11-13

12 장

국제적인 애플리케이션 만들기	12-1
국제화 및 지역화	12-1
국제화	12-1
지역화	12-1
애플리케이션 국제화	12-2
애플리케이션 코드 활성화	12-2
문자 집합	12-2
OEM 및 ANSI 문자 집합	12-2

멀티바이트 문자 집합	12-2
와이드 문자	12-3
애플리케이션에 양방향 기능 포함	12-4
BiDiMode 속성	12-6
로케일 특정 기능	12-9
사용자 인터페이스 디자인	12-9
텍스트	12-9
그래픽 이미지	12-10
형식 및 정렬 순서	12-10
키보드 매핑	12-10
리소스 분리	12-10
리소스 DLL 작성	12-11
리소스 DLL 사용	12-12
리소스 DLL의 동적 전환	12-13
애플리케이션 지역화	12-14
리소스 지역화	12-14

13 장

애플리케이션 배포	13-1
일반 애플리케이션 배포	13-1
설치 프로그램 사용	13-2
애플리케이션 파일 식별	13-2
애플리케이션 파일	13-3
패키지 파일	13-3
Merge 모드	13-3
ActiveX 컨트롤	13-5
Helper 애플리케이션	13-5
DLL 위치	13-5
CLX 애플리케이션 배포	13-6
데이터베이스 애플리케이션 배포	13-6
dbExpress 데이터베이스 애플리케이션 배포	13-7
BDE 애플리케이션 배포	13-8
BDE(Borland Database Engine)	13-8
SQL 연결	13-9
다계층(multi-tier) 데이터베이스 애플리케이션(DataSnap) 배포	13-10
웹 애플리케이션 배포	13-10
Apache 배포	13-11
다양한 호스트 환경을 위한 프로그래밍	13-12
화면 해상도와 색상 수	13-12
동적으로 크기가 조정되지 않는 경우의 고려 사항	13-12
폼과 컨트롤 크기를 동적으로 조정하는 경우의 고려 사항	13-13
다양한 색상 수 지원	13-14
글꼴	13-14
운영 체제 버전	13-14

소프트웨어 사용권 요구 사항	13-15
DEPLOY	13-15
README	13-15
사용권 계약서	13-15
타사(third-party) 제품 설명서	13-16

2 부 데이터베이스 애플리케이션 개발

14 장

데이터베이스 애플리케이션 디자인	14-1
데이터베이스 사용	14-1
데이터베이스의 타입	14-2
데이터베이스 보안	14-4
트랜잭션	14-4
참조 무결성, 내장 프로시저 및 트리거	14-5
데이터베이스 아키텍처	14-6
일반 구조	14-6
사용자 인터페이스 폼	14-6
데이터 모듈	14-6
데이터베이스 서버에 직접 연결	14-8
디스크의 전용 파일 사용	14-9
다른 데이터셋에 연결	14-10
클라이언트 데이터셋을 동일한 애플리케이션의 다른 데이터셋에 연결	14-12
다계층 아키텍처 사용	14-13
방법의 조합	14-14
사용자 인터페이스 디자인	14-15
데이터 분석	14-15
리포트 작성	14-16

15 장

데이터 컨트롤 사용	15-1
공통적인 데이터 컨트롤 기능의 사용	15-2
데이터셋에 데이터 컨트롤 연결	15-3
런타임 시 연결된 데이터셋 변경	15-3
데이터 소스의 사용 가능 및 사용 불가능	15-4
데이터 소스에서 변경 사항에 대한 응답	15-4
데이터 편집과 업데이트	15-5
컨트롤에서 사용자 입력의 편집 사용 가능	15-5
컨트롤에서의 데이터 편집	15-5

데이터 표시 사용 불가능 및 사용 가능	15-6
데이터 표시 새로 고침	15-7
마우스, 키보드 및 타이머 이벤트 사용 가능	15-7
데이터 구성 방법 선택	15-7
단일 레코드 표시	15-7
데이터를 레이블로 표시	15-8
편집 상자의 필드 표시 및 편집	15-8
메모 컨트롤에 텍스트 표시 및 편집	15-9
Rich edit 메모 컨트롤에 텍스트 표시 및 편집	15-9
이미지 컨트롤의 그래픽 필드 표시 및 편집	15-10
리스트 박스와 콤보 박스의 데이터 표시 및 편집	15-10
체크 박스로 부울 필드 값 처리	15-13
라디오 컨트롤로 필드 값 제한	15-14
여러 레코드 표시	15-14
TDBGrid로 데이터 보기 및 편집	15-15
기본 상태에서 그리드 컨트롤 사용	15-16
사용자 지정 그리드 생성	15-17
영구적 열(Persistent Columns) 이해	15-17
영구적 열 생성	15-18
영구적 열 삭제	15-19
영구적 열 순서 정렬	15-19
디자인 타임 시 열 속성 설정	15-20
조회 목록 열 정의	15-21
열에 버튼 두기	15-21
열에 기본값 복구	15-21
ADT 및 배열 필드 표시	15-22
그리드 옵션 설정	15-24
그리드 편집	15-25
그리드 그리기 제어	15-26
런타임 시 사용자 동작에 응답	15-26
기타 data-aware 컨트롤을 포함하는 그리드 생성	15-27
레코드 탐색 및 처리	15-28
표시할 탐색기 버튼 선택	15-29
디자인 타임 시 탐색기 버튼 보이기 및 숨기기	15-29
런타임 시 탐색기 버튼 보이기 및 숨기기	15-30
풍선 도움말 표시	15-30
여러 데이터셋에 단일 탐색기 사용	15-31

16 장

의사 결정 지원(Decision Support)

컴포넌트 사용	16-1
개요	16-1
크로스탭	16-2
1차원 크로스탭	16-3
다차원 크로스탭	16-3
의사 결정 지원 컴포넌트 사용 지침	16-3
의사 결정 지원 컴포넌트에 데이터셋 사용	16-4
TQuery 또는 TTable로 의사 결정	
데이터셋 생성	16-5
Decision Query 에디터로 의사 결정	
데이터셋 생성	16-6
의사 결정 큐브(Decision Cube) 사용	16-7
의사 결정 큐브 속성과 이벤트	16-7
Decision Cube 에디터 사용	16-7
차원 설정의 보기 및 변경	16-7
사용 가능한 차원과 요약의	
최대값 설정	16-8
디자인 옵션의 보기 및 변경	16-8
의사 결정 소스(Decision Source) 사용	16-8
속성과 이벤트	16-9
의사 결정 피벗(Decision Pivot) 사용	16-9
의사 결정 피벗 속성	16-10
의사 결정 그리드(Decision Grid)	
생성 및 사용	16-10
의사 결정 그리드 생성	16-10
의사 결정 그리드 사용	16-11
의사 결정 그리드 필드	
열기와 닫기	16-11
의사 결정 그리드에서 행과 열	
재구성	16-11
의사 결정 그리드의 자세한 내용	
드릴 다운	16-11
의사 결정 그리드의 차원 선택 제한	16-12
의사 결정 그리드 속성	16-12
의사 결정 그래프(Decision Graph)	
생성 및 사용	16-13
의사 결정 그래프 생성	16-13
의사 결정 그래프 사용	16-13
의사 결정 그래프 표시	16-15
의사 결정 그래프 사용자 지정	16-15
의사 결정 그래프 템플릿	
기본값 설정	16-16
의사 결정 그래프 시리즈	
사용자 지정	16-17
런타임의 의사 결정 지원 컴포넌트	16-18
런타임의 의사 결정 피벗	16-18

런타임 시의 의사 결정 그리드	16-19
런타임 시의 의사 결정 그래프	16-19
의사 결정 지원(Decision Support)	
컴포넌트와 메모리 제어	16-19
차원, 요약, 셀의 최대값 설정	16-19
차원 상태 설정	16-20
페이지화된 차원 사용	16-20

17 장

데이터베이스에 연결	17-1
암시적인 연결(Implicit connections)	
사용	17-2
연결 제어	17-2
데이터베이스 서버에 연결	17-3
데이터베이스 서버로부터 연결 해제	17-3
서버 로그인 제어	17-4
트랜잭션 관리	17-6
트랜잭션 시작	17-6
트랜잭션 종료	17-8
성공적인 트랜잭션 종료	17-8
성공적이지 않은 트랜잭션 종료	17-8
트랜잭션 분리 레벨 지정	17-9
서버에 명령 전송	17-10
연결된 데이터셋 작업	17-12
서버로부터 연결을 해제하지 않고	
데이터셋 닫기	17-12
연결된 데이터셋을 통한 반복	17-12
메타데이터 얻기	17-13
사용 가능한 테이블 열거	17-13
테이블의 필드 열거	17-14
사용 가능한 내장 프로시저	
(Stored procedures) 열거	17-14
사용 가능한 인덱스 열거	17-14
내장 프로시저 매개변수 열거	17-14

18 장

데이터셋 이해	18-1
TDataSet 자손 사용	18-2
데이터셋 상태 알아보기	18-3
데이터셋 열기와 닫기	18-4
데이터셋 탐색	18-5
First 및 Last 메소드 사용	18-6
Next 및 Prior 메소드 사용	18-6
MoveBy 메소드 사용	18-7
Eof 및 Bof 속성 사용	18-7
Eof	18-7
Bof	18-8

레코드 표시 및 반환	18-9	범위의 적용 또는 취소	18-34
Bookmark 속성	18-9	마스터/디테일 관계 생성	18-35
GetBookmark 메소드	18-9	테이블을 다른 데이터셋의 디테일로 만들기	18-35
GotoBookmark 및 BookmarkValid 메소드	18-9	중첩 디테일 테이블 사용	18-37
CompareBookmarks 메소드	18-10	테이블에 대한 읽기/쓰기 권한 제어	18-38
FreeBookmark 메소드	18-10	테이블 생성 및 삭제	18-38
북마크 예제	18-10	테이블 생성	18-38
데이터셋 검색	18-10	테이블 삭제	18-41
Locate 사용	18-11	테이블 비우기	18-41
Lookup 사용	18-11	테이블 동기화	18-41
필터를 사용한 데이터 서브셋 표시 및 편집	18-12	쿼리 타입 데이터셋 사용	18-42
필터링 사용 가능 및 사용 불가능	18-13	쿼리 지정	18-43
필터 생성	18-13	SQL 속성을 사용하여 쿼리 지정	18-44
Filter 속성 설정	18-14	CommandText 속성을 사용하여 쿼리 지정	18-44
OnFilterRecord 이벤트 핸들러 작성	18-15	쿼리의 매개변수 사용	18-45
런타임 시 필터 이벤트 핸들러 전환	18-15	디자인 타임에 매개변수 제공	18-45
필터 옵션 설정	18-15	런타임 시 매개변수 제공	18-46
필터링된 데이터셋의 레코드 탐색	18-16	매개변수를 사용하여 마스터/디테일 관계 설정	18-47
데이터 수정	18-17	쿼리 준비	18-48
레코드 편집	18-17	결과 집합(Result set)을 반환하지 않는 쿼리 실행	18-49
새 레코드 추가	18-18	단방향 결과 집합 사용	18-49
레코드 삽입	18-19	내장 프로시저 타입 데이터셋 사용	18-50
레코드 추가	18-19	내장 프로시저 매개변수 작업	18-51
레코드 삭제	18-19	디자인 타임 시 매개변수 설정	18-52
데이터 포스트	18-20	런타임 시 매개변수 사용	18-53
변경 취소	18-21	내장 프로시저 준비	18-54
전체 레코드 수정	18-21	결과 집합(Result set)을 반환하지 않는 내장 프로시저 실행	18-54
필드 계산	18-22	여러 결과 집합 페치	18-54
데이터셋 형식	18-23		
테이블 타입 데이터셋 사용	18-25		
테이블 타입 데이터셋 사용의 장점	18-25		
인덱스로 레코드 정렬	18-26		
인덱스 관련 정보 얻기	18-26		
IndexName으로 인덱스 지정	18-27		
IndexFieldNames로 인덱스 작성	18-27		
인덱스를 사용하여 레코드 검색	18-28		
Goto 메소드로 검색 수행	18-28		
Find 메소드로 검색 수행	18-29		
성공적인 검색 이후 현재 레코드 지정	18-29		
부분 키 검색	18-30		
검색 반복 또는 확장	18-30		
범위로 레코드 제한	18-30		
범위와 필터의 차이점 이해	18-30		
범위 지정	18-31		
범위 수정	18-33		
		19 장	
		필드 컴포넌트 사용	19-1
		동적(Dynamic) 필드 컴포넌트	19-2
		영구적(Persistent) 필드 컴포넌트	19-3
		영구적 필드 생성	19-4
		영구적 필드 정렬	19-5
		새 영구적 필드 정의	19-5
		데이터 필드 정의	19-6
		계산된 필드 정의	19-7
		계산된 필드 프로그래밍	19-8
		조회 필드 정의	19-8
		집계 필드(Aggregates field) 정의	19-10
		영구적 필드 컴포넌트 삭제	19-10
		영구적 필드 속성 및 이벤트 설정	19-11

디자인 타임 시 표시 및 편집		데이터베이스 및 세션 연결과	
속성 설정	19-11	데이터셋의 연결	20-3
런타임 시 필드 컴포넌트 속성 설정	19-12	BLOB 캐시	20-4
필드 컴포넌트의 속성 집합 만들기	19-13	BDE 핸들 얻기	20-4
필드 컴포넌트와 속성 집합 연결	19-13	TTable 사용	20-4
속성 연결 제거	19-14	로컬 테이블의 테이블 타입 지정	20-5
사용자 입력 조정 및 마스킹	19-14	로컬 테이블에 대한 읽기/쓰기	
숫자, 날짜 및 시간 필드에 대한		액세스 제어	20-6
기본 서식 사용	19-15	dBASE 인덱스 파일 지정	20-6
이벤트 처리	19-15	로컬 테이블 이름 재지정	20-8
런타임 시 필드 컴포넌트 메소드 사용	19-16	다른 테이블에서 데이터 가져오기	20-8
필드 값 표시, 변환 및 액세스	19-17	TQuery 사용	20-9
표준 컨트롤의 필드 컴포넌트 값 표시	19-17	이종 쿼리 생성	20-9
필드 값 변환	19-18	편집 가능한 결과 집합(Result set)	
기본 데이터셋 속성으로 필드 값		얻기	20-10
액세스	19-19	읽기 전용 결과 집합 업데이트	20-11
데이터셋의 Fields 속성으로 필드		TStoredProc 사용	20-12
값 액세스	19-19	매개변수 바인딩	20-12
데이터셋의 FieldByName 메소드로		Oracle 오버로드 내장 프로시저	
필드 값 액세스	19-20	사용	20-12
필드에 대한 기본값 설정	19-20	TDatabase를 갖는 데이터베이스에	
제약 조건 작업	19-21	연결	20-13
사용자 지정 제약 조건 만들기	19-21	데이터베이스 컴포넌트와 세션	
서버 제약 조건 사용	19-21	연결	20-13
객체 필드 사용	19-22	데이터베이스와 세션 컴포넌트	
ADT 및 배열 필드 표시	19-23	상호 작용 이해	20-13
ADT 필드 작업	19-23	데이터베이스 식별	20-14
영구적 필드 컴포넌트 사용	19-24	TDatabase를 사용하여 연결 열기	20-15
데이터셋의 FieldByName		데이터 모듈의 데이터베이스	
메소드 사용	19-24	컴포넌트 사용	20-16
데이터셋의 FieldValues 속성 사용	19-24	데이터베이스 세션 관리	20-17
ADT 필드의 FieldValues 속성 사용	19-24	세션 활성화	20-18
ADT 필드의 Fields 속성 사용	19-25	기본 데이터베이스 연결 동작 지정	20-19
배열 필드(Array fields) 작업	19-25	데이터베이스 연결 관리	20-19
영구적 필드 사용	19-25	암호 보호되는 Paradox 및 dBASE	
배열 필드의 FieldValues 속성 사용	19-26	테이블 사용	20-22
배열 필드의 Fields 속성 사용	19-26	Paradox 디렉토리 위치 지정	20-24
데이터셋 필드 표시	19-26	BDE 별칭 사용	20-25
중첩된 데이터셋의 데이터 액세스	19-27	세션에 대한 정보 검색	20-27
참조 필드(Reference fields) 작업	19-27	추가 세션 생성	20-28
참조 필드 표시	19-27	세션 이름 지정	20-29
참조 필드의 데이터 액세스	19-28	다중 세션 관리	20-29
		BDE에서 트랜잭션 사용	20-31
		Passthrough SQL 사용	20-31
		로컬 트랜잭션 사용	20-32
		BDE 사용하여 업데이트 캐시	20-33
		BDE 기반 캐시된 업데이트 활성화	20-34
		BDE 기반 캐시된 업데이트 적용	20-35
20 장			
Borland Database Engine 사용	20-1		
BDE 기반 아키텍처	20-1		
BDE 활성화 데이터셋 사용	20-2		

데이터베이스를 사용하여 캐시된 업데이트 적용	20-36
데이터셋 컴포넌트 메소드로 캐시된 업데이트 적용	20-36
OnUpdateRecord 이벤트 핸들러 생성	20-37
캐시된 업데이트 오류 처리	20-38
업데이트 객체를 사용하여 데이터셋 업데이트	20-40
업데이트 컴포넌트에 대한 SQL 문 생성	20-41
다중 업데이트 객체 사용	20-45
SQL 문 실행	20-46
TBatchMove 사용	20-49
배치 이동 컴포넌트 생성	20-49
배치 이동 모드 지정	20-50
레코드 추가	20-50
레코드 업데이트	20-51
레코드 추가 및 업데이트	20-51
데이터셋 복사	20-51
레코드 삭제	20-51
데이터 타입 매핑	20-51
배치 이동 실행	20-52
배치 이동 오류 처리	20-53
데이터 사전	20-53
BDE 사용을 위한 툴	20-55

21 장

ADO 컴포넌트 사용	21-1
ADO 컴포넌트 개요	21-1
ADO 데이터 저장소에 연결	21-2
TADOConnection을 사용하여 데이터 저장소에 연결	21-3
Connection 객체 액세스	21-4
연결 미세 조정 (fine-tuning)	21-5
강제적인 비동기 (asynchronous) 연결	21-5
시간 초과 제어	21-5
연결이 지원하는 작업 유형 나타내기	21-6
연결의 자동적인 트랜잭션 초기화 여부 지정	21-6
연결 명령 액세스	21-7
ADO 연결 이벤트	21-8
연결 시의 이벤트	21-8
연결 중지 시의 이벤트	21-8
트랜잭션 관리 시의 이벤트	21-8
기타 이벤트	21-9
ADO 데이터셋 사용	21-9

데이터 저장소에 ADO 데이터셋 연결	21-10
레코드셋 사용	21-10
북마크에 기반한 레코드 필터링	21-11
비동기식으로 레코드 페치	21-12
배치 업데이트 사용	21-12
파일에서 데이터 로드와 파일에 데이터 저장	21-15
TADODataset 사용	21-16
Command 객체 사용	21-17
명령 지정	21-17
Execute 메소드 사용	21-18
명령 취소	21-18
명령으로 결과 집합 검색	21-19
명령 매개변수 처리	21-19

22 장

단방향 데이터셋 사용	22-1
단방향 데이터셋의 타입	22-2
데이터베이스 서버에 연결	22-2
TSQLConnection 설정	22-3
드라이버 식별	22-3
연결 매개변수 지정	22-4
연결에 대한 설명 이름 지정	22-4
Connection Editor 사용	22-5
표시할 데이터 지정	22-6
쿼리의 결과 표시	22-6
테이블의 레코드 표시	22-7
TSQLDataSet을 사용하여 테이블 표시	22-7
TSQLTable을 사용하여 테이블 표시	22-7
내장 프로시저의 결과 표시	22-8
데이터 페치	22-8
데이터셋 준비	22-9
여러 데이터셋 페치	22-9
레코드를 반환하지 않는 명령 실행	22-10
실행할 명령 지정	22-10
명령 실행	22-11
서버 메타데이터 생성 및 수정	22-11
마스터/디테일 연결 커서 설정	22-12
스키마 정보 액세스	22-13
메타데이터를 단방향 데이터셋에 페치	22-13
메타데이터에 대한 데이터셋 사용한 후 데이터 페치	22-14
메타데이터 데이터셋의 구조	22-14
dbExpress 애플리케이션 디버깅	22-18
SQL 명령을 모니터링하기 위한 TSQLMonitor 사용	22-18

SQL 명령을 모니터하기 위한 콜백 사용	22-19	소스 데이터셋에서 매개변수 얻기	23-28
23 장		소스 데이터셋에 매개변수 전달	23-28
클라이언트 데이터셋 사용	23-1	쿼리 또는 내장 프로시저 매개변수 보내기	23-29
클라이언트 데이터셋을 사용하는 데이터 작업	23-2	매개변수로 레코드 제한	23-29
클라이언트 데이터셋에서 데이터 탐색	23-2	서버에서 제약 조건 처리	23-30
레코드 표시 제한	23-3	레코드 새로 고침	23-31
데이터 편집	23-5	사용자 지정 이벤트를 사용하여 프로바이더와 통신	23-32
변경 취소	23-6	소스 데이터셋 오버라이드	23-32
변경 내용 저장	23-6	파일 기반 데이터로 클라이언트 데이터셋 사용	23-33
데이터 값 제약 조건	23-7	새로운 데이터셋 생성	23-34
사용자 지정 제약 조건의 지정	23-7	파일이나 스트림에서 데이터 로드	23-34
정렬과 인덱싱	23-8	데이터에 변경 내용 병합	23-35
새 인덱스 추가	23-8	파일이나 스트림에 데이터 저장	23-35
인덱스 삭제와 전환	23-9	24 장	
인덱스를 사용하여 데이터 그룹화	23-10	프로바이더 컴포넌트 사용	24-1
계산된 값 표시	23-11	데이터의 소스 정하기	24-2
클라이언트 데이터셋에서 내부적으로 계산된 필드 사용	23-11	데이터 소스로 데이터셋 사용	24-2
유지 관리되는 집계 (Maintained Aggregates) 속성 사용	23-12	데이터 소스로 XML 문서 사용	24-2
Aggregates 지정	23-12	클라이언트 데이터셋과의 통신	24-3
레코드 그룹의 집계	23-13	데이터셋 프로바이더를 사용하여 업데이트 사항의 적용 방법 선택	24-4
집계 값 얻기	23-14	데이터 패킷에 포함될 정보 제어	24-4
다른 데이터셋에서 데이터 복사	23-14	데이터 패킷에 표시되는 필드 지정	24-5
데이터 직접 할당	23-14	데이터 패킷에 영향을 미치는 옵션 설정	24-5
클라이언트 데이터셋 커서 복제	23-15	데이터 패킷에 사용자 지정 정보 추가	24-7
데이터에 애플리케이션 특정 정보 추가	23-16	클라이언트 데이터 요청에 대한 응답	24-7
클라이언트 데이터셋을 사용하여 업데이트 캐시	23-16	클라이언트 업데이트 요청에 대한 응답	24-8
캐시된 업데이트 사용 개요	23-17	데이터베이스 업데이트 이전의 델타 패킷 편집	24-9
업데이트 캐시에 필요한 데이터셋 타입 선택	23-18	업데이트 적용 방법 변경	24-10
수정된 레코드 표시	23-19	개별적인 업데이트 (Individual updates) 스크린	24-11
레코드 업데이트	23-20	프로바이더에서 업데이트 오류 해결	24-11
업데이트 적용	23-21	단일 테이블을 나타내지 않는 데이터셋에 업데이트 적용	24-12
업데이트가 적용될 때 조정	23-22	클라이언트 생성 이벤트에 응답	24-12
업데이트 오류 해결	23-23	서버 제약 조건 처리	24-13
프로바이더와 함께 클라이언트 데이터셋 사용	23-25	25 장	
프로바이더 지정	23-25	다계층 (multi-tiered) 애플리케이션 생성	25-1
소스 데이터셋이나 문서에서 데이터 요청	23-26	다계층 데이터베이스 모델의 이점	25-2
점진적 페치 (Incremental fetching)	23-27		
요구 즉시 페치 (Fetch-on-demand)	23-27		

프로바이더 기반 다계층 애플리케이션에 대한 이해	25-2
3계층 애플리케이션 개요	25-3
클라이언트 애플리케이션의 구조	25-4
애플리케이션 서버의 구조	25-5
원격 데이터 모듈의 콘텐츠	25-6
트랜잭션 데이터 모듈(Transaction Data Module) 사용	25-6
원격 데이터 모듈 폴링	25-8
연결 프로토콜 선택	25-9
DCOM 연결 사용	25-9
소켓 연결 사용	25-9
웹 연결 사용	25-10
SOAP 연결 사용	25-10
CORBA 연결 사용	25-11
다계층 애플리케이션 구축	25-11
애플리케이션 서버 생성	25-12
원격 데이터 모듈 설정	25-13
TRemoteDataModule 구성	25-13
TMTSDataModule 구성	25-15
TSoapDataModule 구성	25-16
TCorbaDataModule 구성	25-16
애플리케이션 서버의 인터페이스 확장	25-17
애플리케이션 서버의 인터페이스에 콜백 추가	25-18
트랜잭션 애플리케이션 서버의 인터페이스 확장	25-18
다계층 애플리케이션의 트랜잭션 관리	25-19
마스터/디테일 관계 지원	25-19
원격 데이터 모듈의 상태 정보 지원	25-20
다중 원격 데이터 모듈 사용	25-22
애플리케이션 서버 등록	25-23
클라이언트 애플리케이션 생성	25-23
애플리케이션 서버에 연결	25-24
DCOM을 사용하여 연결 지정	25-25
소켓을 사용하여 연결 지정	25-25
HTTP를 사용하여 연결 지정	25-27
SOAP를 사용하여 연결 지정	25-27
CORBA를 사용하여 연결 지정	25-28
연결 브로커링 (Brokering connections)	25-28
서버 연결 관리	25-29
서버에 연결	25-30
서버 연결 끊기 및 바꾸기	25-30
서버 인터페이스 호출	25-30
다중 데이터 모듈을 사용하는 애플리케이션 서버에 대한 연결	25-32
웹 기반 클라이언트 애플리케이션 작성	25-33

클라이언트 애플리케이션을 ActiveX 컨트롤로 배포	25-34
클라이언트 애플리케이션에 Active Form 생성	25-34
InternetExpress를 사용하여 웹 애플리케이션 구축	25-35
InternetExpress 애플리케이션 구축	25-36
자바스크립트 라이브러리 사용	25-37
애플리케이션 서버의 액세스 및 실행 권한 부여	25-38
XML 브로커 사용	25-38
XML 데이터 패킷 페치	25-38
XML 델타 패킷(delta packets)의 업데이트 적용	25-39
InternetExpress 페이지 프로듀서로 웹 페이지 생성	25-40
웹 페이지 에디터 사용	25-41
웹 항목 속성 설정	25-42
InternetExpress 페이지 프로듀서 템플릿 사용자 지정	25-43

26 장

데이터베이스 애플리케이션에서 XML 사용	26-1
변환 정의	26-1
XML 노드와 데이터 패킷 필드 사이의 매핑	26-2
XMLMapper 사용	26-4
XML 스키마 또는 데이터 패킷 로딩	26-4
매핑 정의	26-5
변환 파일 생성	26-5
XML 문서를 데이터 패킷으로 변환	26-6
소스 XML 문서 지정	26-6
변환 지정	26-7
결과 데이터 패킷 가져오기	26-7
사용자 정의 노드 변환	26-7
XML 문서를 프로바이더의 소스로 사용	26-8
프로바이더의 클라이언트로 XML 문서 사용	26-9
프로바이더에서 XML 문서 페치(fetch)	26-9
XML 문서에서 프로바이더로 업데이트 적용	26-10

3 부

인터넷 애플리케이션 작성

27 장

인터넷 애플리케이션 생성	27-1
Web Broker 및 WebSnap 정보	27-1
용어 및 표준	27-2
URL(Uniform Resource Locator)의 각 부분	27-3
URI와 URL 비교	27-4
HTTP 요청 헤더 정보	27-4
HTTP 서버 활동	27-4
클라이언트 요청 구성	27-5
클라이언트 요청에 대한 작업	27-5
클라이언트 요청에 응답	27-6
웹 서버 애플리케이션의 유형	27-6
ISAPI 및 NSAPI	27-6
Apache	27-7
CGI 독립형	27-7
Win-CGI 독립형	27-7
서버 애플리케이션 디버깅	27-7
웹 애플리케이션 디버거 사용	27-7
웹 애플리케이션 디버거를 사용한 애플리케이션 실행	27-8
사용자의 애플리케이션을 다른 유형의 웹 서버 애플리케이션으로 변환	27-8
DLL인 웹 애플리케이션 디버깅	27-9
Windows NT에서의 디버깅	27-9
Windows 2000에서의 디버깅	27-9

28 장

Web Broker 사용	28-1
Web Broker로 웹 서버 애플리케이션 생성	28-1
웹 모듈	28-2
웹 애플리케이션 객체	28-3
Web Broker 애플리케이션의 구조	28-3
웹 디스패처	28-4
디스패처에 액션 추가	28-5
요청 메시지 디스패칭	28-5
액션 항목	28-6
액션 항목 실행 시기 결정	28-6
대상 URL	28-6
요청 메소드 유형	28-6
액션 항목 활성화 및 비활성화	28-7
기본 액션 항목 선택	28-7
액션 항목으로 요청 메시지에 응답	28-8
응답 보내기	28-8
여러 액션 항목 사용	28-8
클라이언트 요청 정보 액세스	28-9
요청 헤더 정보를 포함하는 속성	28-9

대상을 식별하는 속성	28-9
웹 클라이언트를 설명하는 속성	28-9
요청 목적을 식별하는 속성	28-10
예상 응답을 설명하는 속성	28-10
컨텐츠를 설명하는 속성	28-10
HTTP 요청 메시지의 내용	28-11
HTTP 응답 메시지 만들기	28-11
응답 헤더 채우기	28-11
응답 상태 표시	28-11
클라이언트 액션에 대한 요구 표시	28-12
서버 애플리케이션 설명	28-12
컨텐츠 설명	28-12
응답 컨텐츠 설정	28-12
응답 보내기	28-13
응답 메시지 컨텐츠 생성	28-13
페이지 프로듀서 컴포넌트 사용	28-14
HTML 템플릿	28-14
HTML 템플릿 지정	28-15
HTML 투명 태그 변환	28-15
액션 항목에서 페이지 프로듀서 사용	28-15
페이지 프로듀서 간의 연결	28-16
응답에서 데이터베이스 정보 사용	28-17
웹 모듈에 세션 추가	28-18
HTML로 데이터베이스 정보 표시	28-18
데이터셋 페이지 프로듀서 사용	28-18
테이블 프로듀서 사용	28-19
테이블 속성 지정	28-19
행 속성 지정	28-19
열 지정	28-20
HTML 문서에 테이블 포함시키기	28-20
데이터셋 테이블 프로듀서 설정	28-20
쿼리 테이블 프로듀서 설정	28-21

29 장

WebSnap 사용	29-1
WebSnap을 사용한 웹 서버 애플리케이션 생성	29-2
서버 타입	29-2
웹 애플리케이션 모듈 유형	29-3
웹 애플리케이션 모듈 옵션	29-3
애플리케이션 컴포넌트	29-4
웹 모듈	29-5
웹 데이터 모듈	29-5
웹 데이터 모듈 유닛의 구조	29-5
웹 데이터 모듈에 의해 구현되는 인터페이스	29-6
웹 페이지 모듈	29-6

페이지 프로듀서 컴포넌트	29-6	1 단계. 새 모듈 추가	29-21
페이지 이름	29-6	2 단계. 새 모듈 저장	29-21
프로듀서 템플릿	29-6	CountryTable 모듈에 데이터 컴포넌트	
웹 페이지 모듈이 구현하는		추가	29-21
인터페이스	29-7	1 단계. data-aware 컴포넌트 추가	29-21
웹 애플리케이션 모듈	29-7	2 단계. 키 필드 지정	29-22
웹 애플리케이션 데이터 모듈이		3 단계. 어댑터 컴포넌트 추가	29-22
구현하는 인터페이스	29-7	데이터를 표시할 그리드 생성	29-22
웹 애플리케이션 페이지 모듈이		1 단계. 그리드 추가	29-23
구현하는 인터페이스	29-8	2 단계. 그리드에 편집 명령 추가	29-23
어댑터(Adapters)	29-8	편집 폼 추가	29-24
Fields	29-8	1 단계. 새로운 모듈 추가	29-24
Actions	29-8	2 단계. 새 모듈 저장	29-24
Errors	29-8	3 단계. CountryTableU 유닛 사용	29-24
Records	29-8	4 단계. 입력 필드 추가	29-24
페이지 프로듀서	29-9	5 단계. 버튼 추가	29-25
템플릿	29-9	6 단계. 그리드 페이지에 폼 액션	
WebSnap의 서버측 스크립팅	29-9	연결	29-25
활성 스크립팅	29-9	7 단계. 폼 페이지에 그리드 액션	
스크립트 엔진	29-10	연결	29-26
스크립트 블록	29-10	오류 보고 추가	29-26
스크립트 작성	29-10	1 단계. 그리드에 오류 지원 추가	29-26
템플릿 마법사	29-10	2 단계. 폼에 오류 지원 추가	29-27
TAdapterPageProducer	29-10	3 단계. 오류 보고 메커니즘 테스트	29-27
스크립트 편집과 보기	29-11	완성된 애플리케이션 실행	29-27
페이지에 스크립트 포함	29-11		
스크립트 객체	29-11		
요청(Request) 디스패치	29-13		
WebContext	29-13		
디스패처 컴포넌트	29-13		
어댑터 디스패처 작업	29-13		
어댑터 컴포넌트를 사용하여 콘텐츠			
생성	29-13		
어댑터 요청(Request) 및			
응답(Response)	29-15		
액션 요청(Action request)	29-15		
액션 응답(Action response)	29-16		
이미지 요청(Image request)	29-17		
이미지 응답(Image response)	29-17		
액션 항목 디스패칭	29-18		
페이지 디스패처 작업	29-18		
WebSnap 자습서	29-19		
새 애플리케이션 생성	29-19		
1 단계. WebSnap 애플리케이션			
마법사 시작	29-19		
2 단계. 생성된 파일과 프로젝트			
저장	29-20		
3 단계. 애플리케이션 제목 지정	29-20		
CountryTable 페이지 생성	29-20		
		30 장	
		XML 문서 작업	30-1
		Document Object Model 사용	30-2
		XML 컴포넌트 작업	30-3
		TXMLDocument 사용	30-3
		XML 노드 작업	30-4
		노드의 값 사용	30-4
		노드의 속성 작업	30-5
		자식 노드 추가 및 삭제	30-5
		데이터 바인딩 마법사를 이용한 XML	
		문서 추출	30-5
		XML 데이터 바인딩 마법사 사용	30-7
		XML 데이터 바인딩 마법사가 생성하는	
		코드 사용	30-8
		31 장	
		Web Services 사용	31-1
		Web Services를 지원하는 서버 작성	31-2
		Web Services 서버 구축	31-2
		호출 가능한 인터페이스 정의	31-3

호출 가능한 인터페이스에서 복잡한 타입 사용	31-5
구현 생성 및 등록	31-6
Web Services에 대한 사용자 지정 예외 클래스 작성	31-7
Web Services 애플리케이션에 대한 WSDL 문서 생성	31-7
Web Services용 클라이언트 작성	31-8
WSDL 문서 import하기	31-8
호출 가능 인터페이스 호출	31-9

32 장

소켓 작업	32-1
서비스 구현	32-1
서비스 프로토콜 이해	32-2
애플리케이션과 통신	32-2
서비스와 포트	32-2
소켓 연결 유형	32-2
클라이언트 연결	32-3
리스닝(listening) 연결	32-3
서버 연결	32-3
소켓 설명	32-3
호스트 설명	32-4
호스트 이름 또는 IP 주소 선택	32-4
포트 사용	32-5
소켓 컴포넌트 사용	32-5
연결 정보 얻기	32-5
클라이언트 소켓 사용	32-6
원하는 서버 지정	32-6
연결 구성	32-6
연결 정보 얻기	32-6
연결 끊기	32-7
서버 소켓 사용	32-7
포트 지정	32-7
클라이언트 요청 리스닝(listening)	32-7
클라이언트에 연결	32-7
서버 연결 닫기	32-7
소켓 이벤트에 응답	32-8
오류 이벤트	32-8
클라이언트 이벤트	32-8
서버 이벤트	32-9
리스닝(listening) 시의 이벤트	32-9
클라이언트 연결 시의 이벤트	32-9
소켓 연결을 통한 읽기 및 쓰기	32-9
비차단 연결(Non-blocking connections)	32-10
읽기 및 쓰기 이벤트	32-10
차단 연결(Blocking connections)	32-10

4 부

COM 기반 애플리케이션 개발

33 장

COM 기술 개요	33-1
사양 및 구현으로서의 COM	33-1
COM 확장	33-2
COM 애플리케이션의 구성 요소	33-3
COM 인터페이스	33-3
기본 COM 인터페이스, IUnknown	33-4
COM 인터페이스 포인터	33-4
COM 서버	33-5
CoClass 및 클래스 팩토리	33-6
In-process, Out-of-process 및 원격 서버	33-6
마샬링 메커니즘 (Marshaling mechanism)	33-8
집합체	33-9
COM 클라이언트	33-9
COM 확장	33-10
Automation 서버	33-12
Active Server Page	33-13
ActiveX 컨트롤	33-13
활성 문서	33-14
트랜잭션 객체	33-14
타입 라이브러리	33-15
타입 라이브러리의 내용	33-16
타입 라이브러리 생성	33-16
타입 라이브러리 사용 시기	33-16
타입 라이브러리 액세스	33-17
타입 라이브러리 사용의 이점	33-17
타입 라이브러리 도구 사용	33-18
마법사로 COM 객체 구현	33-18
마법사가 생성하는 코드	33-21

34 장

Type Library 작업	34-1
Type Library 에디터	34-2
Type Library 에디터의 각 부분	34-3
툴바	34-3
객체 목록 창	34-5
상태 표시줄	34-5
타입 정보 페이지	34-6
타입 라이브러리 요소	34-8
Interface	34-8
Dispinterface	34-9
CoClasses	34-9

타입 정의	34-9
모듈	34-10
Type Library 에디터 사용	34-10
유효한 타입	34-11
오브젝트 파스칼 또는 IDL 구문 사용	34-13
새로운 타입 라이브러리 생성	34-19
기존 타입 라이브러리 열기	34-19
타입 라이브러리에 interface 추가	34-20
타입 라이브러리를 사용하여 interface 수정	34-20
interface나 dispinterface에 속성 및 메소드 추가	34-21
타입 라이브러리에 CoClass 추가	34-22
CoClass에 인터페이스 추가	34-22
타입 라이브러리에 열거 추가	34-23
타입 라이브러리에 별칭 추가	34-23
타입 라이브러리에 레코드 또는 합집합 추가	34-23
타입 라이브러리에 모듈 추가	34-24
타입 라이브러리 정보의 저장 및 등록	34-24
Apply Updates 대화 상자	34-25
타입 라이브러리 저장	34-25
타입 라이브러리 새로 고침	34-25
타입 라이브러리 등록	34-26
IDL 파일 export하기	34-26
타입 라이브러리 배포	34-26

35 장

COM 클라이언트 생성	35-1
타입 라이브러리 정보 Import하기	35-2
Import Type Library 대화 상자 사용	35-3
Import ActiveX 대화 상자 사용	35-4
타입 라이브러리 정보를 import할 때 생성되는 코드	35-5
import된 객체 제어	35-6
컴포넌트 래퍼 사용	35-6
ActiveX 래퍼	35-6
Automation 객체 래퍼 (object wrappers)	35-7
data-aware ActiveX 컨트롤 사용	35-8
예: Microsoft Word로 문서 인쇄	35-10
1 단계: 이 예제를 위한 Delphi 준비	35-10
2 단계: Word 타입 라이브러리 import하기	35-10

3 단계: Microsoft Word를 제어하기 위해 Vtable 또는 디스패치 인터페이스 객체 사용	35-11
4 단계: 예제 클린업	35-12
타입 라이브러리 정의를 기반으로 클라이언트 코드 작성	35-12
서버에 연결	35-13
이중 인터페이스를 사용하여 Automation 서버 제어	35-13
디스패치 인터페이스를 사용하여 Automation 서버 제어	35-13
Automation 컨트롤러의 이벤트 처리	35-14
타입 라이브러리가 없는 서버에 대한 클라이언트 생성	35-16

36 장

일반 COM 서버 생성	36-1
COM 객체 생성에 대한 개요	36-1
COM 객체 디자인	36-2
COM 객체 마법사 사용	36-2
Automation 객체 마법사 사용	36-4
COM 객체 인스턴스 타입	36-5
스레드 모델 선택	36-6
free thread 모델을 지원하는 객체 작성	36-7
apartment threading model을 지원하는 객체 작성	36-8
Neutral threading model을 지원하는 객체 작성	36-9
COM 객체의 인터페이스 정의	36-9
객체의 인터페이스에 속성 추가	36-9
객체의 인터페이스에 메소드 추가	36-10
클라이언트에 이벤트 노출	36-10
Automation 객체의 이벤트 관리	36-12
Automation 인터페이스	36-12
이중 인터페이스	36-13
디스패치 인터페이스	36-14
사용자 지정 인터페이스	36-15
데이터 마샬링(Marshaling)	36-15
Automation 호환 타입	36-15
자동 마샬링을 위한 타입 제한 사항	36-16
사용자 지정 마샬링	36-16
COM 객체 등록	36-17
In-process 서버 등록	36-17
Out-of-process 서버 등록	36-17
애플리케이션 테스트 및 디버깅	36-18

37 장

Active Server Page 생성	37-1
Active Server Object 생성	37-2
ASP intrinsics 사용	37-3
애플리케이션	37-4
Request	37-4
Response	37-5
Session	37-6
Server	37-6
in-process 또는 out-of-process	
서버용 ASP 생성	37-7
Active Server Object 등록	37-7
In-process 서버 등록	37-8
Out-of-process 서버 등록	37-8
Active Server Page 애플리케이션의	
테스트 및 디버깅	37-8

38 장

ActiveX 컨트롤 생성	38-1
ActiveX 컨트롤 생성에 대한 개요	38-2
ActiveX 컨트롤의 요소	38-2
VCL 컨트롤	38-3
ActiveX 랩퍼	38-3
타입 라이브러리	38-3
속성 페이지	38-3
ActiveX 컨트롤 디자인	38-4
VCL 컨트롤로부터 ActiveX 컨트롤 생성	38-4
VCL 폼을 기반으로 ActiveX 컨트롤 생성	38-5
ActiveX 컨트롤 라이선싱	38-6
ActiveX 컨트롤의 인터페이스 사용자 지정	38-7
다른 속성, 메소드, 이벤트 추가	38-8
속성 및 메소드 추가	38-8
이벤트 추가	38-10
타입 라이브러리를 이용한 간단한	
데이터 바인딩 활성화	38-10
ActiveX 컨트롤에 대한 속성 페이지 생성	38-12
새 속성 페이지 생성	38-12
속성 페이지에 컨트롤 추가	38-12
속성 페이지 컨트롤을 ActiveX 컨트롤	
속성에 연결	38-13
속성 페이지 업데이트	38-13
객체 업데이트	38-13
속성 페이지를 ActiveX 컨트롤에 연결	38-14
ActiveX 컨트롤 등록	38-14
ActiveX 컨트롤 테스트	38-15
웹에 ActiveX 컨트롤 배포	38-15
옵션 설정	38-16

39 장

MTS 또는 COM+ 객체 생성	39-1
트랜잭션 객체에 대한 이해	39-2
트랜잭션 객체에 대한 요구 사항	39-3
리소스 관리	39-3
객체 컨텍스트 액세스	39-4
Just-in-time 활성화	39-4
리소스 풀링	39-5
데이터베이스 리소스 디스펜서	39-5
Shared Property Manager	39-6
리소스 해제	39-8
객체 풀링	39-8
MTS 및 COM+ 트랜잭션 지원	39-8
트랜잭션 속성	39-9
트랜잭션 속성 설정	39-10
상태 있는(stateful) 객체 및	
상태 없는(stateless) 객체	39-11
트랜잭션의 완료 방법에 대한 영향	39-11
트랜잭션 초기화	39-12
클라이언트쪽에 트랜잭션 객체 설치	39-12
서버쪽에 트랜잭션 객체 설치	39-13
트랜잭션 시간 초과	39-14
역할 기반 보안(Role-based security)	39-14
트랜잭션 객체 생성에 대한 개요	39-15
Transactional Object 마법사 사용	39-15
트랜잭션 객체용 스레드 모델 선택	39-17
활동	39-17
COM+에서 이벤트 생성	39-18
Event Object 마법사 사용	39-19
COM+ 이벤트 객체를 사용하여	
이벤트 발생	39-20
객체 참조 전달	39-20
SafeRef 메소드 사용	39-20
콜백	39-21
트랜잭션 객체 디버깅 및 테스트	39-21
트랜잭션 객체 설치	39-22
트랜잭션 객체 관리	39-23

5 부

사용자 지정 컴포넌트 생성

40 장

컴포넌트 생성 개요	40-1
VCL 및 CLX	40-1
컴포넌트 및 클래스	40-2
컴포넌트 생성 방법	40-2
기존 컨트롤 수정	40-3

창 있는 컨트롤 생성	40-3	상속된 속성 게시	42-3
그래픽 컨트롤 생성	40-4	속성 정의	42-3
Windows 컨트롤 하위 분류	40-4	속성 선언	42-3
년비주얼 컴포넌트 생성	40-5	내부 데이터 저장소	42-4
컴포넌트 작성 시 고려 사항	40-5	직접 액세스	42-4
중속성 제거	40-5	액세스 메소드	42-5
속성, 메소드 및 이벤트	40-6	read 메소드	42-6
속성	40-6	write 메소드	42-6
이벤트	40-6	기본 속성 값	42-7
메소드	40-6	기본값을 갖지 않도록 지정	42-7
그래픽 캡슐화	40-7	배열 속성 생성	42-8
등록	40-8	하위 컴포넌트의 속성 생성	42-9
새 컴포넌트 생성	40-8	인터페이스의 속성 생성	42-10
Component 마법사 사용	40-9	속성 저장 및 로드	42-11
수동으로 컴포넌트 생성	40-11	저장 및 로드 메커니즘 사용	42-11
유닛 파일 생성	40-11	기본값 지정	42-11
컴포넌트 파생	40-11	저장할 대상 결정	42-12
컴포넌트 등록	40-12	로드 후 초기화	42-13
설치되지 않은 컴포넌트 테스트	40-12	Published가 아닌 속성 저장 및 로드	42-13
설치된 컴포넌트 테스트	40-14	속성 값을 저장 및 로드하는 메소드 생성	42-14
		DefineProperties 메소드 오버라이드	42-14

41 장

컴포넌트 작성자를 위한 객체 지향 프로그래밍	41-1
새 클래스 정의	41-1
새 클래스 파생	41-2
반복을 피하기 위한 클래스 기본값 변경	41-2
클래스에 새 기능 추가	41-2
새 컴포넌트 클래스 선언	41-3
조상, 자손 및 클래스 계층 구조	41-3
액세스 제어	41-4
구현 세부 사항 숨기기	41-4
컴포넌트 작성자의 인터페이스 정의	41-5
런타임 인터페이스 정의	41-6
디자인 타임 인터페이스 정의	41-6
메소드 디스패칭	41-7
정적 메소드	41-7
가상 메소드	41-8
메소드 오버라이드	41-8
추상 클래스 멤버	41-9
클래스 및 포인터	41-9

42 장

속성 생성	42-1
속성을 생성하는 이유	42-1
속성 타입	42-2

43 장

이벤트 생성	43-1
이벤트의 개념	43-1
이벤트는 메소드 포인터	43-2
이벤트는 속성	43-2
이벤트 타입은 메소드 포인터 타입	43-3
이벤트 핸들러 타입은 프로시저	43-3
이벤트 핸들러는 옵션	43-4
표준 이벤트 구현	43-4
표준 이벤트 식별	43-4
모든 컨트롤에 대한 표준 이벤트	43-5
표준 컨트롤에 대한 표준 이벤트	43-5
이벤트 보이기	43-5
표준 이벤트 처리 변경	43-6
사용자 고유 이벤트 정의	43-6
이벤트 트리거	43-6
두 종류의 이벤트	43-7
핸들러 타입 정의	43-7
간단한 공지	43-7
이벤트 특정 핸들러	43-7
핸들러의 정보 반환	43-8
이벤트 선언	43-8
"On"으로 시작하는 이벤트 이름	43-8
이벤트 호출	43-8

44 장

메소드 생성	44-1
종속성 피하기	44-1
메소드 이름 지정	44-2
메소드 보호	44-2
Public이어야 하는 메소드	44-3
Protected이어야 하는 메소드	44-3
추상 메소드	44-3
가상 메소드 만들기	44-3
메소드 선언	44-4

45 장

컴포넌트에서 그래픽 사용	45-1
그래픽의 개요	45-1
캔버스 사용	45-3
그림 작업	45-3
그림, 그래픽 또는 캔버스 사용	45-4
그래픽 로드 및 저장	45-4
팔레트 처리	45-5
컨트롤에 팔레트 지정	45-5
오프스크린(off-screen) 비트맵	45-6
오프스크린 비트맵 생성 및 관리	45-6
비트맵 이미지 복사	45-7
변화에 대한 반응	45-7

46 장

메시지 처리	46-1
메시지 처리 시스템에 대한 이해	46-1
Windows 메시지 종류	46-2
메시지 디스패칭	46-2
메시지 흐름 추적	46-3
메시지 처리 변경	46-3
핸들러 메소드 오버라이드	46-3
메시지 매개변수 사용	46-4
메시지 트래핑(trapping)	46-4
새 메시지 핸들러 생성	46-5
사용자 고유의 메시지 정의	46-5
메시지 식별자 선언	46-6
메시지 레코드 타입 선언	46-6
새 메시지 처리 메소드 선언	46-7

47 장

디자인 타임 시 컴포넌트 사용	47-1
컴포넌트 등록	47-1
Register 프로시저 선언	47-2

Register 프로시저 작성	47-2
컴포넌트 지정	47-2
팔레트 페이지 지정	47-3
RegisterComponents 함수 사용	47-3
팔레트 비트맵 추가	47-3
사용자의 컴포넌트에 도움말 제공	47-4
도움말 파일 생성	47-4
항목 생성	47-4
문맥에 따른 컴포넌트 도움말 만들기	47-6
컴포넌트 도움말 파일 추가	47-6
속성 편집기 추가	47-6
속성 편집기 클래스 파생	47-7
텍스트로 속성 편집	47-8
속성 값 표시	47-8
속성 값 설정	47-9
전체 속성 편집	47-10
편집기 속성 지정	47-10
속성 편집기 등록	47-12
속성 범주	47-13
한 번에 하나의 속성 등록	47-13
한 번에 여러 속성 등록	47-14
속성 범주 지정	47-14
IsPropertyInCategory 함수 사용	47-15
컴포넌트 에디터 추가	47-15
컨텍스트 메뉴에 항목 추가	47-16
메뉴 항목 지정	47-16
명령 구현	47-17
더블 클릭 동작 변경	47-17
클립보드 형식 추가	47-18
컴포넌트 에디터 등록	47-19
컴포넌트를 패키지로 컴파일	47-19

48 장

기존 컴포넌트 수정	48-1
컴포넌트 생성 및 등록	48-1
컴포넌트 클래스 수정	48-2
생성자 오버라이드	48-2
새 기본 속성 값 지정	48-3

49 장

그래픽 컴포넌트 생성	49-1
컴포넌트 생성 및 등록	49-1
상속된 속성 게시	49-2
그래픽 기능 추가	49-2
그릴 대상 결정	49-3
속성 타입 선언	49-3
속성 선언	49-3

구현 메소드 작성	49-4
생성자 및 소멸자 오버라이드	49-4
기본 속성 값 변경	49-4
펜과 브러시 게시	49-5
클래스 필드 선언	49-5
액세스 속성 선언	49-6
소유된 클래스 초기화	49-7
소유된 클래스의 속성 설정	49-7
컴포넌트 이미지 그리기	49-8
정교한 도형 그리기	49-9

50 장

그리드 사용자 지정	50-1
컴포넌트 생성 및 등록	50-1
상속된 속성 게시	50-2
초기 값 변경	50-3
셀 크기 조정	50-4
셀 채우기	50-5
날짜 추적	50-5
내부 날짜 저장	50-6
일, 월, 연도 액세스	50-6
일(day) 수 생성	50-8
오늘 날짜 선택	50-9
월과 연도 탐색	50-10
일(day) 탐색	50-11
선택 이동	50-11
OnChange 이벤트 제공	50-11
빈 셀 배제	50-12

51 장

Data_aware 컨트롤 만들기	51-1
데이터 찾아보기 컨트롤 생성	51-1
컴포넌트 생성 및 등록	51-2
읽기 전용 컨트롤 만들기	51-2
ReadOnly 속성 추가	51-3
필요한 업데이트 허용	51-3
데이터 연결 추가	51-4
클래스 필드선언	51-4
액세스 속성 선언	51-5
액세스 속성 선언의 예제	51-5
데이터 연결 초기화	51-6
데이터 변경 내용에 응답	51-6
데이터 편집 컨트롤 생성	51-7
FReadOnly의 기본값 변경	51-8
마우스 다운 및 키 다운 메시지 처리	51-8
마우스 다운 메시지에 응답	51-8
키 다운 메시지에 응답	51-9

필드 데이터 연결 클래스 업데이트	51-10
Change 메소드 수정	51-11
데이터셋 업데이트	51-11

52 장

대화 상자를 컴포넌트로 만들기	52-1
컴포넌트 인터페이스 정의	52-1
컴포넌트 생성 및 등록	52-2
컴포넌트 인터페이스 생성	52-3
폼 유닛 포함	52-3
인터페이스 속성 추가	52-3
Execute 메소드 추가	52-4
컴포넌트 테스트	52-6

색인

I-1

표

1.1	글꼴과 기호	1-3	10.4	VCL와 CLX의 유닛	10-10
3.1	중요한 기본 클래스	3-13	10.5	CLX에는 있고 VCL에는 없는 유닛	10-12
3.2	그래픽 컨트롤	3-17	10.6	VCL에만 있는 유닛	10-12
3.3	컴포넌트 팔레트 페이지	3-29	10.7	Linux와 Windows 운영 환경의 차이점	10-14
3.4	텍스트 컨트롤 속성	3-32	10.8	공통된 Linux 디렉토리	10-17
3.5	목록의 생성 및 관리를 위한 컴포넌트	3-49	10.9	시스템 이벤트에 응답하기 위한 TWidgetControl protected 메소드	10-22
4.1	RTL 예외	4-9	10.10	유사한 데이터 액세스 컴포넌트	10-25
4.2	오브젝트 파스칼 문자 타입	4-40	10.11	캐시된 업데이트를 위한 속성, 메소드 및 이벤트	10-29
4.3	문자열 비교 루틴	4-44	11.1	컴파일된 패키지 파일	11-2
4.4	대소문자 변환 루틴	4-44	11.2	패키지 특정 컴파일러 지시어	11-10
4.5	문자열 수정 루틴	4-45	11.3	패키지 특정 명령줄 컴파일러 스위치	11-12
4.6	부분 문자열 루틴	4-45	11.4	컴파일된 패키지 파일	11-12
4.7	문자열 처리 루틴	4-45	12.1	BiDi를 지원하는 VCL 객체	12-4
4.8	문자열에 대한 컴파일러 지시어	4-49	12.2	문자열 길이 추정	12-9
4.9	속성 상수 및 값	4-52	13.1	애플리케이션 파일	13-3
4.10	파일 I/O를 위한 파일 타입	4-54	13.2	Merge 모듈과 증속성	13-4
4.11	개방 모드	4-55	13.3	독립형 실행 파일로 dbExpress 배포	13-7
4.12	공유 모드	4-55	13.4	드라이버 DLL이 있는 dbExpress 배포	13-8
4.13	각 개방 모드에서 사용할 수 있는 공유 모드	4-56	13.5	SQL 데이터베이스 클라이언트 소프트웨어 파일	13-9
5.1	라이브러리용 컴파일러 지시어	5-9	15.1	데이터 컨트롤	15-2
5.2	컴포넌트 팔레트의 Database 페이지	5-10	15.2	열 속성	15-20
5.3	웹 서버 애플리케이션	5-12	15.3	확장된 TColumn Title 속성	15-20
5.4	데이터 모듈에 대한 컨텍스트 메뉴 옵션	5-16	15.4	복합 필드가 나타나는 방식에 영향을 주는 속성	15-23
5.5	Tapplication의 도움말 메소드	5-29	15.5	Expanded TDBGrid Options 속성	15-24
6.1	작업 설정 용어	6-17	15.6	그리드 컨트롤 이벤트	15-26
6.2	Action manager의 PrioritySchedule 속성 기본값	6-22	15.7	선택된 데이터베이스 컨트롤 그리드 속성	15-28
6.3	Action 클래스	6-27	15.8	TDBNavigator 버튼	15-29
6.4	예제 캡션 및 파생된 이름	6-32	17.1	데이터베이스 연결 컴포넌트	17-1
6.5	메뉴 디자이너 컨텍스트 메뉴 명령	6-38	18.1	데이터셋 State 속성의 값	18-3
6.6	스피드 버튼의 모양 설정	6-45	18.2	데이터셋의 탐색 메소드	18-5
6.7	툴 버튼 모양 설정	6-47	18.3	데이터셋의 탐색 속성	18-6
6.8	콜 버튼 모양 설정	6-49	18.4	필터에 나타날 수 있는 비교 및 논리 연산자	18-14
7.1	선택한 텍스트의 속성	7-9	18.5	FilterOptions 값	18-15
7.2	Fixed와 variable owner-draw 스타일	7-13	18.6	필터링된 데이터셋의 탐색 메소드	18-16
8.1	그래픽 객체 형식	8-3	18.7	데이터 삽입, 업데이트 및 삭제를 위한 데이터셋 메소드	18-17
8.2	캔버스 객체의 일반적인 속성	8-4	18.8	전체 레코드에 적용되는 메소드	18-21
8.3	캔버스 객체의 일반적인 속성	8-4	18.9	인덱스 기반 검색 메소드	18-28
8.4	마우스 이벤트 매개변수	8-24			
8.5	멀티미디어 장치 타입 및 기능	8-32			
9.1	스레드 우선 순위	9-3			
9.2	WaitFor 반환 값	9-10			
10.1	포팅 기법	10-3			
10.2	CLX 부분	10-6			
10.3	변경되었거나 다른 기능	10-9			

19.1 데이터 표시에 영향을 주는 TFloatField 속성	19-1	24.1 AppServer 인터페이스 멤버	24-3
19.2 특수한 영구적 필드 종류	19-6	24.2 프로바이더 옵션	24-5
19.3 필드 컴포넌트 속성	19-11	24.3 UpdateStatus 값	24-9
19.4 필드 컴포넌트 서식 루틴	19-15	24.4 UpdateMode 값	24-10
19.5 필드 컴포넌트 이벤트	19-15	24.5 ProviderFlags 값	24-10
19.6 선택된 필드 컴포넌트 메소드	19-16	25.1 다계층 애플리케이션에 사용되는 컴포넌트	25-3
19.7 특수한 변환 결과	19-19	25.2 연결 컴포넌트	25-5
19.8 객체 필드 컴포넌트의 타입	19-22	25.3 자바스크립트 라이브러리	25-37
19.9 공통 객체 필드 자손 속성	19-23	27.1 Web Broker와 WebSnap 비교	27-2
20.1 파일 확장자에 기반한 BDE에 의해 인식되는 테이블 타입	20-5	27.2 웹 서버 애플리케이션 컴포넌트	27-6
20.2 TableType 값	20-6	28.1 MethodType 값	28-6
20.3 BatchMove의 import 모드	20-8	33.1 COM 객체 요구 사항	33-12
20.4 세션 컴포넌트의 데이터베이스 관련 정보 메소드	20-27	33.2 COM, Automation 및 ActiveX 객체를 구현하는 Delphi 마법사	33-20
20.5 TSessionList 속성과 메소드	20-30	33.3 생성된 구현 클래스에 대한 DAX 기본 클래스	33-22
20.6 캐시된 업데이트를 위한 속성, 메소드 및 이벤트	20-33	34.1 Type Library 에디터 파일	34-2
20.7 UpdateKind 값	20-39	34.2 Type Library 에디터의 각 부분	34-3
20.8 배치 이동 모드	20-50	34.3 타입 라이브러리 페이지	34-6
20.9 데이터 사전 인터페이스	20-54	34.4 유효한 타입	34-11
21.1 ADO 컴포넌트	21-2	34.5 속성 구문	34-13
21.2 ADO 연결 모드	21-6	36.1 COM 객체의 스레드 모델	36-6
21.3 ADO 데이터셋의 실행 옵션	21-12	37.1 IApplicationObject 인터페이스 멤버	37-4
21.4 ADO와 클라이언트 데이터셋 캐시된 업데이트 비교	21-12	37.2 IRequest 인터페이스 멤버	37-4
22.1 테이블을 나열하는 메타데이터의 테이블에 있는 열	22-15	37.3 IResponse 인터페이스 멤버	37-5
22.2 내장 프로시저를 나열하는 메타데이터의 테이블에 있는 열	22-15	37.4 ISessionObject 인터페이스 멤버	37-6
22.3 필드를 나열하는 메타데이터의 테이블에 있는 열	22-16	37.5 IServer 인터페이스 멤버	37-6
22.4 인덱스를 나열하는 메타데이터의 테이블에 있는 열	22-17	39.1 트랜잭션 지원을 위한 IObjectContext 메소드	39-11
22.5 매개변수를 나열하는 메타데이터의 테이블에 있는 열	22-17	39.2 트랜잭션 객체용 스레드 모델	39-17
23.1 클라이언트 데이터셋에서의 필터 지원	23-3	39.3 호출 동기화 옵션	39-18
23.2 유지 관리되는 집계의 요약 연산자	23-12	40.1 컴포넌트 생성 기본 작업	40-3
23.3 업데이트 캐시에 필요한 특수 클라이언트 데이터셋	23-18	41.1 객체 내의 가시성 레벨	41-4
		42.1 Object Inspector에서 속성이 표시되는 방법	42-2
		45.1 캔버스 기능 요약	45-3
		45.2 이미지 복사 메소드	45-7
		47.1 이미 정의된 속성 편집기 타입	47-7
		47.2 속성 값을 읽고 쓰기 위한 메소드	47-8
		47.3 속성 편집기 특성 플래그	47-11
		47.4 속성 범주	47-14

그림

3.1	단순한 폼	3-6	15.4	Expanded가 True로 설정된 TDBGrid 컨트롤	15-23
3.2	객체, 컴포넌트 및 컨트롤	3-12	15.5	디자인 타임 시 TDBCtrlGrid.	15-28
3.3	단순화된 계층 구조	3-13	15.6	TDBNavigator 컨트롤의 버튼	15-29
3.4	트랙 표시줄 컴포넌트의 세 가지 보기	3-34	16.1	디자인 타임의 의사 결정 지원 컴포넌트	16-2
3.5	진행 표시줄	3-44	16.2	1차원 크로스탭	16-3
6.1	data-aware 컨트롤과 데이터 소스 컴포넌트를 갖는 프레임	6-15	16.3	3차원 크로스탭	16-3
6.2	Action Manager 에디터.	6-20	16.4	다른 의사 결정 소스에 바인딩된 의사 결정 그래프	16-14
6.3	메뉴 용어	6-29	20.1	BDE 기반 애플리케이션의 컴포넌트	20-2
6.4	MainMenu 컴포넌트 및 PopupMenu 컴포넌트	6-30	25.1	웹 기반 다계층 데이터베이스 애플리케이션	25-33
6.5	팝업 메뉴의 메뉴 디자이너.	6-31	27.1	URL(Uniform Resource Locator)의 각 부분.	27-3
6.6	메인 메뉴의 메뉴 디자이너.	6-31	28.1	서버 애플리케이션의 구조.	28-4
6.7	중첩 메뉴 구조	6-35	29.1	컨텐츠 흐름 생성	29-15
6.8	Select Menu 대화 상자	6-38	29.2	액션 요청과 응답	29-17
6.9	메뉴에 대한 Insert Template 대화 상자	6-39	29.3	요청에 대한 이미지 응답	29-18
6.10	메뉴에 대한 Save Template 대화 상자	6-40	29.4	페이지 디스패칭	29-19
8.1	BMPDlg 유닛의 Bitmap-dimension 대화상자.	8-21	33.1	COM 인터페이스	33-3
12.1	bdLeftToRight로 설정된 TListBox	12-7	33.2	인터페이스 vtable	33-5
12.2	bdRightToLeft로 설정된 TListBox	12-7	33.3	In-process 서버	33-7
12.3	bdRightToLeftNoAlign로 설정된 TListBox	12-7	33.4	Out-of-process 및 원격 서버	33-8
12.4	bdRightToLeftReadingOnly로 설정된 TListBox	12-7	33.5	COM 기반 기술	33-11
14.1	일반 데이터베이스 아키텍처	14-6	33.6	간단한 COM 객체 인터페이스	33-19
14.2	데이터베이스 서버에 직접 연결	14-8	33.7	Automation 객체 인터페이스	33-19
14.3	파일 기반 데이터베이스 애플리케이션	14-9	33.8	ActiveX 객체 인터페이스	33-20
14.4	클라이언트 데이터셋과 다른 데이터셋을 조합하는 아키텍처	14-12	33.9	Delphi ActiveX 프레임워크	33-22
14.5	다계층 데이터베이스 아키텍처	14-13	34.1	Type Library 에디터	34-3
15.1	TDBGrid 컨트롤	15-15	34.2	객체 목록 창	34-5
15.2	False로 설정된 ObjectView를 가지는 TDBGrid 컨트롤	15-23	36.1	이중 인터페이스 VTable	36-14
15.3	Expanded가 False로 설정된 TDBGrid 컨트롤	15-23	38.1	디자인 모드에서의 Mask Edit 속성 페이지	38-13
			40.1	Visual Component Library 클래스 계층 구조.	40-2
			40.2	컴포넌트 마법사	40-9

1

서문

*개발자 안내서*는 클라이언트/서버 데이터베이스 애플리케이션 제작, 사용자 지정 컴포넌트 작성 및 인터넷 웹 서버 애플리케이션 만들기 등과 같은 중급 및 고급 수준의 개발에 관한 주제를 설명합니다. 개발자 안내서를 사용하면 사용자가 SOAP, TCP/IP, COM+ 및 ActiveX를 비롯한 여러 산업 표준에 맞도록 애플리케이션을 만들 수 있습니다. 웹 개발, 고급 XML 기술 및 데이터베이스 개발을 지원하는 많은 고급 기능에는 일부 Delphi 버전에서만 사용할 수 있는 컴포넌트와 마법사가 필요합니다.

*개발자 안내서*를 사용하는 사람은 Delphi를 사용하는 데 익숙하고 기본적인 Delphi 프로그래밍 기술을 이해하고 있다고 가정합니다. Delphi 프로그래밍과 통합 개발 환경(IDE)에 대한 소개는 입문서 설명서와 온라인 도움말을 참조하십시오.

설명서 내용

이 설명서는 다음과 같이 다섯 개의 부로 나뉘어 있습니다.

- **1부 "Delphi 프로그래밍"**에서는 범용 애플리케이션을 제작하는 방법에 대해 설명합니다. 이 부분에서는 애플리케이션에서 사용할 수 있는 프로그래밍 기술에 대한 세부적인 내용을 제공합니다. 예를 들어, 사용자 인터페이스 프로그래밍을 쉽게 해주는 일반적인 비주얼 컴포넌트 라이브러리(VCL)나 크로스 플랫폼용 컴포넌트 라이브러리(C LX, 클릭스)를 사용하는 방법에 대해 설명합니다. 객체는 문자열 처리, 텍스트 조작 및 일반 대화 상자 등에 사용할 수 있습니다. 또한 그래픽, 오류와 예외 처리 DLL 사용, OLE 자동화 및 국제적인 애플리케이션 작성 등의 작업에 대한 장들이 있습니다.

Windows와 Linux 플랫폼 모두에서 컴파일되고 실행될 수 있는 애플리케이션을 개발하기 위해 Borland의 크로스 플랫폼용 컴포넌트 라이브러리(C LX, 클릭스)의 객체를 사용하는 방법을 설명한 장도 있습니다.

배포에 대한 장에서는 애플리케이션 사용자들에게 애플리케이션 배포와 관련된 작업에 대해 자세히 설명합니다. 예를 들어 효과적인 컴파일러 선택, InstallShield Express 사용 및 라이선스 문제에 대한 정보와 애플리케이션의 실제 사용 가능한 버전을 만들 때 사용할 패키지, DLL 및 다른 라이브러리에 대한 정보가 들어 있습니다.

- **2부 "데이터베이스 애플리케이션 개발"**에서는 데이터베이스 툴과 컴포넌트를 사용하여 데이터베이스 애플리케이션을 제작하는 방법에 대해 설명합니다. Delphi는 Paradox나 dBASE와 같은 지역 데이터베이스와 InterBase, Oracle 및 Sybase와 같은 Network SQL 서버 데이터베이스를 비롯하여 여러 가지 유형의 데이터베이스에 액세스할 수 있게 해줍니다. 사용자는 dbExpress, Borland Database Engine, InterbaseExpress 및 ADO를 포함한 다양한 데이터 액세스 메커니즘에서 선택할 수 있습니다. 보다 수준 높은 데이터베이스 애플리케이션을 구현하는 데 필요한 기능들을 모든 Delphi 버전에서 사용할 수 있는 것은 아닙니다.
- **3부 "인터넷 애플리케이션 작성"**에서는 인터넷에서 배포되는 애플리케이션을 만드는 방법에 대해 설명합니다. Delphi는 웹 서버 애플리케이션 제작을 위한 도구들의 와이드 배열을 포함하고 있으며 여기에는 크로스 플랫폼 서버 애플리케이션 작성을 도와 주는 Web Broker 아키텍처, GUI 환경에서의 웹 페이지 디자인을 도와 주는 WebSnap, XML 문서 작업에 대한 지원, SOAP 기반 웹 서비스를 사용하기 위한 아키텍처가 포함됩니다. 많은 인터넷 애플리케이션의 메시징에 대한 낮은 수준의 지원에 대하여 이 단원에서는 소켓 컴포넌트로 작업하는 방법을 설명합니다. 이러한 많은 기능을 구현해 주는 컴포넌트를 모든 Delphi 버전에서 사용할 수 있는 것은 아닙니다.
- **4부 "COM 기반 애플리케이션 개발"**에서는 Windows Shell 확장자나 멀티미디어 애플리케이션과 같이 시스템에서 다른 COM 기반 API 객체들과 함께 이용할 수 있는 애플리케이션을 만드는 방법을 설명합니다. Delphi에는 ActiveX와 COM+를 지원하며 일반적인 목적으로 웹 기반 애플리케이션에서 사용할 수 있는 COM 컨트롤용 COM 기반 라이브러리를 지원하는 컴포넌트들이 있습니다. COM 컨트롤에 대한 지원을 모든 Delphi 에디션에서 사용할 수 있는 것은 아닙니다. ActiveX 컨트롤을 작성하기 위해서는 전문가용 또는 기업용 에디션이 필요합니다.
- **5부 "사용자 지정 컴포넌트 생성"**에서는 사용자의 컴포넌트를 디자인하고 구현하는 방법과 IDE의 컴포넌트 팔레트에서 사용할 수 있는 방법에 대해 설명합니다. 컴포넌트는 사용자가 디자인 타임에 처리하려는 거의 모든 프로그램 요소가 될 수 있습니다. 사용자 지정 컴포넌트의 구현은 VCL이나 CLX 클래스 라이브러리의 기존 클래스 타임에서 새 클래스를 파생함을 의미합니다.

설명서 규칙

이 설명서는 특별한 텍스트를 표시하기 위하여 표 1.1에서 설명한 글꼴과 기호를 사용합니다.

표 1.1 글꼴과 기호

글꼴 또는 기호	의미
Monospace type	고정 폭 텍스트는 텍스트를 화면이나 오브젝트 파스칼 코드에 나타나는 대로 표시합니다. 사용자가 입력해야 하는 내용 역시 이 글꼴로 나타냅니다.
[]	텍스트나 구문 목록에서 대괄호는 옵션 항목을 묶습니다. 이러한 종류의 텍스트는 그대로 입력하면 안 됩니다.
Boldface	텍스트나 코드 목록에서 굵게 쓰여진 글자는 오브젝트 파스칼 키워드 또는 컴파일러 옵션을 나타냅니다.
<i>Italics</i>	텍스트에서 이탤릭체 글자는 변수 또는 타입 이름과 같은 오브젝트 파스칼 식별자를 나타냅니다. 이탤릭체는 새로운 용어와 같은 일부 글자를 강조하기 위해 사용하기도 합니다.
<i>Keycaps</i>	이 글꼴은 키보드에 있는 특정 키를 나타냅니다. 예를 들면, 다음과 같습니다. "메뉴를 종료하려면 <i>Esc</i> 를 누릅니다."

개발자 지원 서비스

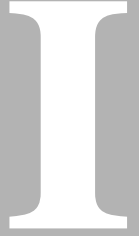
Inprise는 다양한 개발자 커뮤니티의 요구에 맞출 수 있는 다양한 지원 옵션을 제공합니다. Delphi 지원에 대해 알아보려면 <http://www.borland.com/devsupport/delphi>를 참조하십시오.

추가적인 Delphi 기술 정보 문서 및 FAQ에 대한 대답도 이 웹 사이트에서 얻을 수 있습니다.

웹 사이트를 통해 Delphi 개발자들이 정보, 팁, 기술 등을 교환하는 많은 뉴스그룹에 액세스할 수 있습니다. 또한 이 사이트에는 Delphi에 관한 도서 목록이 있습니다.

인쇄된 설명서 주문

추가 설명서 주문에 대한 내용은 shop.borland.com 웹 사이트를 참조하십시오.



Delphi 프로그래밍

"Delphi 프로그래밍"에 포함된 장에서는 제품의 모든 에디션에서 Delphi 애플리케이션을 만드는 데 필요한 개념과 기술에 대해 소개합니다. 또한 *개발자 안내서*의 이후 단원에서 설명될 개념에 대해서도 소개합니다.

2

Delphi를 사용한 애플리케이션 개발

Borland Delphi는 Windows와 Linux용 32 비트 애플리케이션의 신속한 개발을 위한 객체 지향, 비주얼 프로그래밍 환경입니다. Delphi를 사용하면 코딩을 최소화하면서 효율이 높은 애플리케이션을 만들 수 있습니다.

Delphi는 애플리케이션, 폼 템플릿, 프로그래밍 마법사를 비롯한 *비주얼 컴포넌트 라이브러리*(VCL)라는 포괄적인 클래스 라이브러리와 Borland 크로스 플랫폼용 컴포넌트 라이브러리(C LX) 및 신속한 애플리케이션 개발(RAD) 디자인 툴을 제공합니다. Delphi는 진정한 객체 지향 프로그래밍을 지원합니다.

- VCL 클래스 라이브러리에는 다른 유용한 프로그래밍 기술뿐만 아니라 Windows API를 캡슐화하는 객체들이 있습니다(Windows).
- CLX 클래스 라이브러리에는 Qt 라이브러리를 캡슐화하는 객체들이 있습니다(Windows와 Linux).

이 장에서는 Delphi 개발 환경과 이 환경을 개발 라이프 사이클에 맞추는 방법에 대해 간략하게 설명합니다. 이 설명서의 나머지 부분에는 범용 데이터베이스와 인터넷 및 인트라넷 애플리케이션의 개발에 대한 기술적인 세부 사항과 ActiveX와 COM 컨트롤 작성과 고유한 컴포넌트를 작성하는 데 필요한 정보가 들어 있습니다.

통합 개발 환경

Delphi를 시작하면 IDE라는 통합 개발 환경 내에 바로 놓이게 됩니다. 이 환경은 애플리케이션을 디자인, 개발, 테스트, 디버그, 배포하는 데 필요한 모든 툴을 제공합니다.

Delphi의 개발 환경에는 다른 도구들 중 비주얼 폼 디자이너, Object Inspector, Object TreeView, 컴포넌트 팔레트, Project Manager, 소스 코드 에디터 및 디버거 등이 있습니다. 제품의 모든 버전에서 이 도구들을 사용할 수 있는 것은 아닙니다. 폼 디자이너에 있는 객체의 비주얼 표현에서부터 객체의 초기 런타임 상태를 편집하는 Object Inspector 또는 객체의 실행 로직을 편집하는 소스 코드 에디터까지 자유로이 이동할 수 있습니다. Object Inspector에서 이벤트 핸들러의 이름 같은 코드 관련 속성을 변경

하면 해당 소스 코드가 자동으로 바뀝니다. 뿐만 아니라 폼 클래스 선언에서 이벤트 핸들러 메소드의 이름을 재지정하는 것과 같은 소스 코드의 변경 사항은 즉시 Object Inspector에 반영됩니다.

IDE는 디자인에서 배포까지 제품 라이프 사이클의 단계를 통한 애플리케이션 개발을 지원합니다. IDE에 있는 툴을 사용하면 프로토타입을 신속하게 만들 수 있고 개발 기간을 단축할 수 있습니다.

제품과 함께 들어 있는 *입문서*에는 개발 환경에 대한 보다 완벽한 개요가 설명되어 있습니다. 또한 온라인 도움말 시스템에서는 모든 메뉴, 대화 상자, 창에 대한 도움말을 제공합니다.

애플리케이션 디자인

Delphi에는 애플리케이션 디자인을 시작하는 데 필요한 모든 툴이 들어 있습니다

- 애플리케이션용 UI를 디자인하는 *폼*이라고 하는 빈 창
- 재사용 가능한 많은 객체가 들어 있는 광범위한 클래스 라이브러리
- 객체 특성을 조사하고 변경하기 위한 Object Inspector
- 원본으로 사용하는 프로그램 로직에 직접 액세스할 수 있는 코드 에디터
- 하나 이상의 프로젝트를 구성하는 파일을 관리하기 위한 Project Manager
- IDE에서 애플리케이션 개발을 지원하는 툴바의 이미지 에디터와 메뉴의 통합 디버거와 같은 그 밖의 많은 툴
- 컴파일러, 링커 및 기타 다른 유틸리티가 들어 있는 명령줄(command-line) 툴

Delphi를 사용하여 범용 유틸리티에서부터 복잡한 데이터 액세스 프로그램 또는 분산 애플리케이션에 이르기까지 모든 종류의 32비트 애플리케이션을 디자인할 수 있습니다. Delphi의 데이터베이스 툴과 data-aware 컴포넌트를 사용하면 강력한 데스크탑 데이터베이스 및 클라이언트/서버 애플리케이션을 신속하게 개발할 수 있습니다. Delphi의 data-aware 컨트롤을 사용하면 애플리케이션을 디자인하면서 생생한 데이터를 볼 수 있고 데이터베이스 쿼리의 결과와 애플리케이션 인터페이스에 대한 변경 사항을 즉시 볼 수 있습니다.

5장 "애플리케이션, 컴포넌트, 라이브러리 구축"에서는 Delphi가 다른 종류의 애플리케이션을 지원하는 것에 대해 설명합니다.

클래스 라이브러리에 제공된 많은 객체는 IDE의 컴포넌트 팔레트에서 액세스할 수 있습니다. 컴포넌트 팔레트는 폼에 놓을 수 있는 모든 비주얼 컨트롤과 논비주얼(nonvisual) 컨트롤을 보여 줍니다. 각 탭에는 기능별로 그룹화된 컴포넌트가 들어 있습니다. 규칙에 따라 클래스 라이브러리에 있는 객체의 이름은 *TStatusBar*와 같이 T로 시작합니다.

Delphi에서 혁신적인 것 중의 하나는 오브젝트 파스칼을 사용하여 직접 컴포넌트를 만들 수 있다는 것입니다. 제공되는 대부분의 컴포넌트는 오브젝트 파스칼로 작성된 것입니다. 작성한 컴포넌트를 컴포넌트 팔레트에 추가할 수 있고 필요한 경우 새 탭을 포함시켜 팔레트를 사용자 지정하여 사용할 수 있습니다.

CLX를 사용하면 Delphi로 Linux와 Windows에서 크로스 플랫폼을 개발할 수도 있습니다. CLX에 들어 있는 클래스 집합은 VCL의 클래스 대신 사용할 경우, 사용자의 프로그램이 Windows와 Linux 사이에서 포팅할 수 있게 합니다.

애플리케이션 개발

애플리케이션의 사용자 인터페이스를 시각적으로 디자인할 때 Delphi는 애플리케이션을 지원하는 원본으로 사용하는 오브젝트 파스칼 코드를 생성합니다. 컴포넌트와 폼의 속성을 선택하고 수정하면 이러한 변경 사항의 결과가 소스 코드에 자동으로 나타나며 그 반대의 경우도 마찬가지입니다. 기본 제공된 코드 에디터를 비롯한 모든 텍스트 에디터를 사용하여 소스 파일을 직접 수정할 수 있습니다. 변경 사항은 비주얼 환경에도 바로 반영됩니다.

프로젝트 생성

Delphi의 애플리케이션 개발은 모두 프로젝트 중심으로 이루어집니다. Delphi에서 애플리케이션을 만들면 프로젝트를 만드는 것입니다. 프로젝트는 애플리케이션을 구성하는 파일의 모음입니다. 이러한 파일들 중 일부는 디자인 타임 시 생성됩니다. 그 외 다른 파일들은 프로젝트 소스 코드를 컴파일할 때 자동으로 생성됩니다.

Project Manager라는 프로젝트 관리 툴에서 프로젝트 내용을 볼 수 있습니다. Project Manager는 유닛 이름, 유닛에 들어 있는 폼(하나 있는 경우)을 계층적인 보기로 나열하고, 프로젝트에서 파일로 가는 경로를 보여 줍니다. 이러한 많은 파일을 직접 편집할 수도 있지만 Delphi에서 비주얼 툴을 사용하는 것이 때때로 더 쉽고 신뢰성이 있습니다.

그룹 파일은 프로젝트 계층의 맨 위에 있습니다. 여러 프로젝트를 프로젝트 그룹으로 결합할 수 있습니다. 이것을 통해 Project Manager에서 한 번에 둘 이상의 프로젝트를 열 수 있습니다. 프로젝트 그룹을 사용하여 함께 기능하는 애플리케이션 또는 다계층 애플리케이션의 일부와 같은 관련 프로젝트에 대해 구성하고 작업할 수 있습니다. 한 가지 프로젝트에서만 작업 중인 경우에는 애플리케이션을 만드는 데 프로젝트 그룹 파일이 필요하지 않습니다.

개별 프로젝트, 파일 및 연관된 옵션에 대해 설명하는 프로젝트 파일은 .dpr이라는 확장자를 가집니다. 프로젝트 파일에는 애플리케이션이나 공유 객체 빌드에 대한 설명이 들어 있습니다. Project Manager를 사용하여 파일을 추가하고 제거하면 프로젝트 파일이 업데이트됩니다. 폼, 애플리케이션, 컴파일러 등과 같은 프로젝트의 여러 측면에 대한 탭을 가진 Project Options 대화 상자를 사용하여 프로젝트 옵션을 지정합니다. 이러한 프로젝트 옵션은 프로젝트와 함께 프로젝트 파일에 저장됩니다.

유닛과 폼은 Delphi 애플리케이션의 기본 작성 블록입니다. 프로젝트는 프로젝트 디렉토리 트리 밖에 있는 것들을 포함한 모든 기존 폼과 유닛 파일을 공유할 수 있습니다. 여기에는 독립형 루틴으로 작성된 사용자 지정 프로시저와 함수가 들어 있습니다.

프로젝트에 공유 파일을 추가하는 경우 파일이 현재 프로젝트 디렉토리에 복사되는 것이 아니고 현재 위치에 남아 있다는 것을 알 수 있습니다. 현재 프로젝트에 공유 파일을 추가하면 파일 이름과 경로가 프로젝트 파일의 **uses** 절에 기록됩니다. 프로젝트에 유닛을 추가하면 Delphi는 이름과 경로의 기록을 자동으로 처리합니다.

프로젝트를 컴파일할 때 프로젝트를 구성하는 파일들이 있는 위치는 중요하지 않습니다. 컴파일러는 공유 파일을 프로젝트 자체에 의해 생성된 파일과 동일하게 취급합니다.

코드 편집

Delphi 코드 에디터는 완전한 기능을 갖춘 아스키 에디터입니다. 비주얼 프로그래밍 환경을 사용하는 경우 폼은 새 프로젝트의 일부로 자동적으로 표시됩니다. 객체를 폼에 놓고 Object Inspector에서 객체가 작동하는 방식을 수정함으로써 애플리케이션 인터페이스의 디자인을 시작할 수 있습니다. 그러나 객체에 대한 이벤트 핸들러 작성과 같은 다른 프로그래밍 작업은 코드를 입력해야 합니다.

폼의 내용, 폼의 모든 속성, 컴포넌트 및 컴포넌트의 속성은 코드 에디터에서 텍스트로 볼 수 있고 편집할 수 있습니다. 코드 에디터에서 생성된 코드를 조정할 수 있고 코드를 입력하여 에디터 내에서 더 많은 컴포넌트를 추가할 수 있습니다. 에디터에 코드를 입력할 때 컴파일러는 계속 변경된 사항을 스캔하고 폼을 새 레이아웃으로 업데이트합니다. 그런 다음 폼으로 돌아가서 에디터에서 변경 사항을 보고 테스트하며 계속 폼을 조정할 수 있습니다.

Delphi 코드 생성 및 속성 스트리밍 시스템은 검사에 대해 완전히 열려 있습니다. 최종 실행 파일에 들어 있는 모든 것에 대한 소스 코드(모든 VCL 객체, CLX 객체, RTL (Run-Time Library) 소스, 모든 Delphi 프로젝트 파일)를 코드 에디터에서 보고 편집할 수 있습니다.

애플리케이션 컴파일

폼에서 애플리케이션 인터페이스의 디자인, 추가 코드 작성을 원하는 대로 마치면 IDE 또는 명령줄에서 프로젝트를 컴파일할 수 있습니다.

모든 프로젝트는 단일 분산 가능 실행 파일을 대상으로 가집니다. 다양한 개발 단계에서 다음과 같이 애플리케이션을 컴파일, 빌드, 실행함으로써 애플리케이션을 보거나 테스트할 수 있습니다.

- 컴파일 시 마지막 컴파일 이후에 변경된 유닛만 다시 컴파일됩니다.
- 빌드 시 마지막 컴파일 이후의 변경 여부에 관계 없이 프로젝트의 모든 유닛이 컴파일됩니다. 이 기술은 어떤 파일이 변경되었는지 정확하게 모르거나 모든 파일이 현재 동기화되었는지 확인하려 할 때 유용합니다. 또한 전역 컴파일러 지시어를 변경했을 때 Build를 사용하여 모든 코드가 적절한 상태로 컴파일되었는지 확인하는 것도 중요합니다. 프로젝트를 컴파일하지 않고 소스 코드의 유효성을 테스트할 수도 있습니다.
- 실행 시 컴파일한 후에 애플리케이션을 실행합니다. 마지막 컴파일 이후에 소스 코드를 수정한 경우에 컴파일러는 변경된 모듈을 다시 컴파일하고 애플리케이션을 다시 연결합니다.

여러 프로젝트를 함께 그룹화한 경우 단일 프로젝트 그룹에서 한 번에 모든 프로젝트를 컴파일하거나 빌드할 수 있습니다. Project Manager에서 선택한 프로젝트 그룹과 함께 Project|Compile All Projects 또는 Project|Build All Projects를 선택합니다.

애플리케이션 디버깅

Delphi는 애플리케이션의 오류를 찾아서 고치는 데 도움을 주는 통합 디버거를 제공합니다. 통합 디버거를 통해 프로그램 실행을 제어하고, 데이터 구조의 변수 값과 항목을 모니터할 수 있고, 디버깅 중에 데이터 값을 수정할 수 있습니다.

통합 디버거는 런타임 오류와 논리 오류 모두 추적할 수 있습니다. 특정 프로그램 위치에서 실행하고 변수 값, 호출 스택의 함수 값, 프로그램 출력 값을 봄으로써 프로그램의 동작 방식을 모니터할 수 있고 디자인한 대로 동작하지 않는 영역을 찾을 수 있습니다. 디버거에 대해서는 온라인 도움말에서 설명합니다.

또한 예외 처리를 사용하여 오류를 인식하고, 위치를 찾아내고, 처리할 수 있습니다. Delphi에서 예외는 다른 클래스에서 T로 시작하는 것 대신 규칙에 따라 E로 시작하는 것을 제외하면 Delphi에 있는 다른 클래스와 같은 클래스입니다.

애플리케이션 배포

Delphi에는 애플리케이션 배포를 돕는 추가적인 도구들이 들어 있습니다. 예를 들어, InstallShield Express(일부 버전에서는 사용 불가)로 분산 애플리케이션을 실행시키는 데 필요한 모든 파일들을 포함하는 설치 패키지를 만들 수 있습니다. 배포에 대한 특정 내용은 13장 "애플리케이션 배포"를 참조하십시오.

참고 Delphi의 모든 버전에 배포 기능이 있는 것은 아닙니다.

TeamSource 소프트웨어(일부 버전에서는 사용 불가)로 애플리케이션의 업데이트를 추적할 수도 있습니다.

3

컴포넌트 라이브러리 사용

이 장에서는 컴포넌트 라이브러리의 개요를 살펴 보고 애플리케이션 개발 중에 사용할 수 있는 컴포넌트 중 일부를 소개합니다. Delphi에는 비주얼 컴포넌트 라이브러리 (VCL)와 크로스 플랫폼용 컴포넌트 라이브러리 (CLX)가 모두 포함되어 있습니다. VCL은 Windows 개발용이고 CLX는 Windows와 Linux의 크로스 플랫폼 개발용입니다. 이 두 라이브러리는 서로 다른 클래스 라이브러이지만 유사한 점이 많습니다. CLX에 없는 객체, 속성, 메소드 및 이벤트는 "VCL 전용"으로 표시합니다.

컴포넌트 라이브러리 이해

VCL 및 CLX는 객체로 구성된 클래스 라이브러리로서 이 객체 중 일부는 애플리케이션 개발 시 사용하는 컴포넌트 또는 컨트롤입니다. 두 라이브러리는 유사한 점이 많고 동일한 객체를 많이 가지고 있습니다. VCL의 일부 객체(예: 컴포넌트 팔레트의 ADO, BDE, QReport, COM+, Web Services 및 Servers 탭에 있는 객체)는 Windows에서만 사용할 수 있는 기능을 구현합니다. 반면 모든 CLX 객체는 Windows와 Linux에서 사용할 수 있습니다.

VCL 및 CLX 객체는 모든 필요한 데이터와 데이터를 수정하는 "메소드"(코드)를 포함하는 활성 엔티티입니다. 데이터는 객체의 필드와 속성에 저장되고 코드는 필드와 속성 값에 작용하는 메소드들로 구성됩니다. 각 객체는 "클래스"로 선언됩니다. 모든 VCL 객체와 CLX 객체는 오브젝트 파스칼에서 개발하는 객체를 비롯해서 *TObject*라는 조상 객체의 자손입니다.

객체의 서브셋은 컴포넌트입니다. 컴포넌트는 폼 또는 데이터 모듈에 두고 디자인 타임에 처리할 수 있는 객체입니다. 컴포넌트는 컴포넌트 팔레트에 나타납니다. 코드를 작성하지 않고 컴포넌트의 속성을 지정할 수 있습니다. 모든 VCL 또는 CLX 컴포넌트는 *TComponent* 객체의 자손입니다.

컴포넌트는 다음과 같은 이유 때문에 진정한 객체 지향 프로그래밍(OOP)으로 구현된 객체라고 볼 수 있습니다

- 데이터와 데이터 액세스 함수의 집합을 캡슐화합니다.
- 조상 객체로부터 데이터와 행동을 상속받습니다.
- *다형성*이라는 개념을 통해 같은 조상에서 파생된 다른 객체와 상호 교환할 수 있습니다.

대부분의 컴포넌트와 달리 객체는 컴포넌트 팔레트에 나타나지 않습니다. 그 대신 기본 인스턴스 변수는 객체의 유닛에서 선언되거나 사용자가 자체적으로 선언해야 합니다.

컨트롤은 런타임에 사용자가 볼 수 있는 특별한 종류의 컴포넌트입니다. 컨트롤은 컴포넌트의 부분 집합입니다. 또한 컨트롤은 애플리케이션이 실행 중일 때 볼 수 있는 비주얼 컴포넌트입니다. 모든 컨트롤은 *Height*나 *Width*와 같은 비주얼한 속성을 지정하는 공통의 속성을 가집니다. 모든 컨트롤이 공통적으로 가지는 속성, 메소드 및 이벤트는 *TControl*에서 상속됩니다.

크로스 플랫폼 프로그래밍 및 Windows와 Linux 환경 간의 차이점에 대한 자세한 내용은 10장 "크로스 플랫폼 개발을 위한 CLX 사용"을 참조하십시오. 프로그래밍 중에 온라인 도움말을 사용하여 VCL 및 CLX의 모든 객체에 대한 자세한 참조 자료에 액세스할 수 있습니다. 코드 에디터 내에서 객체의 아무 곳이나 커서를 놓은 다음 F1 키를 누르면 VCL 또는 CLX 컴포넌트에 대한 도움말이 나타납니다.

Kylix를 사용하여 크로스 플랫폼 애플리케이션을 개발하는 경우, Linux 환경에 맞게 제작된 *개발자 안내서*가 Kylix에 포함되어 있습니다. Kylix 온라인 도움말에서 이 매뉴얼을 참조하거나 Kylix 제품과 함께 제공된 이 매뉴얼의 인쇄본을 참조할 수 있습니다.

속성, 메소드 및 이벤트

VCL 및 CLX는 모두 애플리케이션을 신속하게 개발할 수 있는 Delphi IDE에 연결된 객체 계층을 구성합니다. 두 컴포넌트 라이브러리의 객체는 속성, 메소드 및 이벤트에 기반합니다. 각각의 객체는 데이터 멤버(속성), 데이터에 작동하는 함수(메소드), 클래스의 사용자와 상호 작용하는 방법(이벤트)을 포함합니다. VCL이 오브젝트 파스칼에서 작성되는 반면, CLX는 C++ 클래스 라이브러리인 Qt에 기반합니다.

속성

속성은 객체의 비주얼한 행동 또는 작동에 영향을 주는 객체의 특성입니다. 예를 들어, *Visible* 속성은 객체가 애플리케이션 인터페이스에서 보이는지 여부를 결정합니다. 속성을 잘 디자인하면 컴포넌트를 유지 관리하기가 쉽고 다른 사용자가 쉽게 사용할 수 있습니다.

속성의 장점은 다음과 같습니다.

- 런타임에만 사용 가능한 메소드와는 달리 디자인 타임에 속성을 보고 변경할 수 있으며 IDE에서 컴포넌트가 변경될 때 즉시 피드백을 얻을 수 있습니다.
- Object Inspector에서 속성에 액세스하여 객체 값을 비주얼하게 수정할 수 있습니다. 디자인 타임에 속성을 설정하여 코드를 직접 작성하는 노력을 줄이고 코드 유지 관리를 쉽게 합니다.

- 데이터가 캡슐화되어 있으므로 데이터가 보호되며 실제 객체에 `private`으로 사용됩니다.
- 값을 얻고 설정하는 실제 호출은 메소드이므로 특별한 처리를 수행하여 객체의 사용자에게 보이지 않도록 할 수 있습니다. 예를 들어, 데이터는 테이블에 상주할 수 있지만 프로그래머에게 일반적인 데이터 멤버로 나타날 수 있습니다.
- 속성에 액세스하는 동안 이벤트를 트리거하거나 다른 데이터를 수정하는 로직을 구현할 수 있습니다. 예를 들어, 특정 속성 값을 변경하려면 다른 속성을 수정해야 합니다. 속성에 대해 만든 메소드를 변경할 수 있습니다.
- 속성은 가상(virtual)일 수 있습니다.
- 속성은 단일 객체에 국한되지 않습니다. 한 객체에 대한 속성을 변경하는 것은 여러 객체들에 영향을 미칠 수 있습니다. 예를 들어, 하나의 라디오 버튼에 `Checked` 속성을 설정하는 것은 그룹의 모든 라디오 버튼에 영향을 줍니다.

메소드

메소드는 클래스에 연결된 프로시저입니다. 메소드는 객체의 행동을 정의합니다. 클래스 메소드는 모든 `public`, `protected`, `private` 속성 및 클래스의 데이터 멤버에 액세스할 수 있고 일반적으로 클래스 메소드를 멤버 함수라고 합니다.

이벤트

이벤트는 프로그램에 의해 탐지된 동작이나 발생한 사건입니다. 대부분의 최근 애플리케이션은 이벤트에 응답하도록 디자인되기 때문에 이벤트 방식이라고 합니다. 프로그램에서 프로그래머는 사용자가 다음에 수행할 일련의 동작을 예측할 수 없습니다. 사용자는 메뉴 항목을 선택하거나 버튼을 클릭하거나 또는 텍스트 일부를 표시할 수도 있습니다. 항상 똑같은 순서로 실행되는 코드를 작성하기보다는 관련되어 있는 이벤트를 처리하는 코드를 작성할 수 있습니다.

이벤트 호출 방법에 상관 없이 Delphi에서는 이벤트를 처리하는 코드를 작성했는지 확인합니다. 사용자가 이벤트를 처리하는 코드를 작성한 경우에는 이 코드를 실행하고, 그렇지 않은 경우에는 기본 이벤트 처리 행동이 실행됩니다.

발생 가능한 이벤트의 종류는 크게 두 개의 범주로 나눌 수 있습니다.

- 사용자 이벤트
- 시스템 이벤트

이벤트 호출 방법에 상관 없이 Delphi는 해당 이벤트를 처리하기 위한 코드를 할당했는지 확인합니다. 할당된 코드가 있으면 이를 실행하고, 그렇지 않으면 아무 일도 일어나지 않습니다.

사용자 이벤트

사용자 이벤트는 사용자에 의해 시작되는 동작입니다. 사용자 이벤트의 예로 `OnClick` (사용자가 마우스 버튼을 클릭했을 때), `OnKeyPress` (사용자가 키보드의 키를 눌렀을 때), `OnDblClick` (사용자가 마우스 버튼을 더블 클릭했을 때) 등을 들 수 있습니다. 이러한 이벤트는 항상 사용자의 동작과 연결됩니다.

시스템 이벤트

시스템 이벤트는 운영 체제가 사용자를 위해 실행하는 이벤트입니다. 시스템 이벤트의 예로 *OnTimer* 이벤트(이미 정의된 시간 간격이 경과할 때마다 Timer 컴포넌트는 이러한 이벤트 중 하나를 실행함), *OnCreate* 이벤트(컴포넌트를 만드는 중임), *OnPaint* 이벤트(컴포넌트나 창을 다시 그려야 함) 등을 들 수 있습니다. 일반적으로 시스템 이벤트는 사용자에 의해 직접 시작되지 않습니다.

오브젝트 파스칼 및 클래스 라이브러리

표준 파스칼에 대한 일련의 객체 지향 확장 형태인 오브젝트 파스칼은 Delphi의 랭귀지입니다. Delphi의 컴포넌트 팔레트 및 Object Inspector를 사용하면 VCL 또는 CLX 컴포넌트를 폼에 놓을 수 있고 코드를 작성할 필요 없이 이러한 컴포넌트의 속성을 처리할 수 있습니다.

모든 객체는 생성, 소멸 및 메시지 처리와 같은 기본 행동을 캡슐화하는 메소드를 갖는 추상 클래스인 *TObject*의 자손입니다. *TObject*는 여러 일반 클래스의 직계 조상입니다.

VCL 또는 CLX의 *컴포넌트*는 추상 클래스인 *TComponent*의 자손입니다. 컴포넌트는 디자인 타임 시 폼에서 처리할 수 있는 객체입니다. 런타임에 화면에 나타나는 *TForm*과 *TSpeedButton*과 같은 비주얼 컴포넌트를 *컨트롤*이라고 하며 *TControl*의 자손입니다.

컴포넌트 라이브러리에는 비주얼 컴포넌트 외에 많은 논비주얼(nonvisual) 객체가 포함되어 있습니다. IDE를 통해 폼에 가져다 놓으면 프로그램에 많은 논비주얼 컴포넌트를 추가할 수 있습니다. 예를 들어, 데이터베이스에 연결하는 애플리케이션을 작성하는 경우에 폼에 *TDataSource* 컴포넌트를 놓을 수 있습니다. *TDataSource*는 논비주얼이지만 런타임에 나타나지 않는 아이콘에 의해 폼에 나타납니다. Object Inspector에서 *TDataSource*의 속성과 이벤트를 비주얼 컨트롤의 경우와 마찬가지로 처리할 수 있습니다.

오브젝트 파스칼에서 고유한 클래스를 작성하는 경우, 이 클래스는 사용하고자 하는 클래스 라이브러리에 있는 *TObject*의 자손이어야 합니다. Windows 애플리케이션을 작성하는 중이면 VCL을 사용하고 크로스 플랫폼 애플리케이션을 작성하는 중이면 CLX를 사용합니다. 새 클래스가 필수 기능을 포함하고 클래스 라이브러리의 다른 클래스를 사용할 수 있도록 해당 기본 클래스(또는 그 자손 중 하나)에서 새 클래스를 파생시킵니다.

객체 모델 사용

객체 지향 프로그래밍은 코드 재사용과 데이터의 기능별 캡슐화를 강조하는 구조적 프로그래밍의 확장입니다. 일단 객체(또는 공식적으로는 클래스)가 생성되면 다른 프로그래머가 생성된 객체를 다른 애플리케이션에 사용할 수 있으므로 개발 시간을 단축하고 생산성을 향상시킬 수 있습니다.

새 컴포넌트를 만드는 방법과 컴포넌트 팔레트에 놓는 방법에 대한 자세한 내용은 40장 "컴포넌트 생성 개요"를 참조하십시오.

객체의 개념

객체나 클래스는 단일 유닛에서 *데이터*와 *데이터에 대한 작업*을 캡슐화하는 데이터 타입입니다. 객체 지향 프로그래밍 개념이 일반화되기 전에는 데이터와 연산(함수)은 별도의 요소로 취급되었습니다.

오브젝트 파스칼의 *레코드* 또는 C의 *구조체*를 이해한다면 객체를 이해하기 시작한 것입니다. 레코드는 데이터를 포함하는 필드로 구성되며 이러한 필드는 각각 고유한 타입을 가집니다. 레코드를 사용하여 다양한 데이터 요소의 모음을 쉽게 참조할 수 있습니다.

또한 객체는 데이터 요소의 모음입니다. 그러나 레코드와 달리 객체에는 데이터에 작동하는 프로시저와 함수가 포함됩니다. 이러한 프로시저와 함수를 *메소드*라고 합니다.

객체의 데이터 요소는 속성을 통해 액세스됩니다. VCL 및 CLX 객체의 속성은 코드를 작성하지 않고 디자인 타임에 변경할 수 있는 값을 가집니다. 속성 값을 런타임에 변경하려면 약간의 코드만 작성하면 됩니다.

단일 유닛의 데이터와 기능의 조합을 *캡슐화(Encapsulation)*라고 합니다. 캡슐화 이외에도 객체 지향 프로그래밍은 *상속(Inheritance)*과 *다형성(Polymorphism)*이라는 특징이 있습니다. 상속은 객체가 *조상*이라고 부르는 객체로부터 기능을 파생시키는 것을 의미합니다. 파생된 객체는 상속된 행동을 수정할 수 있습니다. 다형성은 동일한 조상에서 파생된 다양한 객체들이 동일한 메소드와 속성 인터페이스를 지원한다는 것을 의미하므로 종종 상호 교환 가능하다고 할 수 있습니다.

Delphi 객체 살펴 보기

새 프로젝트가 만들어지면 Delphi에서는 사용자 지정할 수 있는 새 폼이 나타납니다. 코드 에디터에서 Delphi는 폼의 새 클래스 타입을 선언하고 새로운 폼 인스턴스를 생성하는 코드를 만듭니다. 새 Windows 애플리케이션에 대해 생성된 코드는 다음과 같습니다.

```

unit Unit1;
interface

uses Windows, Classes, Graphics, Forms, Controls, Dialogs;

type
  TForm1 = class(TForm) { The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end; { The type declaration of the form ends here }

var
  Form1:TForm1;

implementation { Beginning of implementation part }
  {$R *.DFM}
end.{ End of implementation part and unit }

```

새 클래스 타입은 *TForm1*이고 마찬가지로 클래스인 *TForm* 타입에서 파생됩니다.

클래스는 데이터 필드를 가지고 있다는 점에서 레코드와 유사하지만 클래스는 객체의 데이터에 작동하는 코드인 메소드도 포함합니다. 지금까지 폼에 컴포넌트(새 객체의 필드)를 추가하지 않고 이벤트 핸들러(새 객체의 메소드)를 만들지 않았기 때문에 *TForm1*은 필드나 메소드가 없는 것처럼 보입니다. 타입 선언에서 보이지 않지만 *TForm1*은 상속된 필드와 메소드를 포함합니다.

다음 변수 선언은 새 타입 *TForm1*의 *Form1*이라는 변수를 선언합니다.

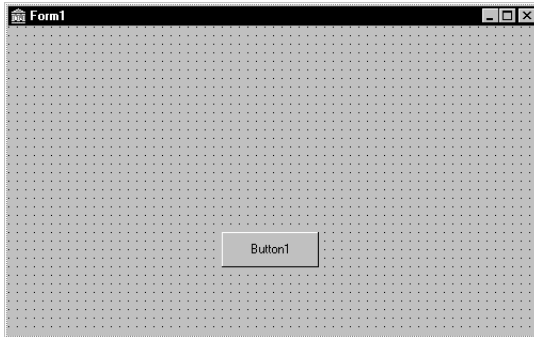
```
var
  Form1:TForm1;
```

*Form1*은 클래스 타입 *TForm1*의 인스턴스 또는 객체를 나타냅니다. 클래스 타입의 인스턴스를 두 개 이상 선언할 수 있습니다. 예를 들어, 다중 문서 인터페이스(MDI) 애플리케이션에서 여러 개의 자식 창을 만들려고 할 때 두 개 이상의 인스턴스를 선언할 수 있습니다. 각 인스턴스는 자체 데이터를 유지하지만 모든 인스턴스는 동일한 코드를 사용하여 메소드를 실행합니다.

폼에 컴포넌트를 추가하거나 코드를 작성하지 않아도 컴파일하고 실행할 수 있는 완전한 Delphi 애플리케이션이 이미 있습니다. 이 애플리케이션을 실행하면 빈 폼이 표시됩니다.

이 폼에 버튼 컴포넌트를 추가하고 사용자가 버튼을 클릭하면 폼의 색상을 변경하는 *OnClick* 이벤트 핸들러를 작성한다고 가정합니다. 그 결과는 다음과 같습니다.

그림 3.1 단순한 폼



사용자가 버튼을 클릭할 때 폼의 색상이 녹색으로 바뀝니다. 다음은 버튼의 *OnClick* 이벤트에 대한 이벤트 핸들러 코드입니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  Form1.Color := clGreen;
end;
```

객체는 데이터 필드로서의 다른 객체를 포함할 수 있습니다. 폼에 컴포넌트를 둘 때마다 새 필드가 폼의 타입 선언에 나타납니다. 위에 설명된 애플리케이션을 만들면 코드 에디터에서 다음 코드를 볼 수 있습니다.

```
unit Unit1;

interface
```

```

uses Windows, Classes, Graphics, Forms, Controls;

type
  TForm1 = class(TForm)
    Button1:TButton;{ New data field }
    procedure Button1Click(Sender:TObject);{ New method declaration }
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1:TForm1;

implementation
  {$R *.DFM}

  procedure TForm1.Button1Click(Sender:TObject);{ The code of the new method }
  begin
    Form1.Color := clGreen;
  end;

end.

```

*TForm1*은 폼에 추가한 버튼에 해당하는 *Button1* 필드를 가집니다. *TButton*은 클래스 타입이므로 *Button1*은 객체를 참조합니다.

Delphi에서 작성한 모든 이벤트 핸들러는 폼 객체의 메소드입니다. 이벤트 핸들러를 만들 때마다 메소드가 폼 객체 타입으로 선언됩니다. 이제 *TForm1* 타입은 *TForm1* 타입 선언에서 선언된 새 메소드인 *Button1Click* 프로시저를 포함합니다. *Button1Click* 메소드를 구현하는 코드는 유닛의 **implementation** 부분에 나타납니다.

컴포넌트 이름 변경

컴포넌트의 이름을 변경하려면 항상 Object Inspector를 사용해야 합니다. 예를 들어, 폼의 이름을 기본 *Form1*에서 *ColorBox*와 같은 보다 구체적인 이름으로 변경하고자 한다고 가정합니다. Object Inspector에서 폼의 *Name* 속성을 변경하면 폼의 .dfm 또는 .xfm 파일(보통은 수동으로 편집하지 않음)과 Delphi가 생성하는 오브젝트 파스칼 소스 코드에 새 이름이 자동으로 반영됩니다.

```

unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls;

type
  TColorBox = class(TForm){ Changed from TForm1 to TColorBox }
    Button1:TButton;
    procedure Button1Click(Sender:TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

```

```

var
  ColorBox:TColorBox;{ Changed Form1 to ColorBox }

implementation
  {$R *.DFM}

procedure TColorBox.Button1Click(Sender:TObject);
begin
  Form1.Color := clGreen;{ The reference to Form1 didn't change! }
end;

end.

```

버튼에 대한 *OnClick* 이벤트 핸들러의 코드가 변경되지 않았다는 것에 유의하십시오. 사용자가 코드를 작성했기 때문에 그 코드를 사용자가 업데이트하고 폼에 대한 참조를 수정해야 합니다.

```

procedure TColorBox.Button1Click(Sender:TObject);
begin
  ColorBox.Color := clGreen;
end;

```

객체로부터 데이터와 코드 상속

설명된 *TForm1* 객체는 단순히 보입니다. *TForm1*은 하나의 필드(*Button1*)와 하나의 메소드(*Button1Click*)를 포함하지만 속성은 포함하지 않습니다. 이제 폼을 보여 주거나 숨기거나 크기를 조정할 수 있고, 표준 테두리 아이콘을 추가하거나 삭제할 수 있으며, MDI(Multiple Document Interface) 애플리케이션의 일부가 되도록 폼을 설정할 수 있습니다. 폼이 컴포넌트 *TForm*의 모든 속성과 메소드를 상속했기 때문에 이러한 작업을 수행할 수 있습니다. 프로젝트에 새 폼을 추가할 때 *TForm*을 시작한 다음에 컴포넌트를 추가하고 속성 값을 변경하고 이벤트 핸들러를 작성하여 폼을 사용자 지정합니다. 객체를 사용자 지정하려면 먼저 기존 객체로부터 새 객체를 파생시킵니다. 프로젝트에 새 폼을 추가하는 경우 Delphi는 자동으로 *TForm* 타입으로부터 새 폼을 파생시킵니다.

```
TForm1 = class(TForm)
```

파생된 객체는 파생시킨 객체의 모든 속성, 이벤트 및 메소드를 상속 받습니다. 파생된 객체는 *자손*이라고 하고, 이 자손을 파생시킨 객체는 *조상*이라고 합니다. 온라인 도움말에서 *TForm*을 조회하면 *TForm*이 조상으로부터 상속받은 것을 포함하여 *TForm*의 속성, 이벤트 및 메소드의 목록을 볼 수 있습니다. 객체는 직계 조상을 단 하나만 가질 수 있지만 직계 자손은 많이 가질 수 있습니다.

유효 범위(scope) 및 한정자

*유효 범위*는 객체의 필드, 속성 및 메소드의 액세스 가능성을 결정합니다. 객체 내에서 선언된 모든 멤버는 객체 및 객체의 자손에서 사용할 수 있습니다. 메소드의 구현 코드가 객체 선언 밖에 나타나더라도 메소드는 객체 선언 내에서 선언되기 때문에 여전히 객체의 유효 범위 내에 있습니다.

메소드가 선언된 객체의 속성, 메소드 또는 필드를 참조하는 메소드를 구현하는 코드를 작성할 때 식별자를 객체 이름으로 시작할 필요는 없습니다. 예를 들어, 새 폼에 버튼을 놓는 경우 버튼의 *OnClick* 이벤트에 대한 이벤트 핸들러를 다음과 같이 작성할 수 있습니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  Color := clFuchsia;
  Button1.Color := clLime;
end;
```

첫 번째 문장은 다음과 같습니다.

```
Form1.Color := clFuchsia
```

Button1Click 메소드가 *TForm1*의 일부이기 때문에 *Form1*을 사용하여 *Color*를 한정할 필요가 없고, 따라서 메소드 몸체의 식별자는 메소드가 호출되는 *TForm1* 인스턴스의 유효 범위 내에 있게 됩니다. 반면 두 번째 문장은 이벤트 핸들러가 선언된 폼이 아닌 버튼 객체의 색상을 참조하므로 한정되어야 합니다.

Delphi는 각 폼에 대하여 별도의 유닛(소스 코드) 파일을 만듭니다. 다른 폼의 유닛 파일로부터 특정 폼의 컴포넌트에 액세스하려면 다음과 같이 컴포넌트 이름을 한정해야 합니다.

```
Form2.Edit1.Color := clLime;
```

동일한 방법으로 다른 폼에 있는 컴포넌트의 메소드에 액세스할 수 있습니다. 예를 들면, 다음과 같습니다.

```
Form2.Edit1.Clear;
```

*Form1*의 유닛 파일에서 *Form2*의 컴포넌트에 액세스하려면 *Form2*의 유닛을 *Form1* 유닛의 **uses** 절에 추가해야 합니다.

객체의 유효 범위가 객체의 자손으로 확장됩니다. 그러나 자손 객체 내에서 필드, 속성 및 메소드를 재선언할 수 있습니다. 그러한 재선언은 상속된 멤버를 숨기거나 오버라이드합니다.

유효 범위, 상속 및 **uses** 절에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.

private, protected, public, published 선언

필드, 속성 또는 메소드를 선언할 때 새 멤버는 키워드 **private**, **protected**, **public** 또는 **published** 중의 하나에 의해 지시되는 *가시성(visibility)*을 갖습니다. 멤버의 가시성은 다른 객체와 유닛에 대한 멤버의 액세스 가능성을 결정합니다.

- **private** 멤버는 멤버가 선언된 유닛 내에서만 멤버에 액세스할 수 있습니다. **private** 멤버를 가끔 클래스 내에서 사용하여 **public** 또는 **published**로 선언된 메소드와 속성을 구현합니다.
- **protected** 멤버는 멤버의 클래스가 선언된 유닛과 자손 클래스 내에서 자손 클래스의 유닛에 상관 없이 액세스할 수 있습니다.

- public 멤버는 멤버가 속하는 객체에 액세스할 수 있는 어디에서라도, 즉 클래스가 선언된 유닛과 그 유닛을 참조하는 모든 유닛에서 액세스할 수 있습니다.
- published 멤버는 public 멤버와 동일한 가시성을 가지지만 컴파일러는 published 멤버에 대해 런타임 타입 정보를 생성합니다. published 속성은 디자인 타임 시 Object Inspector에 나타납니다.

가시성에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.

객체 변수 사용

변수들이 동일한 타입이거나 할당 호환이 있는 경우, 특정 객체 변수를 다른 객체 변수에 할당할 수 있습니다. 특히 할당하는 변수 타입이 할당되는 변수 타입의 조상인 경우에 객체 변수를 다른 객체 변수에 할당할 수 있습니다. 예를 들어, 다음 *TDataForm* 타입 선언 (VCL 전용)의 변수 선언 섹션에서는 두 변수 *AForm*과 *DataForm*을 선언합니다.

```

type
  TDataForm = class(TForm)
    Button1:TButton;
    Edit1:TEdit;
    DataGrid1: TDataGrid;
    Databasel:TDatabase;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  AForm:TForm;
  DataForm: TDataForm;

```

*AForm*은 *TForm* 타입이고, *DataForm*은 *TDataForm* 타입입니다. *TDataForm*은 *TForm*의 자손이므로 다음과 같은 할당문이 적합합니다.

```
AForm := DataForm;
```

버튼의 *OnClick* 이벤트에 대한 이벤트 핸들러를 작성한다고 가정합니다. 버튼을 클릭하면 *OnClick* 이벤트에 대한 이벤트 핸들러가 호출됩니다. 각 이벤트 핸들러에는 다음과 같이 *TObject* 타입의 *Sender* 매개변수가 있습니다.

```

procedure TForm1.Button1Click(Sender:TObject);
begin
  :
end;

```

*Sender*는 *TObject* 타입이므로 모든 객체는 *Sender*에 할당될 수 있습니다. *Sender*의 값은 항상 이벤트에 응답하는 컨트롤이나 컴포넌트입니다. 예약어 **is**를 사용하여 이벤트 핸들러를 호출한 컴포넌트나 컨트롤의 타입을 찾기 위해 *Sender*를 테스트할 수 있습니다. 예를 들어, 다음과 같습니다.

```

if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;

```

객체 생성, 인스턴스화 및 소멸

버튼이나 편집 상자와 같이 Delphi에서 사용하는 객체 중의 다수는 디자인 타임과 런타임 모두에서 보입니다. 공통 대화 상자와 같은 일부 객체는 런타임에만 나타납니다. 그러나 타이머나 데이터소스 컴포넌트와 같은 객체들은 런타임에도 보이지 않습니다.

사용자가 객체를 직접 만들 수도 있습니다. 예를 들어 *Name*, *Title*, *HourlyPayRate* 속성이 포함된 *TEmployee* 객체를 만들 수 있습니다. 그런 다음 *HourlyPayRate*의 데이터를 사용하여 급여를 계산하는 *CalculatePay* 메소드를 추가할 수 있습니다. *TEmployee* 타입 선언은 다음과 같습니다.

```

type
  TEmployee = class(TObject)
  private
    FName:string;
    FTitle:string;
    FHourlyPayRate:Double;
  public
    property Name:string read FName write FName;
    property Title:string read FTitle write FTitle;
    property HourlyPayRate:Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay:Double;
  end;

```

사용자가 정의한 필드, 속성 및 메소드 외에 *TEmployee*는 *TObject*의 모든 메소드를 상속합니다. 유닛의 **interface** 또는 **implementation** 부분에 이와 비슷한 타입 선언을 한 다음 다음과 같이 *TEmployee*가 *TObject*에서 상속받은 *Create* 메소드를 호출하여 새 클래스의 인스턴스를 만들 수 있습니다.

```

var
  Employee:TEmployee;
begin
  Employee := TEmployee.Create;
end;

```

Create 메소드를 *생성자*라고 합니다. 이 메소드는 새 인스턴스 객체에 메모리를 할당하고 객체에 대한 참조를 반환합니다.

폼에 있는 컴포넌트는 Delphi에 의해서 자동으로 생성되고 소멸됩니다. 그러나 객체를 인스턴스화하는 코드를 직접 작성하는 경우에는 객체 소멸 역시 직접해야 합니다. 모든 객체는 *소멸자*라고 하는 *Destroy* 메소드를 *TObject*로부터 상속 받습니다. 하지만 객체를 소멸하려면 *TObject*에서 상속 받은 *Free* 메소드를 호출해야 합니다. 왜냐하면 *Free* 메소드가 *Destroy*를 호출하기 전에 **nil** 참조를 확인하기 때문입니다. 예를 들어,

```
Employee.Free
```

이 문장은 *Employee* 객체를 소멸하고 메모리 할당을 해제합니다.

컴포넌트와 소유권

Delphi는 특정 컴포넌트가 다른 컴포넌트 해제를 책임지는 기본 제공 메모리 관리 메커니즘을 갖습니다. 컴포넌트를 해제하는 컴포넌트는 해제되는 컴포넌트를 *소유(own)*한다고 말합니다. 소유자 (owner)의 메모리가 해제되면 소유자가 소유한 컴포넌트의 메모리는 자동으로 해제됩니다. 컴포넌트의 소유자 (*Owner* 속성 값)는 컴포넌트가 생성될 때 생성자에게 전달된 매개변수에 의해 결정됩니다. 기본적으로 폼에 있는 모든 컴포넌트는 폼이 소유하고 폼은 애플리케이션이 소유합니다. 따라서 애플리케이션이 종료되면 애플리케이션의 모든 폼과 컴포넌트의 메모리는 해제됩니다.

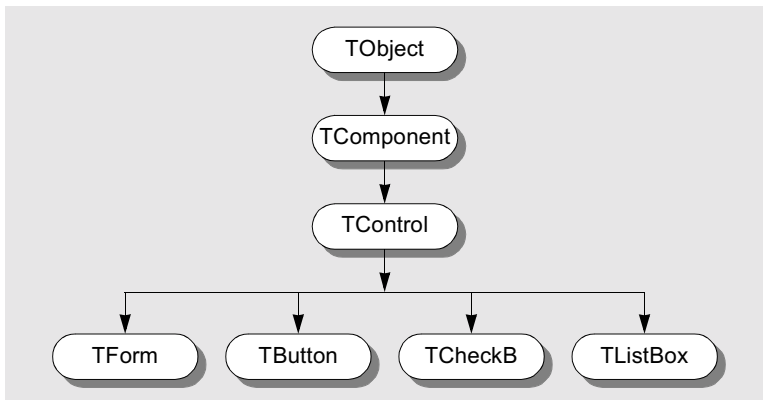
소유권은 *TComponent*와 이 컴포넌트의 자손에만 적용됩니다. 예를 들어, *TStringList* 또는 *TCollection* 객체를 사용자가 생성하는 경우 생성한 객체가 폼에 연결되어 있더라도 사용자가 객체를 해제해야 합니다.

참고 컴포넌트의 *소유자*와 그 부모를 혼동하지 마십시오. 3-19 페이지의 "부모 속성"을 참조하십시오.

객체, 컴포넌트 및 컨트롤

그림 3.2는 객체, 컴포넌트 및 컨트롤 사이의 관계를 나타내는 상속 계층 구조를 아주 간단하게 보여 줍니다.

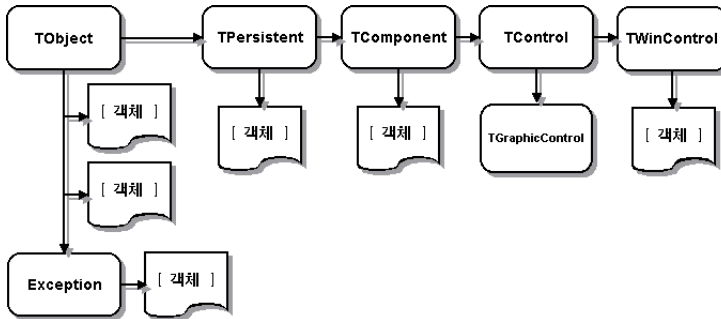
그림 3.2 객체, 컴포넌트 및 컨트롤



모든 객체는 *TObject*로부터 상속 받으며 많은 객체들이 *TComponent*로부터 상속 받습니다. *TControl*로부터 상속 받는 컨트롤에는 런타임 시 자신을 나타내는 기능이 있습니다. *TCheckBox*와 같은 컨트롤은 *TObject*, *TComponent* 및 *TControl*의 모든 기능을 상속하고 고유한 특수 기능을 추가합니다.

그림 3.3은 상속 트리의 주요 분기를 보여 주는 비주얼 컴포넌트 라이브러리 (VCL)의 개요를 나타낸 것입니다. 크로스 플랫폼용 Borland 컴포넌트 라이브러리 (CLX)는 거의 동일한 레벨을 갖지만 *TWinControl*이 *TWidgetControl*로 대체됩니다.

그림 3.3 단순화된 계층 구조



여러 개의 중요한 기본 클래스가 그림에 나타나 있습니다. 다음 표는 기본 클래스를 설명한 것입니다.

표 3.1 중요한 기본 클래스

클래스	설명
<i>TObject</i>	모든 VCL 또는 CLX 객체의 기본 클래스 및 궁극적인 조상을 나타냅니다. <i>TObject</i> 는 객체의 인스턴스 생성, 유지 관리 및 소멸과 같은 기본 기능을 수행하는 메소드를 소개하여 VCL/CLX의 모든 객체에 공통적인 기본 동작을 캡슐화합니다.
<i>Exception</i>	예외와 관련된 모든 클래스의 기본 클래스입니다. 예외는 오류 상태에 대해 일관된 인터페이스를 제공하고 애플리케이션이 오류 상태를 잘 처리하도록 합니다.
<i>TPersistent</i>	속성을 구현하는 모든 객체에 대한 기본 클래스입니다. <i>TPersistent</i> 의 자손 클래스는 데이터를 스트림에 보내는 것을 처리하고 클래스의 할당을 허용합니다.
<i>TComponent</i>	<i>TApplication</i> 과 같은 모든 논비주얼(nonvisual) 컴포넌트에 대한 기본 클래스입니다. <i>TComponent</i> 는 모든 컴포넌트의 공통 조상입니다. 이 클래스를 사용하여 컴포넌트 팔레트에 컴포넌트를 표시할 수 있고, 컴포넌트가 다른 컴포넌트를 소유할 수 있고, 컴포넌트를 폼에서 직접 처리할 수 있습니다.
<i>TControl</i>	런타임에 보이는 모든 컨트롤에 대한 기본 클래스입니다. <i>TControl</i> 은 모든 비주얼 컴포넌트의 공통 조상이고 위치나 커서와 같은 표준 비주얼 컨트롤을 제공합니다. 이 클래스는 또한 마우스 동작에 응답하는 이벤트를 제공합니다.
<i>TWinControl</i>	widget라고도 하는 모든 사용자 인터페이스 객체의 기본 클래스입니다. <i>TWinControl</i> 아래의 컨트롤은 키보드 입력을 캡처할 수 있는 창이 있는 컨트롤입니다. CLX에서는 <i>TWinControl</i> 이 <i>TWidgetControl</i> 로 대체됩니다.

다음 몇 개의 단원에서는 각 분기가 포함하는 클래스 타입에 대한 일반적인 내용을 설명합니다. VCL 객체 계층 구조의 전체적인 개요는 이 제품과 함께 들어 있는 VCL 객체 계층 구조 월 차트(wall chart)를 참조하십시오. CLX에 대한 자세한 설명은 Kylix 제품 및 설명서에 포함된 CLX 객체 계층 구조 월 차트(wall chart)를 참조하십시오.

TObject 분기

TObject 분기에는 *TObject*의 자손인 모든 객체들이 포함되지만 *TPersistent*의 자손 객체는 포함되지 않습니다. 모든 VCL 또는 CLX 객체는 그 메소드가 생성, 소멸, 메시지 또는 시스템 이벤트 처리 등의 기본 동작을 정의하는 추상 클래스인 *TObject*의 자손입니다. VCL 또는 CLX 객체의 강력한 기능 대부분은 *TObject*에 있는 메소드에 의한 것입니다. *TObject*에 다음을 제공하는 메소드가 있어서 VCL 및 CLX의 모든 객체에 공통적인 기본 동작을 캡슐화합니다.

- 객체 인스턴스가 생성되고 소멸될 때 응답하는 기능
- 클래스 타입, 객체에 대한 인스턴스 정보 및 객체의 published 속성에 대한 런타임 타입 정보(RTTI)
- 메시지 처리 지원(VCL 전용)

*TObject*는 여러 일반 클래스의 직계 조상입니다. 이 분기의 클래스들은 일시적인 클래스라는 한 가지 중요한 공통 특성이 있습니다. 이것은 클래스가 소멸되기 전에 상태를 저장하는 메소드를 가지지 않는다는 것을 의미하므로 이 클래스들은 영구적이지 않습니다.

이 분기에 있는 클래스의 주요 그룹 중의 하나는 *Exception* 클래스입니다. 이 클래스는 0으로 나누기 오류, 파일 I/O 오류, 잘못된 타입 변환 그리고 많은 기타 예외 상태를 자동으로 처리하여 대규모의 기본 제공 예외 클래스 집합을 제공합니다.

TObject 분기에 있는 그룹의 다른 타입은 데이터 구조를 캡슐화하는 다음과 같은 클래스들입니다.

- *TBits*, 부울 값 "배열"을 저장하는 클래스
- *TList*, 연결 리스트(linked list) 클래스
- *TStack*, 포인터의 후입선출(LIFO) 배열을 유지 관리하는 클래스
- *TQueue*, 포인터의 선입선출(FIFO) 배열을 유지 관리하는 클래스

또한 VCL에는 Windows 프린터 인터페이스를 캡슐화하는 *TPrinter*와 같은 외부 객체용 래퍼(wrapper)와 시스템 레지스트리 및 여기서 작동하는 함수를 위한 저수준 래퍼인 *TRegistry*가 포함되어 있습니다. 이러한 래퍼는 Windows 환경에서만 사용됩니다.

*TStream*은 이 분기에서 다른 클래스 타입에 대한 좋은 예입니다. *TStream*은 디스크 파일, 동적 메모리 등과 같은 다양한 저장 매체로부터 읽고 쓸 수 있는 스트림 객체에 대한 기본 클래스 타입입니다.

이와 같이 이 분기에는 개발자가 매우 유용하게 사용할 수 있는 다양한 클래스 타입이 들어 있습니다.

TPersistent 분기

VCL 및 CLX의 이 분기에 있는 객체는 *TComponent*의 자손이 아니라 *TPersistent*의 자손입니다. *TPersistent*는 객체에 영구성을 추가합니다. 영구성은 폼 파일이나 데이터 모듈에 저장되는 것과 메모리에서 검색할 때 폼이나 데이터 모듈로 로드되는 것을 결정합니다.

이 분기에 있는 객체들은 컴포넌트의 속성을 구현합니다. 속성은 소유자(owner)가 있는 경우에 폼과 함께 로드되고 저장됩니다. 소유자는 컴포넌트여야 합니다. 이 분기에는 속성의 소유자를 결정할 수 있는 *GetOwner* 함수가 있습니다.

또한 이 분기에 있는 객체들은 속성이 자동으로 로드되고 저장될 수 있는 *published* 섹션을 포함하는 첫 번째 객체입니다. *DefineProperties* 메소드를 사용하여 사용자는 속성을 로드하고 저장하는 방법을 지시할 수도 있습니다.

다음은 계층 구조의 *TPersistent* 분기에 있는 다른 클래스들입니다.

- *TGraphicsObject*는 *TBrush*, *TFont* 및 *TPen*과 같은 그래픽 객체의 추상 기본 클래스입니다.
- *TGraphic*은 비주얼 이미지를 저장 및 표시할 수 있는 아이콘과 비트맵과 같은 객체, 즉 *TBitmap*, *TIcon* 및 Windows 개발 전용인 *TMetafile*의 추상 기본 클래스입니다.
- *TStrings*는 문자열 목록을 나타내는 객체의 기본 클래스입니다.
- *TClipboard*는 애플리케이션에서 잘라내거나 복사한 텍스트나 그래픽을 포함하는 클래스입니다.
- *TCollection*, *TOwnedCollection* 및 *TCollectionItem*은 특별히 정의된 항목의 인덱스된 모음을 유지하는 클래스입니다.

TComponent 분기

TComponent 분기는 *TControl*이 아닌 *TComponent*의 자손인 객체를 포함합니다. 이 분기에 있는 객체는 디자인 타임에 폼에서 처리할 수 있는 컴포넌트입니다. 이 객체는 다음을 수행할 수 있는 영구적인 객체입니다.

- 컴포넌트 팔레트에 나타나고 폼 디자이너에서 변경할 수 있습니다.
- 다른 컴포넌트를 소유하고 관리합니다.
- 객체를 로드하고 저장합니다.

*TComponent*의 여러 메소드는 디자인 타임 동안 컴포넌트가 작동하는 방법과 그 컴포넌트와 함께 저장하는 정보를 지시합니다. VCL 및 CLX의 이 분기에는 스트리밍이 있습니다. Delphi는 대부분의 스트리밍 작업을 자동으로 처리합니다. 속성이 *published* 인 경우 속성은 영구적이며 자동으로 스트리밍됩니다.

TComponent 클래스에는 VCL 및 CLX를 통해 전달되는 소유권의 개념도 있습니다. *Owner*와 *Components* 속성은 소유권 개념을 지원합니다. 각 컴포넌트는 다른 컴포넌트를 자신의 소유자로 참조하는 *Owner* 속성을 가집니다. 컴포넌트는 다른 컴포넌트를 소유할 수 있습니다. 이 경우에 모든 소유된 컴포넌트는 컴포넌트의 *Array* 속성으로 참조됩니다.

컴포넌트의 생성자는 새 컴포넌트의 소유자를 지정하는 데 사용되는 매개변수를 하나 가지고 있습니다. 전달된 소유자가 있으면 새 컴포넌트는 소유자의 컴포넌트 목록에 추가됩니다. 소유된 컴포넌트를 참조하는 컴포넌트 목록을 사용하는 것과 별도로, 이 속성은 소유된 컴포넌트의 자동 소멸 기능도 제공합니다. 컴포넌트의 소유자가 있으면 소유자

가 소멸되면 소유된 컴포넌트도 함께 소멸됩니다. 예를 들어, *TForm*은 *TComponent*의 자손이므로 폼이 소유한 모든 컴포넌트는 소멸되고 컴포넌트의 메모리는 해제됩니다. 이것은 컴포넌트의 소멸자를 호출할 때 폼의 모든 컴포넌트가 제대로 해제된다는 것을 가정합니다.

속성 타입이 *TComponent* 또는 자손인 경우, 스트리밍 시스템은 읽을 때 해당 타입의 인스턴스를 생성합니다. 속성 타입이 *TComponent*가 아닌 *TPersistent*인 경우, 스트리밍 시스템은 속성을 통해 사용 가능한 기존 인스턴스를 사용하고 해당 인스턴스에 대한 속성 값을 읽습니다.

폼 파일(폼에 있는 컴포넌트에 대한 정보를 저장하는 데 사용하는 파일)을 만들 때 폼 디자이너는 폼의 컴포넌트 배열에 폼의 모든 컴포넌트를 저장합니다. 각 컴포넌트는 스트림(이 경우에는 텍스트 파일)에 대한 변경된 속성을 작성하는 방법을 "알고 있습니다". 반면 폼 디자이너는 컴포넌트의 속성을 폼 파일에 로드할 때 컴포넌트 배열에 있는 각 컴포넌트를 로드합니다.

이 분기에서 찾을 수 있는 클래스 타입은 다음과 같습니다.

- *TMainMenu*는 폼에 대한 메뉴 바와 메뉴 바의 드롭다운 메뉴를 제공하는 클래스입니다.
- *TTimer*는 타이머 기능이 들어 있는 클래스입니다.
- *TOpenDialog*, *TSaveDialog*, *TFontDialog*, *TFindDialog*, *TColorDialog* 등은 공통적으로 사용되는 대화 상자를 제공합니다.
- *TActionList*는 메뉴 항목 및 버튼과 같이 컴포넌트 및 컨트롤과 함께 사용되는 동작의 목록을 유지 관리하는 클래스입니다.
- *TScreen*은 애플리케이션에 의해 인스턴스화된 폼과 데이터 모듈, 활성 폼, 폼의 활성 컨트롤, 화면 크기 및 해상도, 애플리케이션에 사용 가능한 커서 및 글꼴 등을 유지 관리하는 클래스입니다.

비주얼 인터페이스가 필요 없는 컴포넌트는 *TComponent*에서 직접 파생될 수 있습니다. *TTimer* 장치와 같은 도구를 만들기 위해 *TComponent*에서 파생시킬 수 있습니다. 이러한 타입의 컴포넌트는 컴포넌트 팔레트에 있지만 런타임 시 사용자 인터페이스에 나타나지 않고 코드를 통해 액세스되는 내부 함수를 수행합니다.

CLX의 *TComponent* 분기에는 *THandleComponent*도 들어 있습니다. 이 클래스는 대화 상자나 메뉴와 같이 기본으로 사용한 Qt 객체에 대한 핸들을 요구하는 논비주얼(nonvisual) 컴포넌트의 기본 클래스입니다.

TControl 분기

TControl 분기는 *TWinControl*(CLX의 경우 *TWidgetControl*)의 자손이 아니라 *TControl*의 자손인 컴포넌트로 구성됩니다. 이 분기의 객체는 애플리케이션 사용자가 런타임 시 보고 처리할 수 있는 비주얼 객체인 컨트롤입니다. 모든 컨트롤은 컨트롤의 위치, 컨트롤의 창(또는 CLX의 widget)에 연결된 커서, 컨트롤을 그리거나 이동하는 메소드 그리고 마우스 동작에 응답하는 이벤트 같은 컨트롤의 모양과 관련된 공통적인 속성, 메소드 및 이벤트를 갖습니다. 컨트롤은 키보드 입력을 받을 수 없습니다.

*TComponent*가 모든 컴포넌트에 대한 행동을 정의하는 반면 *TControl*은 모든 비주얼 컨트롤에 대한 행동을 정의합니다. 여기에는 드로잉 루틴, 표준 이벤트 및 포함 관계 (containership) 등이 포함됩니다.

이 컨트롤의 두 가지 기본 타입은 다음과 같습니다.

- 고유한 창(또는 widget)을 갖는 컨트롤
- "부모"의 창(또는 widget)을 사용하는 컨트롤

고유한 창을 갖는 컨트롤은 "창 있는" 컨트롤 (VCL) 또는 "widget 기반" 컨트롤 (CLX) 이라고 하며 *TWinControl*(CLX의 경우 *TWidgetControl*)의 자손입니다. 버튼과 체크 박스는 이 클래스에 속합니다.

부모 창(또는 widget)을 사용하는 컨트롤은 "그래픽" 컨트롤이라고 하며 *TGraphicControl*의 자손입니다. 이미지와 레이블이 이 클래스에 해당합니다. VCL에서 이러한 타입의 컴포넌트들 간의 주요한 차이점은 그래픽 컨트롤에서 유지 관리하는 창 핸들이 없어서 입력 포커스를 받을 수 없다는 것입니다. CLX의 경우 이러한 타입의 컴포넌트들 간의 주요한 차이점은 그래픽 컨트롤에 연결된 widget이 없어서 입력 포커스를 받을 수 없거나 다른 컨트롤을 포함할 수 없다는 것입니다. 그래픽 컨트롤에 핸들이 필요 없으므로 시스템 리소스에 대한 수요가 감소하고 그래픽 컨트롤을 widget 기반 컨트롤보다 더 빨리 그릴 수 있습니다.

TGraphicControl 컨트롤은 스스로 그려지고 다음과 같은 컨트롤을 포함해야만 합니다.

표 3.2 그래픽 컨트롤

컨트롤	설명
<i>TImage</i>	그래픽 이미지를 표시합니다.
<i>TLabel</i>	폼에 텍스트를 표시합니다.
<i>TBevel</i>	빚면이 있는 액자 모양의 윤곽을 나타냅니다.
<i>TPaintBox</i>	애플리케이션이 이미지를 그리거나 렌더링하는 데 사용할 수 있는 캔버스를 제공합니다.

여기에는 포커스를 받을 필요가 없는 공통 그리기 루틴 (*Repaint*, *Invalidate* 등)을 포함한다는 것에 유의하십시오.

TWinControl/TWidgetControl 분기

TWinControl 분기 (CLX에서는 *TWinControl*이 *TWidgetControl*로 대체됨)는 *TWinControl*의 자손인 모든 컨트롤을 포함합니다. *TWinControl*은 애플리케이션의 사용자 인터페이스에서 사용할 많은 항목을 비롯하여 창이 있는 모든 컨트롤의 기본 클래스입니다.

*TWidgetControl*은 모든 widget 기반 컨트롤 또는 *widgets*에 대한 기본 클래스입니다. widget은 "window"와 "gadget"을 합쳐 만든 용어입니다. widget은 애플리케이션의 사용자 인터페이스 어디에서나 사용할 수 있습니다. widget의 예로는 버튼, 레이블 및 스크롤 막대가 있습니다.

창이 있는 컨트롤과 widget 기반 컨트롤의 특징은 다음과 같습니다.

- 두 컨트롤 모두 애플리케이션이 실행 중일 때 포커스를 받을 수 있습니다.

- 다른 컨트롤에서 데이터를 표시할 수 있지만 사용자는 키보드를 사용하여 창이 있는 컨트롤이나 widget 기반 컨트롤과 상호 작용할 수 있습니다.
- 창이 있는 컨트롤이나 widget 기반 컨트롤은 다른 컨트롤을 포함할 수 있습니다.
- 다른 컨트롤을 포함하는 컨트롤을 부모라고 합니다. 창이 있는 컨트롤이나 widget 기반 컨트롤만이 하나 이상의 자식 컨트롤의 부모가 될 수 있습니다.
- 창이 있는 컨트롤은 창 핸들을 갖습니다. widget 기반 컨트롤은 연결된 widget을 가집니다.

TWinControl(CLX의 경우 *TWidgetControl*)의 자손은 포커스를 받을 수 있는 컨트롤이기 때문에 애플리케이션 사용자의 키보드 입력을 받을 수 있습니다. 이것은 더 많은 표준 이벤트가 컨트롤에 적용된다는 것을 암시합니다.

이 분기에는 자동으로 그려지는 컨트롤(예: *TEdit*, *TListBox*, *TComboBox*, *TPageControl* 등)과 Delphi가 그려야 하는 사용자 지정 컨트롤(예: *TDBNavigator*, *TMediaPlayer*(VCL 전용), *TGauge*(VCL 전용) 등)이 모두 포함되어 있습니다. *TWinControl*(CLX의 경우 *TWidgetControl*)의 직계 자손은 일반적으로 편집 필드, 콤보 박스, 리스트 박스, 페이지 컨트롤 등과 같은 표준 컨트롤을 구현하므로 자신을 그리는 방법을 이미 알고 있습니다.

TCustomControl 클래스는 창 핸들은 필요로 하지만 핸들 자체를 다시 그리는 기능을 포함하는 표준 컨트롤을 캡슐화하지 않는 컴포넌트에 제공됩니다. 컨트롤이 자신을 렌더링하거나 이벤트에 응답하는 방법은 Delphi에서 완벽하게 캡슐화하므로 이 동작을 걱정할 필요는 전혀 없습니다.

다음 단원에서는 컨트롤의 개요를 살펴 봅니다. 컨트롤 사용에 대한 자세한 내용은 7장 "컨트롤 사용"을 참조하십시오.

TControl의 공통 속성

모든 비주얼 컨트롤(*TControl*의 자손)은 다음과 같은 특정 속성을 공유합니다.

- 액션 속성
- 위치, 크기 및 정렬 속성
- 표시 속성
- 부모 속성
- 탐색 속성
- 드래그 앤 드롭 속성
- 드래그 앤 도킹 속성(VCL 전용)

*TControl*에서 상속되는 동안에 이러한 속성이 published로 선언되어 적용 가능한 컴포넌트에 대해서만 Object Inspector에 표시됩니다. 예를 들어, *TImage*는 *Color* 속성의 색이 표시되는 그래픽에 의해 결정되기 때문에 이 속성을 published로 선언하지 않습니다.

액션 속성

이를 통해 애플리케이션 상태에 따라 동작을 중앙 집중화된 단일 방식으로 설정 및 해제할 수 있을 뿐만 아니라 동작을 수행하는 공통적인 코드를 공유할 수 있습니다(예를 들어, 툴바 버튼과 메뉴 항목이 동일한 동작을 수행할 때).

- *Action*은 컨트롤과 연결된 동작을 지정합니다.
- *ActionLink*에는 컨트롤과 연결된 액션 링크 객체가 포함됩니다.

위치, 크기 및 정렬 속성

이 속성 집합은 부모 컨트롤 위에 놓인 컨트롤의 위치와 크기를 정의합니다.

- *Height*는 세로 크기를 설정합니다.
- *Width*는 가로 크기를 설정합니다.
- *Top*은 위쪽 가장자리 위치를 설정합니다.
- *Left*는 왼쪽 가장자리 위치를 설정합니다.
- *AutoSize*는 컨트롤 크기 자체가 내용에 맞게 자동으로 바뀌는지 여부를 지정합니다.
- *Align*은 컨트롤이 컨테이너(부모 컨트롤) 내에서 정렬되는 방법을 결정합니다.
- *Anchor*는 컨트롤이 부모에 앵커(연결)되는 방법을 지정합니다(VCL 전용).

이 속성 집합은 컨트롤 클라이언트 영역의 높이, 너비 및 전체 크기를 결정합니다.

- *ClientHeight*는 컨트롤 클라이언트 영역의 높이를 픽셀 단위로 지정합니다.
- *ClientWidth*는 컨트롤 클라이언트 영역의 너비를 픽셀 단위로 지정합니다.

이러한 속성은 너비주얼 컴포넌트에서 액세스할 수 없지만 Delphi는 폼 위에 컴포넌트 아이콘이 놓인 위치를 추적합니다. 대개는 이러한 속성을 설정 및 변경하기 위해 폼에서 컨트롤 이미지를 처리하거나 Alignment 팔레트를 사용할 것입니다. 하지만 이러한 속성은 런타임 시 변경할 수 있습니다.

표시 속성

다음 속성은 컨트롤의 일반 모양을 제어합니다.

- *Color*는 컨트롤의 배경색을 변경합니다.
- *Font*는 텍스트의 색, 타입, 스타일 또는 크기를 변경합니다.
- *Cursor*는 컨트롤 영역 안쪽으로 이동하는 마우스 포인터를 나타내는 데 사용되는 이미지를 지정합니다.
- *DesktopFont*는 컨트롤에서 텍스트 작성 시 Windows 아이콘 글꼴을 사용하는지 여부를 지정합니다(VCL 전용).

부모 속성

애플리케이션에 걸쳐 일관된 모양을 유지하려면 부모 속성을 *True*로 설정하여 모든 컨트롤을 *부모*라고 부르는 컨테이너와 같은 모양으로 만듭니다.

- *ParentColor*는 컨트롤이 색상 정보를 찾는 위치를 결정합니다.

- *ParentFont*는 컨트롤이 글꼴 정보를 찾는 위치를 결정합니다.
- *ParentShowHint*는 컨트롤이 도움말 힌트의 표시 여부를 찾는 위치를 결정합니다.

탐색 속성

다음 속성은 사용자가 폼에서 컨트롤 사이를 탐색하는 방법을 결정합니다.

- *Caption*에는 컴포넌트의 레이블을 지정하는 텍스트 문자열이 포함됩니다. 문자열의 문자에 밑줄을 그으려면 해당 문자 앞에 앤퍼센드를 포함합니다. 이러한 타입의 문자를 가속키라고 합니다. 사용자는 밑줄이 그어진 문자와 *Alt* 키를 동시에 눌러 컨트롤 또는 메뉴 항목을 선택할 수 있습니다.

드래그 앤 드롭 속성

다음 두 개의 컴포넌트 속성이 드래그 앤 드롭 동작에 영향을 미칩니다.

- *DragMode*는 끌기가 시작되는 방법을 결정합니다. *DragMode*의 기본값은 *dmManual*이며 애플리케이션은 끌기를 시작하기 위해 *BeginDrag* 메소드를 호출해야 합니다. *DragMode*가 *dmAutomatic*이면 끌기는 마우스 버튼을 눌렀을 때 바로 시작됩니다.
- *DragCursor*는 끌 수 있는 컴포넌트 위로 이동했을 때의 마우스 포인터 모양을 결정합니다(VCL 전용).

드래그 앤 도킹 속성(VCL 전용)

다음 속성은 드래그 앤 도킹 동작을 제어합니다.

- *Floating*은 컨트롤이 부동인지 여부를 나타냅니다.
- *DragKind*는 컨트롤 끌기가 일반적인 끌기인지, 도킹용 끌기인지 지정합니다.
- *DragMode*는 컨트롤이 드래그 앤 드롭 또는 드래그 앤 도킹 동작을 시작하는 방법을 결정합니다.
- *FloatingDockSiteClass*는 부동 상태일 때 컨트롤을 호스팅하는 임시 컨트롤의 클래스를 지정합니다.
- *DragCursor*는 끌기 동안에 표시되는 커서입니다.
- *DockOrientation*은 같은 부모에서 도킹된 다른 컨트롤을 기준으로 컨트롤이 도킹되는 방법을 지정합니다.
- *HostDockSite*는 컨트롤이 도킹되는 컨트롤을 지정합니다.

자세한 내용은 7-4 페이지의 "컨트롤에서 드래그 앤 드롭 구현"을 참조하십시오.

TControl의 표준 공통 이벤트

VCL은 컨트롤에 대한 표준 이벤트 집합을 정의합니다. 다음 이벤트는 *TControl* 클래스의 일부로 선언되기 때문에 *TControl*에서 파생된 모든 클래스에서 사용할 수 있습니다.

- *OnClick*은 사용자가 컨트롤을 클릭할 때 발생합니다.
- *OnContextPopup*은 사용자가 마우스 오른쪽 버튼으로 컨트롤을 클릭하거나 키보드 등을 사용하여 팝업 메뉴를 호출할 때 발생합니다.

- *OnCanResize*는 컨트롤 크기를 조정하려는 경우에 발생합니다.
- *OnResize*는 컨트롤 크기를 조정된 직후에 발생합니다.
- *OnConstrainedResize*는 *OnCanResize* 직후에 발생합니다.
- *OnStartDock*은 *dkDock*의 *DragKind*로 컨트롤 끌기를 시작할 때 발생합니다(VCL 전용).
- *OnEndDock*은 객체를 도킹하거나 끌기를 취소하여 객체 끌기를 끝낼 때 발생합니다(VCL 전용).
- *OnStartDrag*는 컨트롤을 마우스 왼쪽 버튼으로 클릭한 상태에서 컨트롤이나 컨트롤에 포함된 객체를 끌기 시작할 때 발생합니다.
- *OnEndDrag*는 객체를 끌어다 놓거나 끌기를 취소하여 객체 끌기를 끝낼 때 발생합니다.
- *OnDragDrop*은 객체를 끌어다 놓을 때 발생합니다.
- *OnMouseMove*는 컨트롤 위에서 마우스 포인터를 이동할 때 발생합니다.
- *OnDbClick*은 컨트롤 위에 마우스 포인터를 놓고 기본 마우스 버튼을 더블 클릭할 때 발생합니다.
- *OnDragOver*는 컨트롤 위에서 객체를 끌 때 발생합니다(VCL 전용).
- *OnMouseDown*은 사용자가 컨트롤 위에 마우스 포인터를 놓고 마우스 버튼을 누를 때 발생합니다.
- *OnMouseUp*은 컴포넌트 위에 마우스 포인터를 놓고 눌렀던 마우스 버튼을 놓았을 때 발생합니다.

TWinControl 및 TWidgetControl의 공통 속성

창이 있는 모든 컨트롤(VCL에서의 *TWinControl* 자손과 CLX에서의 *TWidgetControl* 자손)은 다음과 같은 특정 속성을 공유합니다.

- 컨트롤에 관한 정보
- 테두리 스타일 표시 속성
- 탐색 속성
- 드래그 앤 도킹 속성(VCL 전용)

TWinControl 및 *TWidgetControl*에서 상속되는 동안 이러한 속성이 published로 선언되며 적용 가능한 컨트롤에 대해서만 Object Inspector에 표시됩니다.

일반 정보 속성

일반 정보 속성은 *TWinControl* 및 *TWidgetControl*의 모양, 클라이언트 영역 크기 및 원점, 창에 할당된 정보, 도움말 컨텍스트 등에 대한 정보를 포함합니다.

- *ClientOrigin*은 컨트롤 클라이언트 영역의 왼쪽 위 모서리의 화면 좌표를 픽셀 단위로 지정합니다. *TWinControl*이 아니라 *TControl*의 자손인 특정 컨트롤의 화면 좌표는 *Left* 및 *Top* 속성에 추가되는 해당 컨트롤 부모의 화면 좌표입니다.
- *ClientRect*는 *Top*과 *Left* 속성이 0으로 설정되고 *Bottom*과 *Right* 속성이 각각 컨트롤의 *Height* 및 *Width*로 설정된 경우 사각형을 반환합니다. *ClientRect*는 Rect(0, 0, *ClientWidth*, *ClientHeight*)와 같습니다.

- *Brush*는 컨트롤의 배경을 칠하는 데 사용되는 색과 패턴을 결정합니다.
- *HelpContext*는 문맥에 맞는 온라인 도움말을 호출할 때 사용하는 컨텍스트 번호를 제공합니다.
- *Handle*은 컨트롤의 창 또는 widget 핸들에 대한 액세스를 제공합니다.

테두리 스타일 표시 속성

경사 속성은 애플리케이션의 폼과 창이 있는 컨트롤에 있는 경사진 선, 상자 또는 프레임의 모양을 제어합니다.

VCL에서는 더 많은 객체가 이러한 속성을 *published*로 선언하지만 CLX에서는 이러한 속성 중 일부를 사용할 수 없기 때문에 더 적은 객체에서 테두리 스타일 속성이 *published*로 선언됩니다.

- *InnerBevel*은 내부 경사 모양이 볼록, 오목 또는 평면인지 지정합니다(VCL 전용).
- *BevelKind*는 컨트롤 가장자리가 경사진 경우 경사 타입을 지정합니다(VCL 전용).
- *BevelOuter*는 외부 경사 모양이 볼록, 오목 또는 평면인지 지정합니다.
- *BevelWidth*는 내부 및 외부 경사의 너비를 픽셀 단위로 지정합니다.
- *BorderWidth*는 컨트롤 테두리의 너비를 얻거나 설정하는 데 사용됩니다.
- *BevelEdges*는 컨트롤의 경사진 가장자리를 얻거나 설정하는 데 사용됩니다.

탐색 속성

다음 두 개의 추가적인 속성은 사용자가 폼에서 컨트롤 사이를 탐색하는 방법을 결정합니다.

- *TabOrder*는 컨트롤 위치를 부모의 탭 순서로 나타내며 이 순서는 사용자가 *Tab* 키를 눌렀을 때 컨트롤이 포커스를 받는 순서입니다. 원래 탭 순서는 컴포넌트가 폼에 추가된 순서이지만 *TabOrder*를 변경하여 이 순서를 변경할 수 있습니다. *TabOrder*는 *TabStop*이 *True*인 경우에만 의미가 있습니다.
- *TabStop*은 사용자가 컨트롤에 탭을 지정할 수 있는지 결정합니다. *TabStop*이 *True*이면 컨트롤은 탭 순서를 가집니다.

드래그 앤 도킹 속성(VCL 전용)

다음 속성은 VCL 객체의 드래그 앤 도킹 동작을 관리합니다.

- *UseDockManager*는 드래그 앤 도킹 작업에 도킹 관리자가 사용되는지 여부를 지정합니다.
- *VisibleDockClientCount*는 창이 있는 컨트롤에서 도킹되는 가시적(visible) 컨트롤 개수를 지정합니다.
- *DockManager*는 컨트롤의 도킹 관리자 인터페이스를 지정합니다.
- *DockClients*는 창이 있는 컨트롤에 도킹되는 컨트롤을 나열합니다.
- *DockSite*는 컨트롤이 드래그 앤 도킹 작업의 대상이 될 수 있는지 여부를 지정합니다.

자세한 내용은 7-4 페이지의 "컨트롤에서 드래그 앤 드롭 구현"을 참조하십시오.

TWinControl 및 TWidgetControl의 공통 이벤트

다음은 VCL의 *TWinControl*(Windows가 정의하는 모든 컨트롤도 포함) 및 CLX의 *TWidgetControl*에서 파생되는 모든 컨트롤에 있는 이벤트입니다. 이러한 이벤트는 모든 컨트롤에 존재하는 이벤트 이외의 추가적인 이벤트입니다.

- *OnEnter*는 컨트롤이 포커스를 받으려고 할 때 발생합니다.
- *OnKeyDown*은 키 입력의 다운 스트로크에서 발생합니다.
- *OnKeyPress*는 하나의 문자 키를 눌렀을 때 발생합니다.
- *OnKeyUp*은 눌렀던 키를 놓을 때 발생합니다.
- *OnExit*는 특정 컨트롤에서 다른 컨트롤로 입력 포커스가 전환될 때 발생합니다.
- *OnMouseWheel*은 마우스 휠이 회전할 때 발생합니다.
- *OnMouseWheelDown*은 마우스 휠이 아래쪽으로 회전할 때 발생합니다.
- *OnMouseWheelUp*은 마우스 휠이 위쪽으로 회전할 때 발생합니다.

다음 이벤트는 도킹과 관련된 것으로서 VCL에서만 사용할 수 있습니다.

- *OnUnDock*은 애플리케이션이 창이 있는 컨트롤에 도킹되는 컨트롤을 도킹 해제하려고 할 때 발생합니다(VCL 전용).
- *OnDockDrop*은 다른 컨트롤을 해당 컨트롤로 도킹할 때 발생합니다(VCL 전용).
- *OnDockOver*는 다른 컨트롤을 해당 컨트롤로 끌어올 때 발생합니다(VCL 전용).
- *OnGetSiteInfo*는 컨트롤의 도킹 정보를 반환합니다(VCL 전용).

애플리케이션 사용자 인터페이스 만들기

Delphi의 모든 비주얼 디자인 작업은 폼에서 이루어집니다. Delphi를 열거나 새 프로젝트를 열 때 빈 폼이 화면에 표시됩니다. 이 폼을 사용하여 창, 메뉴 및 일반 대화 상자를 비롯한 애플리케이션 인터페이스 만들기를 시작할 수 있습니다.

버튼, 리스트 박스 등의 비주얼 컴포넌트를 폼에 배치 및 정렬하여 애플리케이션에 대한 그래픽 사용자 인터페이스의 룩앤필(look and feel)을 디자인합니다. Delphi에서는 기본이 되는 프로그래밍 세부 내용을 처리합니다. 또한 보이지 않는 컴포넌트를 폼에 놓아 데이터베이스 정보를 캡처하고, 계산을 수행하고, 기타 상호 작용을 관리할 수 있습니다.

6장 "애플리케이션 사용자 인터페이스 개발"에는 모달 폼을 동적으로 만들고, 폼에 매개변수를 전달하고, 폼으로부터 데이터를 검색하는 것과 같은 폼 사용에 관한 자세한 내용이 나와 있습니다.

Delphi 컴포넌트 사용

컴포넌트 팔레트에 있는 개발 환경 자체에서 많은 비주얼 컴포넌트가 제공됩니다. Delphi의 모든 비주얼 디자인 작업은 폼에서 이루어집니다. Kylix를 열거나 새 프로젝트를 열 때 빈 폼이 화면에 표시됩니다. 컴포넌트 팔레트에서 컴포넌트를 선택하여 폼에 가져다 놓습니다. 폼 위의 버튼과 리스트 박스와 같은 비주얼 컴포넌트를 정렬하여 애플리케이션 사용자 인터페이스의 룩앤필(look and feel)을 디자인합니다. 일단 비주얼 컴포넌트가 폼에 있으면 컴포넌트의 위치, 크기 및 기타 디자인 타임 속성을 조정할 수 있습니다. Delphi에서는 기본이 되는 프로그래밍 세부 내용을 처리합니다.

Delphi 컴포넌트는 기능별로 그룹화되어 컴포넌트 팔레트의 페이지에 다른 정렬되어 있습니다. 예를 들어 메뉴, 편집 상자 또는 버튼을 생성하는 컴포넌트 같이 공통으로 사용되는 컴포넌트는 컴포넌트 팔레트의 Standard 페이지에 있습니다. 타이머, 그림 상자, 미디어 플레이어, OLE 컨테이너 등의 편리한 VCL 컨트롤은 System 페이지에 있습니다.

인뜻 보면 Delphi의 컴포넌트는 다른 클래스와 비슷하게 나타납니다. 그러나 Delphi에 있는 컴포넌트와 많은 프로그래머들이 사용하는 표준 클래스 계층 구조 간에 차이가 있습니다. 몇 가지 차이점은 다음과 같습니다.

- 모든 Delphi 컴포넌트는 *TComponent*의 자손입니다.
- 컴포넌트는 기능을 추가하거나 변경하기 위해 서브 클래스인 "기본 클래스"로 사용하기보다는 대부분 있는 그대로 사용되고 속성을 통해 변경됩니다. 상속된 컴포넌트인 경우, 보통 기존 이벤트 핸들링 멤버 함수에 특정 코드를 추가해야 합니다.
- 컴포넌트는 스택이 아니라 힙에만 할당될 수 있습니다.
- 컴포넌트의 속성은 본질적으로 런타임 타입 정보를 포함합니다.
- 컴포넌트는 Delphi 사용자 인터페이스의 컴포넌트 팔레트에 추가되어 폼에서 조작할 수 있습니다.

표준 클래스에서 일반적으로 수행되는 것보다 더 나은 캡슐화를 컴포넌트가 실현하는 경우가 종종 있습니다. 예를 들어, 푸시 버튼이 포함된 대화 상자를 가정해 보십시오. VCL 컴포넌트를 사용하여 개발한 Windows 프로그램의 경우, 사용자가 이 버튼을 클릭하면 시스템은 WM_LBUTTONDOWN 메시지를 생성합니다. 프로그램은 일반적으로 **switch** 문, 메시지 맵 또는 응답 테이블에 있는 이 메시지를 가져와 메시지에 응답하여 실행할 루틴에 디스패칭해야 합니다.

대부분의 Windows 메시지(VCL)나 시스템 이벤트(CDX)는 Delphi 컴포넌트에 의해 처리됩니다. 메시지에 응답하려면 이벤트 핸들러만 제공하면 됩니다.

컴포넌트 속성 설정

Published 속성은 디자인 타임 시 Object Inspector에서 설정하거나, 경우에 따라 특수 속성 편집기를 사용하여 설정할 수 있습니다.

런타임 시 속성을 설정하려면 애플리케이션 소스 코드에서 속성에 새 값을 할당합니다.

각 컴포넌트의 속성에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

Object Inspector 사용

폼 위의 컴포넌트를 선택하면 Object Inspector는 선택된 컴포넌트의 published 속성을 표시하고 가능한 경우에는 사용자가 편집할 수 있게 해줍니다. Value 열과 Property 열 사이를 토글하려면 *Tab* 키를 사용합니다. 커서가 Property 열에 있으면 속성 이름의 첫 문자를 입력하여 속성을 탐색할 수 있습니다. 부울 및 열거 타입의 속성인 경우, 드롭다운 목록에서 값을 선택하거나 Value 열을 더블 클릭하여 설정을 토글할 수 있습니다.

속성 이름 옆에 더하기(+) 기호가 있을 경우, 더하기 기호를 클릭하거나 속성에 포커스가 있을 때 '+'를 입력하면 해당 속성의 하위 값 목록이 표시됩니다. 마찬가지로 속성 이름 옆에 빼기(-) 기호가 있을 경우, 빼기 기호를 클릭하거나 '-'를 입력하면 하위 값이 숨겨집니다.

기본적으로 Legacy 범주의 속성은 표시되지 않습니다. 표시 필터를 변경하려면 Object Inspector에서 마우스 오른쪽 버튼을 클릭한 다음 View를 선택합니다. 자세한 내용은 온라인 도움말의 "property categories"를 참조하십시오.

둘 이상의 컴포넌트가 선택되면 Object Inspector는 선택된 컴포넌트가 공유하는 모든 속성 (*Name* 제외)을 표시합니다. 선택된 컴포넌트 간에 공유 속성 값이 다른 경우, Object Inspector는 기본값을 표시하거나 처음 선택된 컴포넌트의 값을 표시합니다. 공유 속성을 변경하면 선택된 모든 컴포넌트에 변경 사항이 적용됩니다.

속성 편집기 사용

*Font*와 같은 일부 속성은 특수 속성 편집기를 갖고 있습니다. 이러한 속성은 Object Inspector에서 속성을 선택할 때 그 값 옆에 생략 기호(...)가 나타납니다. 속성 편집기를 열려면 Value 열을 더블 클릭하거나, 생략 기호를 클릭하거나, 속성이나 속성의 값에 포커스가 있을 때 *Ctrl+Enter*를 누릅니다. 일부 컴포넌트의 경우, 폼 위의 컴포넌트를 더블 클릭하면 속성 편집기가 함께 열립니다.

속성 편집기는 단일 대화 상자에서 복잡한 속성을 설정할 수 있게 해줍니다. 속성 편집기는 입력을 확인하고 종종 할당의 결과를 미리 볼 수 있게 해줍니다.

런타임 시 속성 설정

쓰기 가능한 모든 속성은 런타임 시 소스 코드에서 설정할 수 있습니다. 예를 들어, 다음과 같이 폼에 캡션을 동적으로 할당할 수 있습니다.

```
Form1.Caption := MyString;
```

메소드 호출

메소드는 일반적인 프로시저 및 함수와 마찬가지로 호출됩니다. 예를 들어, 비주얼 컨트롤은 화면 상의 컨트롤 이미지를 새로 고치는 *Repaint* 메소드를 갖습니다. 그리기 그리드 객체의 *Repaint* 메소드를 다음과 같이 호출할 수 있습니다.

```
DrawGrid1.Repaint;
```

속성의 경우와 마찬가지로 메소드 이름의 유효 범위(scope)가 한정자의 필요성을 결정합니다. 예를 들어, 다음과 같이 폼의 자식 컨트롤 중 하나의 이벤트 핸들러 내에서 폼을 다시 그리려는 경우에는 폼 이름을 메소드 호출에 추가할 필요가 없습니다.

```
procedure TForm1.Button1Click(Sender:TObject);  
begin  
  Repaint;  
end;
```

유효 범위(scope)에 대한 자세한 내용은 3-8 페이지의 "유효 범위(scope) 및 한정자"를 참조하십시오.

이벤트 및 이벤트 핸들러 작업

Delphi에서 작성되는 거의 모든 코드는 *이벤트*에 응답하여 직접적 또는 간접적으로 실행됩니다. 이벤트는 종종 사용자 동작인 런타임 발생을 나타내는 특수한 종류의 속성입니다. 이벤트에 직접 응답하는 코드, 즉 *이벤트 핸들러*는 오브젝트 파스칼 프로시저입니다. 다음 단원에서는 아래 작업을 수행하는 방법에 대해 설명합니다.

- 새 이벤트 핸들러 생성
- 컴포넌트의 기본 이벤트에 대한 핸들러 생성
- 이벤트 핸들러 찾기
- 이벤트를 기존 이벤트 핸들러에 연결
- 메뉴 이벤트를 이벤트 핸들러에 연결
- 이벤트 핸들러 삭제

새 이벤트 핸들러 생성

Delphi는 폼과 다른 컴포넌트에 대한 뼈대 이벤트 핸들러를 생성할 수 있습니다. 다음과 같은 방법으로 이벤트 핸들러를 작성합니다.

- 1 컴포넌트를 선택합니다.
- 2 Object Inspector의 Event 탭을 클릭합니다. Object Inspector의 Events 페이지가 해당 컴포넌트에 대해 정의된 모든 이벤트를 표시합니다.
- 3 원하는 이벤트를 선택한 다음 Value 열을 더블 클릭하거나 *Ctrl+Enter*를 누릅니다. Delphi는 코드 에디터에서 이벤트 핸들러를 생성하고 **begin...end** 블록 안에 커서를 둡니다.
- 4 **begin...end** 블록 안에 이벤트 발생 시 실행할 코드를 입력합니다.

컴포넌트의 기본 이벤트에 대한 핸들러 생성

일부 컴포넌트는 가장 일반적으로 처리해야 하는 이벤트인 *기본* 이벤트를 갖습니다. 예를 들어, 버튼의 기본 이벤트는 *OnClick*입니다. 기본 이벤트 핸들러를 만들려면 폼 디자이너에서 컴포넌트를 더블 클릭합니다. 이렇게 하면 뼈대 이벤트 처리 프로시저가 생성되고 프로시저 몸체(body)에 커서가 놓인 상태에서 코드 에디터가 열리기 때문에 코드를 쉽게 추가할 수 있습니다.

모든 컴포넌트가 기본 이벤트를 갖는 것은 아닙니다. *TBevel*과 같은 일부 컴포넌트는 어떤 이벤트에도 응답하지 않습니다. 다른 컴포넌트는 폼 디자이너에서 더블 클릭했을 때 다르게 응답합니다. 예를 들어, 많은 컴포넌트가 디자인 타임 시 더블 클릭했을 때 기본 속성 편집기나 기타 대화 상자를 엽니다.

이벤트 핸들러 찾기

폼 디자이너에서 컴포넌트를 더블 클릭하여 컴포넌트에 대한 기본 이벤트 핸들러를 생성한 경우, 동일한 방법으로 기본 이벤트 핸들러를 찾을 수 있습니다. 컴포넌트를 더블 클릭하면 코드 에디터가 열리면서 이벤트 핸들러 본문의 시작 부분에 커서가 위치하게 됩니다.

기본 이벤트 핸들러가 아닌 이벤트 핸들러는 다음과 같은 방법으로 찾습니다.

- 1 폼에서 찾고자 하는 이벤트 핸들러의 컴포넌트를 선택합니다.
- 2 Object Inspector에서 Events 탭을 클릭합니다.
- 3 확인할 핸들러의 이벤트를 선택한 다음 Value 열을 더블 클릭합니다. 코드 에디터가 열리고 이벤트 핸들러 몸체(body)의 시작 부분에 커서가 놓입니다.

이벤트를 기존 이벤트 핸들러에 연결

둘 이상의 이벤트에 응답하는 이벤트 핸들러를 작성하여 코드를 재사용할 수 있습니다. 예를 들면, 많은 애플리케이션에서 드롭다운 메뉴 명령에 해당하는 스피드 버튼을 제공합니다. 버튼이 메뉴 명령과 같은 동작을 일으키면 하나의 이벤트 핸들러를 작성하여 버튼 및 메뉴 항목의 *OnClick* 이벤트에 할당할 수 있습니다.

다음과 같은 방법으로 이벤트를 기존 이벤트 핸들러에 연결합니다.

- 1 폼 위에서 처리하고자 하는 이벤트를 갖는 컴포넌트를 선택합니다.
- 2 Object Inspector의 Events 페이지에서 핸들러를 연결할 이벤트를 선택합니다.
- 3 이벤트 옆에 있는 Value 열에서 아래쪽 화살표를 클릭하여 이전에 작성한 이벤트 핸들러의 목록을 엽니다. 이 목록에는 동일한 폼에 있는 동일한 이름의 이벤트에 대해 작성된 이벤트 핸들러만 포함되어 있습니다. 이벤트 핸들러 이름을 클릭하여 목록에서 선택합니다.

위 절차는 이벤트 핸들러를 재사용할 수 있는 쉬운 방법입니다. 하지만 *액션 리스트*와 *VCL*의 *액션 밴드*는 사용자 명령에 응답하는 코드를 중앙에서 구성하기 위한 강력한 도구를 제공합니다. 액션 밴드와 달리 액션 리스트는 크로스 플랫폼 애플리케이션에서 사용할 수 있습니다. 액션 리스트와 액션 밴드에 대한 자세한 내용은 6-16 페이지의 "툴바 및 메뉴의 작업 구성"을 참조하십시오.

Sender 매개변수 사용

이벤트 핸들러에서 *Sender* 매개변수는 이벤트를 받아 핸들러를 호출한 컴포넌트를 나타냅니다. 호출하는 컴포넌트에 따라 다르게 동작하는 이벤트 핸들러를 여러 컴포넌트가 공유하게 만드는 것은 경우에 따라 유용할 수 있습니다. 이렇게 하려면 *if...then...else* 문에서 *Sender* 매개변수를 사용합니다. 예를 들어, 다음 코드는 *Button1*이 *OnClick* 이벤트를 받은 경우에만 대화 상자의 캡션에 애플리케이션의 제목을 표시합니다.

```
procedure TMainForm.Button1Click(Sender:TObject);  
begin  
  if Sender = Button1 then  
    AboutBox.Caption := 'About ' + Application.Title  
  else  
    AboutBox.Caption := '';  
  AboutBox.ShowModal;  
end;
```

공유 이벤트 표시 및 코딩

컴포넌트가 이벤트를 공유할 경우, Object Inspector에 공유 이벤트를 표시할 수 있습니다. 우선 *Shift* 키를 누른 상태에서 폼 디자이너에서 컴포넌트를 클릭하여 선택한 다음 Object Inspector에서 Events 탭을 선택합니다. 이렇게 하면 Object Inspector의 Value 열에서 공유 이벤트에 대한 새 이벤트 핸들러를 만들거나 기존 이벤트 핸들러를 공유 이벤트에 할당할 수 있습니다.

메뉴 이벤트를 이벤트 핸들러에 연결

MainMenu 및 *PopupMenu* 컴포넌트와 함께 메뉴 디자이너는 애플리케이션에 드롭다운 및 팝업 메뉴를 쉽게 제공할 수 있게 해줍니다. 그러나 메뉴가 작동하려면 사용자가 메뉴 항목을 선택하거나 가속키 또는 단축키를 누를 때마다 발생하는 *OnClick* 이벤트에 각 메뉴 항목이 응답해야 합니다. 이 단원에서는 메뉴 항목에 이벤트 핸들러를 연결하는 방법을 설명합니다. 메뉴 디자이너 및 관련 컴포넌트에 대한 자세한 내용은 6-29 페이지의 "메뉴 생성 및 관리"를 참조하십시오.

다음과 같은 방법으로 메뉴 항목에 대한 이벤트 핸들러를 만듭니다.

- 1 *MainMenu* 또는 *PopupMenu* 객체를 더블 클릭하여 메뉴 디자이너를 엽니다.
- 2 메뉴 디자이너에서 메뉴 항목을 선택합니다. Object Inspector에서 항목의 *Name* 속성에 값이 할당되었는지 확인합니다.
- 3 메뉴 디자이너에서 메뉴 항목을 더블 클릭합니다. Delphi는 코드 에디터에서 이벤트 핸들러를 생성하고 **begin...end** 블록 안에 커서를 놓습니다.
- 4 **begin...end** 블록 안에서 사용자가 메뉴 명령을 선택할 때 실행할 코드를 입력합니다.

다음과 같은 방법으로 메뉴 항목을 기존 *OnClick* 이벤트 핸들러에 연결합니다.

- 1 *MainMenu* 또는 *PopupMenu* 객체를 더블 클릭하여 메뉴 디자이너를 엽니다.
- 2 메뉴 디자이너에서 메뉴 항목을 선택합니다. Object Inspector에서 항목의 *Name* 속성에 값이 할당되었는지 확인합니다.
- 3 Object Inspector의 Events 페이지에서 *OnClick* 옆에 있는 Value 열의 아래쪽 화살표를 클릭하여 이전에 작성된 이벤트 핸들러의 목록을 엽니다. 이 목록에는 이 폼의 *OnClick* 이벤트에 대해 작성된 이벤트 핸들러만 포함되어 있습니다. 이벤트 핸들러 이름을 클릭하여 목록에서 선택합니다.

이벤트 핸들러 삭제

폼 디자이너를 사용하여 폼에서 컴포넌트를 삭제하면 Delphi는 폼의 타입 선언에서 컴포넌트를 제거합니다. 하지만 연결된 모든 메소드가 폼의 다른 컴포넌트에 의해 계속 호출될 수 있기 때문에 유닛 파일에서는 삭제되지 않습니다. 이벤트 핸들러 같은 메소드를 수동으로 삭제할 수 있지만 이 경우 유닛의 **interface** 섹션에 있는 메소드의 forward 선언과 **implementation** 섹션에 있는 그 구현을 모두 삭제해야 합니다. 그렇게 하지 않으면 프로젝트 빌드 시 컴파일러 오류가 발생합니다.

VCL 및 CLX 컴포넌트

컴포넌트 팔레트에는 광범위한 프로그래밍 작업을 처리하는 여러 개의 컴포넌트들이 들어 있습니다. 팔레트에서 컴포넌트를 추가, 제거 및 재정렬할 수 있고 여러 컴포넌트를 그룹화하는 컴포넌트 *템플릿* 및 *프레임*을 만들 수 있습니다.

팔레트에 있는 컴포넌트는 용도와 기능별로 페이지에 정렬되어 있습니다. 기본 구성으로 표시되는 페이지는 현재 실행 중인 Delphi의 버전에 따라 달라집니다. 표 3.3에는 애플리케이션 작성에 사용할 수 있는 일반적인 기본 페이지와 컴포넌트가 나열되어 있습니다. 일부 탭과 컴포넌트는 크로스 플랫폼이 아니며 이 표는 크로스 플랫폼 여부를 표시하고 있습니다. Windows 전용 CLX 애플리케이션에서 일부 VCL 특정 년비주얼 컴포넌트를 사용할 수 있지만 이러한 코드 부분을 분리하지 않으면 애플리케이션은 크로스 플랫폼이 되지 않습니다.

표 3.3 컴포넌트 팔레트 페이지

페이지 이름	설명	크로스 플랫폼 지원 여부
Standard	표준 컨트롤, 메뉴	예
Additional	특수 컨트롤	예 (ApplicationEvents와 CustomizeDlg는 제외)
Win32	Windows 일반 컨트롤	CLX 애플리케이션을 만들 때 대신 나타나는 Common Controls 탭의 동일한 컴포넌트(예: RichEdit, UpDown, HotKey, Animate, DateTimePicker, MonthCalendar, Coolbar, PageScroller 및 ComboBoxEx)는 대부분 크로스 플랫폼이 아닙니다.
System	타이머, 멀티미디어 및 DDE를 비롯한 시스템 수준 액세스를 위한 컴포넌트 및 컨트롤	Timer는 크로스 플랫폼이지만 PaintBox, MediaPlayer, OleContainer 및 Dde 컴포넌트는 크로스 플랫폼이 아닙니다.
Data Access	특정 데이터 액세스 메커니즘에 연결되지 않은 데이터베이스 데이터 작업을 위한 컴포넌트	예
Data Controls	비주얼, data-aware 컨트롤	예 (DBRichEdit, DBCtrlGrid 및 DBChart 제외)

표 3.3 컴포넌트 팔레트 페이지 (계속)

페이지 이름	설명	크로스 플랫폼 지원 여부
dbExpress	동적 SQL 처리를 위한 메소드를 제공하는 크로스 플랫폼의 데이터베이스 독립 계층인 dbExpress를 사용하는 데이터베이스 컨트롤. SQL 서버 액세스를 위한 일반 인터페이스를 정의합니다.	예
DataSnap	다계층 데이터베이스 애플리케이션 작성에 사용되는 컴포넌트	Windows CLX 애플리케이션에서 사용될 수 있습니다.
BDE	Borland Database Engine을 통해 데이터 액세스를 제공하는 컴포넌트	Windows CLX 애플리케이션에서 사용될 수 있습니다.
ADO	ADO 프레임워크를 통해 데이터 액세스를 제공하는 컴포넌트	Windows CLX 애플리케이션에서 사용될 수 있습니다.
InterBase	InterBase에 대한 직접 액세스를 제공하는 컴포넌트	예
InternetExpress	웹 서버 애플리케이션이자 동시에 다계층 데이터베이스 애플리케이션의 클라이언트인 컴포넌트	Windows CLX 애플리케이션에서 사용될 수 있습니다.
Internet	인터넷 통신 프로토콜과 웹 애플리케이션용 컴포넌트	예 (ClientSocket, ServerSocket, QueryTableProducer, XMLDoc 및 WebBrowser 제외)
WebSnap	웹 서버 애플리케이션 작성을 위한 컴포넌트	Windows CLX 애플리케이션에서 사용될 수 있습니다.
FastNet	NetMasters Internet 컨트롤	Windows CLX 애플리케이션에서 사용될 수 있습니다.
QReport	내장 보고서 작성을 위한 QuickReport 컴포넌트	Windows CLX 애플리케이션에서 사용될 수 있습니다.
Dialogs	일반 대화 상자	예 (OpenPictureDialog, SavePictureDialog, PrinterSetupDialog 및 PageSetupDialog 제외)
Win 3.1 Samples	이전 스타일 Win 3.1 컴포넌트	아니오
ActiveX	예제 사용자 지정 컴포넌트	아니오
	예제 ActiveX 컨트롤 (msdn.microsoft.com에서 Microsoft 설명서 참조)	아니오
COM+	COM+ 이벤트를 처리하는 컴포넌트	Windows CLX 애플리케이션에서 사용될 수 있습니다.
WebServices	SOAP 기반 웹 서비스를 구현하거나 사용하는 애플리케이션 작성을 위한 컴포넌트	Windows CLX 애플리케이션에서 사용될 수 있습니다.
Servers	Microsoft Excel, Word 등에 대한 COM 서버 예제 (Microsoft MSDN 설명서 참조)	Windows CLX 애플리케이션에서 사용될 수 있습니다.
Indy Clients	클라이언트용 크로스 플랫폼 인터넷 컴포넌트 (공개 소스 Winshoes Internet 컴포넌트)	예

표 3.3 컴포넌트 팔레트 페이지 (계속)

페이지 이름	설명	크로스 플랫폼 지원 여부
Indy Servers	서버용 크로스 플랫폼 인터넷 컴포넌트 (공개 소스 Winshoes Internet 컴포넌트)	예
Indy Misc	추가적인 크로스 플랫폼 인터넷 컴포넌트 (공개 소스 Winshoes Internet 컴포넌트)	예

온라인 도움말은 컴포넌트 팔레트의 컴포넌트에 관한 정보를 제공합니다. 하지만 ActiveX, Servers 및 Samples 페이지의 일부 컴포넌트는 예제만 제공되고 따로 설명되지 않습니다.

컴포넌트 팔레트에 사용자 지정 컴포넌트 추가

직접 작성했거나 협력 업체에서 작성한 사용자 지정 컴포넌트를 컴포넌트 팔레트에 설치하여 애플리케이션에서 사용할 수 있습니다. 컴포넌트를 작성하려면 5부 "사용자 지정 컴포넌트 생성"을 참조하십시오. 기존의 컴포넌트를 설치하려면 11-5 페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

텍스트 컨트롤

많은 애플리케이션이 텍스트를 사용자에게 보여 주거나 사용자가 텍스트를 입력할 수 있도록 합니다. 이러한 목적으로 사용된 컨트롤 타입은 정보의 크기와 서식에 따라 다릅니다.

컴포넌트	수행 작업
<i>TEdit</i>	한 줄의 텍스트를 편집합니다.
<i>TMemo</i>	여러 줄의 텍스트를 편집합니다.
<i>TMaskEdit</i>	우편 번호 또는 전화 번호 같은 특정 서식을 사용합니다.
<i>TRichEdit</i>	서식있는 텍스트를 사용하여 여러 줄의 텍스트를 편집합니다(VCL 전용).

*TEdit*와 *TMaskEdit*는 정보를 입력할 수 있는 한 줄의 텍스트 편집 상자가 포함된 단순한 텍스트 컨트롤입니다. 편집 상자에 포커스가 있으면 깜빡이는 삽입 지점이 나타납니다.

문자열 값을 텍스트의 *Text* 속성에 할당하여 텍스트를 편집 상자에 포함할 수 있습니다. *Font* 속성에 값을 할당하여 편집 상자에 있는 텍스트 글꼴을 조정합니다. 글꼴 종류, 크기, 색상 및 속성을 지정할 수 있습니다. 속성은 편집 상자의 모든 텍스트에 영향을 미치고 개별 문자에 적용될 수 없습니다.

포함하는 글꼴의 크기에 따라 편집 상자의 크기가 변경되도록 할 수 있습니다. *AutoSize* 속성을 True로 설정하면 글꼴 크기에 따라 편집 상자의 크기가 변합니다. *MaxLength* 속성에 값을 할당하여 편집 상자의 문자 수를 제한할 수 있습니다.

*TMaskEdit*은 텍스트의 유효한 폼을 암호화하는 마스크에 대하여 입력된 텍스트를 확인하는 특별 편집 컨트롤입니다. 마스크는 사용자에게 표시된 텍스트의 서식을 지정할 수 있습니다.

*TMemo*는 여러 줄의 텍스트를 추가할 때 사용합니다.

텍스트 컨트롤 속성

다음은 텍스트 컨트롤의 중요한 속성 중 일부를 보여 줍니다.

표 3.4 텍스트 컨트롤 속성

속성	설명
<i>Text</i>	편집 상자 또는 메모 컨트롤에 나타나는 텍스트를 결정합니다.
<i>Font</i>	편집 상자나 메모 컨트롤에 쓰인 텍스트의 속성 (attribute)을 제어합니다.
<i>AutoSize</i>	현재 선택된 글꼴에 맞게 편집 상자의 높이를 동적으로 변경할 수 있도록 합니다.
<i>ReadOnly</i>	사용자가 텍스트를 변경할 수 있는지 여부를 지정합니다.
<i>MaxLength</i>	단순 텍스트 컨트롤에서 문자 수를 제한합니다.

메모 및 서식있는 텍스트 컨트롤의 속성

여러 줄의 텍스트를 처리하는 메모 및 서식있는 텍스트 컨트롤은 공통된 여러 속성을 갖고 있습니다. 서식있는 텍스트 컨트롤은 크로스 플랫폼에 사용할 수 없습니다.

*TMemo*는 여러 줄의 텍스트를 처리하는 다른 타입의 편집 상자입니다. 메모 컨트롤의 줄은 편집 상자의 오른쪽 테두리 이상으로 확장할 수 있거나 다음 줄로 줄 바꿈할 수 있습니다. *WordWrap* 속성을 사용하여 줄 바꿈 여부를 조정합니다.

메모 및 서식있는 텍스트 컨트롤에는 다음과 같은 다른 속성이 포함되어 있습니다.

- *Alignment*는 컴포넌트에서 텍스트를 정렬하는 방법 (왼쪽, 오른쪽 또는 가운데)을 지정합니다.
- *Text* 속성은 컨트롤에 텍스트를 포함시킵니다. 애플리케이션은 *Modified* 속성을 확인하여 텍스트가 바뀌는지 여부를 알려 줄 수 있습니다.
- *Lines*는 텍스트를 문자열 목록으로 포함합니다.
- *OEMConvert*는 텍스트가 입력될 때 임시로 ANSI에서 OEM으로 변환되는지 여부를 결정합니다. 이것은 파일 이름을 확인하는 데 유용합니다 (VCL 전용).
- *WordWrap*은 텍스트가 오른쪽 여백을 만나면 줄 바꿈할지 여부를 결정합니다.
- *WantReturns*는 사용자가 텍스트에 하드 리턴을 삽입할 수 있는지 여부를 결정합니다.
- *WantTabs*는 사용자가 텍스트에 탭을 삽입할 수 있는지 여부를 결정합니다.
- *AutoSelect*는 컨트롤이 활성화될 때 텍스트가 자동으로 선택되는지 (강조 표시되는지) 여부를 결정합니다.
- *SelText*는 텍스트의 현재 선택된 (강조된) 부분을 포함합니다.
- *SelStart*와 *SelLength*는 텍스트의 선택된 부분의 위치와 길이를 나타냅니다.

런타임에 *SelectAll* 메소드를 사용하여 메모의 모든 텍스트를 선택할 수 있습니다.

서식있는 텍스트 컨트롤(VCL 전용)

서식있는 편집(*TRichEdit*) 컴포넌트는 서식있는 텍스트 형식, 텍스트의 인쇄, 검색, 끌어 놓기 등을 지원하는 메모 컨트롤입니다. 글꼴 속성, 정렬, 탭, 들여쓰기 및 번호 매기기를 지정할 수 있게 해줍니다.

특수한 입력 컨트롤

다음 컴포넌트는 입력을 캡처하는 추가 방법을 제공합니다.

컴포넌트	수행 작업
<i>TScrollBar</i>	연속적인 범위의 값을 선택합니다.
<i>TTrackBar</i>	연속적인 범위의 값을 선택합니다(스크롤 막대보다 시각적으로 더 효과적임).
<i>TUpDown</i>	편집 컴포넌트에 연결된 스피너(spinner)에서 값을 선택합니다(VCL 전용).
<i>THotKey</i>	<i>Ctrl/Shift/Alt</i> 키보드 시퀀스를 입력합니다(VCL 전용).
<i>TSpinEdit</i>	스피너 widget에서 값을 선택합니다(CLX 전용).

스크롤 막대

스크롤 막대 컴포넌트는 창, 폼 또는 다른 컨트롤의 내용을 스크롤하는 데 사용할 수 있는 스크롤 막대를 만듭니다. *OnScroll* 이벤트 핸들러에서 사용자가 스크롤 막대를 움직일 때 컨트롤의 행동을 결정하는 코드를 작성합니다.

많은 비주얼 컴포넌트에는 자체 스크롤 막대가 포함되어 있어 추가 코딩이 필요 없습니다. 따라서 스크롤 막대 컴포넌트를 자주 사용하지 않습니다. 예를 들어, *TForm*에는 자동으로 폼의 스크롤 막대를 만드는 *VertScrollBar*와 *HorzScrollBar* 속성이 있습니다. 폼 내에서 스크롤할 수 있는 영역을 따로 만들려면 *TScrollBox*를 사용하십시오.

트랙 표시줄

트랙 표시줄은 연속적인 범위의 정수 값을 설정할 수 있습니다. 이것은 색상, 부피 및 밝기와 같은 속성을 조정하는 데 유용합니다. 사용자는 슬라이드 지시자를 특정 위치로 끌어 놓거나 표시줄 내부를 클릭해서 슬라이드 지시자를 옮깁니다.

- *Max* 속성과 *Min* 속성을 사용하여 트랙 표시줄의 범위 시작과 끝을 설정합니다.
- *SelEnd*와 *SelStart*를 사용하여 선택 범위를 강조 표시합니다. 그림 3.4를 참조하십시오.
- *Orientation* 속성은 트랙 표시줄이 수직인지 수평인지 결정합니다.
- 기본적으로 트랙 표시줄에는 트랙을 따라 움직이는 틱(tick)이 있습니다. *TickMarks* 속성을 사용하여 틱 위치를 변경합니다. 틱 사이의 간격을 조정하려면 *TickStyle* 속성과 *SetTick* 메소드를 사용합니다.

그림 3.4 트랙 표시줄 컴포넌트의 세 가지 보기

- *Position*은 트랙 표시줄에 대한 기본 위치를 설정하고 런타임 시 위치를 추적합니다.
- 기본적으로 사용자는 위쪽 화살표와 아래쪽 화살표 키를 눌러서 틱을 위나 아래로 움직일 수 있습니다. 틱의 이동 단위를 변경하려면 *LineSize*를 설정합니다.
- *PageSize*를 설정하여 사용자가 *Page Up* 키와 *Page Down* 키를 누를 때 틱의 이동 정도를 결정할 수 있습니다.

Up-down 컨트롤(VCL 전용)

업다운 컨트롤(*TUpDown*)은 사용자가 정수 값을 고정 증분으로 변경할 수 있게 해주는 한 쌍의 화살표 버튼으로 구성됩니다. 현재 값이 *Position* 속성에 의해 주어지고 기본값이 1인 증분은 *Increment* 속성에 의해 지정됩니다. *Associate* 속성을 사용하여 편집 컨트롤 같은 다른 컴포넌트를 업다운 컨트롤에 첨부합니다.

스핀 편집 컨트롤(CLX 전용)

스핀 편집 컨트롤(*TSpinEdit*)은 업다운 widget, 작은 화살표 widget 또는 스핀 버튼이라고도 합니다. 이 컨트롤을 사용하여 위 또는 아래 화살표 버튼을 클릭하여 현재 표시된 값을 증가 또는 감소시키거나 스핀 상자에 직접 값을 입력하여 애플리케이션 사용자가 고정된 증분으로 정수 값을 변경할 수 있습니다.

현재 값은 *Value* 속성에 의해 주어지고 기본값이 1인 증분은 *Increment* 속성에 의해 지정됩니다.

Hot key 컨트롤(VCL 전용)

핫 키 컴포넌트(*THotKey*)를 사용하여 임의의 컨트롤로 포커스를 이동하는 키보드 단축키를 할당합니다. *HotKey* 속성에는 현재 키 조합이 포함되며 *Modifiers* 속성은 *HotKey*에 사용할 수 있는 키를 결정합니다.

핫 키 컴포넌트를 메뉴 항목의 *ShortCut* 속성으로 할당할 수 있습니다. 이렇게 하면 사용자가 *HotKey* 및 *Modifiers* 속성에 의해 지정된 키 조합을 입력할 경우 Windows는 해당 메뉴 항목을 활성화합니다.

스플리터 컨트롤

정렬된 컨트롤 사이에 있는 스플리터(*TSplitter*)를 사용하면 사용자는 컨트롤의 크기를 조정할 수 있습니다. 패널이나 그룹 상자와 같은 컴포넌트와 함께 스플리터를 사용하면 사용자는 각 창에 다양한 컨트롤을 가지는 여러 개의 창으로 폼을 나눌 수 있습니다.

패널이나 다른 컨트롤을 폼에 놓은 후에 컨트롤과 동일한 정렬을 가진 스플리터를 추가합니다. 마지막 컨트롤은 클라이언트 정렬이 되어야 하므로 다른 컨트롤의 크기가 조정되면 나머지 공간을 모두 채웁니다. 예를 들면, 패널을 폼의 왼쪽 가장자리에 둘 수 있고 패널의 *Alignment*를 *alLeft*로 설정한 다음, *alLeft*로 정렬된 스플리터를 패널의 오른쪽에 두고 마지막으로 *alLeft* 또는 *alClient*로 정렬된 다른 패널을 스플리터의 오른쪽에 둡니다.

*MinSize*를 설정하여 이웃하는 컨트롤의 크기를 조정할 때 스플리터가 남겨 두어야 하는 최소 크기를 지정합니다. *Beveled*를 *True*로 설정하여 스플리터의 가장자리를 3차원 모양으로 만들 수 있습니다.

버튼 유형의 컨트롤

메뉴 이외에 버튼도 애플리케이션에서 명령을 호출하는 가장 일반적인 방법을 제공합니다. Delphi는 다음과 같은 몇 가지 버튼 유형의 컨트롤을 제공합니다.

컴포넌트	수행 작업
<i>TButton</i>	텍스트가 있는 버튼에서 명령 선택을 나타냅니다.
<i>TBitBtn</i>	텍스트 및 glyph가 있는 버튼에서 명령 선택을 나타냅니다.
<i>TSpeedButton</i>	그룹화된 툴바 버튼을 만듭니다.
<i>TCheckBox</i>	선택/선택 해제 옵션을 나타냅니다.
<i>TRadioButton</i>	상호 배타적인 선택 집합을 나타냅니다.
<i>TToolBar</i>	행에서 툴 버튼 및 기타 컨트롤을 정렬하고 크기와 위치를 자동으로 조정합니다.
<i>TCoolBar</i>	이동 및 크기 조정이 가능한 밴드 내에서 창이 있는 컨트롤의 모음을 표시합니다(VCL 전용).

버튼 컨트롤

마우스로 버튼 컨트롤을 클릭하여 동작을 시작합니다. 버튼에는 동작을 나타내는 텍스트 레이블이 있습니다. *Caption* 속성에 문자열 값을 할당하여 텍스트를 지정합니다. 또한 단축키를 눌러 대부분의 버튼을 선택할 수 있습니다. 단축키는 버튼에서 밑줄 친 문자로 나타냅니다.

사용자는 버튼 컨트롤을 클릭하여 동작을 시작합니다. *OnClick* 이벤트 핸들러를 만들어 *TButton* 컴포넌트에 동작을 할당할 수 있습니다. 디자인 타임에 버튼을 더블 클릭하여 코드 에디터에 있는 버튼의 *OnClick* 이벤트 핸들러로 이동할 수 있습니다.

- 사용자가 *Esc* 키를 누를 때 버튼의 *OnClick* 이벤트를 트리거하게 하려면 *Cancel*을 *True*로 설정합니다.
- *Enter* 키를 누르면 버튼의 *OnClick* 이벤트를 트리거하도록 하려면 *Default*를 *True*로 설정합니다.

비트맵 버튼

비트맵 버튼(*BitBtn*)은 버튼 위에 비트맵 이미지를 나타내는 버튼 컨트롤입니다.

- 버튼에 표시할 비트맵을 선택하려면 *Glyph* 속성을 설정합니다.
- *Kind*를 사용하여 *glyph*와 기본 행동을 가지는 버튼을 자동으로 구성합니다.
- 기본적으로 *glyph*는 텍스트의 왼쪽에 나타납니다. 이동하려면 *Layout* 속성을 사용합니다.
- *glyph*와 텍스트는 자동으로 버튼의 가운데에 표시됩니다. 표시 위치를 이동하려면 *Margin* 속성을 사용합니다. *Margin*은 이미지의 가장자리와 버튼의 가장자리 사이의 픽셀의 수를 결정합니다.
- 기본적으로 이미지와 텍스트는 4개의 픽셀로 분리됩니다. *Spacing*을 사용하여 간격을 늘리거나 줄입니다.
- 비트맵 버튼에는 up, down 및 held down이라는 3가지 상태가 있습니다. *NumGlyphs* 속성을 3으로 설정하면 각 상태에 대해 각각 다른 비트맵을 보여 줄 수 있습니다.

스피드 버튼

스피드 버튼은 대개 버튼 위에 이미지를 갖고 그룹으로 묶어서 기능할 수 있습니다. 일반적으로 스피드 버튼을 패널과 함께 사용하여 툴바를 만듭니다.

- 스피드 버튼을 그룹으로 작동하게 하려면 모든 버튼의 *GroupIndex* 속성에 0이 아닌 동일한 값을 할당합니다.
- 기본적으로 스피드 버튼은 버튼을 선택하지 않은 상태인 up 상태로 나타납니다. 스피드 버튼이 선택된 상태를 초기 값으로 지정하려면 *Down* 속성을 *True*로 설정합니다.
- *AllowAllUp*이 *True*인 경우에 그룹의 모든 스피드 버튼은 선택되지 않은 상태가 될 수 있습니다. 버튼 그룹을 라디오 그룹처럼 동작하게 하려면 *AllowAllUp*을 *False*로 설정합니다.

스피드 버튼에 대한 자세한 내용은 6-44 페이지의 "패널 컴포넌트를 사용하여 툴바 추가" 단원을 참조하십시오.

체크 박스(Check box)

체크 박스는 사용자가 선택 또는 선택 해제를 선택할 수 있는 토글입니다. 선택하면 체크 박스에 체크 표시가 나타납니다. 선택하지 않으면 체크 박스는 빈 상태가 됩니다. *TCheckBox*를 사용하여 체크 박스를 만듭니다.

- 기본적으로 체크 박스를 선택 표시되게 하려면 *Checked*를 *True*로 설정합니다.
- *AllowGrayed*를 *True*로 설정하여 체크 박스에 선택, 선택 해제 및 비활성(*grayed*)이라는 세 가지의 상태를 부여합니다.
- *State* 속성은 체크 박스가 선택되었는지(*cbChecked*), 선택되지 않았는지(*cbUnchecked*) 또는 비활성인지(*cbGrayed*) 나타냅니다.

참고 체크 박스 컨트롤은 두 가지 바이너리 상태 중 하나를 표시합니다. 다른 설정으로 체크 박스의 현재 값을 결정할 수 없는 경우 미결정 상태가 사용됩니다.

라디오 버튼

라디오 버튼은 상호 배타적인 선택 집합을 나타냅니다. *TRadioButton*을 사용하여 라디오 버튼을 개별적으로 만들거나 *라디오 그룹* 컴포넌트(*TRadioGroup*)를 사용하여 자동으로 라디오 버튼을 그룹으로 정렬합니다. 라디오 버튼을 그룹화하여 사용자는 제한된 선택 집합에서 버튼을 하나 선택할 수 있습니다. 자세한 내용은 3-40 페이지의 "컴포넌트 그룹화"를 참조하십시오.

선택된 라디오 버튼은 가운데에 점이 있는 원으로 나타납니다. 선택되지 않은 경우 라디오 버튼은 빈 원으로 나타납니다. `Checked` 속성에 `True` 또는 `False` 값을 지정하여 라디오 버튼의 비주얼 상태를 변경합니다.

툴바

툴바는 비주얼 컨트롤을 정렬하고 관리하는 쉬운 방법을 제공합니다. 패널 컴포넌트와 스피드 버튼으로 툴바를 만들거나 *ToolBar* 컴포넌트를 사용한 다음 마우스 오른쪽 버튼을 클릭하여 `New Button`을 선택하면 버튼을 툴바에 추가할 수 있습니다.

TToolBar 컴포넌트는 여러 가지 이점을 갖습니다. 다른 컨트롤은 상대적인 위치와 높이를 유지하는 반면 툴바에 있는 버튼은 자동으로 일정한 크기와 간격을 유지하고, 컨트롤들이 수평 정렬되지 않으면 자동으로 래퍼라운드되어 새 줄에서 시작하며, *TToolBar*는 투명도, 팝업 테두리, 공간과 스플리터 등 표시 옵션을 제공하여 컨트롤을 그룹화합니다.

액션 리스트 또는 *액션 밴드*를 사용하여 툴바 및 메뉴에서 중앙 집중화된 동작 집합을 사용할 수 있습니다. 액션 리스트를 버튼 및 툴바와 함께 사용하는 방법에 대한 자세한 내용은 6-23 페이지의 "액션 리스트 사용"을 참조하십시오.

툴바는 편집 상자, 콤보 박스 등과 같은 다른 컨트롤의 부모가 될 수 있습니다.

쿨바(VCL 전용)

쿨바에는 독립적으로 이동 및 크기 조정할 수 있는 자식 컨트롤이 포함되어 있습니다. 각 컨트롤은 개별 밴드에 있습니다. 사용자는 크기 조정 손잡이를 각 밴드의 왼쪽으로 끌어 컨트롤 위치를 지정합니다.

쿨바를 사용하려면 디자인 타임과 런타임 시 버전 4.70 이상의 `COMCTL32.DLL`(일반적으로 `Windows\System` 또는 `Windows\System32` 디렉토리에 위치)이 필요합니다. 크로스 플랫폼 애플리케이션에서는 쿨바를 사용할 수 없습니다.

- *Bands* 속성은 *TCoolBand* 객체의 모음을 유지합니다. 디자인 타임 시 밴드 편집기를 사용하여 밴드를 추가, 제거 또는 수정할 수 있습니다. 밴드 편집기를 열려면 `Object Inspector`에서 *Bands* 속성을 선택한 다음 오른쪽의 `Value` 열을 더블 클릭하거나 `생략(...)` 버튼을 클릭합니다. 팔레트에서 창이 있는 새 컨트롤을 추가하여 밴드를 만들 수도 있습니다.
- *FixedOrder* 속성은 사용자가 밴드를 재정렬할 수 있는지 여부를 결정합니다.
- *FixedSize* 속성은 밴드가 일정한 높이를 유지하는지 여부를 결정합니다.

목록 처리

목록은 선택할 항목들을 사용자에게 보여 줍니다. 목록을 표시하는 일부 컴포넌트는 다음과 같습니다.

컴포넌트	표시 내용
<i>TListBox</i>	텍스트 문자열의 목록
<i>TCheckBoxList</i>	각 항목 앞에 체크 박스가 있는 목록
<i>TComboBox</i>	스크롤할 수 있는 드롭다운 목록을 갖는 편집 상자
<i>TTreeView</i>	계층 구조 목록
<i>TListView</i>	아이콘, 열 및 헤더가 있는 드래그 가능한 항목의 목록
<i>TDateTimePicker</i>	날짜 또는 시간 입력을 위한 리스트 박스(VCL 전용)
<i>TMonthCalendar</i>	날짜를 선택하기 위한 달력(VCL 전용)

년비주얼(nonvisual) *TStringList*와 *TImageList* 컴포넌트를 사용하여 문자열과 이미지 집합을 관리합니다. 문자열 목록에 대한 자세한 내용은 3-49 페이지의 "문자열 목록 사용"을 참조하십시오.

리스트 박스(List box)와 체크 리스트 박스(Check-list box)

리스트 박스(*TListBox*) 및 체크 리스트 박스는 사용자가 항목을 선택할 수 있는 목록을 표시합니다.

- *Items*는 *TStrings*를 사용하여 컨트롤을 값으로 채웁니다.
- *ItemIndex*는 목록에서 선택된 항목을 나타냅니다.
- *MultiSelect*는 한 번에 둘 이상의 항목을 선택할 수 있는지 여부를 지정합니다.
- *Sorted*는 목록이 알파벳순으로 정렬되는지 결정합니다.
- *Columns*는 리스트 컨트롤에 있는 열의 수를 지정합니다.
- *IntegralHeight*는 리스트 박스가 세로 간격에 완전하게 맞는 항목만 표시하는지 여부를 지정합니다(VCL 전용).
- *ItemHeight*는 각 항목의 높이를 픽셀 단위로 지정합니다. *Style* 속성은 *ItemHeight*를 무시하도록 할 수 있습니다.
- *Style* 속성은 리스트 컨트롤이 항목을 표시하는 방법을 결정합니다. 기본적으로 항목은 문자열로 표시됩니다. *Style* 값을 변경하여 항목을 그래픽이나 변하는 높이로 표시하는 *owner-draw* 리스트 박스를 만들 수 있습니다. *owner-draw* 컨트롤에 대한 내용은 7-12 페이지의 "컨트롤에 그래픽 추가"를 참조하십시오.

다음과 같은 방법으로 간단한 리스트 박스를 만듭니다.

- 1 프로젝트 내에서 컴포넌트 팔레트의 리스트 박스 컴포넌트를 폼에 가져다 놓습니다.
- 2 리스트 박스의 크기를 정하고 필요하면 리스트 박스를 정렬합니다.
- 3 *Items* 속성의 오른쪽을 더블 클릭하거나 생략 버튼을 선택하여 String List Editor를 표시합니다.
- 4 편집기를 사용하여 리스트 박스의 내용에 대해 줄로 정렬된 프리 폼 텍스트를 입력합니다.

5 OK를 선택합니다.

리스트 박스에서 여러 항목을 선택하기 위해 *ExtendedSelect* 속성과 *MultiSelect* 속성을 사용할 수 있습니다.

콤보 박스(Combo box)

콤보 박스 (*TComboBox*)는 편집 상자를 스크롤 가능한 목록과 결합합니다. 입력하거나 목록에서 선택하여 데이터를 컨트롤에 입력하면 *Text* 속성 값이 변경됩니다. *AutoComplete*를 사용 가능으로 하면 사용자가 데이터를 입력할 때 애플리케이션은 목록에서 가장 가깝게 일치하는 것을 찾아서 표시합니다.

콤보 박스의 세 가지 종류에는 표준, 드롭다운(기본값) 및 드롭다운 목록이 있습니다.

- *Style* 속성을 사용하여 필요한 콤보 박스의 종류를 선택합니다.
- 드롭다운 목록이 있는 편집 상자를 원할 경우 *csDropDown*을 사용합니다. *csDropDownList*를 사용하여 편집 상자를 읽기 전용(사용자가 목록에서 선택하도록 함)으로 만듭니다. *DropDownCount* 속성을 설정하여 목록에 표시된 항목의 수를 변경합니다.
- *csSimple*을 사용하여 닫히지 않은 고정된 목록을 가진 표준 콤보 박스를 만듭니다. 콤보 박스의 크기를 조정하여 목록 항목이 표시되는지 확인하십시오.
- *csOwnerDrawFixed* 또는 *csOwnerDrawVariable*을 사용하여 항목을 그래픽하게, 또는 변하는 높이로 항목을 표시하는 *owner-draw* 콤보 박스를 만듭니다. *owner-draw* 컨트롤에 대한 내용은 7-12 페이지의 "컨트롤에 그래픽 추가"를 참조하십시오.

런타임 시 CLX 콤보 박스는 VCL 콤보 박스와 다르게 작동합니다. VCL 콤보 박스와 달리 CLX 콤보 박스에서는 콤보 박스의 편집 필드에서 텍스트를 입력하고 Enter 키를 눌러 드롭다운 목록에 항목을 추가할 수 있습니다. *InsertMode*를 *ciNone*으로 설정하여 이 기능을 끌 수 있습니다. 콤보 박스의 목록에 빈(문자열이 아닌) 항목을 추가할 수도 있습니다. 또한 아래쪽 화살표를 계속 누르면 콤보 박스 목록의 마지막 항목에서 계속 스크롤됩니다. 위에서부터 다시 반복됩니다.

트리 뷰(Tree View)

트리 뷰 (*TTreeView*)는 들여쓰기로 항목을 표시합니다. 컨트롤에는 노드를 확장하거나 수축시키는 버튼이 있습니다. 사용자는 항목의 텍스트 레이블이 있는 아이콘을 포함할 수 있고 노드가 확장되거나 축소되는지에 따라 각각 다른 아이콘을 표시할 수 있습니다. 항목에 대한 상태 정보를 반영하는 체크 박스와 같은 그래픽을 포함할 수도 있습니다.

- *Indent*는 항목을 부모로부터 수평으로 분리하는 픽셀 수를 설정합니다.
- *ShowButtons*는 항목 확장 여부를 나타내는 '+' 및 '-' 버튼을 표시합니다.
- *ShowLines*는 계층 관계를 나타내는 연결선을 표시합니다(VCL 전용).
- *ShowRoot*는 상위 레벨 항목에 연결된 선의 표시 여부를 결정합니다(VCL 전용).

디자인 타임에 항목을 트리 뷰 컨트롤에 추가하려면 컨트롤을 더블 클릭하여 *TreeView Items* 편집기를 표시합니다. 추가한 항목은 *Items* 속성 값이 됩니다. *TTreeNode*스 타입의 객체인 *Items* 속성의 메소드를 사용하여 런타임 시 항목을 변경할 수 있습니다. *TTreeNode*스는 트리 뷰의 항목을 추가, 삭제 및 탐색하기 위한 메소드를 제공합니다.

트리 뷰는 vsReport 모드에 있는 리스트 뷰와 유사한 열과 하위 항목을 표시할 수 있습니다.

리스트 뷰(List View)

*TListView*를 사용하여 만든 리스트 뷰는 다양한 형식으로 목록을 표시합니다. 다음과 같이 *ViewStyle* 속성을 사용하여 원하는 목록의 종류를 선택합니다.

- *vsIcon* 및 *vsSmallIcon*은 각 항목을 레이블이 있는 아이콘으로 표시합니다. 사용자는 리스트 뷰 창 안에서 항목을 끌 수 있습니다(VCL 전용).
- *vsList*는 끌어 놓을 수 없는, 레이블이 있는 아이콘으로 항목을 표시합니다.
- *vsReport*는 열로 정렬된 정보를 가진 별도의 행에 항목을 표시합니다. 가장 왼쪽에 있는 열에는 작은 아이콘과 레이블이 있고, 다음 열에는 애플리케이션에 의해 지정된 하위 항목이 있습니다. *ShowColumnHeaders* 속성을 사용하여 열에 대한 헤더를 표시합니다.

날짜/시간 선택(Data-time picker)과 달력(Month calendar)(VCL 전용)

DateTimePicker 컴포넌트는 날짜나 시간 입력을 위한 리스트 박스를 표시하고, *MonthCalendar* 컴포넌트는 날짜나 날짜 범위 입력을 위한 달력을 제공합니다. 이러한 컴포넌트를 사용하려면 디자인 타임과 런타임 시 버전 4.70 이상의 COMCTL32.DLL (일반적으로 Windows\System 또는 Windows\System32 디렉토리에 위치)이 필요합니다. 크로스 플랫폼 애플리케이션에서는 이러한 컴포넌트를 사용할 수 없습니다.

컴포넌트 그룹화

그래픽 인터페이스는 관련 컨트롤과 정보를 그룹화했을 때 사용하기가 쉽습니다. Delphi는 컴포넌트를 그룹화하기 위한 다음과 같은 여러 컴포넌트를 제공합니다.

컴포넌트	수행 작업
<i>TGroupBox</i>	제목을 갖는 표준 그룹 상자
<i>TRadioGroup</i>	라디오 버튼의 단순 그룹
<i>TPanel</i>	시각적으로 더 유연한 컨트롤 그룹
<i>TScrollBar</i>	컨트롤을 포함하는 스크롤 가능한 영역
<i>TTabControl</i>	상호 배타적인 노트북 스타일 탭 집합
<i>TPageControl</i>	탭마다 페이지를 갖는 상호 배타적인 노트북 스타일 탭의 집합(각 페이지는 다른 컨트롤을 포함)
<i>THeaderControl</i>	크기 조정 가능한 열 헤더

그룹 상자와 라디오 그룹

그룹 상자 (*TGroupBox*)는 폼에서 관련 있는 컨트롤을 정렬합니다. 그룹화된 컨트롤 중에 가장 일반적인 것은 라디오 버튼입니다. 그룹 상자를 폼에 둔 후에 컴포넌트 팔레트에서 컴포넌트를 선택하여 그룹 상자에 둡니다. *Caption* 속성은 런타임에 그룹 상자에 레이블을 표시하는 텍스트를 포함합니다.

라디오 그룹 컴포넌트 (*TRadioGroup*)는 라디오 버튼을 조합하여 함께 작동하도록 만드는 작업을 간편하게 합니다. 라디오 버튼을 라디오 그룹에 추가하려면 Object Inspector에서 *Items* 속성을 편집합니다. 즉, *Items*에 있는 각 문자열은 캡션 문자열이 있는 그룹 상자에 라디오 버튼을 나타나게 합니다. *ItemIndex* 속성 값은 현재 선택된 라디오 버튼을 결정합니다. *Columns* 속성 값을 설정하여 단일 열 또는 여러 열에서 라디오 버튼을 표시합니다. 버튼을 재배치하려면 라디오 그룹 컴포넌트의 크기를 조정합니다.

패널

TPanel 컴포넌트는 다른 컨트롤에 대한 일반적인 컨테이너를 제공합니다. 패널은 보통 폼에서 컴포넌트를 시각적으로 그룹화하는 데 사용됩니다. 패널을 폼과 함께 정렬하여 폼의 크기를 조정할 때 동일한 상대적인 위치를 유지할 수 있습니다. *BorderWidth* 속성은 패널 주위의 테두리 너비를 픽셀 단위로 지정합니다.

또한 다른 컨트롤을 패널에 놓고 *Align* 속성을 사용하여 폼에 있는 그룹의 모든 컨트롤의 적절한 위치를 확인합니다. 폼의 크기가 변경되더라도 패널의 위치를 제자리에 두기 위해 패널을 *alTop* 정렬할 수 있습니다.

BevelOuter 속성과 *BevelInner* 속성을 사용하여 패널의 모습을 올라가거나 내려간 모습으로 변경할 수 있습니다. 이러한 속성 값들을 변경하여 다양한 비주얼 3차원 효과를 낼 수 있습니다. 단순히 올라가거나 내려간 빗면을 원한다면 대신 리소스를 덜 사용하는 *TBevel* 컨트롤을 사용하면 됩니다.

또한 하나 이상의 패널을 사용하여 다양한 상태 표시줄 또는 정보 표시 영역을 만들 수 있습니다.

스크롤 상자

스크롤 상자 (*TScrollBar*)는 폼 내에 스크롤 영역을 만듭니다. 경우에 따라 애플리케이션은 지정된 영역에 비해 더 많은 정보를 표시해야 합니다. 리스트 박스, 메모 및 폼 자체 등 일부 컨트롤은 자동으로 내용을 스크롤할 수 있습니다.

스크롤 상자의 다른 용도는 창에서 다양한 스크롤 영역(뷰)을 만드는 것입니다. 뷰는 상용 워드 프로세서, 스프레드시트 및 프로젝트 관리 애플리케이션에서 일반적입니다. 스크롤 상자는 폼의 임의 스크롤 하위 영역을 정의할 수 있는 추가 유연성을 제공합니다.

패널이나 그룹 상자와 마찬가지로 스크롤 상자는 *TButton* 객체 및 *TCheckBox* 객체와 같은 다른 컨트롤을 포함합니다. 그러나 스크롤 상자는 일반적으로 보이지 않습니다. 스크롤 상자에 있는 컨트롤이 시각적인 영역보다 크면 스크롤 상자는 스크롤 막대를 자동으로 표시합니다.

스크롤 상자의 다른 용도는 툴바 또는 상태 표시줄 (*TPanel* 컴포넌트) 과 같은 곳에서 스크롤 영역을 제한하는 것입니다. 툴바와 상태 표시줄이 스크롤되는 것을 방지하려면 스크롤 막대를 숨긴 다음 스크롤 상자를 툴바와 상태 표시줄 사이에 있는 창의 클라이언트 영역에 둡니다. 스크롤 상자에 연결된 스크롤 막대는 창에 속하는 것처럼 보이지만 스크롤 상자 내의 영역만 스크롤합니다.

탭 컨트롤

탭 컨트롤 컴포넌트 (*TTabControl*) 는 노트북 분할기 (divider) 처럼 보이는 탭 집합을 만듭니다. Object Inspector에서 *Tabs* 속성을 편집하여 탭을 만들 수 있는데 *Tabs*에 있는 각 문자열은 탭을 나타냅니다. 탭 컨트롤은 컴포넌트 집합 하나를 가지는 단일 패널입니다. 탭을 클릭할 때 컨트롤 모양을 변경하려면 *OnChange* 이벤트 핸들러를 작성해야 합니다. 멀티페이지 대화 상자를 만들려면 탭 컨트롤 대신 페이지 컨트롤을 사용합니다.

페이지 컨트롤

페이지 컨트롤 컴포넌트 (*TPageControl*) 는 멀티페이지 대화 상자에 적합한 페이지 집합입니다. 페이지 컨트롤은 *TTabSheet* 객체인 여러 개가 오버랩된 페이지를 표시합니다. 컨트롤의 상단에 있는 탭을 클릭하여 사용자 인터페이스에서 페이지를 선택합니다.

디자인 타임에 페이지 컨트롤에서 새 페이지를 만들려면 컨트롤을 마우스 오른쪽 버튼으로 클릭하고 *New Page* 를 선택합니다. 페이지에 대한 객체를 만들고 객체의 *PageControl* 속성을 설정하여 런타임에 새 페이지를 추가합니다.

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

활성 페이지에 액세스하려면 *ActivePage* 속성을 사용합니다. 활성 페이지를 변경하기 위해 *ActivePage* 속성이나 *ActivePageIndex* 속성을 설정할 수 있습니다.

헤더 컨트롤

헤더 컨트롤 (*THeaderControl*) 은 런타임에 선택하거나 크기 조정할 수 있는 열 헤더 집합입니다. 컨트롤의 *Sections* 속성을 편집하여 헤더를 추가하거나 수정합니다. 헤더 섹션은 열 또는 필드 위에 둘 수 있습니다. 예를 들어, 헤더 섹션은 리스트 박스 (*TListBox*) 위에 둘 수 있습니다.

비주얼 피드백 제공

애플리케이션의 상태 정보를 제공하는 방법은 여러 가지가 있습니다. 예를 들어, *TForm*을 비롯한 일부 컴포넌트는 런타임 시 설정할 수 있는 *Caption* 속성을 갖습니다. 대화 상자를 만들어 메시지를 표시할 수도 있습니다. 그 밖에 다음과 같은 컴포넌트가 런타임에 비주얼 피드백을 제공하는 데 특히 유용합니다.

컴포넌트 또는 속성	수행 작업
<i>TLabel</i> 및 <i>TStaticText</i>	편집할 수 없는 텍스트를 표시합니다.
<i>TStatusBar</i>	일반적으로 창 하단에 상태 영역을 표시합니다.
<i>TProgressBar</i>	특정 작업에 대해 완료한 작업량을 보여 줍니다.
<i>Hint</i> 와 <i>ShowHint</i>	플라이바이 (fly-by) 또는 "툴팁" 도움말을 활성화합니다.
<i>HelpContext</i> 와 <i>HelpFile</i>	문맥에 따른 온라인 도움말을 연결합니다.

레이블 및 정적 텍스트 컴포넌트

레이블 (*TLabel*)은 텍스트를 표시하며 일반적으로 다른 컨트롤 옆에 놓입니다. 편집 상자과 같은 다른 컴포넌트를 식별하거나 표시하거나 또는 텍스트를 폼에 포함하고자 할 때 폼에 레이블을 놓습니다. 표준 레이블 컴포넌트인 *TLabel*은 창이 있는 컨트롤(또는 CLX의 경우 widget 기반 컨트롤)이 아니기 때문에 포커스를 받을 수 없습니다. 창 핸들이 있는 레이블이 필요한 경우에는 대신 *TStaticText*를 사용합니다.

Label 속성은 다음과 같습니다.

- *Caption*은 레이블의 텍스트 문자열을 포함합니다.
- *Font*, *Color* 및 다른 속성은 레이블의 모습을 결정합니다. 각 레이블은 글꼴, 크기 및 색상을 하나만 사용할 수 있습니다.
- *FocusControl*은 레이블을 폼에 있는 다른 컨트롤에 연결합니다. *Caption*에 가속키가 있으면 *FocusControl*에 의해 지정된 컨트롤은 가속키를 누를 때 포커스를 받습니다.
- *ShowAccelChar*는 레이블이 밑줄이 그어진 가속키 문자를 표시할 수 있는지 여부를 결정합니다. *ShowAccelChar*가 *True*이면 앞에 앰퍼샌드 (&)가 있는 모든 문자는 밑줄이 그어지고 가속키를 활성화합니다.
- *Transparent*는 그래픽과 같은 레이블 아래의 항목이 보이는지 여부를 결정합니다.

레이블은 보통 애플리케이션 사용자가 변경할 수 없는 읽기 전용 정적 텍스트를 표시합니다. *Caption* 속성에 새 값을 지정하여 애플리케이션은 실행 도중에 텍스트를 변경할 수 있습니다. 사용자가 스크롤하거나 편집할 수 있는 폼에 텍스트를 추가하려면 *TEdit*를 사용합니다.

상태 표시줄(Status bar)

패널을 사용하여 상태 표시줄을 만들 수 있지만 상태 표시줄 컴포넌트를 사용하는 것이 더 간단합니다. 기본적으로 상태 표시줄의 *Align* 속성은 위치와 크기를 모두 관리하는 *alBottom*으로 설정됩니다.

상태 표시줄에서 한 번에 하나의 텍스트 문자열만 표시하려면 *SimplePanel* 속성을 *True*로 설정하고 *SimpleText* 속성을 사용하여 상태 표시줄에서 표시된 텍스트를 조정할 수 있습니다.

상태 표시줄을 패널이라는 여러 개의 텍스트 영역으로 분리할 수 있습니다. 패널을 만들려면 Object Inspector에서 *Panels* 속성을 편집하여 Panels 편집기에서 각 패널의 *Width*, *Alignment* 및 *Text* 속성을 설정합니다. 각 패널의 *Text* 속성에는 패널에 표시된 텍스트가 포함됩니다.

진행 표시줄(Progress bar)

애플리케이션이 시간이 많이 소요되는 작업을 수행할 때 진행 표시줄을 사용하여 작업이 얼마나 진행되었는지 표시할 수 있습니다. 진행 표시줄은 왼쪽에서 오른쪽으로 증가하는 점선을 표시합니다.

그림 3.5 진행 표시줄



Position 속성은 점선의 길이를 지정합니다. *Max*와 *Min*은 *Position*의 범위를 결정합니다. 선을 길게 늘리려면 *StepBy* 또는 *StepIt* 메소드를 호출하여 *Position*을 증가시킵니다. *Step* 속성은 *StepIt*이 사용하는 증가 단위를 결정합니다.

도움말과 힌트 속성

대부분의 비주얼 컨트롤은 런타임 시 플라이바이(fly-by) 도움말뿐만 아니라 문맥에 따른 도움말도 표시할 수 있습니다. *HelpContext* 속성과 *HelpFile* 속성은 컨트롤에 대한 도움말 컨텍스트 번호와 도움말 파일을 만듭니다.

Hint 속성은 마우스 포인터를 컨트롤 또는 메뉴 항목 위로 옮길 때 나타나는 텍스트 문자열을 포함합니다. 힌트를 사용하기 위해 *ShowHint*를 *True*로 설정합니다. *ParentShowHint*를 *True*로 설정하면 컨트롤의 *ShowHint* 속성이 부모와 동일한 값을 가지게 됩니다.

그리드

그리드는 정보를 행과 열로 표시합니다. 데이터베이스 애플리케이션을 작성하는 경우, 15장 "데이터 컨트롤 사용"에 설명된 *TDBGrid* 또는 *TDBCtrGrid* 컴포넌트를 사용합니다. 그외의 경우에는 표준 그리기 그리드나 문자열 그리드를 사용합니다.

그리기 그리드

그리기 그리드(*TDrawGrid*)는 임의의 데이터를 표 형식으로 표시합니다. *OnDrawCell* 이벤트 핸들러를 작성하여 그리드의 셀을 채웁니다.

- *CellRect* 메소드는 지정된 셀의 화면 좌표를 반환하지만 *MouseToCell* 메소드는 지정된 화면 좌표에 있는 셀의 열과 행을 반환합니다. *Selection* 속성은 현재 선택한 셀의 테두리를 나타냅니다.

- *TopRow* 속성은 현재 그리드의 상단에 있는 행을 결정합니다. *LeftCol* 속성은 왼쪽에서 첫 번째 비주열 열을 결정합니다. *VisibleColCount*와 *VisibleRowCount*는 그리드에서 보이는 열과 행의 수입니다.
- *ColWidths* 속성과 *RowHeights* 속성을 사용하여 열이나 행의 너비 또는 높이를 변경할 수 있습니다. *GridLineWidth* 속성을 사용하여 그리드 선의 너비를 설정합니다. *ScrollBars* 속성을 사용하여 스크롤 막대를 그리드에 추가합니다.
- *FixedCols* 속성과 *FixedRows* 속성을 사용하여 고정되거나 스크롤이 없는 열과 행을 갖도록 선택할 수 있습니다. *FixedColor* 속성을 사용하여 고정된 열과 행에 색상을 지정합니다.
- *Options*, *DefaultColWidth* 및 *DefaultRowHeight* 속성은 그리드의 모습과 행동에 영향을 미칠 수 있습니다.

문자열 그리드

문자열 그리드 컴포넌트는 특수한 기능을 추가하여 문자열의 표시를 단순화하는 *TDrawGrid*의 자손입니다. *Cells* 속성은 그리드의 각 셀에 대한 문자열을 나열하고 *Objects* 객체는 각 문자열에 연결된 객체를 나열합니다. 모든 문자열과 특정한 열이나 행에 관련된 객체는 *Cols* 속성 또는 *Rows* 속성을 통해 액세스할 수 있습니다.

값 목록 편집기(VCL 전용)

*TValueListEditor*는 Name=Value 형태로 이름/값 쌍을 포함하는 문자열 목록을 편집하기 위한 특수 그리드입니다. 이름과 값은 *Strings* 속성 값인 *TStrings* 자손으로 저장됩니다. *Values* 속성을 사용하면 모든 이름의 값을 조회할 수 있습니다. 크로스 플랫폼 프로그래밍에서는 *TValueListEditor*를 사용할 수 없습니다.

이 그리드에는 이름과 값을 위한 열이 각각 하나씩 포함되어 있습니다. 기본적으로 Name 열의 이름은 "Key"이고 Value 열의 이름은 "Value"입니다. *TitleCaptions* 속성을 사용하면 이러한 기본 이름을 바꿀 수 있습니다. 또한 컨트롤 크기를 조정할 때 컨트롤 크기 조정을 제어하는 역할도 수행하는 *DisplayOptions* 속성을 사용하면 이러한 열 이름을 생략할 수 있습니다.

KeyOptions 속성을 사용하면 사용자가 Name 열을 편집할 수 있는지 여부를 제어할 수 있습니다. *KeyOptions*에는 이름을 편집 또는 삭제하고, 새 이름을 추가하고, 새 이름이 고유해야 하는지 여부를 제어할 수 있도록 허용하는 별도의 옵션이 포함되어 있습니다.

ItemProps 속성을 사용하면 사용자가 Value 열에서 항목을 편집하는 방법을 제어할 수 있습니다. 각 항목은 다음 작업을 수행할 수 있게 해주는 별도의 *TItemProp* 객체를 가집니다.

- 편집 마스크를 제공하여 유효 입력을 제한합니다.
- 최대값 길이를 지정합니다.
- 값을 읽기 전용으로 표시합니다.

- 값 목록 편집기를 지정하여 사용자가 값을 선택할 수 있도록 선택 목록을 여는 드롭다운 화살표를 표시하거나 사용자가 값을 입력할 수 있도록 대화 상자를 표시하는 데 사용할 수 있는 이벤트를 실행하는 생략 버튼을 표시합니다.

드롭다운 화살표를 표시하도록 지정할 경우, 사용자가 값을 선택할 수 있도록 목록을 제공해야 합니다. 이러한 값 목록은 정적 목록(*TItemProp* 객체의 *PickList* 속성)으로 만들거나 값 목록 편집기의 *OnGetPickList* 이벤트를 사용하여 런타임 시 동적으로 추가할 수 있습니다. 또한 이러한 방법을 결합하여 *OnGetPickList* 이벤트 핸들러가 수정하는 정적 목록을 만들 수도 있습니다.

생략 버튼을 표시하도록 지정할 경우에는 사용자가 생략 버튼을 클릭했을 때 발생하는 응답(해당 사항이 있는 경우 값 설정 포함)을 제공해야 합니다. 이러한 응답은 *OnEditButtonClick* 이벤트 핸들러를 작성하여 제공합니다.

그래픽 표시

다음 컴포넌트를 사용하면 그래픽을 애플리케이션에 쉽게 통합할 수 있습니다.

컴포넌트	표시 내용
<i>TImage</i>	그래픽 파일
<i>TShape</i>	기하학적 도형
<i>TBevel</i>	3차원 선과 프레임
<i>TPaintBox</i>	런타임에 프로그램이 그리는 그래픽
<i>TAnimate</i>	AVI 파일(VCL 전용)

이미지

이미지 컴포넌트는 비트맵, 아이콘 또는 메타파일과 같은 그래픽 이미지를 표시합니다. *Picture* 속성은 표시할 그래픽을 결정합니다. *Center*, *AutoSize*, *Stretch*, *Transparent*를 사용하여 표시 옵션을 설정합니다. 자세한 내용은 8-1 페이지의 "그래픽 프로그래밍 개요"를 참조하십시오.

도형(Shape)

도형 컴포넌트는 기하학적 도형을 표시합니다. 이 컴포넌트는 창이 있는 컨트롤(또는 CLX의 경우 widget 기반 컨트롤)이 아니기 때문에 사용자 입력을 받을 수 없습니다. *Shape* 속성은 컨트롤이 가정하는 도형을 결정합니다. 도형의 색상을 변경하거나 패턴을 추가하려면 *TBrush* 객체의 *Brush* 속성을 사용합니다. 도형을 그리는 방법은 *TBrush*의 *Color* 속성과 *Style* 속성에 따라 다릅니다.

빗면(Bevel)

빗면 컴포넌트(*TBevel*)는 음각과 양각 모습으로 나타날 수 있는 선입니다. *TPanel*과 같은 일부 컴포넌트는 빗면이 있는 액자 모양의 테두리를 만들 수 있는 기본 제공 속성을 가지고 있습니다. 기본 제공 속성을 사용할 수 없을 때 *TBevel*를 사용하여 액자 모양의 윤곽, 상자 또는 프레임을 만듭니다.

그리기 상자

그리기 상자 (*TPaintBox*)를 사용하여 애플리케이션이 폼 위에 그릴 수 있습니다. *OnPaint* 이벤트 핸들러를 작성하여 이미지를 그리기 상자의 *Canvas*에서 직접 렌더링할 수 있습니다. 그리기 상자의 테두리 밖에서 그리는 것이 방지됩니다. 자세한 내용은 8-1 페이지의 "그래픽 프로그래밍 개요"를 참조하십시오.

애니메이션 컨트롤(VCL 전용)

애니메이션 컴포넌트는 AVI(Audio Video Interleaved) 클립을 소리 없이 표시하는 창입니다. AVI 클립은 동영상과 같은 일련의 비트맵 프레임입니다. AVI 클립은 사운드를 가질 수 있지만 애니메이션 컨트롤은 소리가 없는 AVI 클립에서만 작동합니다. 이때 사용하는 파일은 압축되지 않은 AVI 파일이나 RLE(run-length encoding)를 사용하여 압축한 AVI 클립이어야 합니다. 크로스 플랫폼 프로그래밍에서는 애니메이션 컨트롤을 사용할 수 없습니다.

애니메이션 컴포넌트의 일부 속성은 다음과 같습니다.

- *ResHandle*은 AVI 클립을 리소스로 포함하는 모듈에 대한 Windows 핸들입니다. 런타임 시 *ResHandle*을 애니메이션 리소스를 포함하는 모듈의 인스턴스 핸들 또는 모듈 핸들로 설정합니다. *ResHandle*을 설정한 후, *ResID* 또는 *ResName* 속성을 설정하여 애니메이션 컨트롤이 표시해야 할 AVI 클립인 표시된 모듈의 리소스를 지정합니다.
- *AutoSize*를 *True*로 설정하여 애니메이션 컨트롤이 AVI 클립의 프레임 크기에 맞게 자신의 크기를 조정하도록 허용합니다.
- *StartFrame* 및 *StopFrame*은 클립을 시작 및 중지할 프레임을 지정합니다.
- *CommonAVI*를 설정하여 Shell32.DLL에서 제공되는 일반 Windows AVI 클립 중 하나를 표시합니다.
- *Active* 속성을 각각 *True* 및 *False*로 설정하여 애니메이션을 시작 및 중단하는 시기를 지정하고 *Repetitions* 속성을 설정하여 반복 재생 횟수를 지정합니다.
- *Timers* 속성은 타이머를 사용하여 프레임을 표시할 수 있게 해줍니다. 이것은 사운드 트랙 재생과 같은 다른 동작과 애니메이션 시퀀스를 동기화하는 데 유용합니다.

대화 상자 개발

컴포넌트 팔레트의 Dialog 페이지에 있는 대화 상자 컴포넌트들은 애플리케이션에서 다양한 대화 상자를 사용할 수 있게 합니다. 이러한 대화 상자는 파일 열기 및 저장과 같은 공통된 파일 작업을 수행할 수 있는 친숙하고 일관성 있는 인터페이스를 애플리케이션에 제공합니다. 대화 상자는 데이터를 표시하거나 연습니다.

대화 상자는 *Execute* 메소드가 호출되면 열립니다. *Execute*는 부울 값을 반환합니다. OK를 선택하여 대화 상자의 변경 사항을 승인하는 경우 *Execute*는 *True*를 반환하고, Cancel을 선택하여 변경 사항을 저장하지 않고 대화 상자를 빠져 나가는 경우 *Execute*는 *False*를 반환합니다.

크로스 플랫폼 애플리케이션을 개발하는 경우, QDialogs 유닛의 CLX에서 제공하는 대화 상자를 사용할 수 있습니다. 파일 열거나 저장, 글꼴 또는 색 변경 등과 같은 일반 작업을 위한 윈시 대화 상자 유형을 가진 운영 체제의 경우, *UseNativeDialog* 속성을 사용할 수 있습니다. 이러한 환경에서 실행되는 애플리케이션에 대해 Qt 대화 상자 대신 윈시 대화 상자를 사용하려면 *UseNativeDialog*를 *True*로 설정합니다.

열린 대화 상자 사용

공통으로 사용되는 대화 상자 컴포넌트 중의 하나는 *TOpenDialog*입니다. 이 컴포넌트는 보통 폼의 메인 메뉴 바에 있는 File 옵션 아래의 New 또는 Open 메뉴 항목에 의해 호출됩니다. 대화 상자는 와일드카드 문자를 사용하여 파일 그룹을 선택하고 디렉토리를 통해 탐색할 수 있는 컨트롤을 포함합니다.

TOpenDialog 컴포넌트는 Open 대화 상자를 애플리케이션에 사용할 수 있도록 합니다. 이 대화 상자의 용도는 사용자에게 열리는 파일을 선택하도록 하는 것입니다. *Execute* 메소드를 사용하여 대화 상자를 표시합니다.

사용자가 대화 상자에서 OK를 선택하면 사용자의 파일은 *TopenDialog*의 *FileName* 속성에 저장된 후에 원하는 대로 처리할 수 있습니다.

다음 코드를 *Action*에 둘 수 있고 *TMainMenu* 하위 항목의 *Action* 속성에 연결하거나 하위 항목의 *OnClick* 이벤트에 둘 수 있습니다.

```
if OpenDialog1.Execute then
    filename := OpenDialog1.FileName;
```

이 코드는 대화 상자를 보여 주고 사용자가 OK 버튼을 누르면 이전에 선언한 *filename*이라는 *AnsiString* 변수에 파일 이름을 복사합니다.

helper 객체 사용

VCL 및 CLX에는 일반 프로그래밍 작업을 단순화하는 다양한 난비주얼 객체가 포함되어 있습니다. 이 단원에서는 다음 작업을 보다 쉽게 수행할 수 있게 하는 몇 개의 Helper 객체에 대해 설명합니다.

- 목록 사용
- 문자열 목록 사용
- Windows 레지스트리 및 .INI 파일 변경
- 스트림 사용

목록 사용

다음 객체는 목록 생성 및 관리를 위한 기능을 제공합니다.

표 3.5 목록의 생성 및 관리를 위한 컴포넌트

객체	보유
<i>TList</i>	포인터 목록
<i>TObjectList</i>	인스턴스 객체의 메모리 관리 목록
<i>TComponentList</i>	컴포넌트의 메모리 관리 목록 (즉, <i>TComponent</i> 의 자손 클래스의 인스턴스)
<i>TQueue</i>	포인터의 선입선출(FIFO) 목록
<i>TStack</i>	포인터의 후입선출(LIFO) 목록
<i>TObjectQueue</i>	객체의 선입선출 목록
<i>TObjectStack</i>	객체의 후입선출 목록
<i>TClassList</i>	클래스 타입의 목록
<i>TCollection</i> , <i>TOwnedCollection</i> , <i>TCollectionItem</i>	특수하게 정의된 항목의 인덱스된 모음
<i>TStringList</i>	문자열 목록

이러한 객체들에 대한 자세한 내용은 온라인 참조를 보십시오.

문자열 목록 사용

경우에 따라 애플리케이션은 문자열 목록을 관리해야 합니다. 문자열 목록의 예로는 콤보 박스의 항목, 메모의 행, 글꼴 이름, 문자열 그리드의 열과 행 이름이 있습니다. VCL 및 CLX는 *TStrings*라는 객체와 이 객체의 자손인 *TStringList*를 통해 모든 문자열 목록에 공통 인터페이스를 제공합니다. *TStringList*는 *TStrings*에 있는 추상 속성과 메소드를 구현하고, 속성, 이벤트 및 메소드를 사용하여 다음을 수행합니다.

- 목록에서 문자열을 정렬합니다.
- 정렬된 목록에서 문자열 중복을 금합니다.
- 목록 내용의 변경 사항에 응답합니다.

문자열 목록을 보유하는 기능을 제공하는 것 외에도 이러한 객체들은 상호 운용성을 용이하게 합니다. 예를 들어, *TStrings*의 인스턴스인 메모의 행을 편집한 다음 이 행들을 *TStrings*의 인스턴스인 콤보 박스의 항목으로 사용할 수 있습니다.

문자열 목록 속성은 Value 열의 *TStrings*와 함께 Object Inspector에 표시됩니다. *TStrings*를 더블 클릭하여 행을 편집, 추가 또는 삭제할 수 있는 문자열 목록 편집기를 엽니다.

런타임에 문자열 목록 객체를 사용하여 다음과 같은 작업을 수행할 수 있습니다.

- 문자열 목록의 로드 및 저장
- 새 문자열 목록 작성
- 목록에서 문자열 처리
- 문자열 목록에 객체 연결

문자열 목록의 로드 및 저장

문자열 목록 객체는 *SaveToFile* 메소드와 *LoadFromFile* 메소드를 제공하므로 사용자는 문자열 목록을 텍스트 파일에 저장하고 텍스트 파일을 문자열 목록에 로드할 수 있습니다. 텍스트 파일의 각 행은 목록의 문자열에 해당합니다. 이러한 메소드를 사용하면 파일을 메모 컴포넌트에 로드하여 단순한 텍스트 편집기를 만들거나 콤보 박스의 항목 목록을 저장할 수 있습니다.

다음 예제는 WIN.INI 파일 사본을 메모 필드로 로드하고 WIN.BAK 라는 이름의 백업 사본을 만듭니다.

```
procedure EditWinIni;
var
  FileName:string;{ storage for file name }
begin
  FileName := 'C:\WINDOWS\WIN.INI';{ set the file name }
  with Form1.Memo1.Lines do
  begin
    LoadFromFile(FileName);{ load from file }
    SaveToFile(ChangeFileExt(FileName, '.bak'));{ save into backup file }
  end;
end;
```

새 문자열 목록 작성

문자열 목록은 일반적으로 컴포넌트의 일부입니다. 그러나 조회 테이블용 문자열을 저장하는 것과 같이 독립적인 문자열 목록을 만드는 것이 편리할 때가 있습니다. 문자열 목록의 작성 및 관리 방법은 단일 루틴 내에서 생성, 사용 및 소멸되는 목록의 기간이 짧은지 애플리케이션이 종료할 때까지 사용할 만큼 기간이 긴지에 따라 다릅니다. 생성한 문자열 목록의 종류에 상관 없이 사용자는 종료 시 목록을 해제해야 합니다.

기간이 짧은 문자열 목록

문자열 목록을 단일 루틴 기간 동안만 사용하는 경우 그 문자열 목록을 한 루틴 내에서 생성, 사용 및 소멸할 수 있습니다. 이것은 문자열 목록을 사용하는 가장 안전한 방법입니다. 문자열 목록 객체는 목록 자체와 목록의 문자열에 대해 메모리를 할당하기 때문에 예외가 발생하더라도 **try...finally** 블록을 사용하여 메모리가 해제되었는지 확인해야 합니다.

- 1 문자열 목록 객체를 생성합니다.
- 2 **try...finally** 블록의 **try** 부분에서 문자열 목록을 사용합니다.
- 3 **finally** 부분에서 문자열 목록 객체를 해제합니다.

다음 이벤트 핸들러는 문자열 목록을 생성, 사용 및 소멸함으로써 버튼 클릭에 응답합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
var
  TempList:TStrings;{ declare the list }
begin
  TempList := TStringList.Create;{ construct the list object }
```

```

try
  { use the string list }
finally
  TempList.Free;{ destroy the list object }
end;
end;

```

기간이 긴 문자열 목록

애플리케이션을 실행하는 동안 항상 문자열 목록을 사용 가능하도록 하려면 애플리케이션 시작 시 목록을 생성하고 애플리케이션 종료 전에 소멸합니다.

- 1 애플리케이션의 메인 폼에 대한 유닛 파일에서 *TStrings* 타입의 필드를 폼의 선언에 추가합니다.
- 2 메인 폼의 *생성자*에 대한 이벤트 핸들러를 작성합니다. 이 이벤트 핸들러는 폼이 나타나기 전에 실행됩니다. 생성자는 문자열 목록을 만들어서 첫 번째 단계에서 선언한 필드에 할당해야 합니다.
- 3 폼의 *OnClose* 이벤트용 문자열 목록을 해제하는 이벤트 핸들러를 작성합니다.

이 예제에서는 기간이 긴 문자열 목록을 사용하여 메인 폼에서 사용자의 마우스 클릭을 기록한 다음 애플리케이션이 종료하기 전에 목록을 파일에 저장합니다.

```

unit Unit1;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
{For CLX:uses SysUtils, Classes, QGraphics, QControls, QForms, Qialogs;}

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender:TObject);
    procedure FormDestroy(Sender:TObject);
    procedure FormMouseDown(Sender:TObject; Button:TMouseButton;
      Shift:TShiftState; X, Y:Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList:TStrings;{ declare the field }
  end;

var
  Form1:TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender:TObject);
begin
  ClickList := TStringList.Create;{ construct the list }
end;

```

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.LOG'));{ save the list }
  ClickList.Free;{ destroy the list object }
end;

procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  ClickList.Add(Format('Click at (%d, %d)', [X, Y]));{ add a string to the list }
end;

end.
```

목록에서 문자열 처리

다음은 문자열 목록에서 공통으로 수행되는 작업입니다.

- 목록의 문자열 수
- 특정 문자열 액세스
- 목록에서 문자열 위치 찾기
- 목록에서 문자열 반복
- 목록에 문자열 추가
- 목록 내에서 문자열 이동
- 목록에서 문자열 삭제
- 문자열 목록 전체 복사

목록의 문자열 수

읽기 전용 *Count* 속성은 목록의 문자열 수를 반환합니다. 문자열 목록은 인덱스가 0부터 시작하므로 *Count*는 마지막 문자열의 인덱스보다 하나 더 많습니다.

특정 문자열 액세스

Strings 배열 속성에는 0부터 시작하는 인덱스에 의해 참조되는 목록의 문자열을 포함합니다. *Strings*는 문자열 목록의 기본 속성이므로 목록에 액세스할 때 *Strings* 식별자를 생략할 수 있습니다. 따라서,

```
StringList1.Strings[0] := 'This is the first string.';
```

이 구문은 다음과 동일합니다.

```
StringList1[0] := 'This is the first string.';
```

문자열 목록에서 항목 찾기

문자열 목록에서 문자열을 찾으려면 *IndexOf* 메소드를 사용합니다. *IndexOf*는 전달된 매개변수와 일치하는 목록의 첫 번째 문자열 인덱스를 반환하고, 일치하는 매개변수 문자열을 찾지 못하면 -1 을 반환합니다. *IndexOf*는 정확하게 일치하는 문자열만을 찾습니다. 목록에서 일치하는 부분 문자열을 찾으려면 문자열 목록을 반복해야 합니다.

예를 들어, 다음과 같이 *IndexOf*를 사용하면 리스트 박스의 *Items* 중에서 파일 이름이 있는지 알 수 있습니다.

```
if FileListBox1.Items.IndexOf('WIN.INI') > -1 ...
```

목록에서 문자열 반복

목록의 문자열을 반복하려면 0에서 *Count - 1*까지 실행되는 **for** 순환문을 사용합니다.

다음 예제는 리스트 박스에 있는 각 문자열을 대문자로 변환합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
var
  Index:Integer;
begin
  for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
  end;
```

목록에 문자열 추가

문자열을 문자열 목록의 끝에 추가하려면 *Add* 메소드를 호출하여 매개변수로 새 문자열을 전달합니다. 문자열을 목록에 삽입하려면 *Insert* 메소드를 호출하여 다음 두 개의 매개변수인 문자열과 문자열을 삽입할 위치에 해당하는 인덱스를 넘깁니다. 예를 들어, 문자열 "Three"를 목록에서 세 번째 문자열로 만들려면 다음과 같이 합니다.

```
Insert(2, 'Three');
```

문자열을 한 목록에서 다른 목록으로 추가하려면 다음과 같이 *AddStrings*를 호출하십시오.

```
StringList1.AddStrings(StringList2); { append the strings from StringList2 to StringList1 }
```

목록 내에서 문자열 이동

문자열 목록에서 문자열을 이동하려면 *Move* 메소드를 호출하여 매개변수로 문자열의 현재 인덱스와 문자열을 옮길 인덱스를 넘깁니다. 예를 들어, 목록에서 세 번째 문자열을 다섯 번째 위치로 이동하려면 다음과 같이 합니다.

```
Move(2, 4)
```

목록에서 문자열 삭제

문자열 목록에서 문자열을 삭제하려면 목록의 *Delete* 메소드를 호출하여 삭제하려는 문자열의 인덱스를 전달합니다. 삭제하려는 문자열의 인덱스를 모르는 경우 *IndexOf* 메소드를 사용하여 위치를 찾아냅니다. 문자열 목록에서 모든 문자열을 삭제하려면 *Clear* 메소드를 사용합니다.

이 예제에서는 다음과 같이 *IndexOf*와 *Delete*를 사용하여 문자열을 찾아서 삭제합니다.

```
with ListBox1.Items do
begin
```

```

BIndex := IndexOf('bureaucracy');
if BIndex > -1 then
  Delete(BIndex);
end;

```

문자열 목록 전체 복사

Assign 메소드를 사용하여 소스 목록에서 대상 목록으로 문자열을 복사할 수 있습니다. 이 메소드는 대상 목록의 내용을 덮어씁니다. 대상 목록에 덮어쓰지 않고 문자열을 추가하려면 *AddStrings*를 사용합니다. 예를 들면, 다음과 같습니다.

```

Memo1.Lines.Assign(ComboBox1.Items); { overwrite original strings }

```

이 구문은 콤보 박스의 행을 메모에 복사(메모를 덮어 씌)합니다.

```

Memo1.Lines.AddStrings(ComboBox1.Items); { appends strings to end }

```

그러나 이 구문은 콤보 박스의 행을 메모에 추가합니다.

문자열 목록의 로컬 복사본을 만들 때 *Assign* 메소드를 사용합니다. 다음과 같이 한 문자열 목록 변수를 다른 문자열 목록 변수에 할당하는 경우

```

StringList1 := StringList2;

```

원래 문자열 목록 객체를 잃게 되고 가끔 예상치 못한 결과를 가져옵니다.

문자열 목록에 객체 연결

Strings 속성에 저장된 문자열 외에 문자열 목록은 객체에 대한 참조를 유지 관리할 수 있고 *Objects* 속성에 저장합니다. *Strings*와 마찬가지로 *Objects* 속성은 인덱스가 0부터 시작하는 배열입니다. 비트맵을 owner-draw 컨트롤의 문자열에 연결할 때 *Objects* 속성이 가장 많이 사용됩니다.

AddObject 메소드 또는 *InsertObject* 메소드를 사용하여 목록에 문자열과 연결 객체를 한 번에 추가합니다. *IndexOfObject*는 지정된 객체에 연결된 목록에서 첫 번째 문자열의 인덱스를 반환합니다. *Delete*, *Clear* 및 *Move*와 같은 메소드는 문자열과 객체 모두에서 작동합니다. 예를 들어, 문자열을 삭제할 때 해당 객체가 있으면 해당 객체가 제거됩니다.

기존 문자열에 객체를 연결하려면 객체를 동일한 인덱스에 있는 *Objects* 속성에 할당합니다. 해당 문자열을 추가하지 않고 객체를 추가할 수 없습니다.

Windows 레지스트리 및 INI 파일(VCL 전용)

Windows 시스템 레지스트리는 애플리케이션이 구성 정보를 저장하는 계층 데이터베이스입니다. VCL 클래스 *TRegistry*는 레지스트리를 읽고 쓰는 메소드를 제공합니다.

Windows 95까지는 대부분의 애플리케이션이 일반적으로 확장명이 .INI인 초기화 파일에 구성 정보를 저장하였습니다. VCL은 INI 파일을 사용하는 프로그램의 유지 관리 및 마이그레이션을 용이하게 하기 위해 다음과 같은 클래스를 제공합니다.

- 레지스트리를 사용하기 위한 *TRegistry*(VCL 전용)
- INI 파일을 사용하기 위한 *TIniFile*(VCL 전용) 또는 *TMemIniFile*

- 레지스트리 및 INI 파일 모두를 사용하기 위한 *TRegistryIniFile*(VCL 전용)
*TRegistryIniFile*은 *TIniFile*과 유사한 속성 및 메소드를 갖지만 시스템 레지스트리에 대해 읽기와 쓰기를 수행합니다. *TCustomIniFile*(*TIniFile*, *TMemIniFile* 및 *TRegistryIniFile*의 공통 조상) 타입의 변수를 사용하면 호출 위치에 따라 레지스트리 또는 INI 파일을 액세스하는 일반 코드를 작성할 수 있습니다.

크로스 플랫폼 프로그래밍에서는 *TMemIniFile*만 사용할 수 있습니다.

TIniFile 사용(VCL 전용)

INI 파일 형식은 여전히 많이 사용되며 DSK Desktop 설정 파일과 같은 많은 Delphi 구성 파일이 이 형식입니다. 이 파일 형식이 지금까지 많이 사용되기 때문에 VCL은 이러한 파일을 쉽게 읽고 쓸 수 있는 클래스를 제공합니다. 단, 크로스 플랫폼 프로그래밍에서는 *TIniFile*을 사용할 수 없습니다.

TIniFile 객체를 인스턴스화할 때 INI 파일 이름을 매개변수로서 생성자에 전달합니다. 이 파일은 존재하지 않을 경우 자동으로 만들어집니다. 이렇게 하면 *ReadString*, *ReadInteger* 또는 *ReadBool*을 사용하여 값을 자유롭게 읽을 수 있습니다. INI 파일의 전체 섹션을 읽으려는 경우에는 *ReadSection* 메소드를 사용할 수 있습니다. 마찬가지로 *WriteBool*, *WriteInteger* 또는 *WriteString*을 사용하여 값을 쓸 수 있습니다.

각 Read 루틴은 세 개의 매개변수를 가집니다. 첫 번째 매개변수는 INI 파일의 섹션을 식별합니다. 두 번째 매개변수는 읽고자 하는 값을 식별하고, 세 번째 매개변수는 섹션이나 값이 INI 파일에 존재하지 않을 경우 기본값이 됩니다. 마찬가지로 Write 루틴도 섹션 및/또는 값이 존재하지 않을 경우 새로 만듭니다. 다음 예제 코드는 처음 실행될 때 INI 파일을 만듭니다.

```
[Form]
Top=185
Left=280
Caption=Default Caption
InitMax=0
```

이 애플리케이션이 다음 번에 실행되면 폼이 작성되는 동안 INI 값을 읽고 *OnClose* 이벤트에 이 값을 다시 기록합니다.

TRegistry 사용

대부분의 32비트 애플리케이션은 INI 파일 대신 레지스트리에 정보를 저장합니다. 이는 레지스트리가 계층적이고, 더 견고하고, INI 파일과 달리 크기 제한이 없기 때문입니다. *TRegistry* 객체에는 키 열기, 닫기, 저장, 이동, 복사 및 삭제를 위한 메소드가 포함되어 있습니다.

크로스 플랫폼 프로그래밍에서는 *TRegistry*를 사용할 수 없습니다.

자세한 내용은 온라인 도움말의 *TRegistry* 항목을 참조하십시오.

TRegIniFile 사용

INI 파일을 사용하는 데 익숙한 경우, *TRegIniFile* 클래스를 사용하여 구성 정보를 레지스트리로 이동할 수 있습니다. *TRegIniFile*은 레지스트리 항목을 INI 파일 항목처럼 보이도록 디자인된 클래스입니다. *TRegIniFile*에는 *TIniFile*의 모든 메소드(읽기 및 쓰기)가 존재합니다.

TRegIniFile 객체를 생성할 경우, 전달된 매개변수 (*IniFile* 객체의 파일 이름)는 레지스트리의 사용자 키에서 키 값이 되고 모든 섹션과 값은 이 루트에서 분기됩니다. 실제로 이 객체는 레지스트리 인터페이스를 현저하게 단순화하기 때문에 기존 코드를 포팅하는 것이 아니라도 *TRegistry* 컴포넌트 대신 이 객체를 사용할 수 있습니다.

크로스 플랫폼 프로그래밍에서는 *TRegIniFile*을 사용할 수 없습니다.

자세한 내용은 VCL 온라인 참조의 *TRegIniFile* 항목을 참조하십시오.

드로잉 공간 만들기

*TCanvas*는 VCL에서 Windows 장치 컨텍스트를 캡슐화하고 CLX에서 페인트 장치(QT 페인터)를 캡슐화하므로 패널과 같은 비주얼 컨테이너인 폼과 프린터 객체(3-57 페이지의 "인쇄" 참조)의 모든 그리기를 처리합니다.

캔버스 객체를 사용하면 사용자를 위해 모든 할당 및 할당 해제가 처리되므로 펜, 브러시, 팔레트 등의 할당에 신경 쓸 필요가 없습니다.

*TCanvas*는 캔버스를 포함하는 컨트롤에 선, 도형, 다각형 및 글꼴 등을 그리는 여러 가지 기본적인 그래픽 루틴을 포함합니다. 예를 들어, 다음과 같은 버튼 이벤트 핸들러는 폼의 왼쪽 위 모서리에서 중앙으로 선을 그리고 약간의 원시 텍스트를 폼 위에 출력합니다.

```

procedure TForm1.Button1Click(Sender:TObject);
begin
    Canvas.Pen.Color := clBlue;
    Canvas.MoveTo( 10, 10 );
    Canvas.LineTo( 100, 100 );
    Canvas.Brush.Color := clBtnFace;
    Canvas.Font.Name := ?rial?
    Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y,?his is the end of the line?;
end;

```

또한 Windows 애플리케이션에서 *TCanvas* 객체는 장치 컨텍스트, 펜, 브러쉬시 등을 그리기 작업 이전의 값으로 복원하는 경우와 같은 일반 Windows 그래픽 오류를 방지합니다. *TCanvas*는 Delphi에서 그리기가 필요하거나 가능한 곳 어디에서나 사용되며 그래픽 그리기를 쉽게 할 수 있습니다.

속성과 메소드의 전체 목록을 보려면 온라인 도움말의 *TCanvas*를 참조하십시오.

인쇄

VCL *TPrinter* 객체는 Windows 프린터의 세부 정보를 캡슐화합니다. 설치되어 있는 사용 가능한 프린터 목록을 보려면 *Printers* 속성을 사용합니다. *TPrinter* 객체는 프린터에 그리는 그리기 장치입니다. 이 장치는 포스트스크립트를 생성하여 *lpr*, *lp* 또는 다른 프린트 명령에 포스트스크립트를 보냅니다.

프린터 객체는 폼의 *TCanvas*와 동일한 *TCanvas*를 사용하므로 폼에 그릴 수 있는 모든 것을 또한 인쇄할 수 있습니다. 이미지를 인쇄하려면 *BeginDoc* 메소드를 호출한 다음 인쇄하려는 캔버스 그래픽과 텍스트(*TextOut* 메소드 사용)를 나열한 후, *EndDoc* 메소드를 호출하여 인쇄할 작업을 프린터에 보냅니다.

이 예제에서는 폼에 버튼과 메모를 사용합니다. 사용자가 버튼을 클릭하면 메모의 내용은 페이지 주위의 200픽셀 테두리로 인쇄됩니다.

이 예제를 성공적으로 실행하려면 *Printers*를 **uses** 절에 추가합니다.

```

procedure TForm1.Button1Click(Sender:TObject);
var
    r:TRect;
    i:Integer;
begin
    with Printer do
        begin
            r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
            BeginDoc;
            for i := 0 to Mem1.Lines.Count do
                Canvas.TextOut(200,200 + (i *
Canvas.TextHeight(Mem1.Lines.Strings[i])),
                            Mem1.Lines.Strings[i]);
                Canvas.Brush.Color := clBlack;
                Canvas.FrameRect(r);
                EndDoc;
            end;
        end;

```

TPrinter 객체의 사용에 대한 자세한 내용은 *TPrinter*의 온라인 도움말을 보십시오.

스트림 사용

스트림은 데이터를 읽고 쓰는 수단입니다. 스트림은 메모리, 문자열, 소켓 및 BLOB 스트림과 같은 다른 매체에 읽고 쓰기 위한 공통적인 인터페이스를 제공합니다.

다음 스트리밍 예제에서는 스트림을 사용하여 한 파일을 다른 파일에 복사합니다. 애플리케이션에는 두 가지 편집 컨트롤(From 및 To)과 Copy File 버튼이 있습니다.

```

procedure TForm1.CopyFileClick(Sender:TObject);
var
    stream1, stream2:TStream;
begin
    stream1:=TFileStream.Create(From.Text,fmOpenRead or fmShareDenyWrite);
    try
        stream2 := TFileStream.Create(To.Text fmOpenCreate or fmShareDenyRead);
    
```

```
try
    stream2.CopyFrom(Stream1,Stream1.Size);
finally
    stream2.Free;
finally
    stream1.Free
end;
```

저장 매체를 읽고 쓰려면 특수한 스트림 객체를 사용합니다. *TStream*의 각 자손은 디스크 파일, 동적 메모리 등 특정 매체에 액세스하기 위한 메소드를 구현합니다. *TStream* 자손으로는 *TFileStream*, *TStringStream* 및 *TMemoryStream*이 있습니다. 읽기 및 쓰기용 메소드 이외에 이러한 객체는 애플리케이션에서 스트림의 임의 위치를 찾을 수 있습니다. *TStream*의 속성은 크기 및 현재 위치와 같은 스트림에 대한 정보를 제공합니다.

4

일반적인 프로그래밍 작업

이 장에서는 Delphi의 다음과 같은 일반적인 프로그래밍 작업을 수행하는 방법에 대해 설명합니다.

- 클래스 이해
- 클래스 정의
- 예외 처리
- 인터페이스 사용
- 사용자 지정 가변 정의
- 문자열 작업
- 파일 작업
- 측정 변환

클래스 이해

클래스는 속성, 메소드, 이벤트, 클래스에 대한 지역 변수와 같은 클래스 멤버에 대한 추상적인 정의입니다. 클래스의 인스턴스를 만들면 이 인스턴스를 객체라고 합니다. 객체라는 용어는 Delphi 설명서에서 가끔 부정확하게 사용되고 클래스와 클래스의 인스턴스 간의 구별이 중요하지 않은 곳에서는 "객체"가 클래스를 가리키기도 합니다.

Delphi는 자체의 객체 계층에 많은 클래스를 포함하지만 객체 지향 프로그램을 작성하는 경우에는 추가의 클래스를 만들어야 합니다. 사용자가 작성하는 클래스는 *TObject*의 자손이거나 그 자손 중 하나여야 합니다. 클래스 타입 선언에는 클래스의 필드와 메소드 액세스를 제어하는 다음과 같은 세 가지의 이용 가능한 섹션이 있습니다.

```
Type
TClassName = Class(TObject)
    public
        {public fields}
        {public methods}
    protected
        {protected fields}
```

```

        {protected methods}
    private
        {private fields}
        {private methods}
end;

```

- public 섹션은 액세스 제한이 없는 필드와 메소드를 선언하고, 클래스 인스턴스와 자손 클래스는 이러한 필드와 메소드에 액세스할 수 있습니다.
- protected 섹션은 일부 액세스 제한이 있는 필드와 메소드를 포함하고, 자손 클래스는 이러한 필드와 메소드에 액세스할 수 있습니다.
- private 섹션은 액세스가 엄격하게 제한된 필드와 메소드를 선언하고, 클래스 인스턴스와 자손 클래스는 이러한 필드와 메소드에 액세스할 수 없습니다.

클래스를 사용하면 새 클래스를 기존 클래스의 자손으로 생성할 수 있다는 이점이 있습니다. 각 자손 클래스는 그 부모 및 조상 클래스의 필드와 메소드를 상속합니다. 또한 상속된 클래스를 오버라이드하는 새 클래스에서 메소드를 선언하여 좀 더 특별한 새로운 동작을 도입할 수 있습니다.

자손 클래스의 일반 구문은 다음과 같습니다.

```

Type
TClassName = Class (TParentClass)
    public
        {public fields}
        {public methods}
    protected
        {protected fields}
        {protected methods}
    private
        {private fields}
        {private methods}
end;

```

부모 클래스 이름이 지정되지 않은 경우 클래스는 *TObject*로부터 직접 상속합니다. *TObject*는 기본 생성자와 소멸자를 포함한 몇 가지 메소드만 정의합니다.

클래스의 구문, 랭귀지 정의, 규칙에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말의 Class types를 참조하십시오.

클래스 정의

Delphi에서는 클래스를 선언하여 애플리케이션에서 사용해야 하는 프로그래밍 기능을 구현할 수 있습니다. Delphi의 일부 버전에는 클래스 완성 (class completion)이라는 기능이 있는데 이 기능은 사용자가 선언하는 클래스 멤버에 대한 뼈대 코드를 생성하여 새 클래스를 정의하고 구현하는 작업을 간단하게 합니다.

다음과 같이 클래스를 정의합니다.

- 1 IDE에서 프로젝트를 열고 File|New|Unit을 선택하여 새 클래스를 정의할 수 있는 새 유닛을 만듭니다.

- 2 **interface** 섹션에 **uses** 절과 **type** 섹션을 추가합니다.
- 3 **type** 섹션에 클래스 선언을 작성합니다. 모든 멤버 변수, 속성, 메소드 및 이벤트를 선언해야 합니다.

```
TMyClass = class; {This implicitly descends from TObject}
public
.
.
.
.
private
.
.
.
published {If descended from TPersistent or below}
.
.
.
```

참고

사용자 정의 가변 데이터를 갖는 객체는 RTTI로 컴파일해야 합니다. 이는 이 객체를 {\$M+} 컴파일러 지시어 또는 *TPersistent*나 그 아래의 자손을 사용하여 컴파일해야 함을 의미합니다.

특정 클래스의 자손 클래스를 원하는 경우에는 다음과 같이 정의에서 그 클래스를 나타내야 합니다.

```
TMyClass = class(TParentClass); {This descends from TParentClass}
```

예를 들면, 다음과 같습니다.

```
type TMyButton = class(TButton)
  property Size:Integer;
  procedure DoSomething;
end;
```

Delphi 버전에서 클래스 완성을 포함하는 경우, **interface** 섹션에 있는 메소드 정의 안에 커서를 두고 Ctrl+Shift+C를 누르거나 마우스 오른쪽 버튼을 클릭하여 Complete Class at Cursor를 선택합니다. Delphi는 모든 완료되지 않은 속성 선언을 완성하고 **implementation** 섹션에서 필요한 빈 메소드를 생성합니다. 클래스 완성이 없으면 사용자가 직접 코드를 작성하여 속성 선언을 완성하고 메소드를 작성합니다.

위와 같은 예에서 클래스 완성이 있으면 Delphi는 인터페이스 선언에 **read** 지정자와 **write** 지정자를 추가하여 모든 지원 필드나 메소드를 포함시킵니다.

```
type TMyButton = class(TButton)
  property Size:Integer read FSize write SetSize;
  procedure DoSomething;
private
  FSize:Integer;
  procedure SetSize(const Value:Integer);
```

또한 유닛의 **implementation** 섹션에 다음 코드를 추가합니다.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
```

```

end;
procedure TMyButton.SetSize(const Value:Integer);
begin
  FSize := Value;
end;

```

- 4 메소드를 채웁니다. 예를 들면, 버튼에서 DoSomething 메소드를 호출할 때 경고음이 울리게 하려면 **begin**과 **end** 사이에 Beep를 추가합니다.

```

{ TMyButton }
procedure TMyButton.DoSomething;
begin
  Beep;
end;

procedure TMyButton.SetSize(const Value:Integer);
begin
  if fsize < > value then
  begin
    FSize := Value;
    DoSomething;
  end;
end;

```

버튼의 크기를 변경하기 위해 SetSize를 호출할 때도 경고음이 울린다는 것에 유의하십시오.

클래스와 메소드의 구문, 랭귀지 정의, 규칙에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말의 Class types와 methods를 참조하십시오.

예외 처리

Delphi는 일관된 방식으로 오류를 처리하는 메커니즘을 제공합니다. 예외 처리를 통해서 애플리케이션은 데이터나 리소스를 잃지 않고 복구 가능한 오류를 복구하고 필요한 경우 종료합니다. Delphi의 오류 조건은 예외에 의해 나타납니다. 이 단원에서는 예외를 사용하여 안전한 애플리케이션을 만들 수 있는 다음과 같은 작업들을 설명합니다.

- 코드 블록 보호
- 리소스 할당 보호
- RTL(Run-Time Library) 예외 처리
- 컴포넌트 예외 처리
- 외부 소스를 사용한 예외 처리
- 예외 숨기기(Silent exceptions)
- 사용자 고유의 예외 정의

코드 블록 보호

안정적인 애플리케이션을 만들려면 예외가 발생할 때 코드에서 예외를 인식하고 이에 대해 응답해야 합니다. 응답을 지정하지 않으면 애플리케이션은 오류를 설명하는 메시지

상자를 보여 줍니다. 이 때 오류가 발생할 수 있는 곳을 알아내고 오류로 인해 데이터나 시스템 리소스의 손실을 가져올 수 있는 곳에 반드시 응답을 정의해야 합니다.

예외에 대한 응답을 만들 때는 코드의 블록에서도 응답을 만듭니다. 일련의 문장들이 모두 오류에 대한 동일한 종류의 응답이 필요한 경우에는 그 문장들을 블록으로 그룹화한 다음 전체 블록에 적용되는 오류 응답을 정의할 수 있습니다.

예외에 대해 특정한 응답을 가진 블록을 `protected` 블록이라고 하는데 그 이유는 이 블록이 애플리케이션을 종료시키거나 데이터를 손상시키는 오류를 막을 수 있기 때문입니다.

코드의 블록을 보호하려면 다음을 이해해야 합니다.

- 예외에 대한 응답
- 예외와 제어 흐름
- 예외 응답 중첩

예외에 대한 응답

오류 조건이 발생하면 애플리케이션은 예외를 발생시키는데 이것은 애플리케이션이 예외 객체를 만든다는 것을 의미합니다. 예외가 일단 발생하면 애플리케이션은 클린업 코드를 실행하거나 예외를 처리하거나 아니면 둘 다 수행합니다.

클린업 코드 실행

예외에 응답하는 가장 간단한 방법은 일부 클린업 코드가 실행되는 것을 보장하는 것입니다. 이러한 종류의 응답은 오류를 일으키는 조건을 고쳐 주지는 않지만 애플리케이션이 불안정한 상태의 환경에 머무르지 않도록 해줍니다. 보통 이러한 종류의 응답을 사용하여 오류 발생 여부와 상관 없이 애플리케이션이 할당된 리소스를 확실히 해제하도록 합니다.

예외 처리

이것은 특정한 예외에 대한 특정한 응답입니다. 예외를 처리하면 오류 조건을 없애고 예외 객체를 소멸시키며 이를 통해 애플리케이션을 계속 실행할 수 있습니다. 보통 애플리케이션이 오류를 복구하고 계속 실행할 수 있도록 하기 위해 예외 핸들러를 정의합니다. 처리해야 할 예외의 종류에는 존재하지 않는 파일 열기 시도, 용량이 꽉 찬 디스크에 쓰기, 허용 한도를 초과하는 계산 등이 있습니다. 이러한 예외 중에서 "File not found"와 같은 예외는 고치고 재시도하기가 쉽지만 메모리 부족과 같은 예외는 애플리케이션이나 사용자가 고치기가 더 어렵습니다.

예외를 효과적으로 처리하려면 다음을 이해해야 합니다.

- 예외 핸들러 작성
- 예외 처리 문장
- 예외 인스턴스 사용
- 예외 핸들러의 범위
- 기본 예외 핸들러 제공
- 예외 클래스 다루기
- 예외 재발생

예외와 제어 흐름

예외는 코드의 정상적인 흐름을 방해하지 않으므로 오브젝트 파스칼을 사용하여 애플리케이션에 오류 처리 기능을 통합하기가 쉽습니다. 사실, 알고리즘의 주요 과정에 오류 확인과 오류 처리를 두지 않으면 예외는 사용자가 작성하는 코드를 단순화해 줄 것입니다.

보호 블록을 선언할 때 블록 안에서 발생할 수 있는 예외에 대한 특정 응답을 정의하게 됩니다. 그 블록에서 예외가 발생하면 즉시 사용자가 정의한 응답으로 건너서 실행된 다음 그 블록을 벗어납니다.

예 다음 코드는 보호 블록을 포함하고 있습니다. 예외가 protected 블록에서 발생하면 예외 처리 부분으로 건너뛰어 실행되면서 경고음이 울립니다. 블록 밖에서 실행이 재개됩니다.

```

try
  AssignFile(F, FileName);
  Reset(F);
  :
except
  on Exception do Beep;
end;
: { execution resumes here, outside the protected block }

```

예외 응답 중첩

사용자 코드는 블록 내에서 발생하는 예외에 대한 응답을 정의합니다. 파스칼에서는 다른 블록 내에 코드의 블록을 중첩할 수 있기 때문에 사용자 지정된 응답을 이미 포함하고 있는 블록 내에서도 응답을 사용자 지정할 수 있습니다.

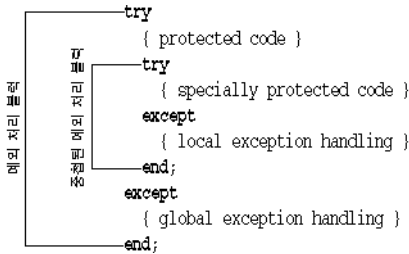
가장 간단한 경우를 예로 들면, 리소스 할당을 보호할 수 있고 그 보호된 블록 내에 다른 리소스를 할당하고 보호하는 블록을 정의할 수 있습니다. 개념적으로 보면 다음과 같습니다.

```

      { allocate first resource }
      try
        { allocate second resource }
        try
          { code that uses both resources }
          finally
            { release second resource }
          end;
        finally
          { release first resource }
        end;
      end;

```


또한 주변 블록 안에서의 예외 처리를 오버라이드하는 특정 예외에 대한 지역 처리를 정의하기 위해서 중첩 블록을 사용할 수 있습니다. 개념적으로 보면 다음과 같습니다.



또한 예외 처리 블록 내에 리소스 보호를 중첩시키면서 다른 종류의 예외 응답 블록을 혼합할 수 있으며 그 반대로도 할 수 있습니다.

리소스 할당 보호

강력한 애플리케이션을 만드는 한 가지 중요한 요소는 예외가 발생하더라도 할당한 리소스를 확실히 해제하도록 하는 것입니다. 예를 들어, 애플리케이션이 메모리를 할당할 경우 반드시 메모리도 해제하도록 해야 합니다. 애플리케이션이 파일을 열 경우 나중에 파일도 반드시 닫도록 해야 합니다.

예외는 코드에서만 발생하는 것이 아니라는 점을 명심하십시오. 예를 들면, RTL 루틴 호출이나 애플리케이션의 다른 컴포넌트가 예외를 발생시킬 수 있습니다. 개발자가 작성한 코드는 이러한 조건이 발생했을 경우 할당된 리소스를 확실히 해제하도록 해야 합니다.

리소스를 효과적으로 보호하기 위해서는 다음 사항을 이해해야 합니다.

- 보호가 필요한 리소스 종류
- 리소스 보호 블록 만들기

보호가 필요한 리소스 종류

일반적인 환경에서는 애플리케이션이 할당과 해제에 대한 코드를 포함하여 할당된 리소스를 해제하도록 할 수 있습니다. 그러나 예외가 발생할 때에도 애플리케이션이 리소스 해제 코드를 실행하게 해야 합니다.

항상 해제를 염두에 두어야 하는 몇 가지 일반적인 리소스는 다음과 같습니다.

- 파일
- 메모리
- Windows 리소스(VCL 전용)
- 객체

예 다음 이벤트 핸들러는 메모리를 할당한 다음 오류를 생성하므로 메모리를 해제하는 코드를 절대 실행하지 않습니다.

```

procedure TForm1.Button1Click(Sender:TComponent);
var
  APointer:Pointer;
  AnInteger, ADividend:Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  AnInteger := 10 div ADividend;{ this generates an error }
  FreeMem(APointer, 1024);{ it never gets here }
end;

```

대개의 오류가 이렇게 명백한 것은 아니지만 위의 예제는 중요한 점을 보여 줍니다. 0으로 나누는 오류가 발생하면 블록 밖으로 건너뛰어 실행되므로 *FreeMem* 문장이 절대로 메모리를 해제하지 못합니다.

*FreeMem*이 *GetMem*에 의해 할당된 메모리를 확실히 해제하도록 하려면 코드를 리소스 보호 블록에 넣어야 합니다.

리소스 보호 블록 만들기

예외인 경우에도 할당된 리소스를 확실히 해제하려면 보호된 블록 안에 리소스 사용 코드를 포함시키고 이 블록의 특별한 부분에 리소스 해제 코드를 추가해야 합니다. 전형적인 보호된 리소스 할당 형태는 다음과 같습니다.

```

{ allocate the resource }
try
  { statements that use the resource }
finally
  { free the resource }
end;

```

try..finally 블록의 주요 특징은 protected 블록에서 예외가 발생하더라도 애플리케이션이 블록의 **finally** 부분에 있는 모든 문장을 항상 실행한다는 것입니다. 블록의 **try** 부분의 코드 또는 **try** 부분의 코드에 의해 호출된 루틴이 예외를 발생시키면 그 시점에서 실행이 중지됩니다. 예외 핸들러가 일단 발견되면 클린업 코드라는 **finally** 부분으로 건너뛰어 실행됩니다. **finally** 부분이 실행되고 나면 예외 핸들러가 호출됩니다. 예외가 발생하지 않으면 클린업 코드가 정상적인 순서로 **try** 부분의 모든 문장 뒤에서 실행됩니다.

예 다음 코드에서는 메모리를 할당하고 오류를 생성하면서 할당된 메모리를 해제하는 이벤트 핸들러의 예를 보여 줍니다.

```

procedure TForm1.Button1Click(Sender:TComponent);
var
  APointer:Pointer;
  AnInteger, ADividend:Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an error }
  finally
    FreeMem(APointer, 1024);{ execution resumes here, despite the error }
  end;
end;

```

finally 블록의 문장은 예외 발생에 영향을 받지 않습니다. **try** 부분의 문장에서 예외가 발생하지 않으면 **finally** 블록을 거쳐 계속 실행됩니다.

RTL(Run-Time Library) 예외 처리

수학 함수나 파일 처리 프로시저와 같은 런타임 라이브러리 (RTL)의 루틴을 호출하는 코드를 작성할 경우, RTL은 예외의 형태로 애플리케이션에 오류를 보고합니다. 기본적으로 RTL 예외는 애플리케이션이 사용자에게 보여 주는 메시지를 생성합니다. RTL 예외를 다른 방법으로 처리하는 자신만의 예외 핸들러를 정의할 수 있습니다.

기본적으로 메시지를 보여 주지 않는 예외 숨기기 (silent exceptions)도 있습니다.

RTL 예외는 다른 모든 예외처럼 처리됩니다. RTL 예외를 효과적으로 처리하려면 다음 사항을 이해해야 합니다.

- RTL 예외
- 예외 핸들러 작성
- 예외 처리 문장
- 예외 인스턴스 사용
- 예외 핸들러의 범위
- 기본 예외 핸들러 제공
- 예외 클래스 다루기
- 예외 재발생

RTL 예외

런타임 라이브러리의 예외는 *SysUtils* 유닛에서 정의되며 이 예외는 모두 Exception이라는 일반적인 예외 객체 타입의 자손입니다. Exception은 RTL 예외가 기본적으로 보여 주는 메시지에 대한 문자열을 제공합니다.

다음 표에서 설명한 대로 여러 종류의 예외가 RTL에 의해 발생합니다.

표 4.1 RTL 예외

오류 타입	원인	의미
I/O	파일 또는 I/O 장치 액세스 오류	대부분의 I/O 예외는 파일에 액세스할 때 반환된 오류 코드와 관련이 있습니다.
힙	동적 메모리 사용 오류	사용 가능한 메모리가 불충분하거나 애플리케이션이 힙 외부로 가리키는 포인터를 처리할 때 힙 오류가 발생할 수 있습니다.
정수 함수	정수 타입 표현식의 부적합한 연산	오류에는 0으로 나누기, 범위 밖의 숫자나 표현식, 오버플로 등이 있습니다.
부동 소수점 수학	실수 타입 표현식의 부적합한 연산	부동 소수점 오류는 하드웨어 보조프로세서 또는 소프트웨어 에뮬레이터가 원인입니다. 오류에는 잘못된 명령, 0으로 나누기, 오버플로 또는 언더플로 등이 있습니다.
타입 변환 (Typecast)	as 연산자를 사용하는 잘못된 타입 변환	객체는 호환 가능한 타입으로만 타입 변환할 수 있습니다.

표 4.1 RTL 예외 (계속)

오류 타입	원인	의미
변환	잘못된 타입 변환	IntToStr, StrToInt, StrToFloat 등과 같은 타입 변환 함수는 매개변수를 원하는 타입으로 변환할 수 없을 때 변환 예외를 발생시킵니다.
하드웨어	시스템 상태	하드웨어 예외는 프로세서 또는 사용자가 액세스 위반, 스택 오버플로 또는 키보드 인터럽트 등과 같은 어떤 오류 조건이나 인터럽트를 생성했음을 나타냅니다.
가변	부적합한 타입 강제 변환	가변(variant)을 호환 가능한 타입으로 강제 변환할 수 없는 표현식에서 가변을 참조하면 오류가 발생합니다.

RTL 예외 타입의 목록은 *SysUtils* 유닛의 코드를 참조하십시오.

예외 핸들러 작성

예외 핸들러는 특정 예외 또는 코드의 `protected` 블록 안에서 발생하는 예외를 처리하는 코드입니다. 크로스 플랫폼 프로그래밍에서는 예외 핸들러를 작성해야 하는 경우가 거의 드물습니다. 대부분의 예외는 4-4 페이지의 "코드 블록 보호"와 4-7 페이지의 "리소스 할당 보호"에서 설명한 대로 `try..finally` 블록을 사용하여 처리할 수 있습니다.

예외 핸들러를 정의하려면 보호하려는 코드를 예외 처리 블록 안에 포함시키고 블록의 `except` 부분에 예외 처리 문장을 지정합니다. 전형적인 예외 처리 블록의 형태는 다음과 같습니다.

```
try
  { statements you want to protect }
except
  { exception-handling statements }
end;
```

애플리케이션은 예외가 `try` 부분의 문장이 실행되는 동안에 발생하는 경우에만 `except` 부분의 문장을 실행합니다. `try` 부분 문장의 실행에는 `try` 부분의 코드에 의해 호출되는 루틴이 포함됩니다. 즉, `try` 부분의 코드가 자체 예외 핸들러를 정의하지 않는 루틴을 호출하면 예외 처리 블록으로 돌아와서 실행됩니다.

`try` 부분의 문장이 예외를 발생시키면 즉시 `except` 부분으로 건너뛰어 실행되고 여기서 현재 예외에 적용되는 핸들러를 찾을 때까지 특정 예외 처리 문장 또는 예외 핸들러를 통한 단계를 거칩니다.

일단 애플리케이션이 예외를 처리하는 예외 핸들러를 찾게 되면 그 문장을 실행한 후 예외 객체를 자동적으로 소멸시킵니다. 그러면 현재 블록의 끝에서 계속 실행됩니다.

예외 처리 문장

`try..except` 블록에서 `except` 부분의 각 `on` 문장들은 특별한 유형의 예외 처리 코드를 정의합니다. 이러한 예외 처리 문장의 형태는 다음과 같습니다.

```
on <type of exception> do <statement>;
```

예 0으로 나누기에 대한 기본 결과를 제공하는 예외 핸들러를 다음과 같이 정의할 수 있습니다.

```
function GetAverage(Sum, NumberOfItems:Integer):Integer;
begin
  try
    Result := Sum div NumberOfItems;{ handle the normal case }
  except
    on EDivByZero do Result := 0;{ handle the exception only if needed }
  end;
end;
```

함수를 호출할 때마다 0에 대한 검사를 하는 것보다는 위와 같이 하는 것이 더 명확합니다. 다음은 예외를 이용하지 않는 동일한 함수입니다.

```
function GetAverage(Sum, NumberOfItems:Integer):Integer;
begin
  if NumberOfItems <> 0 then{ always test }
    Result := Sum div NumberOfItems{ use normal calculation }
  else Result := 0;{ handle exceptional case }
end;
```

위의 두 가지 함수의 차이는 예외를 가진 프로그래밍과 갖지 않은 프로그래밍 간의 차이를 정의합니다. 이러한 예는 매우 간단하지만 수십 개의 입력한 내용 중 하나가 잘못되면 수백 개의 단계 중의 하나가 실패할 수 있는 경우와 같은 더 복잡한 계산을 생각해 볼 수 있습니다.

예외를 사용하면 알고리즘의 "정상적인" 표현식을 모두 작성한 다음 이 알고리즘이 적용되지 않는 예외적인 경우를 만들 수 있습니다. 예외를 사용하지 않으면 프로그램의 매 단계를 검사하여 계산의 각 단계를 진행할 수 있는지 확인해야 합니다.

예외 인스턴스 사용

대부분의 경우 예외 핸들러는 타입 이외의 예외에 대한 정보를 필요로 하지 않으므로 다음과 같은 `on..do` 문장은 예외의 타입에만 한정됩니다. 그러나 어떤 경우에는 예외 인스턴스에 포함된 정보가 다소 필요할 수도 있습니다.

예외 핸들러에서 예외 인스턴스에 대한 특정 정보를 읽으려면 예외 인스턴스에 액세스할 수 있게 해주는 `on..do`의 특별한 변형을 사용합니다. 이 특별한 폼에는 인스턴스를 유지하기 위한 임시 변수를 제공해야 합니다.

예 단일 폼을 포함하는 새 프로젝트를 만들 경우, 스크롤 막대와 명령 버튼을 폼에 추가할 수 있습니다. 버튼을 더블 클릭하고 클릭 이벤트 핸들러에 다음 줄을 추가합니다.

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

스크롤 막대의 최대값이 최소값을 항상 초과해야 하기 때문에 이 줄은 예외를 발생시킵니다. 애플리케이션에 대한 기본 예외 핸들러는 예외 객체에 메시지를 포함하는 대화 상자를 엽니다. 이 핸들러에서는 예외 처리를 오버라이드할 수 있으며 예외 메시지 문자열을 포함하는 자신의 메시지 상자를 만들 수 있습니다.

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
```

```

except
  on E:EInvalidOperation do
    MessageDlg('Ignoring exception:' + E.Message, mtInformation, [mbOK], 0);
  end;

```

임시 변수(이 예에서 E)는 콜론 뒤에 지정된 타입입니다(이 예에서 *EInvalidOperation*). 필요한 경우에 예외를 더 특정한 타입으로 타입 변환하기 위해 as 연산자를 사용할 수 있습니다.

참고 임시 예외 객체를 절대 소멸시키지 마십시오. 예외 처리는 자동으로 예외 객체를 소멸시킵니다. 사용자가 직접 객체를 소멸시키면 애플리케이션이 객체를 다시 소멸하여 액세스 위반을 일으킵니다.

예외 핸들러의 범위

모든 블록에 포함된 모든 종류의 예외에 핸들러를 제공할 필요는 없습니다. 실제로 특정 블록 안에서 특별하게 처리하고자 하는 예외에 대한 핸들러만 있으면 됩니다.

블록이 특정 예외를 처리하지 않으면 그 블록을 떠나 예외를 발생시킨 채 블록을 호출한 코드가 들어 있는 블록으로 돌아가서 실행됩니다. 이 과정은 실행이 애플리케이션의 가장 바깥 범위에 도달하거나 블록이 어느 정도 예외를 처리할 때까지 범위를 넓혀 반복합니다.

기본 예외 핸들러 제공

하나의 기본 예외 핸들러를 제공하여 특정 핸들러를 제공하지 않은 모든 예외를 처리할 수 있습니다. 이를 위해 예외 처리 블록의 **except** 부분에 **else** 부분을 추가합니다.

```

try
  { statements }
except
  on ESomething do
    { specific exception-handling code };
  else
    { default exception-handling code };
end;

```

기본 예외 처리를 블록에 추가하면 블록이 어떤 방법으로든지 모든 예외를 처리해 주므로 포함된 블록의 모든 처리를 오버라이드합니다.

주의 모든 것을 다 처리하는 기본 예외 핸들러는 사용하지 않는 것이 좋습니다. **else** 절은 개발자가 모르고 있는 것까지 포함하여 모든 예외를 처리합니다. 코드는 일반적으로 처리 방법을 알고 있는 예외만을 처리해야 합니다. 클린업을 처리한 다음 예외와 그 처리 방법에 대해 더 많은 정보를 가지고 있는 코드가 예외를 처리하게 하려면 다음과 같이 **try..finally** 블록을 포함시키는 방법을 사용합니다.

```

try
  try
    { statements }
  except
    on ESomething do { specific exception-handling code };
  end;
finally
  {cleanup code };
end;

```

다른 예외 처리 방법은 예외 재발생을 참조하십시오.

예외 클래스 다루기

예외 객체는 계층 구조의 일부이기 때문에 예외 객체 부분의 조상인 예외 클래스에 대한 핸들러를 제공함으로써 계층 구조의 전체 부분에 대한 핸들러를 지정할 수 있습니다.

예 다음 블록은 모든 정수 수학 예외를 처리하는 예를 보여 줍니다.

```
try
  { statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

좀더 특정한 예외에 대한 특정 핸들러를 지정할 수 있지만 이러한 핸들러는 일반적인 핸들러 위에 위치해야 합니다. 왜냐하면 애플리케이션은 핸들러를 나타난 순서대로 찾기 때문에 처음 찾는 적용 가능한 핸들러를 실행합니다. 예를 들면, 다음 블록에서는 범위 오류에 대한 특별한 처리와 다른 모든 정수 수학 오류에 대한 처리를 제공합니다.

```
try
  { statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

*EIntError*를 위한 핸들러가 *ERangeError*를 위한 핸들러보다 먼저 오는 경우 *ERangeError*의 특정 핸들러까지 실행되지 않음을 유의해야 합니다.

예외 재발생

가끔 지역적으로 예외를 처리할 때 예외를 대체하지 않고 바깥쪽 블록에서의 처리를 원할 수 있습니다. 물론 로컬 핸들러가 예외 처리를 완료하는 경우, 예외 인스턴스를 소멸 시키므로 바깥쪽 블록의 핸들러는 절대 작동하지 않게 됩니다. 하지만 핸들러가 예외를 소멸시키는 것을 막아서 바깥쪽 핸들러에게 응답할 기회를 줄 수 있습니다.

예 예외가 발생할 때 사용자에게 메시지를 표시하거나 오류를 로그 파일에 기록한 후에 표준 처리를 진행하길 원할 수 있습니다. 그렇게 하려면 메시지를 보여 주는 로컬 예외 핸들러를 선언하고 나서 예약어 `raise`를 호출합니다. 다음 예에서 예외 재발생을 보여 줍니다.

```
try
  { statements }
  try
    { special statements }
  except
    on ESomething do
      begin
        { handling for only the special statements }
        raise; { reraise the exception }
      end;
    end;
  except
    on ESomething do ...; { handling you want in all cases }
  end;
```

{ statements } 부분에 있는 코드가 *ESomething* 예외를 발생시키면 바깥에 있는 **except** 부분의 핸들러만 실행됩니다. 하지만 { special statements } 부분에 있는 코드가 *ESomething* 예외를 발생시키면 안쪽 **except** 부분에 있는 처리가 실행되고 나서 바깥 **except** 부분의 보다 일반적인 처리가 실행됩니다.

예외를 재발생시킴으로써 기존 핸들러를 잃거나 복제하지 않고 특별한 경우의 예외에 대한 특별한 처리를 쉽게 만들 수 있습니다.

컴포넌트 예외 처리

Delphi의 컴포넌트는 오류 상황을 나타내기 위해 예외를 발생시킵니다. 대부분의 컴포넌트 예외는 이것이 발생하지 않으면 런타임 오류를 생성시킬 프로그래밍 오류를 나타냅니다. 컴포넌트 예외 처리 메커니즘은 RTL 예외 처리와 같습니다.

예 컴포넌트에서 오류의 공통적인 원인은 인덱스된 속성에서의 범위 오류입니다. 예를 들어, 리스트 박스의 목록에 세 개의 항목(0..2)이 있고 애플리케이션이 항목 번호 3에 액세스하려 할 경우, 리스트 박스에서 "List index out of bounds" 예외를 발생시킵니다.

다음 이벤트 핸들러는 사용자에게 리스트 박스의 잘못된 인덱스 액세스를 공지하는 예외 핸들러를 포함합니다.

```

procedure TForm1.Button1Click(Sender:TObject);
begin
  ListBox1.Items.Add('a string');{ add a string to list box }
  ListBox1.Items.Add('another string');{ add another string... }
  ListBox1.Items.Add('still another string');{ ...and a third string }
  try
    Caption := ListBox1.Items[3];{ set form caption to fourth string in list box }
  except
    on EStringListError do
      MessageDlg('List box contains fewer than four strings', mtWarning, [mbOK], 0);
    end;
  end;

```

버튼을 한 번 클릭하면 리스트 박스에는 세 개의 문자열만 있으므로 네 번째 문자열(Items[3])에 액세스하면 예외가 발생합니다. 두 번째로 버튼을 클릭하면 더 많은 문자열을 목록에 추가하므로 더 이상 예외를 일으키지 않습니다.

외부 소스를 사용한 예외 처리

*HandleException*은 애플리케이션의 예외의 기본 처리를 제공합니다. 일반적으로 크로스 플랫폼 애플리케이션 개발 시에는 *TApplication.HandleException*을 호출할 필요가 없습니다. 그렇지만 공유 객체 파일 또는 callback 함수를 작성할 때에는 필요합니다. 특히 예외를 지원하지 않는 외부 소스로부터 코드가 호출되고 있을 때 예외가 코드에서 벗어나지 않도록 하기 위해 *TApplication.HandleException*을 사용할 수 있습니다.

예를 들어, 예외가 애플리케이션 코드에 있는 모든 **try** 블록을 통과하면 애플리케이션은 자동적으로 *HandleException* 메소드를 호출하고, 이 메소드는 오류가 발생했다는 것을 나타내는 대화 상자를 표시합니다. 다음과 같은 방법으로 *HandleException*을 사용할 수 있습니다.


```

try
  { statements }
except
  Application.HandleException(Self);
end;

```

*EAbort*를 제외한 모든 예외에 대해 *HandleException*은 *OnException* 이벤트 핸들러가 있을 경우 그것을 호출합니다. 따라서 예외도 처리하고 기본 제공된 컴포넌트에서 하는 이러한 기본 동작을 제공하려면 다음과 같이 사용자의 코드에 *HandleException*에 대한 호출을 추가합니다.

```

try
  { special statements }
except
  on ESomething do
  begin
    { handling for only the special statements }
    Application.HandleException(Self); { call HandleException }
  end;
end;

```

참고 스레드의 예외 처리 코드 내에서 *HandleException*을 호출하지 마십시오. 자세한 내용은 도움말 색인에서 "예외 처리 루틴"을 참조하십시오.

예외 숨기기(Silent exceptions)

Delphi 애플리케이션은 예외 객체에서 메시지 문자열을 보여 주는 메시지를 표시함으로써 사용자의 코드가 특별하게 다루지 않은 대부분의 예외를 처리합니다. 또한 기본적으로 애플리케이션이 오류 메시지를 보여 주지 않도록 하는 예외 "숨기기"를 정의할 수 있습니다.

예외 숨기기는 예외를 사용자에게 보고하지 않고 작업을 중지하고자 할 때 유용합니다. 연산 중지는 *Break* 또는 *Exit* 프로시저를 사용하여 블록을 빠져 나가는 것과 유사하지만 여러 단계로 중첩된 레벨의 블록을 빠져 나가지는 못합니다.

예외 숨기기는 모두 표준 예외 타입 *EAbort*의 자손입니다. Delphi VCL 및 CLX 애플리케이션용 기본 예외 핸들러는 *EAbort*의 자손을 제외한 모든 예외들에 대해서 오류 메시지 대화 상자를 표시합니다.

참고 콘솔 애플리케이션의 경우 오류 메시지 대화 상자는 처리되지 않은 *EAbort* 예외에 표시됩니다.

간단하게 예외 숨기기를 만들 수 있는 방법은 다음과 같습니다. 수동으로 객체를 구성하는 대신 *Abort* 프로시저를 호출합니다. 그러면 *Abort*는 자동적으로 *EAbort* 예외를 발생시키면서 오류 메시지를 보여 주지 않고 현재 연산을 빠져 나갑니다.

예 다음 코드는 작업 중지에 대한 간단한 예를 보여 줍니다. 빈 리스트 박스와 버튼을 포함하는 폼에서 다음 코드를 버튼의 *OnClick* 이벤트에 연결합니다.

```

procedure TForm1.Button1Click(Sender: TObject);
var
    I:Integer;
begin
    for I := 1 to 10 do{ loop ten times }
    begin
        ListBox1.Items.Add(IntToStr(I));{ add a numeral to the list }
        if I = 7 then Abort;{ abort after the seventh one }
    end;
end;

```

사용자 고유의 예외 정의

코드를 런타임 라이브러리와 다양한 컴포넌트에 의해 생성된 예외로부터 보호하는 것 이외에 자신의 코드에서 동일한 메커니즘을 사용하여 사용자 고유의 코드에서 예외 조건을 관리할 수 있습니다.

코드에서 예외를 사용하려면 다음 단계들을 완료해야 합니다.

- 예외 객체 타입 선언
- 예외 발생

예외 객체 타입 선언

예외는 객체이므로 새로운 종류의 예외를 정의하는 것은 새 객체 타입을 선언하는 것만큼 간단합니다. 모든 객체 인스턴스를 예외로 발생시킬 수 있지만 표준 예외 핸들러는 *Exception*의 자손인 예외만 처리합니다.

새로운 예외 타입은 *Exception*이나 다른 표준 예외 중의 하나로부터 파생되어야 하는 것이 규칙입니다. 이런 방식으로 그 예외에 대한 특정 예외 핸들러가 보호하지 않는 코드 블록에서 새로운 예외를 발생시키면 표준 핸들러 중 하나가 이 예외를 대신 처리합니다.

예 예를 들면, 다음과 같은 선언을 생각해 봅시다.

```

type
    EMyException = class(Exception);

```

*EMyException*을 발생시키고 그에 대한 특정 핸들러를 제공하지 않으면 *Exception*에 대한 핸들러 또는 기본 예외 핸들러가 그 예외를 처리합니다. *Exception*에 대한 표준 처리가 발생된 예외의 이름을 표시하기 때문에 발생된 것이 새 예외라는 것을 알 수 있습니다.

예외 발생

애플리케이션에서 오류 상태를 나타내기 위해서는 예외 타입의 인스턴스를 생성하고 예약어 **raise**를 호출하는 것을 포함하는 예외를 발생시키면 됩니다.

예외를 발생시키려면 예외 객체의 인스턴스 앞에 예약어 `raise`를 호출합니다. 이런 방법으로 특별한 주소에서 오는 예외를 발생시킬 수 있습니다. 예외 핸들러가 실제로 예외를 처리할 때 예외 인스턴스를 소멸하면 예외가 완료되므로 그 작업을 직접 수행할 필요는 없습니다.

예외 주소 설정은 시스템 유닛의 `ErrorAddr` 변수를 통해 이루어집니다. 예를 들어, 예외 핸들러에 있는 `ErrorAddr`을 참조하여 오류가 발생한 장소를 사용자에게 알려 줄 수 있습니다. 또한 예외가 발생할 때 `ErrorAddr`에 나타나는 `raise` 절에 값을 지정할 수 있습니다.

경고 `ErrorAddr`에 직접 값을 할당하지 마십시오. 이것은 읽기 전용입니다.

예외의 오류 주소를 지정하려면 식별자 같은 주소 표현식 앞에 있는 예외 인스턴스 뒤에 예약어 `at`을 추가합니다.

예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
type
  EPasswordInvalid = class(Exception);
```

다음과 같이 `EPasswordInvalid`의 인스턴스를 사용하여 `raise`를 호출하면 언제든지 "password invalid" 예외를 발생시킬 수 있습니다.

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Incorrect password entered');
```

인터페이스 사용

Delphi의 **interface** 키워드를 사용하면 애플리케이션에서 인터페이스를 만들어 사용할 수 있습니다. 인터페이스는 오브젝트 파스칼의 단일 상속 모델을 확장하는 방법으로써 단일 클래스를 사용하여 둘 이상의 인터페이스를 구현할 수 있고 다른 기반의 자손인 여러 클래스를 사용하여 동일한 인터페이스를 공유할 수 있습니다. 인터페이스는 스트리밍과 같은 동일한 집합의 작업을 광범위한 객체에 걸쳐 사용할 때 유용합니다. 인터페이스는 또한 COM(컴포넌트 객체 모델, Component Object Model)과 CORBA(공통 객체 요청 브로커 아키텍처, Common Object Request Broker Architecture) 분산 객체 모델의 기본입니다.

랭귀지 기능의 인터페이스

인터페이스는 추상 메소드와 기능의 명확한 정의만 포함하는 클래스와 유사합니다. 엄밀하게 말하면 인터페이스 메소드 정의에는 매개변수의 개수와 타입, 반환 타입, 기대 동작 등이 포함됩니다. 인터페이스 메소드의 이름은 대개 인터페이스의 목적을 나타내도록 지정됩니다. 인터페이스의 이름을 그 동작에 따라 지정하고 대문자 `I`로 시작하는 것이 규칙입니다. 예를 들어, `IMalloc` 인터페이스는 메모리를 할당, 해제, 관리합니다. 이와 마찬가지로 `IPersist` 인터페이스는 자손들에 대한 일반적인 기본 인터페이스로 사용될 수 있으며 각 자손은 객체의 상태를 저장 장치나 스트림 또는 파일에 저장하고 로드하는 특정 메소드의 프로토타입을 정의합니다.

인터페이스는 다음과 같은 구문을 갖습니다.

```

IMyObject = interface
  Procedure MyProcedure;
end;

```

다음은 인터페이스를 선언하는 간단한 예입니다.

```

type
  IEdit = interface
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo:Boolean; stdcall;
  end;

```

추상 클래스와 같이 인터페이스 자체는 결코 인스턴스화될 수 없습니다. 인터페이스를 사용하려면 구현하는 클래스에서 인터페이스를 얻어야 합니다.

인터페이스를 구현하려면 조상 목록에서 인터페이스를 선언하는 클래스를 정의해야 하며 해당 인터페이스의 모든 메소드를 구현해야 합니다.

```

TEditor = class(TInterfacedObject, IEdit)
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo:Boolean; stdcall;
end;

```

인터페이스는 메소드의 동작과 시그니처를 정의하지만 구현을 정의하지는 않습니다. 클래스에서 이를 실제적으로 구현하게 되는데 클래스의 구현이 인터페이스 정의에 따르는 한 인터페이스는 완전한 다형성을 가지며 인터페이스의 액세스와 사용이 인터페이스의 모든 구현에 있어 동일하다는 것을 의미합니다.

클래스 간의 인터페이스 공유

인터페이스를 사용하면 클래스 사용 방법과 클래스 구현 방법을 구분하는 설계 방법을 제공합니다. 두 가지 클래스는 동일한 기본 클래스의 자손이 아니라도 동일한 인터페이스를 구현할 수 있습니다. 관련이 없는 객체에 대한 동일한 메소드의 이러한 다형적 호출은 객체가 동일한 인터페이스를 구현하는 한 가능합니다. 예를 들어, 다음과 같은 인터페이스를 생각해 봅시다.

```

IPaint = interface
  procedure Paint;
end;

```

그리고 다음 두 가지 클래스도 생각해 봅시다.

```

TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;

```

두 클래스가 공통 조상을 공유하는지 여부에 관계 없이 두 클래스는 다음과 같이 *IPaint*의 변수와 여전히 호환하는 할당입니다.

```
var
  Painter:IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

위의 예는 가상 메소드 *Paint*를 구현하는 *TFigure*의 자손인 *TCircle*과 *TSquare*를 가짐으로써 이루어질 수 있습니다. 그런 다음 *TCircle*과 *TSquare*가 *Paint* 메소드를 오버라이드합니다. 위의 예에서 *IPaint*는 *TFigure*로 대체할 수 있습니다. 하지만 다음과 같은 인터페이스를 생각해 봅시다.

```
IRotate = interface
  procedure Rotate(Degrees:Integer);
end;
```

여기서는 원이 아닌 사각형을 지원하도록 하는 것이 바람직합니다. 클래스는 다음과 같습니다.

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

나중에 *TFilledCircle* 클래스를 만들 수 있으며 *IRotate* 인터페이스를 구현하면 원에 회전을 추가하지 않고 원을 채우는 데 사용되는 패턴을 회전할 수 있습니다.

참고 이러한 예에서는 직계 기본 클래스나 조상 클래스가 참조 카운팅을 관리하는 *IInterface* 메소드를 구현한 것으로 가정할 수 있습니다. 자세한 내용은 4-20 페이지의 "IInterface 구현" 및 4-24 페이지의 "인터페이스 객체의 메모리 관리"를 참조하십시오.

프로시저와 함께 인터페이스 사용

인터페이스는 또한 객체가 특별한 기본 클래스의 자손이 아니라도 객체를 처리할 수 있는 일반적인 프로시저를 작성할 수 있게 합니다. 위의 *IPaint*와 *IRotate* 인터페이스를 사용하여 다음과 같은 프로시저를 작성할 수 있습니다.

```
procedure PaintObjects(Painters: array of IPaint);
var
  I:Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;

procedure RotateObjects(Degrees: Integer; Rotaters:array of IRotate);
var
```

```

I:Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
  end;

```

*RotateObjects*를 사용하면 객체를 저절로 그려지게 하지 않아도 되며 *PaintObjects*를 사용하면 객체를 회전시키는 방법을 몰라도 됩니다. 이러한 이유로 *TFigure* 클래스에 대해서만 작동하도록 작성된 경우보다 위의 일반적인 프로시저가 더 자주 사용됩니다.

인터페이스의 구문, 랭귀지 정의 및 규칙에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말의 Object interfaces 단원을 참조하십시오.

Interface 구현

모든 인터페이스는 *IInterface* 인터페이스로부터 직접적 또는 간접적으로 파생됩니다. 이 인터페이스는 동적 쿼리와 수명 관리(lifetime management)라는 핵심적인 인터페이스 기능을 제공합니다. 이러한 기능은 다음 세 가지 *IInterface* 메소드를 통해 실현됩니다.

- *QueryInterface*는 주어진 객체에 동적으로 쿼리하고 그 객체가 지원하는 인터페이스에 대한 인터페이스 참조를 얻을 수 있는 메소드를 제공합니다.
- *_AddRef*는 *QueryInterface*로의 호출이 성공할 때마다 횡수를 증가시키는 참조 카운팅 메소드입니다. 참조 카운트가 0이 아니면 객체는 메모리에 남아 있게 됩니다.
- *_Release*는 *_AddRef*와 함께 객체가 자신의 수명을 추적하고 객체 자체를 안전하게 삭제하는 시기를 결정하는 데 사용됩니다. 참조 카운트가 일단 0이 되면 객체는 메모리에서 해제됩니다.

인터페이스를 구현하는 모든 클래스는 다른 조상 인터페이스가 선언한 모든 메소드 및 인터페이스 자체가 선언한 모든 메소드뿐만 아니라 위의 세 가지 *IInterface* 메소드도 구현해야 합니다. 그러나 개발자는 클래스에서 선언한 인터페이스의 메소드 구현을 상속할 수 있습니다.

자신이 직접 이러한 메소드들을 구현함으로써 수명 관리의 대체 수단을 제공하여 참조 카운팅 사용 불가능으로 설정할 수 있습니다. 이 방법은 인터페이스를 참조 카운팅에서 분리할 수 있는 강력한 기법입니다.

TInterfacedObject

Delphi는 *IInterface*의 메소드를 구현하기 때문에 편리하게 기본 클래스로 사용할 수 있는 일반(simple) 클래스 *TInterfacedObject*를 정의합니다. *TInterfacedObject* 클래스는 다음과 같이 *시스템* 유닛에 선언됩니다.

```

type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount:Integer;
    function QueryInterface(const IID:TGUID; out Obj): HRESULT; stdcall;
    function _AddRef:Integer; stdcall;

```

```

function _Release:Integer; stdcall;
public
  procedure AfterConstruction; override;
  procedure BeforeDestruction; override;
  class function NewInstance: TObject; override;
  property RefCount: Integer read FRefCount;
end;

```

*TInterfacedObject*에서 직접 파생할 수 있습니다. 다음과 같은 선언에서 *TDerived*는 *TInterfacedObject*의 직접적인 자손이고 가상적인 *IPaint* 인터페이스를 구현합니다.

```

type
  TDerived = class(TInterfacedObject, IPaint)
    ...
  end;

```

여기서 *IInterface*의 메소드를 구현하기 때문에 *TInterfacedObject*는 자동적으로 참조 카운팅과 인터페이스 객체의 메모리 관리를 처리합니다. 자세한 내용은 4-24 페이지의 "인터페이스 객체의 메모리 관리"를 참조하십시오. 여기서도 인터페이스를 구현하는 사용자 고유의 클래스 작성에 대해 설명하고 있지만 *TInterfacedObject*에서의 참조 카운팅 메커니즘을 따르지는 않습니다.

as 연산자 사용

인터페이스를 구현하는 클래스는 **as** 연산자를 인터페이스에서의 동적 바인딩에 사용할 수 있습니다. 다음과 같은 예를 들 수 있습니다.

```

procedure PaintObjects(P:TinterfacedObject)
var
  X:IPaint;
begin
  X := P as IPaint;
  { statements }
end;

```

여기서 *TInterfacedObject* 타입의 변수 *P*는 *IPaint* 인터페이스 참조인 변수 *X*에 할당될 수 있습니다. 동적 바인딩은 이러한 할당을 가능하게 만듭니다. 컴파일러는 이러한 할당에 대해 *P*의 *IInterface* 인터페이스의 *QueryInterface* 메소드를 호출하는 코드를 생성합니다. 왜냐하면 컴파일러는 *P*의 선언 타입으로는 *P*의 인스턴스가 실제로 *IPaint*를 지원하는지 알 수 없기 때문입니다. 런타임에 *P*는 *IPaint* 참조로 확인되거나 그렇지 않으면 예외가 발생합니다. 어떤 경우라도 *P*를 *X*에 할당하는 것은 *P*가 *IInterface*를 구현하지 않은 클래스 타입일 때처럼 컴파일 타임 오류를 발생시키지 않습니다.

인터페이스에서 **as** 연산자를 동적 바인딩에 사용하는 경우, 다음과 같은 요구 사항을 알고 있어야 합니다.

- 명시적인 *IInterface* 선언: 모든 인터페이스가 *IInterface*에서 파생된다 하더라도 **as** 연산자를 사용하고자 한다면 클래스가 단순히 *IInterface*의 메소드를 구현하는 것으로는 불충분합니다. 이러한 상황은 클래스가 명시적으로 선언한 인터페이스를 구현하는 경우에도 마찬가지입니다. 클래스는 인터페이스 목록 안에 *IInterface*를 명시적으로 선언해야 합니다.

- IID 사용: 인터페이스는 GUID(globally unique identifier)에 기반한 식별자를 사용할 수 있습니다. GUID는 인터페이스를 식별하는 데 사용되므로 인터페이스 식별자(IID)라고 합니다. 인터페이스에서 **as** 연산자를 사용하고 있을 경우에는 연결된 IID가 있어야 합니다. 소스 코드에서 새 GUID를 생성하려면 *Ctrl+Shift+G* 편집 단축키를 사용합니다.

코드 재사용 및 위임(Delegation)

인터페이스와 함께 코드를 재사용하는 한 가지 방법은 어떤 객체를 포함하거나 다른 객체에 포함된 객체를 갖도록 하는 것입니다. 객체 타입인 속성을 포함 관계와 코드 재사용에 대한 방법으로 사용합니다. 인터페이스를 위한 이러한 설계를 지원하기 위해서 오브젝트 파스칼에는 하위 객체에 대한 인터페이스의 전부 또는 일부를 위임하는 코드를 쉽게 작성할 수 있는 키워드 **implements**가 들어 있습니다. 추상화는 포함과 위임을 통해 코드를 재사용하는 또 다른 방법입니다. 추상화에서 외부 객체는 외부 객체에 의해서만 노출되는 인터페이스를 구현하는 내부 객체를 포함합니다. VCL과 CLX는 추상화를 지원하는 클래스를 가지고 있습니다.

위임을 위한 implements 사용

많은 클래스들은 하위 객체인 속성을 갖습니다. 인터페이스를 속성 타입으로 사용할 수도 있습니다. 속성이 인터페이스 타입 또는 인터페이스의 메소드를 구현하는 클래스 타입일 경우, 키워드 **implements**를 사용하여 그 인터페이스의 메소드가 속성 인스턴스인 인터페이스 참조 또는 객체에 위임되는지 지정할 수 있습니다. 위임은 메소드에 대한 구현만 제공하면 됩니다. 인터페이스 지원을 선언할 필요는 없습니다. 속성을 포함하는 클래스는 조상 목록에 인터페이스를 포함해야 합니다.

기본적으로 키워드 **implements**를 사용하면 모든 인터페이스 메소드를 위임합니다. 하지만 메소드 확인절을 사용하거나 클래스에서 이러한 기본 동작을 오버라이드하기 위해서 일부 인터페이스 메소드를 구현하는 메소드를 선언할 수 있습니다.

다음 예제는 8비트 RGB 색상 값을 *Color* 참조로 변환하는 색상 어댑터 객체 설계에서 **implements** 키워드를 사용합니다.

```
unit cadapt;

type
  IRGB8bit = interface
    ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red:Byte;
    function Green:Byte;
    function Blue:Byte;
  end;

  IColorRef = interface
    ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color:Integer;
  end;

{ TRGB8ColorRefAdapter  map an IRGB8bit to an IColorRef }
TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
private
```



```

FRGB8bit:IRGB8bit;
FPalRelative:Boolean;
public
  constructor Create(rgb:IRGB8bit);
  property RGB8Intf:IRGB8bit read FRGB8bit implements IRGB8bit;
  property PalRelative:Boolean read FPalRelative write FPalRelative;
  function Color:Integer;
end;

implementation

constructor TRGB8ColorRefAdapter.Create(rgb:IRGB8bit);
begin
  FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color:Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
  end;
end.

```

implements 키워드의 구문, 구현 세부 사항, 랭귀지 규칙 등에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말의 Object interfaces 단원을 참조하십시오.

추상화(Aggregation)

추상화는 자신을 포함하는 객체의 기능을 정의하지만 해당 객체의 구현 세부 사항을 숨기는 하위 객체를 통해 코드를 재사용하는 모듈화된 방법을 제공합니다. 추상화에서 외부 객체는 하나 이상의 인터페이스를 구현합니다. 유일한 필수 작업 사항은 *IInterface*를 구현하는 것입니다. 내부 객체나 객체는 하나 이상의 인터페이스를 구현하지만 외부 객체만이 인터페이스를 노출시킵니다. 이러한 인터페이스에는 외부 객체가 구현하는 인터페이스와 포함된 객체에 의해 구현되는 인터페이스가 있습니다. 클라이언트는 내부 객체에 대해 전혀 모릅니다. 외부 객체는 내부 객체 인터페이스에 대한 액세스를 제공하면서도 자신의 구현은 완전히 투명합니다. 따라서 외부 객체 클래스는 동일한 인터페이스를 구현하는 모든 클래스의 내부 객체 클래스 타입을 교환할 수 있습니다. 이에 상응하여 내부 객체 클래스의 코드는 이를 사용하려는 다른 클래스가 공유할 수 있습니다.

추상화를 위한 구현 모델은 위임을 사용하여 *IInterface*를 구현하기 위한 명시적인 규칙을 정의합니다. 내부 객체는 자신에 대해 내부 객체의 참조 카운트를 제어하는 *IInterface*를 구현합니다. 이러한 *IInterface*의 구현은 외부와 내부 객체 간의 관계를 추적합니다. 예를 들어 해당 타입의 객체(내부 객체)가 만들어질 때 이러한 생성은 *IInterface* 타입의 요청된 인터페이스에 대해서만 성공합니다. 내부 객체는 또한 자신이 구현하는 모든 인터페이스에 대해 두 번째 *IInterface*를 구현합니다. 이들이 외부 객체에 의해 노출되는 인터페이스입니다. 이 두 번째 *IInterface*는 *QueryInterface*, *AddRef* 및 *Release*로의 호출을 외부 객체로 위임합니다. 외부 *IInterface*는 "Unknown 제어"라고 합니다.

추상화를 만드는 규칙에 대한 자세한 내용은 MS 온라인 도움말을 참조하십시오. 자기 자신의 추상 클래스를 작성할 때에는 *TComObject* 내의 *IInterface*의 구현 세부 사항도 참조하십시오. *TComObject*는 추상화를 지원하는 COM 클래스입니다. COM 애플리케이션을 작성하는 경우 *TComObject*를 기본 클래스로 직접 사용할 수도 있습니다.

인터페이스 객체의 메모리 관리

인터페이스 디자인의 목적 중 하나는 인터페이스를 구현하는 객체의 수명 관리를 확실히 하는 데 있습니다. *IInterface*의 *_AddRef* 메소드와 *_Release* 메소드는 이러한 수명 관리를 구현하는 방법을 제공합니다. *_AddRef*와 *_Release*는 인터페이스 참조가 클라이언트에게 전달될 때 객체에서 참조 카운트를 증가시켜 객체의 수명을 추적하고 참조 카운트가 0일 때 객체를 소멸시킵니다.

분산 애플리케이션의 COM 객체를 작성하는 경우(Windows 환경에만 해당) 참조 카운팅 규칙을 엄격히 준수해야 합니다. 하지만 애플리케이션 내에서만 인터페이스를 사용하는 경우 객체의 성격과 사용 방법 결정에 따라 선택권이 있습니다.

참조 카운팅 사용

Delphi는 인터페이스 쿼리와 참조 카운팅의 구현을 통해 대부분의 *IInterface* 메모리 관리를 개발자 대신 제공합니다. 따라서 인터페이스에 의해 존재하고 소멸하는 객체를 가지고 있는 경우, 이러한 클래스로부터 파생시켜서 참조 카운팅을 쉽게 사용할 수 있습니다. *TInterfacedObject*는 이러한 동작을 제공하는 non-CoClass입니다. 참조 카운팅을 사용하기로 결정했다면 객체를 인터페이스 참조로만 유지하고 참조 카운팅이 일관성을 갖도록 유의해야 합니다. 예를 들면,

```
procedure beep(x:ITest);
function test_func()
var
  y:ITest;
begin
  y := TTest.Create; // because y is of type ITest, the reference count is one
  beep(y); // the act of calling the beep function increments the reference count
            // and then decrements it when it returns
  y.something; // object is still here with a reference count of one
end;
```

위의 예는 메모리 관리에 대한 가장 깨끗하고 안전한 방법이고 *TInterfacedObject*를 사용하는 경우에는 자동으로 처리됩니다. 이러한 규칙을 따르지 않으면 객체가 갑자기 사라질 수 있습니다. 다음 코드는 이에 대한 예제입니다.

```
function test_func()
var
  x:TTest;
begin
  x := TTest.Create; // no count on the object yet
  beep(x as ITest); // count is incremented by the act of calling beep
                    // and decremented when it returns
  x.something; // surprise, the object is gone
end;
```

참고 위의 예에서 *beep* 프로시저는 선언된 대로 매개변수에 대한 참조 카운트를 증가시키지만 (*_AddRef* 호출) 다음 선언에서는 증가시키지 않습니다.

```
procedure beep(const x: ITest);
```

또는

```
procedure beep(var x: ITest);
```

이러한 선언은 더 작고 더 빠른 코드를 생성합니다.

참조 카운팅을 일관성 있게 적용할 수 없어서 사용할 수 없는 경우가 있다면 객체가 컴포넌트이거나 다른 컴포넌트에 의해 소유된 컨트롤인 경우입니다. 그러한 상황에서는 인터페이스를 사용할 수도 있지만 그 객체의 수명이 인터페이스에 의해 결정되지 않기 때문에 참조 카운팅을 사용하지 않아야 합니다.

참조 카운팅을 사용하지 않음

객체가 컴포넌트이거나 다른 컴포넌트가 소유한 컨트롤인 경우, 객체는 *TComponent*에 기반한 다른 메모리 관리 시스템의 일부가 됩니다. VCL 및 CLX 컴포넌트의 객체 수명 관리 방법과 인터페이스 참조 카운팅을 혼합해서는 안 됩니다. 인터페이스를 지원하는 컴포넌트를 만들고자 한다면 다음과 같이 *IInterface* *_AddRef* 및 *_Release* 메소드를 빈 함수로 구현하여 인터페이스 참조 카운팅 메커니즘을 회피할 수 있습니다.

```
function TMyObject._AddRef: Integer;
begin
  Result := -1;
end;

function TMyObject._Release: Integer;
begin
  Result := -1;
end;
```

이 경우에도 여전히 *QueryInterface*를 구현하여 객체에 대한 동적 쿼리를 제공할 것입니다.

주목할 점은 *QueryInterface*를 구현하기 때문에 인터페이스 식별자 (IID)를 생성하는 한 계속해서 컴포넌트에서 인터페이스를 위한 **as** 연산자를 사용할 수 있다는 것입니다. 추상화를 사용할 수도 있습니다. 외부 객체가 컴포넌트인 경우, 내부 객체는 "Unknown 제어"에게 위임함으로써 일반적인 경우처럼 참조 카운팅을 구현합니다. *_AddRef*와 *_Release* 메소드를 회피하고 컴포넌트 기반의 접근을 통해 메모리 관리를 처리하는 결정을 하는 것은 외부, 컴포넌트 객체의 수준입니다. 사실 컴포넌트를 포함한 외부 객체로 가지는, 분리의 내부 객체를 위한 기본 클래스로 *TInterfacedObject*를 사용할 수도 있습니다.

참고 "Unknown 제어"는 외부 객체에 의해 구현된 *IUnknown*이며 전체 객체의 참조 카운트를 유지하기 위한 것입니다. *IUnknown*은 *IInterface*와 동일하지만 COM 기반의 애플리케이션 (Windows 전용)에서는 *IUnknown*이 대신 사용됩니다. 내부 및 외부 객체에 의한 *IUnknown*이나 *IInterface* 인터페이스의 다양한 구현을 구분하는 자세한 내용은 23 페이지의 "추상화 (Aggregation)"와 Microsoft 온라인 도움말 항목 중 "Controlling Unknown"을 참조하십시오.

분산 애플리케이션에서 인터페이스 사용(VCL 전용)

인터페이스는 COM, SOAP 및 CORBA 분산 객체 모델에서 기본적인 요소입니다. Delphi는 *IInterface* 인터페이스 메소드를 구현하는 *TInterfacedObject*에서 기본 인터페이스 기능을 확장하는 이러한 기술을 위한 기본 클래스를 제공합니다.

COM 사용 시 클래스와 인터페이스는 *IInterface*보다는 *IUnknown*의 관점에서 정의됩니다. *IUnknown*과 *IInterface* 간에 구문 상의 차이점은 없으며 *IUnknown*을 사용하는 것은 Delphi 인터페이스를 COM 정의에 적용시키는 방법일 뿐입니다. COM 클래스는 클래스 팩토리와 클래스 식별자(CLSID) 사용을 위한 기능을 추가합니다. 클래스 팩토리는 CLSID를 통해 클래스 인스턴스를 생성합니다. CLSID는 COM 클래스의 등록과 처리에 사용됩니다. 클래스 팩토리와 클래스 식별자를 가지는 COM 클래스를 CoClasses라고 부릅니다. CoClasses는 *QueryInterface*의 버전 기능의 이점을 취하여 소프트웨어 모듈이 업데이트될 때 *QueryInterface*가 런타임 시 호출되어 객체의 현재 기능을 쿼리합니다.

모든 새로운 인터페이스나 객체의 기능은 물론 이전의 인터페이스의 새 버전도 새로운 클라이언트에 즉시 사용할 수 있습니다. 동시에 객체는 기존 클라이언트 코드와의 완전한 호환성을 유지하며 인터페이스 구현이 숨겨져 있으면서도 메소드와 매개변수는 상수인 채로 남아 있어 체크파일이 필요하지 않습니다. COM 애플리케이션에서 개발자는 성능을 향상시키거나 내부적인 이유로 해당 인터페이스에 의존하는 모든 클라이언트 코드를 손상시키지 않고 구현을 변경할 수 있습니다. COM 인터페이스에 대한 자세한 내용은 33장 "COM 기술 개요"를 참조하십시오.

SOAP를 이용하여 애플리케이션을 분산하는 경우, 인터페이스는 자신의 런타임 타입 정보(RTTI)를 가지고 있어야 합니다. {\$M+} 스위치를 사용하여 컴파일 시 컴파일러에서는 RTTI를 인터페이스에 추가할 뿐입니다. 이러한 인터페이스를 *호출 가능한 인터페이스*라고 합니다. 호출 가능한 인터페이스의 자손도 호출할 수 있습니다. 하지만 호출 가능한 인터페이스가 호출이 불가능한 다른 인터페이스의 자손인 경우 클라이언트 애플리케이션은 호출 가능한 인터페이스와 해당 자손에 정의된 메소드를 호출만 할 수 있습니다. 호출 불가능한 조상으로부터 상속된 메소드는 타입 정보와 함께 컴파일되지 않으므로 클라이언트에 의해 호출될 수 없습니다.

호출 가능한 인터페이스를 정의하는 가장 쉬운 방법은 인터페이스가 *IInvokable*의 자손이 되도록 정의하는 것입니다. *IInvokable*은 {\$M+} 스위치를 사용하여 컴파일된다는 점만 제외하면 *IInterface*와 같습니다. SOAP를 사용하여 분산된 웹 서비스 애플리케이션과 호출 가능한 인터페이스에 대한 자세한 내용은 31장 "Web Services 사용"을 참조하십시오.

또 다른 분산 애플리케이션 기술은 CORBA입니다. CORBA 애플리케이션에서의 인터페이스 사용은 클라이언트상의 stub 클래스와 서버 상의 skeleton 클래스에 의해 중재됩니다. 이러한 stub 및 skeleton 클래스는 마샬링 인터페이스 호출의 세부 사항을 처리하여 매개변수 값과 반환 값이 올바르게 전송되도록 합니다. 애플리케이션은 stub이나 skeleton 클래스를 사용하거나 모든 매개변수를 자신의 타입 정보를 지닐 수 있게 특수한 가변으로 변환시키는 동적 호출 인터페이스(DII)를 채택해야 합니다.

사용자 지정 가변 정의

오브젝트 파스칼 랭귀지의 기본 제공 타입 중 강력한 것은 가변 타입입니다. 가변은 컴파일 시 타입이 정해지지 않은 값을 나타냅니다. 그 대신 해당 값의 타입이 런타임 시 변할 수 있습니다. 가변 타입은 다른 가변 타입 및 표현식과 할당문의 정수, 실수, 문자열, 부울 값과 혼합할 수 있습니다. 컴파일러에서 자동으로 타입을 변환합니다.

기본적으로 가변은 레코드, 집합, 정적 배열, 파일, 클래스, 클래스 참조 또는 포인터의 값을 가질 수 없습니다. 하지만 이러한 타입과 함께 사용할 수 있도록 가변 타입을 확장할 수 있습니다. 확장을 위해서는 가변 타입이 표준 작업을 수행하는 방법을 나타내는 *TCustomVariantType* 클래스의 자손을 만들기만 하면 됩니다.

다음과 같은 방법으로 가변 타입을 생성합니다.

- 1 가변 데이터의 저장소를 *TVarData* 레코드에 매핑합니다.
- 2 *TCustomVariantType*의 자손인 클래스를 선언합니다. 새 클래스에 타입 변환 규칙을 포함한 모든 필요한 동작을 구현합니다.
- 3 사용자 지정 가변의 인스턴스 생성과 해당 타입 인식을 위한 유틸리티 메소드를 작성합니다.

위의 절차는 가변 타입을 확장하여 표준 연산자가 새로운 타입을 사용하고 새 가변 타입이 다른 데이터 타입으로 타입 변환될 수 있게 합니다. 새 가변 타입을 더 확장시켜 사용자가 정의하는 속성과 메소드를 지원하게 할 수 있습니다. 속성이나 메소드를 지원하는 가변 타입을 생성할 때 *TCustomVariantType*보다는 *TInvokeableVariantType*이나 *TPublishableVariantType*을 기본 클래스로 사용합니다.

사용자 지정 가변 타입 데이터 저장

가변은 *TVarData* 레코드 타입에 자신의 데이터를 저장합니다. 이 타입은 16바이트를 포함하는 레코드입니다. 처음 워드(Word)는 가변의 타입을 나타내는 데 쓰이며 나머지 14바이트는 데이터 저장에 사용할 수 있습니다. 새로운 가변 타입을 *TVarData* 레코드와 함께 직접 사용할 수 있지만 새 타입에 대한 이름을 갖는 멤버의 레코드 타입을 정의하고 새 타입을 *TVarData* 레코드 타입으로 타입 변환하는 것이 일반적으로 더 쉬운 방법입니다.

예를 들어, *VarConv* 유닛은 측정을 나타내는 사용자 지정 가변 타입을 정의합니다. 이 타입의 데이터는 값(더블)뿐만 아니라 측정 유닛(*TConvType*)을 포함합니다. 다음과 같이 *VarConv* 유닛은 자신의 타입을 정의하여 이러한 값을 나타냅니다.

```
TConvertVarData = packed record
    VType:TVarType;
    VConvType:TConvType;
    Reserved1, Reserved2:Word;
    VValue:Double;
end;
```

이 타입은 *TVarData* 레코드와 똑같은 크기입니다. 새 타입의 사용자 지정 가변을 사용하는 경우, 가변(또는 해당 *TVarData* 레코드)은 *TConvertVarData*로 타입 변환되고

사용자 지정 가변 타입은 *TConvertVarData* 타입인 것처럼 *TVarData* 레코드를 사용합니다.

참고 이러한 방법으로 *TVarData* 레코드로 매핑하는 레코드를 정의하는 경우 압축된 레코드로 정의하는지 확인하십시오.

새로운 사용자 지정 가변 타입이 자신의 데이터를 저장하기 위해 14바이트 이상을 필요로 하는 경우, 포인터나 객체 인스턴스를 포함하는 새로운 레코드 타입을 정의할 수 있습니다. 예를 들어, *VarCmplx* 유닛은 클래스 *TComplexData*의 인스턴스를 사용하여 복잡한 값으로 된 가변 내의 데이터를 나타냅니다. 따라서 *VarCmplx* 유닛은 다음과 같이 *TComplexData* 객체로의 참조를 포함하는 *TVarData*와 같은 크기의 레코드 타입을 정의합니다.

```
TComplexVarData = packed record
  VType:TVarType;
  Reserved1, Reserved2, Reserved3:Word;
  VComplex: TComplexData;
  Reserved4: LongInt;
end;
```

객체 참조는 실제로는 포인터(2워드)이므로 이 타입은 *TVarData* 레코드와 크기가 같습니다. 이전과 마찬가지로 복잡한 사용자 지정 가변 또는 가변의 *TVarData* 레코드는 *TComplexVarData*로 타입 변환될 수 있으며 사용자 지정 가변 타입은 *TComplexVarData* 타입인 것처럼 *TVarData* 레코드를 사용합니다.

사용자 지정 가변 타입을 활성화하기 위한 클래스 생성

사용자 지정 가변은 사용자 지정 타입의 가변이 표준 작업을 수행하는 방법을 나타내는 특별한 helper 클래스를 사용하여 동작합니다. 이러한 helper 클래스는 *TCustomVariantType*의 자손을 작성하여 생성할 수 있습니다. 여기에는 *TCustomVariantType*의 해당 가상 메소드를 오버라이드하는 작업이 포함됩니다.

타입 변환 활성화

사용자가 구현하는 사용자 지정 가변 타입의 가장 중요한 기능 중 하나는 타입 변환입니다. 가변의 유연성은 암시적인 타입 변환에서 부분적으로 발생합니다.

사용자 지정 가변 타입이 타입 변환을 수행할 수 있도록 사용자가 구현하는 다음의 두 가지 메소드가 있습니다. 다른 가변 타입을 사용자 지정 가변으로 변환하는 *Cast*와 사용자 지정 가변 타입을 다른 타입의 가변으로 변환하는 *CastTo*가 있습니다.

이들 메소드 구현 시 기본 제공 가변 타입으로부터 논리적 변환을 수행하는 것은 비교적 쉽습니다. 하지만 타입 변환을 하는 대상이나 원본이 다른 사용자 지정 가변 타입일 가능성을 고려해야 합니다. 이러한 상황을 처리하기 위해 중간 단계로서 기본 제공 가변 타입 중 하나로의 타입 변환을 시도할 수 있습니다.

예를 들어, *TComplexVariantType* 클래스로의 다음과 같은 *Cast* 메소드는 *Double* 타입을 중간 타입으로 사용합니다.

```
procedure TComplexVariantType.Cast(var Dest:TVarData; const Source:TVarData);
var
```

```

LSource, LTemp:TVarData;
begin
  VarDataInit(LSource);
  try
    VarDataCopyNoInd(LSource, Source);
    if VarDataIsStr(LSource) then
      TComplexVarData(Dest).VComplex := TComplexData.Create(VarDataToStr(LSource))
    else
      begin
        VarDataInit(LTemp);
        try
          VarDataCastTo(LTemp, LSource, varDouble);
          TComplexVarData(Dest).VComplex := TComplexData.Create(LTemp.VDouble, 0);
        finally
          VarDataClear(LTemp);
        end;
      end;
    Dest.VType := VarType;
  finally
    VarDataClear(LSource);
  end;
end;

```

Double 을 중간 가변 타입으로 사용하는 것에 덧붙여 이러한 구현에서 알아야 할 것이 몇 가지 있습니다.

- 이 메소드의 마지막 단계는 반환된 *TVarData* 레코드의 *VType* 멤버를 설정합니다. 이 멤버가 Variant 타입 코드를 부여합니다. 가변 타입 코드는 사용자 지정 가변에 할당된 가변 타입 코드인 *TComplexVariantType*의 *VarType* 속성으로 설정됩니다.
- 사용자 지정 가변의 데이터 (*Dest*)는 *TVarData*에서 데이터를 저장 (*TComplexVarData*) 하는 데 실질적으로 사용되는 레코드 타입으로 타입 변환됩니다. 이로 인해 데이터로 작업 하는 것이 쉬워집니다.
- 메소드는 데이터와 직접 작동하지 않고 소스 가변의 로컬 사본을 만듭니다. 이는 소스 데이터에 영향을 끼치는 부작용을 막습니다.

복잡한 가변에서 다른 타입으로 변환할 때 *CastTo* 메소드도 다음과 같이 Double의 중간 타입(문자열을 제외한 모든 대상 타입)을 사용합니다.

```

procedure TComplexVariantType.CastTo(var Dest:TVarData; const Source:TVarData;
const AVarType: TVarType);
var
  LTemp:TVarData;
begin
  if Source.VType = VarType then
    case AVarType of
      varOleStr:
        VarDataFromOleStr(Dest, TComplexVarData(Source).VComplex.AsString);
      varString:
        VarDataFromStr(Dest, TComplexVarData(Source).VComplex.AsString);
    else
      VarDataInit(LTemp);
    try

```

```

    LTemp.VType := varDouble;
    LTemp.VDouble := TComplexVarData(LTemp).VComplex.Real;
    VarDataCastTo(Dest, LTemp, AVarType);
finally
    VarDataClear(LTemp);
end;
end
else
    RaiseCastError;
end;

```

CastTo 메소드에는 소스 가변 데이터가 *VarType* 속성과 일치하는 타입 코드를 가지지 않는 경우가 있다는 점에 유의합니다. 이러한 경우는 빈(할당되지 않은) 소스 가변의 경우에만 일어납니다.

이항 연산 구현

사용자 지정 가변 타입을 표준 이항 연산자(시스템 유닛에 나열되어 있는 +, -, *, /, div, mod, shl, shr, and, or, xor)와 함께 사용하려면 *BinaryOp* 메소드를 오버라이드해야 합니다. *BinaryOp*는 세 가지 매개변수, 즉 왼쪽 피연산자의 값, 오른쪽 피연산자의 값 그리고 연산자를 갖습니다. 이 메소드를 구현하여 연산을 수행하고 왼쪽 피연산자를 포함하는 동일한 가변을 사용하여 결과를 반환합니다.

예를 들어, 다음 *BinaryOp* 메소드는 *VarCmplx* 유닛에 정의된 *TComplexVariantType*에서 옵니다.

```

procedure TComplexVariantType.BinaryOp(var Left:TVarData; const Right:TVarData;
const Operator: TVarOp);
begin
    if Right.VType = VarType then
        case Left.VType of
            varString:
                case Operator of
                    opAdd: Variant(Left) := Variant(Left) +
TComplexVarData(Right).VComplex.AsString;
                    else
                        RaiseInvalidOp;
                    end;
                end
            else
                if Left.VType = VarType then
                    case Operator of
                        opAdd:
                            TComplexVarData(Left).VComplex.DoAdd(TComplexVarData(Right).VComplex);
                        opSubtract:
                            TComplexVarData(Left).VComplex.DoSubtract(TComplexVarData(Right).VComplex);
                        opMultiply:
                            TComplexVarData(Left).VComplex.DoMultiply(TComplexVarData(Right).VComplex);
                        opDivide:
                            TComplexVarData(Left).VComplex.DoDivide(TComplexVarData(Right).VComplex);
                    else
                        RaiseInvalidOp;
                    end
                end
            else

```



```

        RaiseInvalidOp;
    end
else
    RaiseInvalidOp;
end;

```

이 구현에서 몇 가지 유의해야 할 것은 다음과 같습니다.

이 메소드는 연산자 오른쪽의 가변이 복잡한 수를 나타내는 사용자 지정 가변인 경우에 처리만 합니다. 왼쪽 피연산자가 복잡한 가변이고 오른쪽 피연산자는 복잡한 가변이 아닌 경우, 복잡한 가변은 오른쪽 피연산자를 먼저 복잡한 가변으로 강제로 타입 변환되도록 합니다. 이는 *RightPromotion* 메소드를 오버라이드하므로 항상 *VarType* 속성의 타입을 필요로 합니다.

```

function TComplexVariantType.RightPromotion(const V:TVarData;
const Operator:TVarOp; out RequiredVarType:TVarType):Boolean;
begin
    { Complex Op TypeX }
    RequiredVarType := VarType;
    Result := True;
end;

```

더하기 연산자는 복잡한 값을 문자열로 타입 변환하고 연결하여 문자열과 복잡한 수에 대해 구현되고 더하기, 빼기, 곱하기 및 나누기 연산자는 복잡한 가변의 데이터에 저장된 *TComplexData* 객체의 메소드를 사용하여 두 개의 복잡한 수에 대해 구현됩니다. 이는 *TVarData* 레코드를 *TComplexVarData* 레코드로 타입 변환하고 해당 *VComplex* 멤버를 사용함으로써 액세스됩니다.

다른 연산자 또는 타입의 조합을 시도하면 메소드가 *RaiseInvalidOp* 메소드를 호출하게 되어 런타임 오류가 발생합니다. *TCustomVariantType* 클래스는 사용자 지정 가변 타입의 구현에 사용될 수 있는 *RaiseInvalidOp*와 같은 많은 유틸리티 메소드를 포함합니다.

*BinaryOp*는 문자열 및 기타 복잡한 가변 등 제한된 수의 타입만 다룹니다. 하지만 복잡한 수와 다른 숫자 타입 간에 연산을 수행하는 것은 가능합니다. *BinaryOp* 메소드가 동작하려면 값이 이 메소드로 전달되기 전에 피연산자가 복잡한 가변으로 타입 변환되어야 합니다. 위에서 우리는 이미 *RightPromotion* 메소드를 사용하여 왼쪽 피연산자가 복잡한 경우 강제로 오른쪽 피연산자를 복잡한 가변이 되도록 하는 것을 보았습니다. 유사한 메소드인 *LeftPromotion*은 오른쪽 피연산자가 복잡할 때 왼쪽 피연산자를 다음과 같이 강제로 타입 변환합니다.

```

function TComplexVariantType.LeftPromotion(const V:TVarData;
const Operator:TVarOp; out RequiredVarType:TVarType):Boolean;
begin
    { TypeX Op Complex }
    if (Operator = opAdd) and VarDataIsStr(V) then
        RequiredVarType := varString
    else
        RequiredVarType := VarType;
        Result := True;
    end;

```

이 *LeftPromotion* 메소드는 *LeftPromotion*이 피연산자가 문자열로 있도록 하는 경우 문자열과 연산이 더하기가 아니면 왼쪽 피연산자를 다른 복잡한 가변으로 강제로 타입 변환합니다.

비교 연산 구현

사용자 지정 가변 타입이 비교 연산자(=, <>, <, <=, >, >=)를 지원하도록 하는 두 가지 방법이 있습니다. *Compare* 메소드를 오버라이드하거나 *CompareOp* 메소드를 오버라이드하면 됩니다.

사용자 지정 가변 타입이 비교 연산자의 모든 범위를 지원하는 경우, *Compare* 메소드가 가장 쉽습니다. *Compare*는 다음 세 가지 매개변수를 취합니다. 왼쪽 피연산자, 오른쪽 피연산자 그리고 이 둘 사이의 관계를 반환하는 var 매개변수가 그것입니다. 예를 들어 VarConv 유닛 내의 *TConvertVariantType* 객체는 다음과 같은 *Compare* 메소드를 구현합니다.

```

procedure TConvertVariantType.Compare(const Left, Right:TVarData;
var Relationship: TVarCompareResult);
const
    CRelationshipToRelationship: array [TValueRelationship] of TVarCompareResult =
        (crLessThan, crEqual, crGreaterThan);
var
    LValue:Double;
    LType:TConvType;
    LRelationship: TValueRelationship;
begin
    // supports...
    // convvar cmp number
    // Compare the value of convvar and the given number
    // convvar1 cmp convvar2
    // Compare after converting convvar2 to convvar1's unit type
    // The right can also be a string. If the string has unit info then it is
    // treated like a varConvert else it is treated as a double
    LRelationship := EqualsValue;
case Right.VType of
    varString:
        if TryStrToConvUnit(Variant(Right), LValue, LType) then
            if LType = CIllegalConvType then
                LRelationship := CompareValue(TConvertVarData(Left).VValue, LValue)
            else
                LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                    TConvertVarData(Left).VConvType, LValue, LType)
            else
                RaiseCastError;
    varDouble:
        LRelationship := CompareValue(TConvertVarData(Left).VValue,
            TConvertVarData(Right).VDouble);
    else
        if Left.VType = VarType then
            LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                TConvertVarData(Left).VConvType,
                TConvertVarData(Right).VValue,
                TConvertVarData(Right).VConvType)

```

```

    else
      RaiseInvalidOp;
    end;
    Relationship := CRelationshipToRelationship[LRelationship];
  end;
end;

```

하지만 사용자 지정 타입이 "더 큰" 또는 "더 적은"이라는 개념을 지원하지 않고 "같은" 또는 "같지 않은" 만 지원하는 경우 *Compare*는 *crLessThan*, *crEqual* 또는 *crGreaterThan*을 반환해야 하기 때문에 *Compare* 메소드를 구현하기 어렵습니다. 유일한 유효 응답이 "같지 않은"인 경우 *crLessThan*이나 *crGreaterThan* 중 어느 것을 반환할 것인지 알 수 없습니다. 따라서 정렬의 개념을 지원하지 않는 타입의 경우에는 *CompareOp* 메소드를 대신 오버라이드할 수 있습니다.

*CompareOp*는 다음 세 개의 매개변수를 갖습니다. 왼쪽 피연산자의 값, 오른쪽 피연산자의 값 그리고 비교 연산자가 그것입니다. 이 메소드를 구현하여 연산을 수행하고 비교가 *True*인지를 표시하는 부울 값을 반환합니다. 그런 다음 비교가 의미가 없을 때 *RaiseInvalidOp* 메소드를 호출할 수 있습니다.

예를 들어, 다음 *CompareOp* 메소드는 *VarCmplx* 유닛 내의 *TComplexVariantType* 객체로부터 옵니다. *CompareOp* 메소드는 다음과 같이 동등과 부등의 테스트만 지원합니다.

```

function TComplexVariantType.CompareOp(const Left, Right:TVarData;
  const Operator:Integer):Boolean;
begin
  Result := False;
  if (Left.VType = VarType) and (Right.VType = VarType) then
    case Operator of
      opCmpEQ:
        Result :=
          TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
      opCmpNE:
        Result := not
          TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
  end;
end;

```

이 두 가지 구현이 지원하는 피연산자의 타입이 매우 제한적이라는 것에 유의하십시오. 이항 연산과 함께 *RightPromotion* 및 *LeftPromotion* 메소드를 사용하여 *Compare*나 *CompareOp*가 호출되기 전에 강제로 변환하여 고려할 경우를 제한해야 합니다.

단항 연산 구현

사용자 지정 가변 타입을 표준 단항 연산자 ($-$, *not*) 와 함께 동작하도록 하려면 *UnaryOp* 메소드를 오버라이드해야 합니다. *UnaryOp*은 피연산자의 값과 연산자, 두 개의 매개변수를 갖습니다. 이 메소드를 구현하여 연산을 수행하고 피연산자를 포함한 동일한 가변을 사용하여 결과를 반환합니다.

예를 들어, 다음 *UnaryOp* 메소드는 *VarCmplx* 유닛에 정의된 *TComplexVariantType*에서 옵니다.

```

procedure TComplexVariantType.UnaryOp(var Right:TVarData; const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Operator of
      opNegate:
        TComplexVarData(Right).VComplex.DoNegate;
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
  end;

```

논리적인 **not** 연산자의 경우 복잡한 가변에 대해서는 의미가 없으므로 이 메소드는 *RaiseInvalidOp*를 호출하여 런타임 오류를 일으킵니다.

사용자 지정 가변의 복사 및 지우기

타입 변환과 연산자의 구현뿐만 아니라 사용자 지정 가변 타입의 가변을 복사하고 지우는 방법을 지시해야 합니다.

가변의 값을 복사하는 방법을 지시하기 위해 *Copy* 메소드를 구현합니다. 가변의 값을 갖기 위해 사용하는 모든 클래스나 구조에 대해 메모리를 할당하는 것을 기억해야 하기는 하지만 일반적으로 이 작업은 쉽습니다.

```

procedure TComplexVariantType.Copy(var Dest:TVarData; const Source:TVarData;
  const Indirect:Boolean);
begin
  if Indirect and VarDataIsByRef(Source) then
    VarDataCopyNoInd(Dest, Source)
  else
    with TComplexVarData(Dest) do
      begin
        VType := VarType;
        VComplex := TComplexData.Create(TComplexVarData(Source).VComplex);
      end;
    end;
  end;

```

참고 *Copy* 메소드의 *Indirect* 매개변수는 가변이 자신의 데이터에 대해 간접적인 참조만 가질 경우를 고려해서 복사해야 함을 알립니다.

팁 사용자 지정 가변 타입이 데이터를 보유할 메모리를 할당하지 않은 경우(데이터가 *TVarData* 레코드에 전체적으로 들어 맞는 경우), *Copy* 메소드의 구현은 단순히 *SimplisticCopy* 메소드를 호출합니다.

가변의 값을 지우는 방법을 지시하기 위해 *Clear* 메소드를 구현합니다. *Copy* 메소드 사용 시 단 한 가지 까다로운 점은 가변의 데이터를 저장하기 위해 할당된 모든 리소스를 다음과 같이 해제해야 한다는 것입니다.

```

procedure TComplexVariantType.Clear(var V:TVarData);
begin
  V.VType := varEmpty;
  FreeAndNil(TComplexVarData(V).VComplex);
end;

```

또한 *IsClear* 메소드도 구현해야 합니다. 유효하지 않은 값이나 "공백" 데이터를 나타내는 특수한 값을 다음과 같은 방법으로 탐지할 수 있습니다.

```

function TComplexVariantType.IsClear(const V:TVarData):Boolean;
begin
  Result := (TComplexVarData(V).VComplex = nil) or
    TComplexVarData(V).VComplex.IsZero;
end;

```

사용자 지정 가변 값 로드 및 저장

기본적으로 사용자 지정 가변이 *published* 속성의 값으로 할당되는 경우 해당 속성이 폼 파일로 저장될 때 문자열로 타입 변환되고 폼 파일에서 속성을 읽을 때 문자열로부터 다시 변환됩니다. 하지만 좀더 자연스러운 방법으로 사용자 지정 가변 값을 로드하고 저장하는 사용자만의 메커니즘을 제공할 수도 있습니다. 그렇게 하려면 *TCustomVariantType* 자손은 *Classes.pas*에서 *IVarStreamable* 인터페이스를 구현해야 합니다.

*IVarStreamable*은 가변의 값을 스트림으로부터 읽고 스트림에 쓰는 작업을 위해 *StreamIn*과 *StreamOut*, 두 개의 메소드를 정의합니다. 예를 들어, *VarCmplx* 유닛에서 *TComplexVariantType*은 *IVarStreamable* 메소드를 다음과 같이 구현합니다.

```

procedure TComplexVariantType.StreamIn(var Dest:TVarData; const Stream:TStream);
begin
  with TReader.Create(Stream, 1024) do
    try
      with TComplexVarData(Dest) do
        begin
          VComplex := TComplexData.Create;
          VComplex.Real := ReadFloat;
          VComplex.Imaginary := ReadFloat;
        end;
      finally
        Free;
      end;
    end;
end;

procedure TComplexVariantType.StreamOut(const Source:TVarData; const Stream:TStream);
begin
  with TWriter.Create(Stream, 1024) do
    try
      with TComplexVarData(Source).VComplex do
        begin
          WriteFloat(Real);
          WriteFloat(Imaginary);
        end;
      finally
        Free;
      end;
    end;
end;

```

이들 메소드는 *Stream* 매개변수를 위한 Reader나 Writer 객체를 생성하여 읽기 및 쓰기 값의 세부 사항을 처리합니다.

TCustomVariantType 자손 사용

TCustomVariantType 자손을 정의하는 유닛의 초기화 섹션에서 사용자 고유 클래스의 인스턴스를 생성합니다. 객체를 인스턴스화할 때 가변 처리 시스템에 자신을 자동으로 등록하여 새로운 가변 타입을 사용할 수 있도록 합니다. 예를 들어, 다음은 *VarCmplx* 유닛의 초기화 섹션입니다.

```
initialization
  ComplexVariantType := TComplexVariantType.Create;
```

TCustomVariantType 자손을 정의하는 유닛의 완료 섹션에서 클래스의 인스턴스를 해제합니다. 이렇게 하면 가변 타입을 자동으로 등록 해제합니다. 다음은 *VarCmplx* 유닛의 완료 섹션입니다.

```
finalization
  FreeAndNil(ComplexVariantType);
```

사용자 지정 가변 타입을 사용하는 유틸리티 작성

일단 *TCustomVariantType* 자손을 생성하여 사용자 지정 가변 타입을 구현하면 애플리케이션에서 새로운 가변 타입을 사용하는 것이 가능합니다. 하지만 몇 가지 유틸리티가 없으면 이 작업은 그리 쉽지 않습니다.

예를 들어, 유틸리티 함수가 없는 경우 사용자 지정 가변 타입의 인스턴스를 생성하는 유일한 방법은 다른 타입의 소스 가변에 대해 전역 *VarCast* 프로시저를 사용하는 것입니다. 적절한 값이나 값 집합에서 사용자 지정 가변 타입의 인스턴스를 생성하는 메소드를 생성하는 것이 좋습니다. 이러한 함수나 함수 집합은 사용자가 정의한 구조를 기입하여 사용자 지정 가변 데이터를 저장합니다. 예를 들어, 다음과 같은 함수를 사용하여 복잡한 값으로 된 가변을 생성할 수 있습니다.

```
function VarComplexCreate(const AReal, AImaginary:Double):Variant;
begin
  VarClear(Result);
  TComplexVarData(Result).VType := ComplexVariantType.VarType;
  TComplexVarData(ADest).VComplex := TComplexData.Create(ARead, AImaginary);
end;
```

이 함수는 *VarCmplx* 유닛에 실제로 존재하지는 않지만, 존재하는 메소드를 합성한 것이며 예를 단순화하기 위해 제공된 것입니다. 반환된 가변은 *TVarData* 구조 (*TComplexVarData*)로 매핑하기 위해 정의되었던 레코드로 변환된 후 채워집니다.

또 하나 생성해야 할 것은 새로운 가변 타입의 가변 타입 코드를 반환하는 다른 유용한 유틸리티입니다. 이 타입 코드는 상수가 아닙니다. 이 타입 코드는 *TCustomVariantType* 자손을 인스턴스화할 때 자동으로 생성됩니다. 그러므로 사용자 지정 가변 타입의 타입 코드를 쉽게 결정할 수 있는 방법을 제공하는 데 유용합니다. *VarCmplx* 유닛의 다음과 같은 함수는 *TCustomVariantType* 자손의 *VarType* 속성을 단순히 반환하여 타입 코드를 작성하는 법을 보여 줍니다.

```
function VarComplex:TVarType;
begin
    Result := ComplexVariantType.VarType;
end;
```

대부분의 사용자 지정 가변에 대해 제공된 두 개의 다른 표준 유틸리티는 주어진 가변이 사용자 지정 타입으로부터 온 것인지 검사하고 임의의 가변을 새로운 사용자 지정 타입으로 변환합니다. 다음은 `VarCmplx` 유닛으로부터의 이러한 유틸리티 구현의 예입니다.

```
function VarIsComplex(const AValue:Variant):Boolean;
begin
    Result := (TVarData(AValue).VType and varTypeMask) = VarComplex;
end;

function VarAsComplex(const AValue:Variant):Variant;
begin
    if not VarIsComplex(AValue) then
        VarCast(Result, AValue, VarComplex)
    else
        Result := AValue;
end;
```

이 유틸리티들은 다음과 같은 모든 가변의 표준 기능을 사용합니다. `TVarData` 레코드의 `VType` 멤버와 데이터 변환을 위한 `TCustomVariantType` 자손 내에 구현된 메소드로 인해 동작하는 `VarCast` 함수가 그것입니다.

위에서 언급한 표준 유틸리티 외에 새로운 사용자 지정 가변 타입에 특정한 유틸리티를 작성할 수 있습니다. 예를 들어, `VarCmplx` 유닛은 복잡한 값의 가변에 대해 수학적 연산을 구현하는 많은 함수를 정의합니다.

사용자 지정 가변의 지원 속성 및 메소드

일부 가변은 속성과 메소드를 가집니다. 예를 들어, 가변의 값이 인터페이스일 때 가변을 사용하여 해당 인터페이스에 대한 속성의 값을 읽거나 쓰고 메소드를 호출할 수 있습니다. 사용자 지정 가변 타입이 인터페이스를 나타내지 않더라도 동일한 방식으로 애플리케이션이 사용할 수 있는 속성과 메소드를 주려고 할 수도 있습니다.

TInvokeableVariantType 사용

메소드에 속성 지원을 제공하려면 새로운 사용자 지정 가변 타입을 활성화하는 클래스가 `TCustomVariantType`으로부터 직접 오는 것이 아니라 `TInvokeableVariantType`의 자손이어야 합니다.

`TInvokeableVariantType`은 다음의 네 가지 메소드를 정의합니다.

- `DoFunction`
- `DoProcedure`
- `GetProperty`
- `SetProperty`

이 메소드들은 사용자 지정 가변 타입에 대한 속성과 메소드를 지원하기 위해 구현할 수 있습니다.

예를 들면, `VarConv` 유닛은 `TConvertVariantType`에 대한 기본 클래스로 `TInvokeableVariantType`을 사용하여 결과 값인 사용자 지정 가변이 속성을 지원할 수 있게 합니다. 다음 예는 이러한 속성에 대한 속성 가져오기를 보여 줍니다.

```
function TConvertVariantType.GetProperty(var Dest:TVarData;
    const V:TVarData; const Name:String):Boolean;
var
    LType:TConvType;
begin
    // supports...
    // 'Value'
    // 'Type'
    // 'TypeName'
    // 'Family'
    // 'FamilyName'
    // 'As[Type]'
    Result := True;
    if Name = 'VALUE' then
        Variant(Dest) := TConvertVarData(V).VValue
    else if Name = 'TYPE' then
        Variant(Dest) := TConvertVarData(V).VConvType
    else if Name = 'TYPENAME' then
        Variant(Dest) := ConvTypeToDescription(TConvertVarData(V).VConvType)
    else if Name = 'FAMILY' then
        Variant(Dest) := ConvTypeToFamily(TConvertVarData(V).VConvType)
    else if Name = 'FAMILYNAME' then
        Variant(Dest) :=
ConvFamilyToDescription(ConvTypeToFamily(TConvertVarData(V).VConvType))
    else if System.Copy(Name, 1, 2) = 'AS' then
        begin
            if DescriptionToConvType(ConvTypeToFamily(TConvertVarData(V).VConvType),
                System.Copy(Name, 3, MaxInt), LType) then
                VarConvertCreateInto(Variant(Dest), Convert(TConvertVarData(V).VValue,
                    TConvertVarData(V).VConvType, LType), LType)
            else
                Result := False;
            end
        else
            Result := False;
        end;
end;
```

`GetProperty` 메소드는 어느 속성이 필요한지 결정하기 위해 `Name` 매개변수를 검사합니다. 그런 다음 `Variant(V)`의 `TVarData` 레코드에서 정보를 검색하고 이를 `Variant(Dest)`로서 반환합니다. 이 메소드는 사용자 지정 가변의 현재 값에 기반하여 런타임 시 동적으로 이름이 생성되는 속성(`As[Type]`)을 지원합니다.

이와 유사하게 `SetProperty`, `DoFunction` 및 `DoProcedure` 메소드는 동적으로 메소드 이름을 생성하거나 가변 숫자와 매개변수의 타입에 반응할 수 있을 정도로 일반적입니다.

TPublishableVariantType 사용

사용자 지정 가변 타입이 자신의 데이터를 객체 인스턴스를 사용해 저장하는 경우, 이 데이터가 가변의 데이터를 나타내는 객체의 속성이라면 속성을 구현하는 더 쉬운 방법이 있습니다. *TPublishableVariantType*을 사용자 지정 가변 타입의 기본 클래스로 사용한다면 단지 *GetInstance* 메소드만 구현하면 되며 가변의 데이터를 나타내는 객체의 모든 *published* 속성은 사용자 지정 가변에 대해 자동으로 구현됩니다.

예를 들어, 4-27 페이지의 "사용자 지정 가변 타입 데이터 저장"에서 보듯이 *TComplexVariantType*은 복잡한 값을 갖는 가변의 데이터를 *TComplexData*의 인스턴스를 사용하여 저장합니다. *TComplexData*는 복잡한 값에 대한 정보를 제공하는 많은 *published* 속성 (*Real*, *Imaginary*, *Radius*, *Theta* 및 *FixedTheta*)을 갖습니다. *TComplexVariantType*은 *TPublishableVariantType*의 자손이며 *GetInstance* 메소드를 구현하여 복잡한 값으로 된 가변의 *TVarData* 레코드에 저장된 *TComplexData* 객체 (TypInfo.pas 내)를 반환합니다.

```
function TComplexVariantType.GetInstance(const V:TVarData):TObject;
begin
  Result := TComplexVarData(V).VComplex;
end;
```

*TPublishableVariantType*은 나머지 일을 합니다. *TPublishableVariantType*은 *GetProperty*와 *SetProperty* 메소드를 오버라이드하여 속성 값을 가져오고 설정하는 데 *TComplexData* 객체의 런타임 타입 정보 (RTTI)를 사용합니다.

참고 *TPublishableVariantType*이 동작하려면 사용자 지정 가변의 데이터를 갖는 객체를 RTTI과 함께 컴파일해야 합니다. 다시 말해 {\$M+} 컴파일러 지시어를 사용해서 컴파일하거나 *TPersistent*의 자손이어야 합니다.

문자열 작업

Delphi에는 오브젝트 파스칼 언어의 개발을 통해 도입된 많은 다른 문자와 문자열 타입이 있습니다. 이 단원은 이러한 타입, 용도 및 사용에 대한 개요를 담고 있습니다. 언어에 대한 자세한 내용은 오브젝트 파스칼 언어 온라인 도움말의 String types를 참조하십시오.

문자 타입(Character types)

Delphi에는 다음 세 가지 문자 타입인 *Char*, *AnsiChar*, *WideChar*가 있습니다.

Char 문자 타입은 표준 파스칼에서 유래하여 터보 파스칼에서 사용되고 나서 오브젝트 파스칼에서 사용되었습니다. 후에 오브젝트 파스칼은 *AnsiChar*와 *WideChar*를 Windows 운영 체제에서의 문자 표현을 위한 표준을 지원하는 데 사용되었던 특정 문자 타입으로 추가했습니다. *AnsiChar*는 8비트 문자 ANSI 표준을, *WideChar*는 16비트 유니코드 표준을 지원하기 위해 도입되었습니다. *WideChar*라는 이름은 유니코드 문자가 와이드 문자로 알려져 있어서 그렇게 사용합니다. 와이드 문자는 한 바이트가 아니라 두 바이트이므로 문자 집합으로 훨씬 많은 다른 문자들을 표현할 수 있습니다. *AnsiChar*와 *WideChar*를 사용하게 되면서 *Char*는 현재 권장되는 구현을 나타내는 기본

문자 타입이 되었습니다. 애플리케이션에서 *Char*를 사용할 경우 해당 구현이 Delphi의 추후 버전에서 달라질 수 있습니다.

참고 크로스 플랫폼 프로그램용: Linux `wchar_t` `widechar`는 문자당 23비트입니다. 오브젝트 파스칼 `widechar`가 지원하는 16비트 유니코드 표준은 Linux와 GNU 라이브러리가 지원하는 32비트 UCS 표준의 서브셋입니다. 파스칼 `widechar` 데이터는 `wchar_t`로 OS 함수에 전달되기 전에 문자당 32비트로 확장되어야 합니다.

다음 표에서는 이러한 문자 타입에 대해 요약합니다.

표 4.2 오브젝트 파스칼 문자 타입

타입	바이트	내용	용도
Char	1	단일 문자	기본 문자 타입
AnsiChar	1	단일 문자	8비트 문자
WideChar	2	단일 유니코드 문자	16비트 유니코드 표준

이러한 문자 타입 사용에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말에서 `Character types`를 참조하십시오. 유니코드 문자에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말에서 `About extended character sets`를 참조하십시오.

문자열 타입(String types)

Delphi에는 문자열 작업 시 사용할 수 있는 타입이 세 가지 있습니다.

- 문자 포인터
- 문자열 타입
- 문자열 클래스

이 단원에서는 문자열 타입에 대해 요약하고, 문자열 포인터와 함께 문자열 타입 사용에 대해 설명합니다. 문자열 클래스 사용에 대한 내용은 온라인 도움말의 `TStrings`를 참조하십시오.

Delphi에는 세 개의 문자열 구현, 즉 짧은 문자열, 긴 문자열, 와이드 문자열이 있습니다. 여러 다른 문자열 타입은 이러한 구현을 나타냅니다. 또한 현재 권장하는 문자열 구현을 기본으로 설정하는 예약어 **string**이 있습니다.

짧은 문자열

String은 터보 파스칼에서 사용되었던 첫 번째 문자열 타입입니다. **String**은 원래 짧은 문자열로 구현되었습니다. 짧은 문자열은 1바이트에서 256바이트까지 할당되고 그 중 첫 번째 바이트에는 문자열의 길이에 대한 정보가 들어 있으며 나머지 바이트에는 문자열의 문자가 들어 있습니다.

S: `string[0..n]`// the original string type

긴 문자열을 구현할 때 **string**은 기본으로 긴 문자열 구현으로 바뀌고 `ShortString`이 역호환성 타입으로 도입됩니다. 다음과 같이 `ShortString`은 최대 길이 문자열에 대한 이미 정의된 타입입니다.

S: `string[255]`// the ShortString type

*ShortString*에 대해 할당되는 메모리의 크기는 정적이고, 이는 크기가 컴파일 타임에 결정된다는 것을 의미합니다. 하지만 예를 들어, *ShortString*에 대한 포인터인 *PShortString*를 사용하는 경우에 *ShortString*에 대한 메모리 위치가 동적으로 할당될 수 있습니다. 짧은 문자열 타입 변수가 차지한 저장 장치의 바이트 수는 짧은 문자열 타입의 최대 길이에 한 바이트를 더한 것이 됩니다. 이미 정의된 *ShortString* 타입의 크기는 256바이트입니다.

구문 `string[0..n]`을 사용하여 선언된 짧은 문자열과 이미 정의된 *ShortString* 타입은 모두 이전 버전의 Delphi 및 Borland 파스칼과의 역호환성을 위해 주로 존재합니다.

컴파일러 지시어 `$H`는 예약어 `string`이 짧은 문자열 또는 긴 문자열을 나타내는지 여부를 제어합니다. 기본 상태 `{$H+}`에서 `string`은 긴 문자열을 나타냅니다. `{$H-}` 지시어를 사용하여 긴 문자열을 *ShortString*으로 변경할 수 있습니다. `{$H-}` 상태는 기본으로 짧은 문자열을 사용하는 오브젝트 파스칼 버전의 코드를 사용하는 데 가장 유용합니다. 하지만 짧은 문자열은 고정 크기 컴포넌트가 필요한 데이터 구조나 *ShareMem* 유닛을 사용하고 싶지 않을 경우 DLL에서 유용하게 사용될 수 있습니다(온라인 도움말의 Memory Management 참조). 짧은 문자열의 생성을 확인하기 위해 문자열 타입 정의의 의미를 지역적으로 오버라이드할 수 있습니다. 또한 짧은 문자열 타입의 선언을 명확하고 `$H` 설정에 무관한 `string[255]` 또는 *ShortString*으로 변경할 수 있습니다.

짧은 문자열과 *ShortString* 타입에 대한 자세한 내용은 *오브젝트 파스칼 랭기지 안내서* 온라인 도움말의 Short strings를 참조하십시오.

긴 문자열

긴 문자열은 최대 2기가바이트의 길이를 갖는 동적으로 할당된 문자열이지만 실제 한계는 사용 가능한 메모리의 양에 따라 다릅니다. 짧은 문자열과 마찬가지로 긴 문자열도 8비트 Ansi 문자를 사용하고 길이 지시자를 가집니다. 그러나 짧은 문자열과 달리 긴 문자열에는 동적 문자열 길이를 포함하는 영순위 요소가 없습니다. 긴 문자열의 길이를 알아내려면 *Length* 표준 함수를 사용해야 하고, 긴 문자열의 길이를 설정하려면 *SetLength* 표준 프로시저를 사용해야 합니다. 또한 긴 문자열은 참조 카운트되고 *PChars*처럼 Null 종료됩니다. 긴 문자열의 구현에 대한 자세한 내용은 *오브젝트 파스칼 랭기지 안내서* 온라인 도움말에서 Long strings를 참조하십시오.

긴 문자열은 예약어 `string`과 이미 정의된 식별자 *AnsiString*으로 표시합니다. 새 애플리케이션에서 긴 문자열 타입을 사용하는 것이 좋습니다. VCL의 모든 컴포넌트는 `string`을 사용하여 이 상태에서 컴파일됩니다. 컴포넌트를 작성하는 경우에는 문자열 타입 속성에서 데이터를 받는 모든 코드에서와 마찬가지로 긴 문자열을 사용해야 합니다. 항상 긴 문자열을 사용하는 특정 코드를 작성하려면 *AnsiString*을 사용해야 합니다. 새 문자열 구현이 표준이 될 때 타입을 쉽게 변경할 수 있는 융통성 있는 코드를 작성하고자 한다면 `string`을 사용해야 합니다.

WideString

WideChar 타입을 통해 와이드 문자열을 *WideChars*의 배열로 나타낼 수 있습니다. 와이드 문자열은 16비트 유니코드 문자로 구성된 문자열입니다. 긴 문자열과 함께 와이드 문자열은 최대 2기가바이트의 길이로 동적으로 할당할 수 있지만 실제 한계는 사용 가능한

메모리의 양에 따라 다릅니다. Delphi에서 와이드 문자열은 참조 카운팅되지 않습니다. 와이드 문자열을 와이드 문자열 var로 할당하면 문자열 데이터의 사본을 생성합니다. Kylix에서는 WideStrings가 참조 카운팅됩니다.

문자열을 포함하는 동적으로 할당된 메모리는 와이드 문자열이 유효 범위 밖으로 벗어날 때 할당 해제됩니다. 모든 점에서 와이드 문자열은 긴 문자열과 동일한 속성을 갖습니다. *WideString* 타입은 이미 정의된 식별자 *WideString*에 의해 표시됩니다.

32비트 버전의 OLE(Windows 전용)에서는 모든 문자열에 대해 유니코드를 사용하므로 문자열은 모든 OLE automated 속성과 메소드 매개변수에서 와이드 문자열 타입이어야 합니다. 또한 대부분의 OLE API 함수는 Null 종료 와이드 문자열을 사용합니다.

자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 항목의 *WideString*을 참조하십시오.

PChar 타입

*PChar*는 *Char* 타입의 Null 종료 문자열에 대한 포인터입니다. 또한 세 가지 문자 타입 모두 다음과 같은 기본 제공 포인터 타입을 가집니다.

- *PChar*는 8비트 문자의 Null 종료 문자열에 대한 포인터입니다.
- *PAnsiChar*는 8비트 문자의 Null 종료 문자열에 대한 포인터입니다.
- *PWideChar*는 16비트 문자의 Null 종료 문자열에 대한 포인터입니다.

*PChars*는 짧은 문자열과 함께 원래의 오브젝트 파스칼 문자열 타입 중 하나입니다. *PChars*는 주로 C 랭귀지와 Windows API 호환 타입으로서 생성됩니다.

OpenString

*OpenString*은 이제 사용하지 않지만 예전의 코드에서는 볼 수 있습니다. 이 문자열 타입은 16비트 호환이고 매개변수에서만 사용할 수 있습니다. *OpenString*은 긴 문자열이 구현되기 전에 길이가 지정되지 않은 짧은 문자열을 매개변수로 전달하는 데 사용되었습니다. 예를 들면,

```
procedure a(v :openstring);
```

어떠한 길이의 문자열도 매개변수로 전달될 수 있으며 여기서 실제 매개변수와 정식 매개변수의 문자열 길이가 정확하게 일치해야 합니다. 자신이 작성하는 새 애플리케이션에서는 *OpenString*을 사용하면 안 됩니다.

또한 49 페이지의 "문자열에 대한 컴파일러 지시어"에서 {\$P+/-} 스위치를 참조하십시오.

런타임 라이브러리 문자열 처리 루틴

런타임 라이브러리는 특정 문자열 타입에 해당되는 많은 특수한 문자열 처리 루틴을 제공합니다. 이 루틴들은 와이드 문자열, 긴 문자열, Null 종료 문자열(*PChars*를 의미함)을 위한 것입니다. *PChar* 타입을 처리하는 루틴은 Null 종료를 사용하여 문자열 길이를 결정합니다. Null 종료 문자열에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말에서 Working with null-terminated strings를 참조하십시오.

또한 런타임 라이브러리에는 문자열 서식 루틴의 범주가 포함됩니다. *ShortString* 타입에 대한 루틴 범주는 없습니다. 그러나 일부 기본 제공 컴파일러는 *ShortString* 타입을 처리합니다. 예를 들어, 여기에는 *Low* 표준 함수와 *High* 표준 함수가 포함됩니다.

와이드 문자열과 긴 문자열은 일반적으로 사용되는 타입이므로 나머지 단원에서 이러한 루틴들을 설명합니다.

와이드 문자 루틴

문자열을 사용할 때 애플리케이션에 있는 코드가 다양한 대상 로케일에서 집하게 될 문자열을 처리할 수 있는지 확인해야 합니다. 가끔 와이드 문자와 와이드 문자열을 사용해야 할 때가 있습니다. 실제로 표의 문자 집합을 사용하는 한 가지 방법은 모든 문자를 유니코드와 같은 와이드 문자 인코딩 구성으로 변환하는 것입니다. 런타임 라이브러리에는 표준 싱글 바이트 문자열(또는 MBCS 문자열)과 유니코드 문자열 간의 변환을 위한 다음과 같은 와이드 문자열 함수가 들어 있습니다.

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

와이드 문자 인코딩 구성을 사용하면 MBCS(멀티바이트 문자 집합) 시스템에서는 작동하지 않는 문자열에 대한 일반적인 가정을 만들 수 있는 이점이 있습니다. 문자열의 바이트 수와 문자 수 사이에는 직접적인 관계가 있습니다. 따라서 문자가 반으로 잘리거나 문자의 뒷부분을 다른 문자의 앞부분으로 오인하는 실수에 대해서 걱정할 필요가 없습니다.

와이드 문자열로 작업하는 데 있어 단점은 Windows 95가 와이드 문자 API 함수 호출을 지원하지 않는다는 것입니다. 이로 인해 VCL 컴포넌트는 모든 문자열 값을 단일 바이트 또는 MBCS 문자열로 나타냅니다. 문자열 속성이나 해당 값을 읽을 때마다 와이드 문자 시스템과 MBCS 시스템 간의 변환을 하려면 엄청난 양의 추가 코드가 필요하며 애플리케이션이 느려집니다. 하지만 문자(character)와 *WideChars* 간에 1:1 매핑을 이용해야 할 필요가 있을 경우 어떤 특별한 문자열 처리 알고리즘에 대한 와이드 문자로 변환시키려 할 수도 있습니다.

참고 일반적으로 CLX 컴포넌트는 문자열 값을 와이드 문자열로 표시합니다.

일반적으로 사용되는 긴 문자열 루틴

긴 문자열 처리 루틴은 여러 함수 영역을 처리합니다. 이 영역 중 일부는 동일한 목적으로 사용되는데 처리 루틴들 간의 차이는 계산에서의 특정 기준 사용 여부입니다. 다음 표는 이러한 기능적 영역에 따라 루틴을 나열한 것입니다.

- 비교
- 대소문자 변환
- 수정
- 부분 문자열

적당한 곳에서 테이블은 또한 루틴이 다음 기준을 만족하는지 여부를 나타내는 열을 제공합니다.

- 대소문자 구별 사용: 로케일 설정이 사용된 경우 대소문자의 정의를 결정합니다. 루틴이 로케일 설정을 사용하지 않으면 분석은 문자의 순서 값에 기반합니다. 루틴이 대소문자를 구별하지 않으면 이미 정의된 패턴에 의해 결정된 대소문자를 논리적으로 병합합니다.
- 로케일 설정 사용: 로케일 설정을 사용하면 특히 아시아 언어 환경과 같은 특정 로케일에 대해 애플리케이션을 사용자 지정할 수 있습니다. 대부분의 로케일 설정에서는 소문자가 해당 대문자보다 값이 더 적다고 간주합니다. 이것은 소문자가 대문자보다 값이 더 큰 ASCII 순서와는 반대입니다. Windows 로케일을 사용하는 루틴은 앞에 Ansi가 옵니다 (즉, AnsiXXX).
- MBCS(멀티바이트 문자 집합) 지원: MBCS는 극동 아시아 로케일용 코드를 작성할 때 사용됩니다. 멀티바이트 문자는 1바이트 및 2바이트의 문자 코드의 혼합으로 표현되므로 바이트 길이가 반드시 문자열의 길이와 일치하는 것은 아닙니다. MBCS를 지원하는 루틴은 1바이트와 2바이트 문자 구문 분석으로 작성됩니다.

*ByteType*과 *StrByteType*은 두 바이트 문자의 선행 바이트를 결정합니다. 멀티바이트 문자를 사용할 때는 2바이트 문자가 절반으로 잘리지 않도록 유의해야 합니다. 문자 크기는 미리 결정할 수 없으므로 문자를 함수 또는 프로시저에 매개변수로 전달하지 마십시오. 그 대신 문자 또는 문자열에 대한 포인터를 전달합니다. MBCS에 대한 자세한 내용은 12장 "국제적인 애플리케이션 만들기"의 12-2 페이지의 "애플리케이션 코드 활성화"를 참조하십시오.

표 4.3 문자열 비교 루틴

루틴	대소문자 구별	로케일 설정 사용	MBCS 지원
AnsiCompareStr	예	예	예
AnsiCompareText	아니오	예	예
AnsiCompareFileName	아니오	예	예
CompareStr	예	아니오	아니오
CompareText	아니오	아니오	아니오

표 4.4 대소문자 변환 루틴

루틴	로케일 설정 사용	MBCS 지원
AnsiLowerCase	예	예
AnsiLowerCaseFileName	예	예
AnsiUpperCaseFileName	예	예
AnsiUpperCase	예	예
LowerCase	아니오	아니오
UpperCase	아니오	아니오

문자열 파일 이름에 사용되는 루틴: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, *AnsiUpperCaseFileName*은 모두 Windows 로케일을 사용합니다. 파일 이름에 사용되는 로케일(문자 집합)은 기본 사용자 인터페이스와 다를 수 있으므로 항상 포팅 가능한 파일 이름을 사용해야 합니다.

표 4.5 문자열 수정 루틴

루틴	대소문자 구별	MBCS 지원
AdjustLineBreaks	해당 없음	예
AnsiQuotedStr	해당 없음	예
StringReplace	플래그별 옵션	예
Trim	해당 없음	예
TrimLeft	해당 없음	예
TrimRight	해당 없음	예
WrapText	해당 없음	예

표 4.6 부분 문자열 루틴

루틴	대소문자 구별	MBCS 지원
AnsiExtractQuotedStr	해당 없음	예
AnsiPos	예	예
IsDelimiter	예	예
IsPathDelimiter	예	예
LastDelimiter	예	예
QuotedStr	아니오	아니오

표 4.7 문자열 처리 루틴

루틴	대소문자 구별	MBCS 지원
AnsiContainsText	아니오	예
AnsiEndsText	아니오	아니오
AnsiIndexText	아니오	예
AnsiMatchText	아니오	예
AnsiResemblesText	아니오	아니오
AnsiStartsText	아니오	예
IfThen	해당 없음	예
LeftStr	예	아니오
RightStr	예	아니오
SoundEx	해당 없음	아니오
SoundExInt	해당 없음	아니오

표 4.7 문자열 처리 루틴 (계속)

루틴	대소문자 구별	MBCS 지원
DecodeSoundExInt	해당 없음	아니오
SoundExWord	해당 없음	아니오
DecodeSoundExWord	해당 없음	아니오
SoundExSimilar	해당 없음	아니오
SoundExCompare	해당 없음	아니오

문자열 선언 및 초기화

긴 문자열은 다음과 같이 선언합니다.

```
S:string;
```

긴 문자열을 초기화할 필요는 없습니다. 긴 문자열은 비어 있는 것으로 자동적으로 초기화됩니다. 문자열이 비어 있는지 검사하려면 다음과 같이 *EmptyStr* 변수를 사용합니다.

```
S = EmptyStr;
```

아니면 다음과 같이 빈 문자열을 검사합니다.

```
S = '';
```

빈 문자열에는 유효한 데이터가 없습니다. 따라서 다음과 같이 빈 문자열을 인덱스 참조하려는 것은 **nil**에 액세스하려는 것과 같고 액세스 위반을 초래합니다.

```
var
  S:string;
begin
  S[i]; // this will cause an access violation
  // statements
end;
```

이와 마찬가지로 빈 문자열을 *PChar*로 변환하면 그 결과는 **nil** 포인터가 됩니다. 따라서 그러한 *PChar*를 읽거나 써야 하는 루틴에 전달할 경우에는 다음과 같이 루틴이 **nil**을 처리할 수 있는지 확인하십시오.

```
var
  S:string; // empty string
begin
  proc(PChar(S)); // be sure that proc can handle nil
  // statements
end;
```

만일 루틴이 처리할 수 없다면 다음과 같이 문자열을 초기화합니다.

```
S := 'No longer nil'
proc(PChar(S)); // proc does not need to handle nil now
```

아니면 다음과 같이 *SetLength* 프로시저를 사용하여 길이를 설정합니다.

```
SetLength(S, 100); // sets the dynamic length of S to 100
proc(PChar(S)); // proc does not need to handle nil now
```


*SetLength*를 사용할 경우 문자열의 기존 문자는 유지되지만 모든 새로 할당된 공간의 내용은 정의되어 있지 않습니다. *SetLength*를 호출한 다음 *S*는 참조 카운트가 1인 고유의 문자열을 반드시 참조합니다. 문자열의 길이를 알려면 *Length* 함수를 사용합니다.

다음과 같이 **string**을 선언하는 경우를 생각해 봅시다.

```
S:string[n];
```

이 구문은 *n* 길이의 긴 문자열이 아닌 짧은 문자열을 암시적으로 선언합니다. 특별히 *n* 길이의 긴 문자열을 선언하려면 **string** 타입의 변수를 선언하고 *SetLength* 프로시저를 사용하십시오.

```
S:string;
SetLength(S, n);
```

문자열 타입의 혼합 및 변환

짧은 문자열, 긴 문자열, 와이드 문자열은 할당문과 표현식에서 혼합할 수 있고 컴파일러는 필요한 문자열 타입 변환을 수행하기 위한 코드를 자동으로 생성합니다. 하지만 문자열 값을 짧은 문자열 변수에 할당할 때 문자열 값이 짧은 문자열 변수의 선언된 최대 길이보다 더 긴 경우에는 문자열 값이 잘린다는 점에 유의하십시오.

긴 문자열은 미리 동적으로 할당됩니다. *PAnsiString*, *PString* 또는 *PWideString*과 같은 기본 제공 포인터 타입 중의 하나를 사용하는 경우 간접적인 또 다른 레벨의 방법을 도입하는 것임을 알아야 합니다. 의도한 대로 되었는지 확인하십시오.

기본으로 하는 Qt 문자열 타입과 CLX 문자열 타입을 변환하기 위한 추가적인 함수 (*CopyQStringListToTstrings*, *Copy TStringsToQStringList*, *QStringListToTStringList*)가 제공됩니다. 이러한 함수들은 *QtTypes.pas*에 있습니다.

문자열에서 PChar로 변환

긴 문자열은 *PChar*로 자동 변환되지 않습니다. 문자열과 *PChars* 간의 몇 가지 차이로 인해 다음과 같은 변환 문제점이 야기될 수 있습니다.

- 긴 문자열은 참조 카운트되지만 *PChars*는 참조 카운트되지 않습니다.
- 문자열에 할당하면 데이터를 복사하지만 *PChar*가 메모리에 대한 포인터입니다.
- 긴 문자열은 Null 종료되고 또한 문자열의 길이를 포함하지만 *PChars*는 단순히 Null 종료됩니다.

이러한 차이로 인해 미묘한 오류가 발생할 수 있는 상황을 이 단원에서 설명합니다.

문자열 종속성

예를 들면, *PChar*를 취하는 함수를 사용할 때 긴 문자열을 Null 종료된 문자열로 변환해야 할 경우가 있을 것입니다. 문자열을 *PChar*로 변환해야 하는 경우, 결과 *PChar*의 수명을 관리해야 한다는 점에 유의하십시오. 긴 문자열은 참조 카운트되기 때문에 문자열을 *PChar*로 타입 변환하면 실제로 참조 카운트를 증가시키지 않고 문자열에 대한 종속을 하나 증가시킵니다. 참조 카운트가 0이면 문자열은 이에 대한 추가 종속이 있더라도

소멸됩니다. 이 문자열을 전달하는 루틴이 계속 이 타입 변환을 이용하고 있는 동안 *PChar* 변환이 사라질 수도 있습니다. 예를 들면, 다음과 같습니다.

```
procedure my_func(x:string);
begin
  // do something with x
  some_proc(PChar(x)); // cast the string to a PChar
  // you now need to guarantee that the string remains
  // as long as the some_proc procedure needs to use it
end;
```

PChar 지역 변수 반환

*PChars*를 사용할 때 공통적으로 나타나는 오류는 지역 변수를 데이터 구조에 저장하거나 값으로 반환하는 것입니다. *PChar*는 단순히 메모리에 대한 포인터이지 문자열의 참조 카운트된 복사가 아니기 때문에 루틴이 종료되면 사라집니다. 예를 들면, 다음과 같습니다.

```
function title(n:Integer):PChar;
var
  s:string;
begin
  s := Format('title - %d', [n]);
  Result := PChar(s); // DON'T DO THIS
end;
```

이 예에서는 *title* 함수가 반환될 때 해제되는 문자열 데이터에 대한 포인터를 반환합니다.

지역 변수를 PChar로 전달

*PChar*를 취하는 함수를 호출하여 초기화할 필요가 있는 지역 문자열 변수가 있는 경우를 생각해 봅시다. 한 가지 방법은 다음과 같이 지역 **array of char**를 만들어 함수에 전달한 다음 지역 문자열 변수를 문자열에 할당하는 것입니다.

```
// VCL version
// assume MAXSIZE is a predefined constant
var
  i:Integer;
  buf:array[0..MAX_SIZE] of char;
  S:string;
begin
  i := GetModuleFilename(0, @buf, SizeOf(buf)); // treats @buf as a PChar
  S := buf;
  // statements
end;
```

또는 크로스 플랫폼 프로그램의 경우에도 다음과 같이 코드가 거의 동일합니다.

```
// assume FillBuffer is a predefined function
function FillBuffer(Buf: PChar;Count: Integer): Integer
begin
  . . .
end;

// assume MAX_SIZE is a predefined constant
```

```

var
  i:Integer;
  buf:array[0..MAX_SIZE] of char;
  S:string;
begin
  i := FillBuffer(0, @buf, SizeOf(buf)); // treats @buf as a PChar
  S := buf;
  // statements
end;

```

이 방법은 버퍼가 스택에 할당되기 때문에 버퍼 크기가 상대적으로 작은 경우에 유용합니다. 또한 **array of char** 타입과 **string** 타입 간의 변환이 자동 수행되므로 이 방법은 안전합니다. *GetModuleFilename*(또는 크로스 플랫폼 버전의 *FillBuffer*) 반환 시 문자열의 *Length*는 *buf*에 쓰여진 바이트의 수를 정확히 나타냅니다.

버퍼 복사의 오버헤드를 제거하기 위해 문자열을 *PChar*로 변환할 수 있습니다(루틴에서 *PChar*를 메모리에 남겨 둘 필요가 없다고 확신할 경우). 그러나 **array of char**를 **string**에 할당할 때처럼 문자열 길이의 동기화가 자동적으로 발생하지는 않습니다. 실제 문자열의 길이를 반영하도록 문자열 *Length*를 다시 설정해야 합니다. 복사된 바이트 수를 반환하는 함수를 사용하는 경우에는 다음 한 줄의 코드로 이 작업을 안전하게 수행할 수 있습니다.

```

var
  S: string;
begin
  SetLength(S, MAX_SIZE); // when casting to a PChar, be sure the string is not empty
  SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
  // statements
end;

```

문자열에 대한 컴파일러 지시어

다음의 컴파일러 지시어는 문자와 문자열 타입에 영향을 줍니다.

표 4.8 문자열에 대한 컴파일러 지시어

지시어	설명
{ <i>\$H</i> +/-}	컴파일러 지시어 <i>\$H</i> 는 예약어 string 이 짧은 문자열 또는 긴 문자열을 나타내는지 여부를 제어합니다. 기본 상태 { <i>\$H</i> +}에서 string 은 긴 문자열을 나타냅니다. { <i>\$H</i> -} 지시어를 사용하면 긴 문자열을 <i>ShortString</i> 으로 바꿀 수 있습니다.
{ <i>\$P</i> +/-}	<i>\$P</i> 지시어는 { <i>\$H</i> -} 상태에서 컴파일된 코드에서만 의미가 있으며 역호환성을 위해 제공됩니다. <i>\$P</i> 는 { <i>\$H</i> -} 상태에서 string 키워드를 사용하여 선언된 변수 매개변수의 의미를 제어합니다. string 키워드를 사용하여 선언된 변수 매개변수는 { <i>\$P</i> -} 상태에서는 일반적인 변수 매개변수이지만 { <i>\$P</i> +} 상태에서는 개방형(open) 문자열 매개변수입니다. <i>\$P</i> 지시어의 설정에 상관 없이 <i>OpenString</i> 식별자는 항상 개방형 문자열 매개변수를 선언하는 데 사용될 수 있습니다.

표 4.8 문자열에 대한 컴파일러 지시어 (계속)

지시어	설명
<code>{S\$V+/-}</code>	<p><code>\$V</code> 지시어는 변수 매개변수로 전달된 짧은 문자열에 대한 타입 확인을 제어합니다. <code>{S\$V+}</code> 상태에서는 형식 매개변수와 실제 매개변수의 문자열 타입이 동일해야 하는 엄격한 타입 확인이 수행됩니다.</p> <p><code>{S\$V-}</code> (완화된) 상태에서 모든 짧은 문자열 타입 변수는 선언된 최대 길이가 형식 매개변수의 길이와 동일하지 않더라도 실제 매개변수로 허용됩니다. 이렇게 하면 메모리 손상을 유발할 수 있다는 점에 유의하십시오. 예를 들면, 다음과 같습니다.</p> <pre>var S:string[3]; procedure Test(var T:string); begin T := '1234'; end; begin Test(S); end.</pre>
<code>{S\$X+/-}</code>	<p><code>{S\$X+}</code> 컴파일러 지시어는 기본 제공 <i>PChar</i> 타입과 인덱스가 0부터 시작하는 문자 배열에 적용하는 특별한 규칙을 활성화하여 Null 종료 문자열을 지원할 수 있습니다. 이러한 규칙을 통해 인덱스가 0부터 시작하는 배열과 문자 포인터를 시스템 유닛의 <i>Write</i>, <i>Writeln</i>, <i>Val</i>, <i>Assign</i>, <i>Rename</i> 등과 함께 사용할 수 있습니다.</p>

문자열과 문자: 관련 주제

*오브젝트 파스칼 랭귀지 안내서*의 다음 항목에서 문자열과 문자 집합에 대해 설명합니다. 12장 "국제적인 애플리케이션 만들기"도 참조하십시오.

- 확장 문자 집합 정보(국제 문자 집합에 대해 설명)
- Null 종료 문자열 사용(문자 배열에 대한 내용 포함)
- 문자열
- 문자 포인터
- 문자열 연산자

파일 작업

이 단원에서는 파일 작업에 대해 설명하고 디스크에서 파일을 처리하는 것과 파일 읽기 및 쓰기와 같은 I/O 작업을 구별합니다. 첫 번째 단원에서는 디스크 상의 파일 처리를 포함하는 공통적인 프로그래밍 작업에서 사용하는 API 루틴과 런타임 라이브러리에 대해 설명합니다. 다음 단원은 파일 I/O와 함께 사용되는 파일 형식에 대한 개요입니다. 마지막 단원에서는 파일 I/O 작업에 권장되는 방법인 파일 스트림 사용 방법을 중점적으로 다룹니다.

오브젝트 파스칼 랭귀지는 대소문자를 구별하지 않지만 Linux 운영 체제는 구별합니다. 크로스 플랫폼 애플리케이션에서 파일을 사용할 때에는 대소문자에 주의하십시오.

참고 오브젝트 파스칼 랭귀지의 이전 버전에서는 요즘 보편적으로 사용되는 파일 이름 매개 변수가 아닌 파일 자체에 대한 작업을 수행하였습니다. 이러한 파일 형식을 사용할 경우에는, 예를 들어 파일 이름을 재지정하기 전에 파일을 찾아 파일 변수에 할당해야 했습니다.

파일 처리

몇 가지 일반적인 파일 작업들은 오브젝트 파스칼의 런타임 라이브러리에 기본 제공되어 있습니다. 파일 작업에 관한 프로시저와 함수는 수준 높은 작업을 수행합니다. 대부분의 루틴에서는 파일 이름을 지정하면 루틴이 개발자 대신 운영 체제에 필요한 호출을 합니다. 그 대신 파일 핸들을 사용하는 경우도 있습니다. 오브젝트 파스칼은 대부분의 파일 처리용 루틴을 제공합니다. 만일 제공되지 않는 경우에는 대체 루틴에 대해 설명합니다.

주의 오브젝트 파스칼 랭귀지는 대소문자를 구별하지 않지만 Linux 운영 체제는 구별합니다. 크로스 플랫폼 애플리케이션에서 파일을 사용할 때에는 대소문자에 주의하십시오.

파일 삭제

파일 삭제는 디스크에서 파일을 지우고 디스크의 디렉토리에서 항목을 제거합니다. 삭제된 파일을 복구하는 작업은 없으므로 일반적으로 사용자가 애플리케이션에서 파일의 삭제를 확인할 수 있습니다. 파일을 삭제하려면 다음과 같이 파일 이름을 *DeleteFile* 함수로 전달합니다.

```
DeleteFile(FileName);
```

*DeleteFile*은 파일을 삭제한 경우 *True*를 반환하며, 삭제하지 않은 경우(예: 파일이 없거나 읽기 전용인 경우)에는 *False*를 반환합니다. *DeleteFile*은 *FileName*에 의해 명명된 파일을 디스크에서 지웁니다.

파일 찾기

파일을 찾는 데 사용되는 세 가지 루틴은 *FindFirst*, *FindNext*, *FindClose*입니다. *FindFirst*는 지정된 디렉토리에서 주어진 속성 집합을 가지고 파일 이름의 첫 번째 인스턴스를 찾습니다. *FindNext*는 이전 *FindFirst* 호출에서 지정된 이름과 속성에 맞는 다음 항목을 반환합니다. *FindClose*는 *FindFirst*에 의해 할당된 메모리를 해제합니다. 항상 *FindClose*를 사용하여 *FindFirst/FindNext*를 종료해야 합니다. 파일이 존재하는지 알고 싶을 때에는 *FileExists* 함수를 사용하면 파일이 존재하는 경우 *True*를 반환하고 존재하지 않는 경우 *False*를 반환합니다.

세 가지 파일 찾기 루틴은 *TSearchRec*을 매개변수 중의 하나로 사용합니다. *TSearchRec*은 *FindFirst* 또는 *FindNext*로 찾아낸 파일 정보를 정의합니다. 다음은 *TSearchRec*에 대한 선언입니다.

```
type
  TFileName = string;
  TSearchRec = record
    Time: Integer;//Time contains the time stamp of the file.
    Size: Integer;//Size contains the size of the file in bytes.
```

```

Attr: Integer;//Attr represents the file attributes of the file.
Name: TFileName;//Name contains the filename and extension.
ExcludeAttr: Integer;
FindHandle: THandle;
FindData: TWin32FindData;//FindData contains additional information such as
//file creation time, last access time, long and short filenames.
end;

```

파일을 찾으려면 *TSearchRec* 타입의 매개변수 필드는 찾아낸 파일을 설명하도록 수정됩니다. 파일에 특정한 속성이 있는지 확인하기 위해 다음 속성 상수 또는 값에 대한 *Attr*을 검사할 수 있습니다.

표 4.9 속성 상수 및 값

상수	값	설명
<i>faReadOnly</i>	\$00000001	읽기 전용 파일
<i>faHidden</i>	\$00000002	숨김 파일
<i>faSysFile</i>	\$00000004	시스템 파일
<i>faVolumeID</i>	\$00000008	볼륨 ID 파일
<i>faDirectory</i>	\$00000010	디렉토리 파일
<i>faArchive</i>	\$00000020	아카이브 파일
<i>faAnyFile</i>	\$0000003F	모든 파일

속성에 대해 테스트하려면 속성 상수를 갖는 *Attr* 필드의 값을 **and** 연산자와 조합합니다. 파일에 해당 속성이 있으면 결과 값은 0보다 큼니다. 예를 들어, 발견된 파일이 숨김 파일인 경우 다음 표현식은 *True*로 평가됩니다. (*SearchRec.Attr* and *faHidden* >) 속성은 자신의 상수나 값을 OR를 사용하여 결합할 수 있습니다. 예를 들어, 일반 파일 외에 읽기 전용이나 숨김 파일을 찾으려면 (*faReadOnly* or *faHidden*)을 *Attr* 매개변수로 넘깁니다.

예 이 예는 폼에서 레이블과 *Search*라는 버튼, *Again*이라는 버튼을 사용합니다. 사용자가 *Search* 버튼을 누르면 지정된 경로에 있는 첫 번째 파일이 발견되고 파일의 이름과 바이트 수가 레이블의 캡션에 나타납니다. 사용자가 *Again* 버튼을 클릭할 때마다 그 다음으로 일치하는 파일 이름과 크기가 레이블에 나타납니다.

```

var
  SearchRec:TSearchRec;

procedure TForm1.SearchClick(Sender:TObject);
begin
  FindFirst('c:\Program Files\delphi6\bin\*.*', faAnyFile, SearchRec);
  Labell.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in
size';
end;

procedure TForm1.Edit1Click(Sender:TObject);
begin
  if (FindNext(SearchRec) = 0)
    Labell.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in
size';
  else
    FindClose(SearchRec);
end;

```

크로스플랫폼 애플리케이션의 경우, c:\Program Files\delphi6\bin*. *와 같은 모든 하드코딩된 경로명을 시스템의 정확한 경로명으로 교체하거나 이를 나타내기 위해 (Tools|Environment Options 선택 시 Environment Variables 페이지에서) 환경 변수를 사용해야 합니다.

파일 이름 재지정

파일 이름을 바꾸려면 다음과 같이 *RenameFile* 함수를 사용하면 됩니다.

```
function RenameFile(const OldFileName, NewFileName:string):Boolean;
```

여기서 *OldFileName*으로 식별한 파일 이름을 *NewFileName*에서 지정한 파일 이름으로 변경합니다. 이 작업이 성공하면 *RenameFile*은 *True*를 반환합니다. 만일 파일 이름을 재지정하지 못하면 예를 들어 *NewFileName*이라는 파일이 이미 있을 경우, *False*를 반환합니다. 예를 들면, 다음과 같습니다.

```
if not RenameFile('OLDNAME.TXT', 'NEWNAME.TXT') then
  ErrorMsg('Error renaming file!');
```

여기서 *RenameFile*을 사용하여 다른 드라이브로 파일을 이름 재지정(이동)할 수 없습니다. 먼저 파일을 복사한 다음 이전 파일을 삭제해야 합니다.

참고 VCL 내의 *RenameFile*은 Windows API *MoveFile* 함수에 기초를 둔 래퍼(wrapper)이므로 *MoveFile*은 드라이브 간에 동작하지 않습니다.

파일 date-time 루틴

FileAge, *FileGetDate*, *FileSetDate* 루틴은 운영 체제 date-time 값을 사용합니다. *FileAge*는 파일의 date-and-time 스탬프를 반환하거나 파일이 없으면 -1을 반환합니다. *FileSetDate*는 지정된 파일의 date-and-time 스탬프를 설정한 다음 성공하면 0을 반환하고 실패하면 오류 코드를 반환합니다. *FileGetDate*는 지정된 파일의 date-and-time 스탬프를 반환하거나 핸들이 유효하지 않으면 -1을 반환합니다.

대부분의 파일 처리 루틴처럼 *FileAge*는 문자열 파일 이름을 사용합니다. 그러나 *FileGetDate*와 *FileSetDate*는 *Handle* 타입을 매개변수로 사용합니다. Windows 파일 *Handle*에 액세스하려면 다음과 같이 합니다.

- Windows API *CreateFile* 함수를 호출합니다. *CreateFile*은 파일을 열거나 생성하는 데 사용하는 32비트 전용 함수이며 파일을 액세스하는 데 사용할 수 있는 *Handle*을 반환합니다.
- *TFileStream*을 인스턴스화하여 파일을 생성하거나 엽니다. 그런 다음 Windows의 파일 *Handle*과 마찬가지로 *Handle* 속성을 사용합니다. 자세한 내용은 4-54 페이지의 "파일 스트림 사용"을 참조하십시오.

파일 복사

런타임 라이브러리는 파일 복사를 위한 어떠한 루틴도 제공하지 않습니다. 하지만 Windows 전용 애플리케이션을 작성하는 경우 Windows API *CopyFile* 함수를 직접 호출하여 파일을 복사할 수 있습니다. 대부분의 Delphi 런타임 라이브러리 파일 루틴과 마찬가지로 *CopyFile*은 파일 이름을 *Handle*이 아닌 매개변수로 취합니다. 파일 복사 시

기존 파일의 파일 속성은 새 파일로 복사되지만 보안 속성은 복사되지 않습니다. *CopyFile* 은 또한 Delphi *RenameFile* 함수나 Windows API *MoveFile* 함수가 드라이브 간의 이름 변경/이동을 할 수 없으므로 드라이브 간에 파일을 이동할 때 유용합니다. 자세한 내용은 Microsoft Windows 온라인 도움말을 참조하십시오.

파일 I/O와 파일 형식

파일 I/O 작업을 할 때 파스칼 파일 형식, 파일 핸들, 파일 스트림 객체의 세 가지 타입을 사용할 수 있습니다. 다음 표는 이러한 형식들을 요약한 것입니다.

표 4.10 파일 I/O를 위한 파일 타입

파일 형식	설명
파스칼 파일 형식	시스템 유닛에 있습니다. 이러한 형식은 대개 "F: Text:" 또는 "F: File" 형식의 파일 변수에 사용됩니다. 파일은 타입이 지정된 (typed), 텍스트, 타입이 지정되지 않은 (untyped) 이라는 세 가지 형식이 있습니다. <i>AssignPrn</i> 과 <i>writeln</i> 과 같은 많은 파일 처리 루틴에서 이 형식을 사용합니다. 이러한 파일 형식은 이제 쓰이지 않으며 Windows 파일 핸들과 호환되지 않습니다. 이 형식들을 사용하려면 <i>오브젝트 파스칼 랭귀지 안내서</i> 를 참조하십시오.
파일 핸들	시스템 유닛에 있습니다. 많은 루틴이 핸들을 사용하여 파일을 식별합니다. 파일을 열거나 생성할 때 (예를 들어, <i>FileOpen</i> 또는 <i>FileCreate</i> 사용 시) 핸들을 얻습니다. 일단 핸들이 있으면 그 핸들을 사용하여 파일의 내용을 작업할 수 있는 루틴이 생깁니다 (한 줄 작성, 텍스트 읽기 등). Windows 프로그래밍에서 오브젝트 파스칼 파일 핸들은 Windows 파일 핸들 타입의 래퍼입니다. Windows 파일 Handle 을 사용하는 런타임 라이브러리 파일 처리 루틴은 일반적으로 Windows API 함수에 기초를 둔 래퍼입니다. 예를 들어, <i>FileRead</i> 는 Windows <i>ReadFile</i> 함수를 호출합니다. Delphi 함수는 오브젝트 파스칼 구문을 사용하고 경우에 따라 기본 매개변수 값을 제공하므로 Windows API에 편리한 인터페이스입니다. 이러한 루틴을 사용하는 것은 직접적이며 Windows API 파일 루틴에 익숙하고 편하다면 파일 I/O 작업 시 사용할 수도 있습니다.
파일 스트림	파일 스트림은 디스크 파일의 정보를 액세스하는 데 사용되는 <i>TFileStream</i> 클래스의 객체 인스턴스입니다. 파일 스트림은 파일 I/O를 위한 포팅 가능한 수준 높은 방법입니다. <i>TFileStream</i> 에는 파일 핸들에 액세스할 수 있는 <i>Handle</i> 속성이 있습니다. 다음 단원에서는 <i>TFileStream</i> 에 대해 설명합니다.

파일 스트림 사용

*TFileStream*은 애플리케이션에서 디스크상의 파일을 읽고 쓸 수 있게 하는 클래스입니다. 이것은 파일 스트림의 수준 높은 객체 표현에 사용됩니다. *TFileStream*은 영구성, 다른 스트림과의 상호 작용, 파일 I/O와 같은 많은 기능을 제공합니다.

- *TFileStream*은 스트림 클래스의 자손입니다. 이와 같은 파일 스트림 사용상의 한 가지 이점은 파일 스트림이 컴포넌트 속성을 영구적으로 저장하는 능력을 상속한다는 것입니다. 스트림 클래스는 *TFileer* 클래스, *TReader*, *TWriter*를 사용하여 객체를 디스크로 스트림합니다. 따라서 파일 스트림을 가질 때 컴포넌트 스트리밍 메커니즘을 위한 동일한 코드를 사용할 수 있습니다. 컴포넌트 스트리밍 시스템 사용에 대한 자세한

내용은 *TStream*, *TFile*, *TReader*, *TWriter*, *TComponent* 클래스 등에 대한 온라인 도움말을 참조하십시오.

- *TFileStream*은 다른 스트림 클래스와 쉽게 상호 작용할 수 있습니다. 예를 들어, *TFileStream*과 *TMemoryStream*을 사용하면 동적 메모리 블록을 디스크에 덤프할 수 있습니다.
- *TFileStream*은 파일 I/O용 기본 메소드와 속성을 제공합니다. 나머지 단원에서 파일 스트림의 이러한 면을 중점적으로 다룹니다.

파일 생성 및 열기

파일을 생성하거나 열고 파일 핸들에 대한 액세스 권한을 얻기 위해서는 단순히 *TFileStream*을 인스턴스화하면 됩니다. 이렇게 하면 명명된 파일을 열거나 작성하며 파일을 읽고 쓸 수 있는 메소드를 제공할 수 있습니다. 파일이 열리지 않으면 *TFileStream*은 예외를 발생시킵니다.

```
constructor Create(const filename: string; Mode:Word);
```

Mode 매개변수는 파일 스트림을 생성할 때 파일을 여는 방법을 지정합니다. *Mode* 매개변수는 개방 모드와 공유 모드 두 가지로 구성됩니다. 개방 모드는 다음 값 중의 하나여야 합니다.

표 4.11 개방 모드

값	의미
fmCreate	TFileStream은 파일을 주어진 이름으로 생성합니다. 주어진 이름을 가진 파일이 있을 경우 쓰기 모드에서 파일을 엽니다.
fmOpenRead	읽기 전용으로 파일을 엽니다.
fmOpenWrite	쓰기 전용으로 파일을 엽니다. 파일에 쓰는 내용은 현재 내용을 완전히 대체합니다.
fmOpenReadWrite	현재 내용을 대체하기보다는 수정하기 위해 파일을 엽니다.

공유 모드는 아래에 나열된 제한 사항을 가진 다음 값 중의 하나가 될 수 있습니다.

표 4.12 공유 모드

값	의미
fmShareCompat	공유는 FCB가 열리는 방법에 호환됩니다.
fmShareExclusive	다른 애플리케이션에서 파일을 절대 열 수 없습니다.
fmShareDenyWrite	다른 애플리케이션에서 읽기 위해 파일을 열 수는 있지만 쓰기 위해 열 수는 없습니다.
fmShareDenyRead	다른 애플리케이션에서 쓰기 위해 파일을 열 수는 있지만 읽기 위해 열 수는 없습니다.
fmShareDenyNone	다른 애플리케이션에서 파일 읽기나 쓰기가 금지되어 있습니다.

사용할 수 있는 공유 모드는 사용했던 개방 모드에 따라 다르다는 것에 유의하십시오. 다음 표는 각 개방 모드에서 사용할 수 있는 공유 모드를 나타냅니다.

표 4.13 각 개방 모드에서 사용할 수 있는 공유 모드

개방 모드	fmShare Compat	fmShare Exclusive	fmShareDeny Write	fmShareDeny Read	fmShareDeny None
fmOpenRead	사용 불가	사용 불가	사용 가능	사용 불가	사용 가능
fmOpenWrite	사용 가능	사용 가능	사용 불가	사용 가능	사용 가능
fmOpenReadWrite	사용 가능	사용 가능	사용 가능	사용 가능	사용 가능

파일 개방 모드 및 공유 모드 상수는 SysUtils 유닛에 정의되어 있습니다.

파일 핸들 사용

*TFileStream*을 인스턴스화하면 파일 핸들에 대한 액세스를 얻게 됩니다. 파일 핸들은 *Handle* 속성에 포함되어 있습니다. *Handle* 속성은 읽기 전용이며 파일이 열린 모드를 나타냅니다. 파일 *Handle*의 속성을 변경하려면 새로운 파일 스트림 객체를 만들어야 합니다.

일부 파일 처리 루틴은 window의 파일 핸들을 매개변수로 사용합니다. 일단 파일 스트림을 가지면 window의 파일 핸들을 사용하는 모든 상황에서 *Handle* 속성을 사용할 수 있습니다. 파일 스트림은 핸들 스트림과 달리 객체가 소멸될 때 파일 핸들을 닫는다는 점에 유의하십시오.

파일 읽기 및 쓰기

*TFileStream*에는 파일을 읽고 쓰는 몇 가지 다른 방법이 있습니다. 이러한 방법들은 다음을 수행하는지에 따라 구별됩니다.

- 읽거나 쓴 바이트 수를 반환합니다.
- 바이트 수를 알아야 합니다.
- 오류에 대한 예외를 발생시킵니다.

*Read*는 파일 스트림과 연결된 파일을 현재 *Position*에서 시작하여 *Count* 바이트까지 *Buffer*에 읽는 함수입니다. *Read*는 읽은 다음 파일의 현재 위치를 실제로 전송된 바이트 수만큼 앞으로 나가게 합니다. 다음은 *Read*에 대한 프로토타입입니다.

```
function Read(var Buffer; Count:Longint):Longint; override;
```

*Read*는 파일의 바이트 수를 모를 때 유용합니다. *Read*는 실제로 옮겨진 바이트 수를 반환하는데 파일 표시의 끝에 도달하면 *Count*보다 작아집니다.

*Write*는 *Position*에서 시작하여 파일 스트림과 연결된 파일에 *Buffer*로부터 나온 *Count* 바이트를 쓰는 함수입니다. 다음은 *Write*에 대한 프로토타입입니다.

```
function Write(const Buffer; Count:Longint):Longint; override;
```

파일에 기록한 후 *Write*는 현재 위치를 기록된 바이트 수만큼 앞으로 나가게 하고 실제로 기록된 바이트 수를 반환하는데 버퍼의 끝에 도달하면 *Count*보다 작아집니다.

그 반대 프로시저는 *ReadBuffer*와 *WriteBuffer*이며 *Read*, *Write*와 달리 읽거나 기록된 바이트 숫자를 반환하지 않습니다. 이러한 프로시저는 구조체에서 읽을 때처럼 바이트 수가 알려져 있고 또 필요한 경우에 유용합니다. *ReadBuffer*와 *WriteBuffer*는 오류시 (*EReadError*와 *EWriteError*) 예외를 발생시키지만 *Read* 메소드와 *Write* 메소드는 발생시키지 않습니다. 다음은 *ReadBuffer*와 *WriteBuffer*에 대한 프로토타입입니다.

```
procedure ReadBuffer(var Buffer; Count:Longint);
procedure WriteBuffer(const Buffer; Count:Longint);
```

이러한 메소드들은 *Read* 메소드와 *Write* 메소드를 호출하여 실제 읽기와 쓰기를 수행합니다.

문자열 읽기 및 쓰기

문자열을 *read* 함수 또는 *write* 함수에 전달하는 경우 정확한 구문을 알고 있어야 합니다. *read* 루틴과 *write* 루틴에 대한 *Buffer* 매개변수는 각각 **var** 타입과 **const** 타입입니다. 이 매개변수들은 타입이 지정되지 않았으므로 루틴은 변수의 주소를 사용합니다.

문자열을 사용할 때 가장 많이 사용하는 타입은 긴 문자열입니다. 그러나 긴 문자열을 *Buffer* 매개변수로 전달하면 올바른 결과를 얻을 수 없습니다. 긴 문자열에는 문자열에 있는 문자의 크기, 참조 카운트, 포인터가 들어 있습니다. 따라서 긴 문자열을 역참조 (*dereference*)하여 포인터 요소에만 그 결과를 나타내는 것은 아닙니다. 사용자가 먼저 해야 할 일은 문자열을 *Pointer* 또는 *PChar* 로 변환한 다음 문자열을 역참조하는 것입니다. 예를 들면, 다음과 같습니다.

```
procedure caststring;
var
  fs:TFileStream;
const
  s:string = 'Hello';
begin
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s)); // this will give you garbage
  fs.Write(PChar(s)^, Length(s)); // this is the correct way
end;
```

파일 찾기

가장 전형적인 파일 I/O 메커니즘에는 파일 안의 특정 위치에서 읽고 쓰기 위해 파일을 찾는 프로세스가 있습니다. 이러한 목적으로 *TFileStream*은 *Seek* 메소드를 갖고 있습니다. 다음은 *Seek*에 대한 프로토타입입니다.

```
function Seek(Offset:Longint; Origin:Word):Longint; override;
```

Origin 매개변수는 *Offset* 매개변수를 해석하는 방법을 나타냅니다. *Origin* 매개변수는 다음 값 중의 하나가 되어야 합니다.

값	의미
soFromBeginning	리소스의 시작 지점에서 <i>Offset</i> 이 시작합니다. <i>Seek</i> 은 <i>Offset</i> 위치로 이동합니다. <i>Offset</i> 은 ≥ 0 이어야 합니다.
soFromCurrent	리소스의 현재 위치에서 <i>Offset</i> 이 시작합니다. <i>Seek</i> 은 $Position + Offset$ 으로 이동합니다.
soFromEnd	리소스의 끝에서 <i>Offset</i> 이 시작합니다. 파일 끝으로부터의 바이트 수를 나타내려면 <i>Offset</i> 은 ≤ 0 이어야 합니다.

*Seek*은 스트림의 현재 *Position*을 재설정하여 표시된 오프셋만큼 이동합니다. *Seek*은 *Position* 속성의 새로운 값, 즉 리소스에서 새로운 현재 위치를 반환합니다.

파일 위치 및 크기

*TFileStream*은 파일의 현재 위치와 크기를 포함하는 속성을 가집니다. 이러한 속성들은 *Seek*, *read*, *write* 등의 메소드에 의해 사용됩니다.

*TFileStream*의 *Position* 속성은 현재 오프셋을 스트림에(스트림 데이터의 시작부터) 바이트 단위로 나타내는 데 사용됩니다. 다음은 *Position*에 대한 선언입니다.

```
property Position:Longint;
```

Size 속성은 스트림의 크기를 바이트 단위로 표시합니다. 이 속성은 파일을 잘라내기 위한 파일 끝 표시로 이용됩니다. 다음은 *Size*에 대한 선언입니다.

```
property Size:Longint;
```

*Size*는 스트림에 읽고 쓰는 루틴에서 내부적으로 사용됩니다.

Size 속성을 설정하면 파일 크기가 변경됩니다. 파일의 *Size*가 변경되지 않으면 예외가 발생합니다. 예를 들어, *fmOpenRead* 모드로 열린 파일의 *Size*를 변경하려고 하면 예외가 발생합니다.

복사

*CopyFrom*은 지정된 바이트 수만큼 한 파일 스트림에서 다른 스트림으로 복사합니다.

```
function CopyFrom(Source:TStream; Count:Longint):Longint;
```

*CopyFrom*을 사용하면 데이터를 복사할 때 버퍼를 만들고, 버퍼로 읽어 들이고, 버퍼에서 옮겨 작성하고, 버퍼를 해제할 필요가 없습니다.

*CopyFrom*은 *Source*에서 스트림으로 *Count* 바이트만큼 복사합니다. 그 다음 *CopyFrom*은 현재 위치를 *Count*만큼 이동하고 복사된 바이트 수를 반환합니다. *Count*가 0인 경우, *CopyFrom*은 읽기 전에 *Source* 위치를 0으로 설정한 다음 *Source*의 전체 내용을 스트림에 복사합니다. *Count*가 0보다 더 크거나 더 작은 경우, *CopyFrom*은 *Source*에 있는 현재 위치에서 읽습니다.

측정 변환

ConvUtils 유닛은 하나의 단위 집합에서 다른 단위 집합으로 측정을 변환하는 일반적인 용도로 사용할 수 있는 *Convert* 함수를 선언합니다. 피트나 인치 또는 일이나 주 등과 같은 호환 가능한 측정 단위 간의 변환을 수행할 수 있습니다. 동일한 유형의 것을 측정하는 단위는 동일한 *변환 패밀리*에 속합니다. 변환하려는 단위는 동일한 변환 패밀리에 있어야 합니다. 변환에 대한 자세한 내용은 다음 단원인 변환 수행과 온라인 도움말의 *Convert*를 참조하십시오.

StdConvs 유닛은 여러 변환 패밀리 및 각 변환 패밀리 내의 측정 단위를 정의합니다. 또한 *RegisterConversionType*과 *RegisterConversionFamily* 함수를 사용하여 사용자 지정된 변환 패밀리와 관련 단위를 생성할 수 있습니다. 변환 및 변환 단위 확장에 대한 자세한 내용은 새 측정 타입 추가 단원과 온라인 도움말의 *Convert*를 참조하십시오.

변환 수행

Convert 함수를 사용하여 간단하거나 복잡한 변환을 수행할 수 있습니다. 변환에는 복잡한 측정 유형 간의 변환을 수행하기 위한 간단한 구문과 보조 구문이 있습니다.

간단한 변환 수행

Convert 함수를 사용하여 하나의 단위 집합에서 다른 단위 집합으로 측정을 변환할 수 있습니다. *Convert* 함수는 동일한 유형의 측정 단위 간에 변환을 합니다(거리, 면적, 시간, 온도 등).

*Convert*를 사용하려면 변환할 단위와 대상 단위를 지정해야 합니다. *TConvType* 타입을 사용하여 측정 단위를 확인합니다.

예를 들어, *TConvType*은 다음과 같이 온도를 화씨에서 켈빈 온도로 변환합니다.

```
TempInKelvin := Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

복잡한 변환 수행

Convert 함수를 사용하여 두 측정 타입의 비율 간의 좀더 복잡한 변환을 수행할 수 있습니다. 복잡한 변환이 필요한 경우의 예로는 속도를 계산하기 위해 시간당 마일을 분당 미터로 변환하는 것과 유출량을 계산하기 위해 분당 갤론을 시간당 리터로 변환하는 것을 들 수 있습니다.

예를 들어, 다음 호출은 갤론당 마일을 리터당 킬로미터로 변환합니다.

```
nKPL := Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

변환할 단위는 동일한 것을 측정하는 동일한 변환 패밀리 내에 있어야 합니다. 단위가 호환되지 않는 경우 *Convert*는 *EConversionError* 예외를 일으킵니다. 두 *TConvType* 값이 동일한 변환 패밀리 내에 있는지 여부는 *CompatibleConversionTypes*를 호출하여 확인할 수 있습니다.

StdConvs 유닛은 *TConvType* 값의 여러 변환 패밀리를 정의합니다. 측정 단위의 이미 정의된 변환 패밀리 목록과 각 변환 패밀리 내의 측정 단위에 대한 자세한 내용은 온라인 도움말의 Conversion family variables를 참조하십시오.

새 측정 타입 추가

StdConvs 유닛에 이미 정의되어 있지 않은 측정 단위 간의 변환을 수행하려면 측정 단위 (*TConvType* 값)를 나타내는 새로운 변환 패밀리를 생성해야 합니다. 두 *TConvType* 값이 동일한 변환 패밀리에 등록된 경우 *Convert* 함수는 *TConvType* 값에서 나타나는 단위를 사용하여 측정 간의 변환을 할 수 있습니다.

RegisterConversionFamily 함수를 사용하여 변환 패밀리를 등록함으로써 *TConvFamily* 값을 먼저 얻어야 합니다. 새 변환 패밀리를 등록하거나 StdConvs 유닛 내의 전역 변수를 사용하여 등록함으로써 *TConvFamily* 값을 얻은 후 *RegisterConversionType* 함수를 사용하여 변환 패밀리에 새 단위를 추가할 수 있습니다. 다음 예제는 이를 수행하는 방법을 보여 줍니다.

더 자세한 예제는 표준 변환 유닛 (stdconvs.pas)의 소스 코드를 참조하십시오.(이 소스는 Delphi의 일부 버전에는 포함되어 있지 않습니다.)

간단한 변환 패밀리 생성과 단위 추가

새 변환 패밀리를 생성하고 새 측정 타입을 추가하는 경우의 한 가지 예가 정확성의 손실이 발생할 수 있는 긴 기간 사이의 변환(예를 들어, 월에서 세기로의 변환)을 수행할 때입니다.

더 자세히 설명하면 *cbTime* 패밀리는 일(day)을 기본 단위로 사용합니다. 기본 단위는 해당 패밀리 내의 모든 변환을 수행할 때 사용되는 것입니다. 따라서 모든 변환은 일을 기준으로 수행해야 합니다. 일과 월, 일과 연 등등 간의 정확한 변환이 없기 때문에 월 단위 이상의 단위(월, 연, 10년, 세기, 천년)를 사용하면 변환 시 정확성이 떨어질 수 있습니다. 월은 길이가 다르며 연의 경우도 윤년을 보정하기 위한 윤년, 윤달 등이 있습니다.

월보다 크거나 같은 측정 단위만을 사용하는 경우, 연 단위를 기본 단위로 하는 더 정확한 변환 패밀리를 생성할 수 있습니다. 이 예는 *cbLongTime*이라는 새 변환 패밀리를 생성합니다.

변수 선언

먼저 식별자의 변수를 선언해야 합니다. 식별자는 LongTime 변환 패밀리와 해당 멤버인 측정 단위에 사용됩니다.

```
var
  cbLongTime:TConvFamily;
  ltMonths:TConvType;
  ltYears:TConvType;
  ltDecades:TConvType;
  ltCenturies:TConvType;
  ltMillennia:TConvType;
```

변환 패밀리 등록

그런 다음 변환 패밀리를 등록합니다.

```
cbLongTime := RegisterConversionFamily ('Long Times');
```

*UnregisterConversionFamily*가 제공된다 하더라도 변환 패밀리를 정의하는 유닛이 런타임 시 제거되지 않는 한, 변환 패밀리를 등록 해제할 필요가 없습니다. 변환 패밀리는 애플리케이션이 종료될 때 자동으로 지워집니다.

측정 단위 등록

다음으로 방금 생성한 변환 패밀리 내에 측정 단위를 등록해야 합니다. 지정된 패밀리에 측정 단위를 등록하는 *RegisterConversionType* 함수를 사용합니다. 예제에서 연으로 되어 있는 기본 단위를 정의하고 다른 단위는 기본 단위와의 연관성을 나타내는 요소를 사용하여 정의됩니다. 따라서 LongTime 패밀리의 기본 단위가 연이므로 *ltMonths*의 요소는 1/12입니다. 변환할 단위의 설명도 포함시킵니다.

측정 단위를 등록하는 코드는 다음과 같습니다.

```
ltMonths:=RegisterConversionType(cbLongTime, 'Months', 1/12);
ltYears:=RegisterConversionType(cbLongTime, 'Years', 1);
ltDecades:=RegisterConversionType(cbLongTime, 'Decades', 10);
ltCenturies:=RegisterConversionType(cbLongTime, 'Centuries', 100);
ltMillennia:=RegisterConversionType(cbLongTime, 'Millennia', 1000);
```

새 단위 사용

이제 새로 등록한 단위를 사용하여 변환을 수행할 수 있습니다. 전역 *Convert* 함수는 *cbLongTime* 변환 패밀리에 등록한 모든 변환 타입 간의 변환을 할 수 있습니다.

따라서 다음 *Convert* 호출을 사용하는 대신,

```
Convert(StrToFloat(Edit1.Text), tuMonths, tuMillennia);
```

정확성이 더 높은 아래 것을 사용할 수 있습니다.

```
Convert(StrToFloat(Edit1.Text), ltMonths, ltMillennia);
```

변환 함수 사용

변환이 더 복잡한 경우에는 변환 계수를 사용하는 대신 변환을 수행하기 위한 함수를 지정하기 위해 다른 구문을 사용할 수 있습니다. 예를 들어, 서로 다른 온도 단위는 근본이 다르므로 변환 계수를 사용하여 변환할 수 없습니다.

StdConvs 유닛에서의 이 예제는 기본 단위를 변환하거나 기본 유닛으로 변환하기 위한 함수를 제공함으로써 변환 타입을 등록하는 방법을 보여 줍니다.

변수 선언

먼저 식별자의 변수를 선언합니다. 식별자는 *cbTemperature* 변환 패밀리에 사용되며 측정 단위는 해당 멤버입니다.

```
var
    cbTemperature:TConvFamily;
    tuCelsius:TConvType;
    tuKelvin:TConvType;
    tuFahrenheit:TConvType;
```

참고 여기 나열된 측정 단위는 실제로는 *StdConvs* 유닛에 등록된 온도 단위의 서브셋입니다.

변환 패밀리 등록

그런 다음 변환 패밀리를 등록합니다.

```
cbTemperature := RegisterConversionFamily ('Temperature');
```

기본 단위 등록

그런 다음 예에서 접시로 되어 있는 변환 패밀리의 기본 단위를 정의하고 등록합니다. 기본 단위의 경우에는 실제로 변환할 것이 없으므로 간단한 변환 계수를 사용할 수 있습니다.

```
tuCelsius:=RegisterConversionType(cbTemperature,'Celsius',1);
```

기본 단위로 또는 기본 단위를 다른 단위로 변환하기 위한 메소드 작성

접시로 또는 접시로부터의 변환을 수행하는 코드를 작성해야 하는데 이유는 간단한 변환 계수에 의존하지 않기 때문입니다. 이 함수는 다음과 같이 *StdConvs* 유닛에서 가져옵니다.

```
function FahrenheitToCelsius(const AValue:Double):Double;
begin
    Result := ((AValue - 32) * 5) / 9;
end;

function CelsiusToFahrenheit(const AValue:Double):Double;
begin
    Result := ((AValue * 9) / 5) + 32;
end;

function KelvinToCelsius(const AValue:Double):Double;
begin
    Result := AValue - 273.15;
end;

function CelsiusToKelvin(const AValue:Double):Double;
begin
    Result := AValue +273.15;
end;
```


기타 단위 등록

변환 함수가 있으므로 변환 패밀리 내에 다른 측정 단위를 등록할 수 있습니다. 단위의 설명을 포함시킬 수도 있습니다.

다음은 패밀리 안에 다른 유닛을 등록시키는 코드를 보여 줍니다.

```
tuKelvin := RegisterConversionType(cbTemperature, 'Kelvin', KelvinToCelsius,
    CelsiusToKelvin);
tuFahrenheit := RegisterConversionType(cbTemperature, 'Fahrenheit',
    FahrenheitToCelsius, CelsiusToFahrenheit);
```

새로운 단위 사용

이제 새로 등록한 유닛을 사용하여 애플리케이션에서 변환을 수행할 수 있습니다. 전역 *Convert* 함수는 *cbTemperature* 변환 패밀리에 등록했던 모든 변환 타입 간에 변환할 수 있습니다. 예를 들어, 다음 코드는 화씨에서 켈빈 온도로 값을 변환합니다.

```
Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

클래스를 사용한 변환 관리

변환 함수를 사용하면 언제든지 변환 단위를 등록할 수 있습니다. 그러나 본질적으로 동일한 작업을 하는 불필요한 많은 함수를 생성해야 하는 경우가 있습니다.

매개변수나 변수의 값만 다른 일련의 변환 함수를 작성할 수 있는 경우, 클래스를 만들어 변환을 처리할 수 있습니다. 예를 들어, 유로화(Euro)의 도입 이후 다양한 유럽 통화 간의 변환을 위한 표준 기술 집합이 있습니다. 달러와 유로화 간의 변환 계수와 달리 변환 계수가 상수일지라도 유럽 통화 간에 적절히 변환하는 간단한 변환 계수 방식을 사용할 수 없는 이유는 다음 두 가지입니다.

- 변환은 통화별 자릿수로 반올림해야 합니다.
- 변환 계수 방식은 표준 유로화 변환에 의해 지정된 계수의 역계수(reverse factor)를 사용합니다.

하지만 다음과 같은 변환 함수에 의해 모두 처리될 수 있습니다.

```
function FromEuro(const AValue: Double, Factor, FRound):Double;
begin
    Result := RoundTo(AValue * Factor, FRound);
end;

function ToEuro(const AValue: Double, Factor):Double;
begin
    Result := AValue / Factor;
end;
```

문제는 이 방식은 변환 함수에 대한 추가 매개변수를 필요로 한다는 것입니다. 즉, 모든 유럽 통화에 동일한 함수를 등록할 수 없다는 것을 의미합니다. 모든 유럽 통화마다 두 개의 새로운 변환 함수를 작성해야 하는 것을 피하기 위해서 클래스의 멤버로 만들면 동일한 두 함수를 사용할 수 있습니다.

변환 클래스 만들기

클래스가 *TConvTypeFactor*의 자손이어야 합니다. *TConvTypeFactor*에는 변환 패밀리
의 기본 단위 (지금의 경우에는 유로화) 변환을 위한 두 개의 메소드, *ToCommon*
과 *FromCommon*이 있습니다. 변환 단위 등록 시 직접 사용하는 함수와 마찬가지로 이
러한 메소드에는 추가 매개변수가 없으므로 반올림할 자릿수와 변환 클래스의 private
멤버로서의 변환 계수를 보충해야 합니다. demos\ConvertIt 디렉토리 (euroconv.pas
참조)에 있는 EuroConv 예제에 나타나 있습니다.

```

type
  TConvTypeEuroFactor = class(TConvTypeFactor)
  private
    FRound: TRoundToRange;
  public
    constructor Create(const AConvFamily:TConvFamily;
      const ADescription:string; const AFactor:Double;
      const ARound: TRoundToRange);
    function ToCommon(const AValue:Double):Double; override;
    function FromCommon(const AValue:Double):Double; override;
  end;
end;

```

생성자는 그러한 private 멤버에 값을 할당합니다.

```

constructor TConvTypeEuroFactor.Create(const AConvFamily:TConvFamily;
  const ADescription:string; const AFactor:Double;
  const ARound: TRoundToRange);
begin
  inherited Create(AConvFamily, ADescription, AFactor);
  FRound := ARound;
end;

```

다음 두 변환 함수는 이러한 private 멤버를 사용합니다.

```

function TConvTypeEuroFactor.FromCommon(const AValue:Double):Double;
begin
  Result := SimpleRoundTo(AValue * Factor, FRound);
end;

function TConvTypeEuroFactor.ToCommon(const AValue:Double):Double;
begin
  Result := AValue / Factor;
end;

```

변수 선언

변환 클래스가 있으므로 식별자를 선언하여 다른 변환 패밀리를 사용하여 시작합니다.

```

var
  euEUR: TConvType; { EU euro }
  euBEF: TConvType; { Belgian francs }
  euDEM: TConvType; { German marks }
  euGRD: TConvType; { Greek drachmas }
  euESP: TConvType; { Spanish pesetas }
  euFFR: TConvType; { French francs }

```

```

euIEP: TConvType; { Irish pounds }
euITL: TConvType; { Italian lire }
euLUF: TConvType; { Luxembourg francs }
euNLG: TConvType; { Dutch guilders }
euATS: TConvType; { Austrian schillings }
euPTE: TConvType; { Portuguese escudos }
euFIM: TConvType; { Finnish marks }
euUSD: TConvType; { US dollars }
euGBP: TConvType; { British pounds }
euJPY: TConvType; { Japanese yen }

```

변환 패밀리 및 기타 단위 등록

이제 새 변환 클래스를 사용하여 변환 패밀리 및 유럽 화폐 단위를 등록할 준비가 되었습니다.

```

cbEuro := RegisterConversionFamily ('European currency');
...
// Euro's various conversion types
euEUR := RegisterEuroConversionType(cbEuro, SEURDescription, EURToEUR, EURSubUnit);
euBEF := RegisterEuroConversionType(cbEuro, SBEFDescription, BEFToEUR, BEFSubUnit);
euDEM := RegisterEuroConversionType(cbEuro, SDEMDescription, DEMToEUR, DEMSubUnit);
euGRD := RegisterEuroConversionType(cbEuro, SGRDDescription, GRDToEUR, GRDSubUnit);
euESP := RegisterEuroConversionType(cbEuro, SESPDescription, ESPToEUR, ESPSubUnit);
euFFR := RegisterEuroConversionType(cbEuro, SFFRDescription, FFRToEUR, FFRSubUnit);
euIEP := RegisterEuroConversionType(cbEuro, SIEPDescription, IEPToEUR, IEPSubUnit);
euITL := RegisterEuroConversionType(cbEuro, SITLDescription, ITLToEUR, ITLSubUnit);
euLUF := RegisterEuroConversionType(cbEuro, SLUFDescription, LUFToEUR, LUFSubUnit);
euNLG := RegisterEuroConversionType(cbEuro, SNLGDescription, NLGToEUR, NLGSubUnit);
euATS := RegisterEuroConversionType(cbEuro, SATSDescription, ATSToEUR, ATSSubUnit);
euPTE := RegisterEuroConversionType(cbEuro, SPTEDescription, PTEToEUR, PTESubUnit);
euFIM := RegisterEuroConversionType(cbEuro, SFIMDescription, FIMToEUR, FIMSubUnit);
euUSD := RegisterEuroConversionType(cbEuro, SUSDDescription,
    ConvertUSDToEUR, ConvertEURToUSD);
euGBP := RegisterEuroConversionType(cbEuro, SGBPDescription,
    ConvertGBPToEUR, ConvertEURToGBP);
euJPY := RegisterEuroConversionType(cbEuro, SJPYDescription,
    ConvertJPYToEUR, ConvertEURToJPY);

```

*RegisterEuroConversionType*은 화폐 타입의 등록을 단순화시키는 랩퍼 함수입니다. 자세한 내용은 예제 코드를 참조하십시오.

새 단위 사용

이제 새로 등록한 단위를 사용하여 애플리케이션에서 변환을 수행할 수 있습니다. 전역 *Convert* 함수는 새로운 *cbEuro* 패밀리에 등록했던 모든 유럽 통화 간에 변환할 수 있습니다. 예를 들어, 다음 코드는 이탈리아 리라 값을 독일의 마르크로 변환합니다.

```
Edit2.Text = FloatToStr(Convert(StrToFloat(Edit1.Text), euITL, euDEM));
```

데이터 타입 정의

오브젝트 파스칼에는 이미 정의된 데이터 타입이 많이 들어 있습니다. 이러한 이미 정의된 타입을 사용하여 애플리케이션에서 필요로 하는 내용에 따라 새로운 타입을 만들 수 있습니다. 타입에 대한 개요는 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.

5

애플리케이션, 컴포넌트, 라이브러리 구축

이 장에서는 Delphi를 사용하여 애플리케이션, 라이브러리 및 컴포넌트를 만드는 방법에 대한 개요를 제공합니다.

애플리케이션 생성

Delphi의 주된 용도는 다음과 같은 애플리케이션을 디자인하고 구축하는 것입니다.

- GUI 애플리케이션
- 콘솔 애플리케이션
- 서비스 애플리케이션 (Windows 애플리케이션에만 해당)
- 패키지 및 DLL

GUI 애플리케이션은 일반적으로 사용하기 쉬운 인터페이스를 갖습니다. 콘솔 애플리케이션은 콘솔 창에서 실행합니다. 서비스 애플리케이션은 Windows 서비스로서 실행합니다. 이러한 타입의 애플리케이션은 시동 코드를 가지고 실행 파일로서 컴파일합니다.

결과적으로 패키지나 동적 연결 라이브러리를 만드는 패키지 및 DLL과 같은 다른 유형의 프로젝트를 만들 수 있습니다. 이러한 애플리케이션은 시동 코드 없이 실행 코드를 만듭니다. 9 페이지의 "패키지와 DLL 생성"을 참조하십시오.

GUI 애플리케이션

GUI(그래픽 사용자 인터페이스) 애플리케이션은 창, 메뉴, 대화 상자와 같은 그래픽 기능과 애플리케이션을 사용하기 쉽게 하는 기능을 사용하여 디자인된 애플리케이션입니다. GUI 애플리케이션을 컴파일할 때 시동 코드와 함께 실행 파일이 생성됩니다. 실행 파일은 보통 사용자 프로그램의 기본적인 기능을 제공하고 간단한 프로그램은 대체로

실행 파일 하나로만 구성됩니다. DLL, 패키지 및 실행 파일을 지원하는 파일을 호출하여 애플리케이션을 확장할 수 있습니다.

Delphi는 다음 두 가지 애플리케이션 UI 모델을 제공합니다.

- 단일 문서 인터페이스(SDI)
- 다중 문서 인터페이스(MDI)

애플리케이션의 구현 모델 외에 프로젝트의 디자인 타임 행동과 애플리케이션의 런타임 행동은 IDE에서 프로젝트 옵션을 설정하여 처리할 수 있습니다.

사용자 인터페이스 모델

모든 폼은 단일 문서 인터페이스(SDI) 또는 다중 문서 인터페이스(MDI) 폼으로 구현될 수 있습니다. MDI 애플리케이션에서는 단일 부모 창에서 둘 이상의 문서나 자식 창을 열 수 있습니다. MDI는 스프레드시트나 워드 프로세서와 같은 애플리케이션에서는 일반적인 인터페이스입니다. 반면 SDI 애플리케이션은 단일 문서 뷰를 가집니다. 폼을 SDI 애플리케이션으로 만들려면 *Form* 객체의 *FormStyle* 속성을 *fsNormal*로 설정합니다.

애플리케이션용 UI 개발에 대한 자세한 내용은 6장 "애플리케이션 사용자 인터페이스 개발"을 참조하십시오.

SDI 애플리케이션

새 SDI 애플리케이션을 만들려면 다음과 같이 합니다.

- 1 File|New|Other를 선택하여 New Items 대화 상자를 나타냅니다.
- 2 Projects 페이지를 클릭하고 SDI Application을 선택합니다.
- 3 OK를 클릭합니다.

기본적으로 *Form* 객체의 *FormStyle* 속성은 *fsNormal*로 설정되므로 Delphi에서는 모든 새 애플리케이션이 SDI 애플리케이션이라고 가정합니다.

MDI 애플리케이션

새 MDI 애플리케이션을 생성하려면 다음과 같이 합니다.

- 1 File|New|Other를 선택하여 New Items 대화 상자를 나타냅니다.
- 2 Projects 페이지를 클릭하고 MDI Application을 선택합니다.
- 3 OK를 클릭합니다.

MDI 애플리케이션은 SDI 애플리케이션보다 디자인하기가 다소 복잡하므로 보다 계획적이어야 합니다. MDI 애플리케이션은 클라이언트 창 내에 상주하는 자식 창을 생성하고 메인 폼은 자식 폼을 포함합니다. *TForm* 객체의 *FormStyle* 속성을 설정하여 폼이 자식(*fsMDIForm*)인지 또는 메인 폼(*fsMDIChild*)인지 지정합니다. 자식 폼의 속성 재설정을 피하려면 자식 폼의 기본 클래스를 정의하고 이 클래스에서 각 자식 폼을 파생시키는 것이 좋습니다.

IDE, 프로젝트 및 컴파일 옵션 설정

Project|Options 메뉴를 선택해서 프로젝트에 다양한 옵션을 지정할 수 있습니다. 자세한 내용은 온라인 도움말을 참조하십시오.

기본 프로젝트 옵션 설정

차후의 모든 프로젝트에 적용될 기본 옵션을 변경하려면 Project Options 대화 상자에서 옵션을 설정하고 창의 오른쪽 하단에 있는 Default 상자를 선택 표시합니다. 모든 새 프로젝트는 기본적으로 현재 옵션을 사용합니다.

프로그래밍 템플릿

프로그래밍 템플릿은 공통적으로 사용되는 "뼈대" 구조이므로 소스 코드에 추가하여 채워 넣을 수 있습니다. Delphi에는 배열, 클래스, 함수 선언 및 많은 문장 등의 몇 가지 표준 코드 템플릿이 들어 있습니다.

자주 사용하는 코드 구조에 대해 사용자 고유의 템플릿을 작성할 수도 있습니다. 예를 들어, 코드에서 **for** 루프를 사용하고 싶을 경우 다음과 같은 템플릿을 삽입할 수 있습니다.

```
for := to do
begin

end;
```

코드 에디터에 코드 템플릿을 삽입하려면 *Ctrl+J*를 누르고 사용하려는 템플릿을 선택합니다. 사용자가 직접 만든 템플릿도 이 컬렉션에 추가할 수 있습니다. 다음과 같은 방법으로 템플릿을 추가합니다.

- 1 Tools|Editor Options를 선택합니다.
- 2 Code Insight 탭을 클릭합니다.
- 3 템플릿 섹션에서 Add를 클릭합니다.
- 4 단축 이름 뒤에 템플릿 이름을 입력하고 새 템플릿에 대한 간단한 설명을 입력합니다.
- 5 템플릿 코드를 Code 텍스트 상자에 추가합니다.
- 6 OK를 클릭합니다.

콘솔 애플리케이션

콘솔 애플리케이션은 그래픽 인터페이스 없이 일반적으로 콘솔 창에서 실행되는 32비트 프로그램입니다. 콘솔 애플리케이션은 대체로 사용자 입력이 많지 않으며 제한된 함수 집합을 수행합니다.

새 콘솔 애플리케이션을 생성하려면 다음과 같이 합니다.

- 1 File|New|Other를 선택한 다음 New Items 대화 상자에서 Console Application을 선택합니다.

Delphi는 콘솔형 소스 파일에 대한 프로젝트 파일을 생성하고 코드 에디터를 표시합니다.

참고 새 콘솔 애플리케이션을 생성하면 IDE는 새 폼을 만들지 않고 코드 에디터만 표시됩니다.

서비스 애플리케이션

서비스 애플리케이션은 클라이언트 애플리케이션의 요청을 받아서 이 요청을 처리하고 클라이언트 애플리케이션에 정보를 반환합니다. 서비스 애플리케이션은 일반적으로 백그라운드에서 실행되며 사용자 입력을 많이 필요로 하지 않습니다. 서비스 애플리케이션의 예로는 웹, FTP 또는 전자 메일 서버가 있습니다.

Win32 서비스를 구현하는 애플리케이션을 작성하려면 File|New를 선택한 다음 New Items 페이지에서 Service Application을 선택합니다. 이렇게 하면 *TServiceApplication* 타입의 *Application*이라는 전역 변수가 프로젝트에 추가됩니다.

서비스 애플리케이션을 만들면 서비스 (*TService*)에 해당하는 디자이너의 창이 보입니다. Object Inspector에서 속성 및 이벤트 핸들러를 설정하여 서비스를 구현합니다. New Items 대화 상자에서 Service를 선택하여 서비스 애플리케이션에 다른 서비스를 추가할 수 있습니다. 서비스 애플리케이션이 아닌 애플리케이션에 서비스를 추가하지 않도록 합니다. 그 애플리케이션에 *TService* 객체를 추가할 수는 있지만 그럴 경우 필요한 이벤트를 생성하지 못하거나 서비스 대신 적절한 Windows 호출을 하지 못할 것입니다.

서비스 애플리케이션을 구축하면 서비스 제어 관리자 (SCM)에 해당 서비스를 설치할 수 있습니다. 그러면 다른 애플리케이션은 SCM에 요청을 보내어 그 서비스를 실행할 수 있습니다.

애플리케이션의 서비스를 설치하려면 /INSTALL 옵션으로 서비스 애플리케이션을 실행합니다. 그러면 애플리케이션은 해당 서비스를 설치하고 종료합니다. 이 때 서비스가 성공적으로 설치되었으면 확인 메시지를 나타냅니다. /SILENT 옵션으로 서비스 애플리케이션을 실행하면 확인 메시지를 나타내지 않게 할 수 있습니다.

서비스를 설치 해제하려면 명령줄에서 /UNINSTALL 옵션으로 서비스 애플리케이션을 실행합니다 (설치 해제 시에도 /SILENT 옵션으로 확인 메시지를 나타내지 않게 할 수 있습니다).

예 다음 예제 서비스는 포트가 80으로 설정된 *TServerSocket*을 갖습니다. 이 포트는 웹 서버에 요청하기 위한 웹 브라우저와 웹 브라우저에 응답하기 위한 웹 서버를 위한 기본 포트입니다. 이 예제는 C:\Temp 디렉토리에 WebLogxxx.log라는 텍스트 문서를 만듭니다 (여기서 xxx는 ThreadID입니다). 주어진 포트에는 하나의 서버만 수신 대기해야 합니다. 따라서 웹 서버가 있을 경우 수신 대기하지 않는지 (서비스가 중지되었는지) 확인해야 합니다.

결과를 보려면 로컬 시스템에서 웹 브라우저를 열고 주소에 'localhost'(인용 부호 없이)를 입력합니다. 브라우저는 시간 초과가 발생하겠지만 C:\temp 디렉토리에 weblogxxx.log라는 파일이 생깁니다.

1 이 예제를 만들려면 File|New 메뉴를 선택하고 New Items 대화 상자에서 Service Application을 선택합니다. Service1이라는 창이 나타납니다. 컴포넌트 팔레트의 Internet 페이지에서 ServerSocket 컴포넌트를 서비스 창 (Service1)에 추가합니다.

- 2 그런 다음 TService1 클래스에 TMemoryStream 타입의 private 데이터 멤버를 추가합니다. 유닛의 interface 섹션은 다음과 같이 보일 것입니다.

```
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;

type
  TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender:TObject;
      Socket:TCustomWinSocket);
    procedure Service1Execute(Sender:TService);
  private
    { Private declarations }
    Stream: TMemoryStream; // Add this line here
  public
    function GetServiceController: PServiceController; override;
    { Public declarations }
  end;

var
  Service1: TService1;
```

- 3 그런 다음 1 단계에서 추가했던 컴포넌트인 ServerSocket1을 선택합니다. Object Inspector에서 OnClientRead 이벤트를 더블 클릭하고 다음과 같은 이벤트 핸들러를 추가합니다.

```
procedure TService1.ServerSocket1ClientRead(Sender:TObject;
  Socket:TCustomWinSocket);
var
  Buffer:PChar;

begin
  Buffer := nil;

while Socket.ReceiveLength > 0 do begin
  Buffer := AllocMem(Socket.ReceiveLength);
  try
    Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
    Stream.Write(Buffer^, StrLen(Buffer));
  finally
    FreeMem(Buffer);
  end;

  Stream.Seek(0, soFromBeginning);
  Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
end;
end;
```

- 4 마지막으로 ServiceSocket이 아닌 창의 클라이언트 영역을 클릭하여 Service1을 선택합니다. Object Inspector에서 OnExecute 이벤트를 더블 클릭하고 다음과 같은 이벤트 핸들러를 추가합니다.

```
procedure TService1.Service1Execute(Sender:TService);
begin
  Stream := TMemoryStream.Create;
```

```

try
  ServerSocket1.Port := 80; // WWW port
  ServerSocket1.Active := True;

  while not Terminated do begin
    ServiceThread.ProcessRequests(True);
  end;

  ServerSocket1.Active := False;
finally
  Stream.Free;
end;
end;

```

서비스 애플리케이션 작성 시 다음 사항을 잘 알고 있어야 합니다.

- 서비스 스레드
- 서비스 이름 속성
- 서비스 디버깅

서비스 스레드

각 서비스는 자체의 스레드 (*TServiceThread*) 를 가지고 있습니다. 따라서 서비스 애플리케이션이 하나 이상의 서비스를 구현할 경우 서비스를 thread-safe로 구현해야 합니다. *TServiceThread*가 디자인되어 있으므로 *TService OnExecute* 이벤트 핸들러에서 서비스를 구현할 수 있습니다. 서비스 스레드는 새로운 요청을 처리하기 전에 서비스의 *OnStart* 및 *OnExecute* 핸들러를 호출하는 루프를 포함하는 자체의 *Execute* 메소드를 갖습니다.

서비스 요청은 시간이 오래 걸릴 수 있고 서비스 애플리케이션은 하나 이상의 클라이언트로부터 동시에 요청을 받기 때문에 각 요청에 대해 *TServiceThread*가 아닌, *TThread*로부터 파생된 스레드를 새로 생성하고 그 서비스의 구현을 새 스레드의 *Execute* 메소드에 옮기는 것이 훨씬 효율적입니다. 이를 통해 서비스의 *OnExecute* 핸들러를 완료할 때까지 기다리지 않고 서비스 스레드의 *Execute* 루프에서 계속적으로 새로운 요청을 처리할 수 있습니다. 다음 예제에서 보여 줍니다.

예 이 서비스는 0.5초마다 표준 스레드 안에서 경고음을 냅니다. 그리고 서비스에 대한 명령에 따라 스레드의 일시 정지, 계속, 멈춤 등을 처리합니다.

- 1 File|New|Other를 선택한 다음 New Items 대화 상자에서 Service Application을 선택합니다. Service1이라는 창이 나타납니다.
- 2 유닛의 interface 섹션에서 TSparkyThread라는 *TThread*의 새 자손을 선언합니다. 이것은 서비스를 처리하는 스레드입니다. 다음과 같이 선언합니다.

```

TSparkyThread = class(TThread)
  public
    procedure Execute; override;
end;

```

- 3 그런 다음 유닛의 implementation 섹션에 TSparkyThread 인스턴스에 대한 전역 변수를 만듭니다.

```

var
  SparkyThread: TSparkyThread;

```

- 4 TSparkyThread Execute 메소드(스레드 함수)를 실행하려면 implementation 섹션에 다음 코드를 추가합니다.

```
procedure TSparkyThread.Execute;
begin
  while not Terminated do
  begin
    Beep;
    Sleep(500);
  end;
end;
```

- 5 서비스 창(Service1)을 선택하고 Object Inspector에서 OnStart 이벤트를 더블 클릭합니다. 다음과 같은 OnStart 이벤트 핸들러를 추가합니다.

```
procedure TService1.Service1Start(Sender: TService; var Started:Boolean);
begin
  SparkyThread := TSparkyThread.Create(False);
  Started := True;
end;
```

- 6 Object Inspector에서 OnContinue 이벤트를 더블 클릭합니다. 다음과 같은 OnContinue 이벤트 핸들러를 추가합니다.

```
procedure TService1.Service1Continue(Sender: TService; var Continued:Boolean);
begin
  SparkyThread.Resume;
  Continued := True;
end;
```

- 7 Object Inspector에서 OnPause 이벤트를 더블 클릭합니다. 다음과 같은 OnPause 이벤트 핸들러를 추가합니다.

```
procedure TService1.Service1Pause(Sender: TService; var Paused:Boolean);
begin
  SparkyThread.Suspend;
  Paused := True;
end;
```

- 8 마지막으로 Object Inspector에서 OnStop 이벤트를 더블 클릭하고 다음과 같은 OnStop 이벤트 핸들러를 추가합니다.

```
procedure TService1.Service1Stop(Sender: TService; var Stopped:Boolean);
begin
  SparkyThread.Terminate;
  Stopped := True;
end;
```

서버 애플리케이션을 개발할 때 제공되는 서비스의 성격, 예상 연결 수 및 서비스를 실행하는 컴퓨터의 예상 프로세서 수에 따라 새로운 스레드를 갖도록 선택합니다.

서비스 이름 속성

VCL은 Windows 플랫폼에 서비스 애플리케이션을 만드는 클래스를 제공합니다(크로스 플랫폼 애플리케이션에 대해서는 사용 불가). 이러한 클래스에는 *TService*와 *TDependency*가 포함됩니다. 이 클래스들을 사용할 때 여러 가지 이름 속성이 혼란스러울 수 있습니다. 이 단원에서 그 차이점을 설명합니다.

서비스는 암호, 관리자좌와 에디터 윈도우에 표시하는 표시 이름 및 실제 서비스의 이름과 관련된 서비스 시작 이름이라는 사용자 이름을 갖습니다. 종속성은 서비스일 수 있고 그룹의 순서를 매기면서 로드될 수 있습니다. 또한 이름 및 표시 이름을 갖습니다. 그리고 서비스 객체는 *TComponent*에서 파생된 것이므로 *Name* 속성을 상속합니다. 다음 단원에서는 이름 속성을 요약합니다.

TDependency 속성

*TDependency DisplayName*은 표시 이름이면서 서비스의 실제 이름입니다. *TDependency Name* 속성과 거의 항상 동일합니다.

TService 이름 속성

TService Name 속성은 *TComponent*에서 상속됩니다. 이것은 컴포넌트의 이름이면서 서비스의 이름이기도 합니다. 종속성이 서비스인 경우, 이 속성은 *TDependency Name* 및 *DisplayName* 속성과 동일합니다.

*TService*의 *DisplayName*은 서비스 관리자 창에 표시되는 이름입니다. 이 이름이 때로는 실제 서비스 이름 (*TService.Name*, *TDependency.DisplayName*, *TDependency.Name*)과 다를 수 있습니다. 종속성의 *DisplayName*과 서비스의 *DisplayName*이 일반적으로 다르다는 것에 유의하십시오.

서비스 시작 이름은 서비스 표시 이름이나 실제 서비스 이름과 다릅니다. *ServiceStartName*은 서비스 제어 관리자에서 선택된 시작 대화 상자에 입력하는 사용자 이름입니다.

서비스 디버깅

서비스 애플리케이션 디버깅은 시간 간격이 짧기 때문에 까다로울 수 있습니다.

- 1 먼저 디버거 안에서 애플리케이션을 실행하고 로딩을 완료할 때까지 기다립니다.
- 2 제어판이나 명령줄에서 빨리 서비스를 시작합니다.

```
start MyServ
```

서비스가 실행되지 않으면 애플리케이션이 종료되기 때문에 서비스를 애플리케이션 시작 후 15-30초 이내에 빨리 실행해야 합니다.

또 다른 방법은 서비스 애플리케이션이 이미 실행 중일 때 서비스 애플리케이션 프로세스를 덧붙이는 것입니다. (즉, 서비스를 먼저 시작하고 난 다음 디버거에 덧붙이는 것입니다.) 서비스 애플리케이션 프로세스에 덧붙이려면 Run|Attach To Process를 선택한 다음 대화 상자에서 서비스 애플리케이션을 선택합니다.

어떤 경우에는 권한이 충분하지 않아서 이 두 번째 방법이 실패할 수도 있습니다. 그럴 경우 서비스 제어 관리자를 사용하여 서비스가 디버거와 함께 작동하게 할 수 있습니다.

- 1 먼저 다음의 레지스트리 위치에 **Image File Execution Options**라는 키를 만듭니다.
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
- 2 서비스와 동일한 이름을 갖는 하위 키를 만듭니다 (예를 들면, MYSERV.EXE). 이 하위 키에 Debugger라는 이름을 갖는 REG_SZ 타입의 값을 추가합니다. 문자열 값으로 Delphi32.exe의 전체 경로를 사용합니다.
- 3 서비스 제어판 애플릿에서 서비스를 선택한 다음 시작을 클릭하고 Allow Service to Interact with Desktop을 선택 표시합니다.

패키지와 DLL 생성

DLL(동적 연결 라이브러리)은 애플리케이션에 기능을 제공하기 위해 실행 파일과 함께 작업하는 컴파일된 코드의 모듈입니다. 크로스 플랫폼 프로그램에서 DLL을 만들 수 있습니다. 하지만 Linux에서 DLL과 패키지는 공유 객체로 다시 컴파일됩니다.

패키지는 Delphi 애플리케이션, IDE 또는 두 가지 모두에서 사용되는 특별한 DLL입니다. 패키지의 종류에는 런타임 패키지와 디자인 타임 패키지가 있습니다. 런타임 패키지는 프로그램 실행 중에 프로그램에 기능을 제공합니다. 디자인 타임 패키지는 IDE의 기능을 확장합니다.

다음과 같은 컴파일러 지시어를 라이브러리 프로젝트 파일에 둘 수 있습니다.

표 5.1 라이브러리용 컴파일러 지시어

컴파일러 지시어	설명
{\$LIBPREFIX 'string'}	출력 파일 이름에 지정된 접두어를 추가합니다. 예를 들어, 디자인 타임 패키지에 {\$LIBPREFIX 'dcl'}을 지정하거나 {\$LIBPREFIX '}'를 사용하여 접두어를 없앨 수 있습니다.
{\$LIBSUFFIX 'string'}	출력 파일 이름의 확장자 앞에 지정된 접미어를 추가합니다. 예를 들어, something.pas에서 {\$LIBSUFFIX '-2.1.3'}을 사용하여 something-2.1.3.bpl을 만듭니다.
{\$LIBVERSION 'string'}	출력 파일 이름에서 .bpl 확장자 뒤에 두 번째 확장자를 추가합니다. 예를 들어, something.pas에서 {\$LIBVERSION '2.1.3'}을 사용하여 something.bpl.2.1.3을 만듭니다.

자세한 내용은 11장 "패키지와 컴포넌트 사용"을 참조하십시오.

패키지와 DLL 사용하는 경우

Delphi로 작성된 대부분의 애플리케이션에 대해서 패키지는 더 많은 유연성을 제공하고 DLL보다 더 쉽게 생성될 수 있습니다. 그러나 패키지보다 DLL이 더 적합한 경우도 있습니다.

- 코드 모듈이 Delphi 이외의 애플리케이션에서 호출되는 경우
- 웹 서버의 기능을 확장하는 경우
- 타사 개발자가 사용할 코드 모듈을 생성하는 경우
- 프로젝트가 OLE 컨테이너인 경우

런타임 타입 정보(RTTI)를 DLL 간에 또는 DLL에서 실행 파일로 전달할 수는 없습니다. 왜냐하면 모든 DLL은 자기 자신의 심볼 정보를 갖고 있기 때문입니다. DLL로부터 *TStrings* 객체를 전달해야 하는 경우에는 **is** 또는 **as** 연산자를 사용하므로 DLL보다는 패키지로 만들어야 합니다. 패키지들은 심볼 정보를 공유합니다.

데이터베이스 애플리케이션 개발

참고 Delphi의 모든 버전이 데이터베이스 지원을 포함하지는 않습니다.

Delphi의 장점 중 하나는 고급 데이터베이스 애플리케이션에 대한 지원입니다. Delphi는 애플리케이션 간에 데이터를 공유할 수 있게 하면서 SQL 서버 및 Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix, DB2와 같은 데이터베이스에 연결할 수 있게 하는 틀을 지원합니다.

Delphi에는 데이터베이스에 액세스하여 들어 있는 정보를 나타내는 데 사용하는 컴포넌트가 많이 있습니다. 컴포넌트 팔레트에서 데이터베이스 컴포넌트는 데이터 액세스 메커니즘과 기능에 따라 그룹화됩니다.

표 5.2 컴포넌트 팔레트의 Database 페이지

Palette 페이지	내용
BDE	데이터베이스와의 상호 작용을 위해 대규모 API와 BDE(Borland Database Engine)를 사용하는 컴포넌트. BDE는 폭넓은 범위의 함수들을 지원하고 부수적으로 Database Desktop, Database Explorer, SQL Monitor 및 BDE Administrator를 포함하여 대부분의 지원 유틸리티가 있습니다. 자세한 내용은 20장 "Borland Database Engine 사용"을 참조하십시오.
ADO	Microsoft에서 개발된 ADO(ActiveX Data Objects)를 사용하여 데이터베이스 정보에 액세스하는 컴포넌트. 많은 ADO 드라이버는 다른 데이터베이스 서버에 연결할 때 유용합니다. ADO 기반 컴포넌트는 ADO 기반 환경에 애플리케이션을 통합할 수 있게 합니다. 자세한 내용은 21장 "ADO 컴포넌트 사용"을 참조하십시오.
dbExpress	dbExpress를 사용하여 데이터베이스 정보에 액세스하는 크로스 플랫폼 컴포넌트. dbExpress 드라이버는 데이터베이스에 빠른 액세스를 제공하지만 업데이트를 수행하는 데 <i>TClientDataSet</i> 및 <i>TDataSetProvider</i> 를 사용해야 합니다. 자세한 내용은 22장 "단방향 데이터셋 사용"을 참조하십시오.
InterBase	별도의 엔진 레이어를 거치지 않고 직접 InterBase 데이터베이스에 액세스하는 컴포넌트. InterBase 컴포넌트 사용에 관한 자세한 내용은 온라인 도움말 참조하십시오.
Data Access	<i>TClientDataSet</i> 및 <i>TDataSetProvider</i> 와 같은 데이터 액세스 메커니즘이 사용될 수 있는 컴포넌트. 클라이언트 데이터셋에 관한 내용은 23장 "클라이언트 데이터셋 사용"을 참조하십시오. 프로바이더에 관한 내용은 24장 "프로바이더 컴포넌트 사용"을 참조하십시오.
Data Controls	데이터 소스의 정보를 액세스할 수 있는 Data-aware 컨트롤. 자세한 내용은 15장 "데이터 컨트롤 사용"을 참조하십시오.

데이터베이스 애플리케이션을 디자인할 때에는 어떤 데이터 액세스 메커니즘을 사용할 것인지 결정해야 합니다. 각 데이터 액세스 메커니즘은 기능 지원, 배포 용이성 및 다른 데이터베이스 서버 지원을 위한 드라이버 가용성에 대해 각기 다른 범위를 가집니다.

Delphi를 사용하여 데이터베이스 클라이언트 애플리케이션과 애플리케이션 서버를 만드는 방법에 대한 자세한 내용은 이 안내서의 2부 "데이터베이스 애플리케이션 개발"을 참조하십시오. 배포 정보는 13-6 페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오.

분산 데이터베이스 애플리케이션

Delphi는 컴포넌트들을 조합하여 데이터베이스 애플리케이션을 만드는 것을 지원합니다. 분산 데이터베이스 애플리케이션은 DCOM, CORBA, TCP/IP 및 SOAP 등을 비롯한 다양한 통신 프로토콜에 기반하여 구축할 수 있습니다.

분산 데이터베이스 애플리케이션을 만드는 자세한 내용은 25장 "다계층(multi-tiered) 애플리케이션 생성"을 참조하십시오.

분산 데이터베이스 애플리케이션은 때때로 애플리케이션 파일 외에 BDE(Borland 데이터베이스 엔진) 배포를 필요로 합니다. BDE 배포에 관한 내용은 13-6 페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오.

웹 서버 애플리케이션 만들기

웹 서버 애플리케이션은 인터넷 상의 HTML 웹 페이지나 XML 문서와 같은 웹 콘텐츠를 전달하는 서버에서 실행하는 애플리케이션입니다. 웹 서버 애플리케이션의 예에는 웹 사이트에 대한 액세스 제어, 구매 주문 생성, 정보 요청에 대한 응답 등의 기능을 가진 것이 있습니다.

다음과 같은 Delphi 기술을 사용하여 여러 가지 다른 유형의 웹 서버 애플리케이션을 만들 수 있습니다.

- Web Broker
- WebSnap
- InternetExpress
- Web Services

Web Broker 사용

Web Broker(NetCLX 아키텍처라고도 함)는 CGI 애플리케이션이나 동적 연결 라이브러리(DLL)와 같은 웹 서버 애플리케이션을 만드는 데 사용할 수 있습니다. 이러한 웹 서버 애플리케이션에는 모든 닌비주얼 컴포넌트를 포함할 수 있습니다. 컴포넌트 팔레트의 Internet 페이지에 있는 컴포넌트는 이벤트 핸들러를 생성하고, 프로그램에서 HTML 또는 XML 문서를 생성하고, 그 문서를 클라이언트에 전송할 수 있게 합니다.

웹 브로커 아키텍처를 사용하여 새로운 웹 서버 애플리케이션을 만들려면 File|New|Other를 선택하고 New Items 대화 상자에서 Web Server Application을 선택합니다. 그런 다음 웹 서버 애플리케이션 유형을 선택합니다.

표 5.3 웹 서버 애플리케이션

웹 서버 애플리케이션 유형	설명
ISAPI 및 NSAPI 동적 연결 라이브러리	ISAPI 및 NSAPI 웹 서버 애플리케이션은 웹 서버에 의해 로드되는 DLL입니다. 클라이언트 요청 정보는 구조로서 DLL에 전달되고 TISAPIApplication에 의해 평가됩니다. 각각의 요청 메시지는 별개의 실행 스레드에서 처리됩니다. 이 유형의 애플리케이션을 선택하면 프로젝트 파일의 라이브러리 헤더와 필수 항목을 프로젝트 파일의 uses 목록과 exports 절에 추가합니다.
CGI 독립형 실행 파일	CGI 웹 서버 애플리케이션은 표준 입력에서 클라이언트의 요청을 받아 처리한 다음, 클라이언트에 보낼 결과를 표준 출력으로 서버에 돌려 보내는 콘솔 애플리케이션입니다. 이 유형의 애플리케이션을 선택하면 필요한 항목이 프로젝트 파일의 uses 절에 추가되고 적절한 \$APPTYPE 지시어가 소스에 추가됩니다.
Win-CGI 독립형 실행 파일	Win-CGI 웹 서버 애플리케이션은 서버가 작성한 구성 설정 (INI) 파일로부터 클라이언트의 요청을 받는 Windows 애플리케이션으로 이 서버는 클라이언트에 다시 전달하는 파일에 결과를 작성합니다. INI 파일은 TCGIApplication에 의해 값이 구해집니다. 각 요청 메시지들은 별도의 애플리케이션 인스턴스에 의해 처리됩니다. 이 유형의 애플리케이션을 선택하면 필요한 항목이 프로젝트 파일의 uses 절에 추가되고 적절한 \$APPTYPE 지시어가 소스에 추가됩니다.
Apache 공유 모듈 (DLL)	이 유형의 애플리케이션을 선택하면 프로젝트를 DLL로 설치합니다. Apache 웹 서버 애플리케이션은 웹 서버에 의해 로드되는 DLL입니다. 정보가 DLL에 전달되고 처리되며 웹 서버에 의해 클라이언트에 반환됩니다.
Web App Debugger 독립형 실행 파일	이 유형의 애플리케이션을 선택하면 웹 서버 애플리케이션을 개발하고 테스트하기 위한 환경이 설정됩니다. Web App Debugger 애플리케이션은 웹 서버에 의해 로드되는 실행 파일입니다. 이 유형의 애플리케이션은 배포용이 아닙니다.

CGI와 Win-CGI 애플리케이션은 서버에서 더 많은 시스템 리소스를 사용하므로 복잡한 애플리케이션은 ISAPI, NSAPI 또는 Apache DLL 애플리케이션으로 만드는 것이 더 좋습니다. 크로스 플랫폼 애플리케이션을 작성하는 경우, CGI 독립형 또는 웹 서버 개발용 Apache 공유 모듈(DLL)을 선택해야 합니다. WebSnap과 Web Services 애플리케이션을 만들 때에도 적용됩니다.

웹 서버 애플리케이션 구축에 대한 자세한 내용은 27장 "인터넷 애플리케이션 생성"을 참조하십시오.

WebSnap 애플리케이션 만들기

WebSnap에서는 웹 브라우저와 상호 작용하는 고급 웹 서버를 구축하기 위한 컴포넌트 집합과 마법사를 제공합니다. WebSnap 컴포넌트는 HTML 또는 다른 Web 페이지용 mime 콘텐츠를 생성합니다. WebSnap은 서버측 개발을 위한 것입니다. 따라서 현재까지는 WebSnap을 크로스 플랫폼 애플리케이션에서 사용할 수 없습니다.

새로운 WebSnap 애플리케이션을 만들려면 File|New|Other를 선택한 후 New Items 대화 상자에서 WebSnap 탭을 선택합니다. WebSnap Application을 선택합니다. 그런 다음 웹 서버 애플리케이션 유형(ISAPI/NSAPI, CGI, Win-CGI, Apache)을 선택합니다. 자세한 내용은 표 5.3 "웹 서버 애플리케이션"을 참조하십시오.

WebSnap에 대한 자세한 내용은 29장 "WebSnap 사용"을 참조하십시오.

InternetExpress 사용

InternetExpress는 애플리케이션 서버의 클라이언트 역할을 하기 위해 기본 Web 서버 애플리케이션 아키텍처를 확장하는 컴포넌트 집합입니다. 클라이언트에서 실행하는 동안 브라우저 기반 클라이언트가 프로바이더로부터 데이터를 폐치하고 프로바이더에 업데이트를 확인할 수 있는 애플리케이션에 InternetExpress를 사용합니다.

InternetExpress 애플리케이션은 HTML, XML 및 Javascript가 조합되어 있는 HTML 페이지를 생성합니다. HTML은 최종 사용자의 브라우저에서 보여지는 페이지의 레이아웃과 외관을 결정합니다. XML은 데이터베이스 정보를 나타내는 데이터 패킷과 델타 패킷을 인코딩합니다. Javascript를 통해 HTML 컨트롤은 클라이언트 시스템에서 XML 데이터 패킷의 데이터를 해석하고 처리할 수 있습니다.

InternetExpress에 대한 자세한 내용은 25-34 페이지의 "InternetExpress를 사용하여 웹 애플리케이션 구축"을 참조하십시오.

Web Services 애플리케이션 만들기

Web Services는 월드 와이드 웹과 같은 네트워크 상에서 게시 및 호출할 수 있는 모듈이 자체적으로 포함되어 있는 애플리케이션입니다. Web Services는 제공된 서비스를 설명하는 명확하게 정의된 인터페이스를 제공합니다. Web Services는 XML, XML Schema, SOAP(Simple Object Access Protocol) 및 WSDL(Web Services Definition Language)과 같은 새로운 표준을 사용하여 인터넷 상에 프로그래밍할 수 있는 서비스를 만들거나 소비하는 데 사용됩니다.

Web Services는 분산 환경에서 정보를 교환하는 표준 경량급(lightweight) 프로토콜인 SOAP를 사용합니다. 통신 프로토콜은 HTTP를 사용하고 원격 프로시저 호출을 인코딩하는데 XML을 사용합니다.

Delphi를 사용하여 Web Services를 구현하는 서버 및 그 서비스에서 호출하는 클라이언트를 구축할 수 있습니다. SOAP 메시지에 응답하는 Web Services를 구현하는 임의의 서버 및 임의의 클라이언트에 사용할 목적으로 Web Services를 게시하는 Delphi 서버에 대한 클라이언트를 작성할 수 있습니다.

Web Services에 관한 자세한 내용은 31장 "Web Services 사용"을 참조하십시오.

COM을 사용한 애플리케이션 작성

COM(Component Object Model)은 인터페이스라는 이미 정의된 루틴을 사용해서 객체 상호 운용성을 제공하기 위해 디자인된 Windows 기반의 분산 객체 구조입니다. 개별 시스템에서 DCOM을 사용하는 경우, COM 애플리케이션은 다른 프로세서에서 구현하는 객체를 사용합니다. COM+, ActiveX 및 Active Server 페이지를 사용할 수도 있습니다.

COM은 랭귀지와 무관하게 사용할 수 있는 소프트웨어 컴포넌트 모델로서 Windows 플랫폼에서 실행되는 소프트웨어 컴포넌트와 애플리케이션 간의 상호 작용을 가능하게 합니다. COM의 핵심은 명확하게 정의되어 있는 인터페이스를 통해 컴포넌트들 사이에서, 애플리케이션들 사이에서, 또 클라이언트와 서버 사이에서의 통신을 가능하게 한다는 것입니다. 인터페이스를 통해 클라이언트는 COM 컴포넌트가 런타임 시 지원하는 기능을 알아볼 수 있습니다. 사용자 컴포넌트에 기능을 추가하려면 원하는 기능에 대한 추가 인터페이스를 간단히 추가하면 됩니다.

COM과 DCOM 사용

Delphi에는 COM, OLE 또는 ActiveX 애플리케이션을 쉽게 만들 수 있게 하는 클래스 및 마법사가 있습니다. COM 객체, 자동화 서버(Active Server Object 포함), ActiveX 컨트롤 또는 ActiveForm를 구현하는 COM 클라이언트나 서버를 만들 수 있습니다. COM은 또한 Automation, ActiveX 컨트롤, 활성 문서 및 액티브 디렉토리 등 그외의 기술에 대한 기초가 됩니다.

Delphi를 사용해서 COM 기반의 애플리케이션을 만들면 광범위한 가능성이 제공되는데 애플리케이션 안에서 내부적으로 인터페이스를 이용함으로써 소프트웨어 디자인을 개선할 수 있고, Win9x 셸 확장 및 DirectX 멀티미디어 지원 같은 시스템에 있는 COM 기반의 다른 API 객체와 상호 작용할 수 있는 객체도 만들 수 있습니다. 애플리케이션은 애플리케이션과 동일한 컴퓨터에 있거나 DCOM(Distributed COM)이라는 메커니즘을 사용하여 네트워크 상의 다른 컴퓨터에 있는 COM 컴포넌트의 인터페이스에 액세스할 수 있습니다.

COM 및 Active X 컨트롤에 대한 자세한 내용은 33장 "COM 기술 개요", 38장 "ActiveX 컨트롤 생성" 및 25-33 페이지의 "클라이언트 애플리케이션을 ActiveX 컨트롤로 배포"를 참조하십시오.

DCOM에 대한 자세한 내용은 25-9 페이지의 "DCOM 연결 사용"을 참조하십시오.

MTS 및 COM+ 사용

COM 애플리케이션은 대규모 분산 환경에서 객체를 관리하는 특별한 서비스로 확대될 수 있습니다. 이러한 서비스에는 Windows 2000 이전 버전에서는 MTS(Microsoft Transaction Server)에 의해, Windows 2000 이후 버전에서는 COM+에 의해 제공되는 트랜잭션 서비스, 보안 및 리소스 관리가 포함됩니다.

MTS 및 COM+에 대한 자세한 내용은 39장 "MTS 또는 COM+ 객체 생성" 및 25-6 페이지의 "트랜잭션 데이터 모듈(Transaction Data Module) 사용"을 참조하십시오.

데이터 모듈 사용

데이터 모듈은 논비주얼(nonvisual) 컴포넌트를 포함하는 특별한 폼과도 같습니다. 데이터 모듈의 모든 컴포넌트는 비주얼 컨트롤과 함께 일반적인 폼에 둘 수 있습니다. 그러나 데이터베이스와 시스템 객체 그룹을 재사용하거나 데이터베이스 연결과 비즈니스 로직을 처리하는 애플리케이션의 일부를 분리하려는 경우, 데이터 모듈은 간편한 구성 틀을 제공합니다.

가지고 있는 Delphi 에디션에 따라 표준, 원격, 웹 모듈, 애플릿 모듈 및 서비스 등을 포함하여 여러 가지 타입의 데이터 모듈이 있습니다. 각 데이터 모듈의 타입은 특별한 목적을 수행합니다.

- 표준 데이터 모듈은 단일 및 2계층(2 tier) 데이터베이스 애플리케이션에 특히 유용하지만 모든 애플리케이션에서 논비주얼(nonvisual) 컴포넌트를 구성하는 데 사용할 수 있습니다. 자세한 내용은 5-16 페이지의 "표준 데이터 모듈 생성 및 편집"을 참조하십시오.
- 원격 데이터 모듈은 다계층 데이터베이스 애플리케이션에서 애플리케이션 서버의 기초를 형성합니다. 모든 에디션에서 사용할 수 있는 것은 아닙니다. 원격 데이터 모듈은 애플리케이션의 논비주얼 컴포넌트를 보유하는 것뿐만 아니라 애플리케이션 서버와 통신하기 위해 클라이언트가 사용하는 인터페이스를 노출시킵니다. 원격 데이터 모듈에 대한 자세한 내용은 5-19 페이지의 "원격 데이터 모듈을 애플리케이션 서버 프로젝트에 추가"를 참조하십시오.
- 웹 모듈은 웹 서버 애플리케이션의 기본을 형성합니다. 웹 모듈은 HTTP 응답 메시지의 내용을 만드는 컴포넌트를 유지하는 것 외에 클라이언트 애플리케이션에서 HTTP 메시지의 디스패칭을 처리합니다. 웹 모듈 사용에 대한 자세한 내용은 27장 "인터넷 애플리케이션 생성"을 참조하십시오.
- 애플릿 모듈은 제어판 애플릿의 기본을 형성합니다. 애플릿 모듈은 제어판 애플릿을 구현하는 논비주얼 컨트롤을 보유할 뿐만 아니라 제어판에 애플릿의 아이콘을 나타내는 방법을 결정하는 속성을 정의하고 사용자가 애플릿을 실행할 때 호출되는 이벤트를 포함합니다. 애플릿 모듈에 대한 자세한 내용은 온라인 도움말을 참조하십시오.
- 서비스는 NT 서비스 애플리케이션에 있는 각각의 서비스들을 캡슐화합니다. 서비스는 서비스를 구현하기 위해 사용된 논비주얼 컨트롤을 보유할 뿐만 아니라 서비스가 시작되거나 중지될 때 호출되는 이벤트를 포함합니다. 서비스에 대한 자세한 내용은 5-4 페이지의 "서비스 애플리케이션"을 참조하십시오.

표준 데이터 모듈 생성 및 편집

프로젝트를 위한 표준 데이터 모듈을 작성하려면 File|New|Data Module을 선택합니다. Delphi는 바탕 화면에 있는 데이터 모듈 컨테이너를 열어 코드 에디터에서 새 모듈에 대한 유닛 파일을 나타낸 다음 현재 프로젝트에 모듈을 추가합니다.

디자인 타임 시 데이터 모듈은 배경 색이 하얗고 정렬 그리드가 없는 표준 Delphi 폼처럼 보입니다. 폼에서와 마찬가지로 컴포넌트 팔레트에 있는 논비주얼 컴포넌트를 모듈에 두고 Object Inspector에서 속성을 편집할 수 있습니다. 추가한 컴포넌트를 수용하도록 데이터 모듈의 크기를 재조정할 수 있습니다.

컨텍스트 메뉴를 표시하기 위해 마우스 오른쪽 버튼을 클릭할 수도 있습니다. 다음 표는 데이터 모듈에 대한 컨텍스트 메뉴 옵션을 요약한 것입니다.

표 5.4 데이터 모듈에 대한 컨텍스트 메뉴 옵션

메뉴 항목	용도
<i>Edit</i>	데이터 모듈의 컴포넌트를 잘라내기, 복사, 붙여넣기, 삭제 및 선택할 수 있는 컨텍스트 메뉴를 나타냅니다.
<i>Position</i>	(<i>Align To Grid</i>) 또는 Alignment 대화 상자(<i>Align</i>)에서 제공하는 기준에 따라 논비주얼 컴포넌트를 모듈의 보이지 않는 그리드에 정렬합니다.
<i>Tab Order</i>	탭 키를 눌렀을 때 컴포넌트에서 포커스가 이동하는 순서를 변경할 수 있게 합니다.
<i>Creation Order</i>	시동 시 데이터 액세스 컴포넌트가 생성되는 순서를 변경할 수 있게 합니다.
<i>Revert to Inherited</i>	Object Repository에서 다른 모듈로부터 상속 받은 모듈에 변경된 내용을 버리고 원래의 상속 받은 모듈로 되돌아갑니다.
<i>Add to Repository</i>	Object Repository에서 데이터 모듈에 대한 연결을 저장합니다.
<i>View as Text</i>	데이터 모듈 속성의 텍스트 표현을 나타냅니다.
<i>View DFM</i>	특정 폼이 저장되는 형식(바이너리 또는 텍스트) 사이를 토글합니다.

데이터 모듈에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

데이터 모듈과 유닛 파일 이름 지정

데이터 모듈의 제목 표시줄은 모듈의 이름을 표시합니다. 데이터 모듈의 기본 이름은 "DataModuleN"이며 여기서 N은 프로젝트에서 사용되지 않은 가장 적은 숫자를 나타냅니다. 예를 들어, 새 프로젝트를 시작하면 다른 애플리케이션을 구축하기 전에 모듈이 프로젝트에 추가되고 데이터 모듈의 이름은 기본적으로 "DataModule2"가 됩니다. 그리고 *DataModule2*에 대한 해당 유닛 파일은 기본적으로 "Unit2"가 됩니다.

디자인 타임에 데이터 모듈과 해당 유닛 파일의 이름을 좀더 서술적인 이름으로 다시 재지정해야 합니다. 특히 Repository 또는 모듈을 사용한 애플리케이션의 다른 데이터 모듈과의 이름 충돌을 막기 위해서 Object Repository에 추가한 데이터 모듈의 이름을 재지정해야 합니다.

다음과 같은 방법으로 데이터 모듈의 이름을 재지정합니다.

1 모듈을 선택합니다.

2 Object Inspector에서 모듈의 *Name* 속성을 편집합니다.

모듈의 새 이름이 제목 표시줄에 표시되고 이 때 Object Inspector에서 *Name* 속성은 더 이상 포커스를 갖지 않습니다.

디자인 타임에 데이터 모듈의 이름을 변경하면 코드의 interface 섹션에 있는 해당 변수의 이름이 변경됩니다. 또한 프로시저 선언에 있는 타입 이름의 사용이 변경됩니다. 작성한 코드에 있는 데이터 모듈에 대한 참조를 수동으로 변경해야 합니다.

다음과 같은 방법으로 데이터 모듈에 대한 유닛 파일의 이름을 재지정합니다.

1 유닛 파일을 선택합니다.

컴포넌트 배치 및 이름 지정

폼 위에 비주얼 컴포넌트를 두는 것과 같은 방식으로 닌비주얼 컴포넌트를 둡니다. 컴포넌트 팔레트의 적절한 페이지에서 원하는 컴포넌트를 클릭한 다음, 컴포넌트를 두기 위해 데이터 모듈을 클릭합니다. 데이터 모듈에는 그리드와 같은 비주얼 컨트롤을 둘 수 없습니다. 데이터 모듈에 비주얼 컨트롤을 두려고 하면 에러 메시지를 받습니다.

쉽게 사용하도록 하기 위해 데이터 모듈 안에 컴포넌트의 이름이 표시됩니다. 컴포넌트를 처음 두면 Delphi에서는 어떤 종류의 컴포넌트인지 식별해 주는 일반 이름이 할당되고 그 끝에 *I*이 옵니다. 예를 들어, *TDataSource* 컴포넌트는 *DataSource1*이라는 이름이 할당됩니다. 이렇게 하면 사용하려는 특정 컴포넌트의 속성과 메소드를 쉽게 선택할 수 있습니다.

컴포넌트 타입과 용도를 반영하는 다른 이름으로 컴포넌트의 이름을 지정하기를 원할 수도 있습니다.

다음과 같은 방법으로 데이터 모듈의 컴포넌트 이름을 변경합니다.

1 컴포넌트를 선택합니다.

2 Object Inspector에서 컴포넌트의 *Name* 속성을 편집합니다.

Object Inspector에서 *Name* 속성이 더 이상 포커스를 가지지 않으면 곧바로 데이터 모듈의 아이콘 아래에 컴포넌트의 새 이름이 표시됩니다.

예를 들어, CUSTOMER 테이블을 사용하는 데이터베이스 애플리케이션을 가정해 봅시다. 테이블에 액세스하기 위해서는 데이터 소스 컴포넌트 (*TDataSource*) 및 테이블 컴포넌트 (*TClientDataSet*)의 최소 두 개의 데이터 액세스 컴포넌트가 필요합니다. 데이터 모듈에 이러한 컴포넌트를 놓을 때 Delphi는 *DataSource1* 및 *ClientDataSet1* 이름을 할당합니다. 컴포넌트의 타입과 액세스하는 데이터베이스 CUSTOMER를 반영하기 위해 이 이름을 *CustomerSource* 및 *CustomerTable*로 변경할 수 있습니다.

데이터 모듈의 컴포넌트 속성 및 이벤트 사용

데이터 모듈에 컴포넌트를 놓으면 전체 애플리케이션에 대한 동작이 중앙 집중화됩니다. 예를 들어, 그러한 데이터셋을 사용하는 데이터 소스 컴포넌트에 사용 가능한 데이터를 제어하는 *TClientDataSet*과 같은 데이터셋 컴포넌트의 속성을 사용할 수 있습니다. 데이터셋에 대한 *ReadOnly* 속성을 *True*로 설정하면 사용자가 폼에 있는 data-aware 비주얼 컨트롤에서 보이는 데이터를 편집하는 것을 방지합니다. 또한 *ClientDataSetI*을 더블 클릭하여 데이터셋에 대한 Fields Editor를 호출해서 테이블 내의 필드 또는 데이터 소스, 즉 폼에 있는 data-aware 컨트롤에 사용 가능한 쿼리를 제한할 수 있습니다. 데이터 모듈에서 컴포넌트에 대해 설정한 속성은 모듈을 사용하는 애플리케이션의 모든 폼에 일관성 있게 적용됩니다.

속성과 아울러 컴포넌트의 이벤트 핸들러를 작성할 수 있습니다. 예를 들어, *TDataSource* 컴포넌트에는 *OnDataChange*, *OnStateChange* 및 *OnUpdateData* 등 세 개의 사용 가능한 이벤트가 있습니다. *TClientDataSet* 컴포넌트에는 20개 이상의 잠재적인 이벤트가 들어 있습니다. 이러한 이벤트를 사용하여 애플리케이션에서의 데이터 조작을 지배하는 일관된 비즈니스 룰의 집합을 만들 수 있습니다.

데이터 모듈에서 비즈니스 룰 만들기

데이터 모듈의 컴포넌트에 대한 이벤트 핸들러를 작성하는 것 외에 데이터 모듈에 대한 유닛 파일에 메소드를 직접 코딩할 수 있습니다. 이러한 메소드는 비즈니스 룰로 데이터를 사용하는 폼에 적용될 수 있습니다. 예를 들어 월말, 분기말, 연말 부기를 수행하는 프로시저를 작성할 수도 있습니다. 데이터 모듈의 컴포넌트에 대한 이벤트 핸들러에서 프로시저를 호출할 수도 있습니다. 데이터 모듈을 위해 작성하는 프로시저 및 함수의 프로토타입은 모듈의 **type** 선언에 나타나야 합니다.

```

type
  TCustomerData = class(TDataModule)
    Customers:TClientDataSet;
    Orders:TClientDataSet;
    ...
  private
    { Private declarations }
  public
    { Public declarations }
    procedure LineItemsCalcFields(DataSet: TDataSet); { A procedure you add }
  end;

var
  CustomerData: TCustomerData;
  
```

작성하는 프로시저와 함수에서 모듈 부분은 코드의 implementation 섹션에 따라야 합니다.

폼에서 데이터 모듈 액세스

폼에서 비주얼 컨트롤을 데이터 모듈에 연결하려면 먼저 데이터 모듈을 폼의 **uses** 절에 추가해야 합니다. 이 작업을 다음과 같은 방법으로 수행할 수 있습니다.

- 코드 에디터에서 폼의 유닛 파일을 열고 데이터 모듈의 이름을 **interface** 섹션에 있는 **uses**절에 추가합니다.
- 폼의 유닛 파일을 클릭하고 File|Use Unit을 선택한 다음 모듈의 이름을 입력하거나 Use Unit 대화 상자의 리스트 박스로부터 이름을 선택하십시오.
- 데이터베이스 컴포넌트의 경우, 데이터 모듈에서 Fields Editor를 열기 위해 데이터셋 또는 쿼리 컴포넌트를 클릭하고 에디터에서 기존 필드를 폼으로 끌어 놓습니다. Delphi는 모듈을 폼의 **uses** 절에 추가할지 사용자에게 물어 본 다음, 편집 상자와 같은 필드용 컨트롤을 만듭니다.

예를 들어, *TClientDataSet* 컴포넌트를 데이터 모듈에 추가한 경우 더블 클릭하면 Fields Editor가 열립니다. 필드를 선택하고 폼에 끌어 놓습니다. 편집 상자 컴포넌트가 나타납니다.

데이터 소스가 아직 정의되지 않았으므로 Delphi는 새 데이터 소스 컴포넌트 *DataSource1*을 폼에 추가하고 편집 상자의 *DataSource* 속성을 *DataSource1*로 설정합니다. 데이터 소스는 데이터 모듈의 데이터셋 컴포넌트 *ClientDataSet1*에 *DataSet* 속성을 설정합니다.

데이터 모듈에 *TDataSource* 컴포넌트를 추가하여 필드를 폼에 끌어 놓기 전에 데이터 소스를 정의할 수 있습니다. 데이터 소스의 *DataSet* 속성을 *ClientDataSet1*로 설정합니다. 필드를 폼에 끌어 놓은 후에 편집 상자는 *DataSource1*로 이미 설정되어 있는 *TDataSource* 속성을 나타냅니다. 이 방식은 데이터 액세스 모델을 보다 더 명확하게 합니다.

원격 데이터 모듈을 애플리케이션 서버 프로젝트에 추가

Delphi의 일부 에디션에서는 애플리케이션 서버 프로젝트에 *원격 데이터 모듈*을 추가할 수 있습니다. 원격 데이터 모듈은 다계층 애플리케이션의 클라이언트가 네트워크를 통해 액세스할 수 있는 인터페이스를 갖습니다.

다음과 같은 방법으로 프로젝트에 원격 데이터 모듈을 추가합니다.

- 1 File|New|Other를 선택합니다.
- 2 New Items 대화 상자에서 Multitier 페이지를 선택합니다.
- 3 원하는 모듈 타입(CORBA Data Module, Remote Data Module 또는 Transactional Data Module)을 더블 클릭하여 Remote Data Module 마법사를 엽니다.

원격 데이터 모듈을 프로젝트에 추가하면 표준 데이터 모듈과 같은 방식으로 사용할 수 있습니다.

다계층 데이터베이스 애플리케이션에 대한 자세한 내용은 25장 "다계층(multi-tiered) 애플리케이션 생성"을 참조하십시오.

Object Repository 사용

Object Repository (Tools|Repository)를 사용하면 폼, 대화 상자, 프레임 및 데이터 모듈을 쉽게 공유할 수 있습니다. Object Repository는 폼과 프로젝트를 만들 때 사용자를 안내하는 마법사 및 새 프로젝트에 대한 템플릿도 제공합니다. 레포지토리는 Repository 및 New Items 대화 상자에 나타나는 항목에 대한 참조를 포함하는 텍스트 파일인 DELPHI32.DRO(기본적으로 BIN 디렉토리에 있음)에서 유지 관리됩니다.

프로젝트 내에서 항목 공유

항목을 Object Repository에 추가하지 않고 프로젝트 *내에서* 공유할 수 있습니다. New Items 대화 상자(File|New|Other)를 열면 현재 프로젝트의 이름을 갖는 페이지 탭을 볼 수 있습니다. 이 페이지에서는 프로젝트에 있는 모든 폼, 대화 상자 및 데이터 모듈을 나열합니다. 기존 항목에서 새 항목을 파생시키고 필요하면 항목을 사용자 지정할 수 있습니다.

항목 추가: Object Repository

사용자의 프로젝트, 폼, 프레임 및 데이터 모듈을 Object Repository에 추가할 수 있습니다. 다음과 같은 방법으로 항목을 Object Repository에 추가합니다.

- 1 항목이 프로젝트이거나 프로젝트에 있는 경우에 해당 프로젝트를 엽니다.
- 2 프로젝트에서 Project|Add To Repository를 선택합니다. 폼 또는 데이터 모듈에서 항목을 마우스 오른쪽 버튼으로 클릭하고 Add To Repository를 선택합니다.
- 3 설명, 제목 및 작성자를 입력합니다.
- 4 New Items 대화 상자에서 항목을 나타내려는 페이지를 결정한 다음, 페이지의 이름을 입력하거나 Page 콤보 박스에서 페이지의 이름을 선택합니다. 존재하지 않는 페이지 이름을 입력한 경우, Delphi는 새 페이지를 생성합니다.
- 5 Object Repository에 있는 객체를 나타내는 아이콘을 선택하려면 Browse를 선택합니다.
- 6 OK를 선택합니다.

팀 환경에서 객체 공유

네트워크 상에서 레포지토리를 사용 가능하게 하여 작업 그룹 또는 개발 팀과 객체를 공유할 수 있습니다. 공유 레포지토리를 사용하려면 팀원 모두 Environment Options 대화 상자에서 동일한 Shared Repository 디렉토리를 선택해야 합니다.

- 1 Tools|Environment Options를 선택합니다.
- 2 Preferences 페이지에서 Shared Repository 패널을 찾습니다. Directory 편집 상자에서 공유 레포지토리를 검색하려는 디렉토리를 입력합니다. 반드시 모든 팀원이 액세스할 수 있는 디렉토리를 지정합니다.

항목이 레포지토리에 처음으로 추가되면 DELPHI32.DRO 파일이 없는 경우 Delphi는 Shared Repository 디렉토리에 이 파일을 만듭니다.

프로젝트에서 Object Repository 항목 사용

Object Repository의 항목에 액세스하려면 File|New|Other를 선택합니다. New Items 대화 상자가 나타나 모든 사용 가능한 항목을 보여 줍니다. 사용하려는 항목의 종류에 따라 프로젝트에 항목을 추가하기 위한 다음 세 가지 옵션이 있습니다.

- Copy
- Inherit
- Use

항목 복사

Copy를 선택하여 선택한 항목의 정확한 복사본을 만들어 복사본을 프로젝트에 추가합니다. Object Repository에 있는 항목의 차후 변경 내용은 복사본에 반영되지 않고, 복사본의 변경 내용은 원래의 Object Repository 항목에 영향을 미치지 않습니다.

Copy는 프로젝트 템플릿에서 사용할 수 있는 유일한 옵션입니다.

항목 상속

Inherit를 선택하고 Object Repository에서 선택한 항목에서 새 클래스를 파생시켜 프로젝트에 추가합니다. 프로젝트를 다시 컴파일할 때 Object Repository에서 항목의 변경된 내용은 모두 프로젝트의 항목 변경 내용과 더불어 파생된 클래스에 반영됩니다. 파생된 클래스의 변경 내용은 Object Repository의 공유 항목에 영향을 주지 않습니다.

상속은 폼, 대화 상자 및 데이터 모듈에 사용할 수 있지만 프로젝트 템플릿에는 사용할 수 없습니다. 상속은 동일한 프로젝트 내에서 항목을 재사용할 때 사용하는 유일한 옵션입니다.

항목 사용

선택된 항목 자체를 프로젝트의 일부가 되게 하려면 Use를 선택합니다. 프로젝트에서 항목의 변경된 내용은 Inherit 옵션 또는 Use 옵션으로 항목을 추가했던 다른 모든 프로젝트에 나타납니다. 이 옵션은 신중하게 선택하십시오.

Use 옵션은 폼, 대화 상자 및 데이터 모듈에 사용할 수 있습니다.

프로젝트 템플릿 사용

템플릿은 애플리케이션 개발의 첫 단계에서 사용할 수 있도록 미리 디자인됩니다. 다음과 같은 방법으로 템플릿에서 새 프로젝트를 만듭니다.

- 1 File|New|Other를 선택하여 New Items 대화 상자를 표시합니다.
- 2 Projects 탭을 선택합니다.

3 원하는 프로젝트 템플릿을 선택하고 OK를 클릭합니다.

4 Select Directory 대화 상자에서 새 프로젝트 파일의 디렉토리를 지정합니다.

Delphi는 템플릿 파일을 지정된 디렉토리에 복사하는데 이 템플릿 파일은 수정할 수 있습니다. 프로젝트 파일 원본은 변경된 내용에 영향을 받지 않습니다.

공유 항목 수정

Object Repository에서 항목을 수정하는 경우, 변경된 내용은 Use 옵션 또는 Inherit 옵션으로 항목을 추가했던 기존 프로젝트뿐만 아니라 동일한 항목을 사용하는 향후의 모든 프로젝트에 영향을 미칩니다. 변경된 내용이 다른 프로젝트에 전파되는 것을 막으려면 다음 중 하나를 수행합니다.

- 항목을 복사하고 현재 프로젝트 내에서만 수정합니다.
- 항목을 현재 프로젝트로 복사하고 수정한 다음 항목을 다른 이름으로 Object Repository에 추가합니다.
- 항목으로부터 컴포넌트, DLL, 컴포넌트 템플릿 또는 프레임을 만듭니다. 컴포넌트나 DLL을 만드는 경우, 다른 개발자와 공유할 수 있습니다.

기본 프로젝트, 새 폼 및 메인 폼 지정

기본적으로 File|New|Application 또는 File|New|Form을 선택하면 Delphi는 빈 폼을 표시합니다. 다음과 같이 레포지토리를 재구성하여 빈 폼을 다른 폼으로 변경할 수 있습니다.

- 1 Tools|Repository를 선택합니다.
- 2 기본 프로젝트를 지정하려면 Projects 페이지를 선택하고 Objects 아래에 있는 항목을 선택합니다. 그런 다음 New Project 체크 박스를 선택합니다.
- 3 기본 폼을 지정하려면 Forms와 같은 Repository 페이지를 선택한 다음 Objects에 있는 폼을 선택합니다. 기본적인 새 폼 (File|New|Form) 을 지정하려면 New Form 체크 박스를 선택합니다. 새 프로젝트에 사용할 기본 메인 폼을 지정하려면 Main Form 체크 박스를 선택합니다.
- 4 OK를 클릭합니다.

애플리케이션에서 도움말 사용

VCL과 CLX 모두 도움말 요청이 여러 가지 외부 도움말 뷰어 중 하나로 전달되도록 하는 객체 기반 메커니즘을 사용하여 애플리케이션의 도움말 표시를 지원합니다. 이를 지원하려면 애플리케이션에서 *ICustomHelpViewer* 인터페이스와 경우에 따라서는 그 자손 중 하나인 인터페이스를 구현하는 클래스를 포함해야 하며 전역 Help Manager에 등록되어 있어야 합니다.

VCL의 경우 이러한 인터페이스를 모두 구현하고 애플리케이션과 WinHelp의 연결을 제공하는 *TWinHelpViewer*의 인스턴스를 모든 애플리케이션에 제공하는 반면 CLX에서는 애플리케이션 개발자가 자체적으로 구현해야 합니다.

Help Manager는 등록된 뷰어의 목록을 유지 관리하고 다음과 같은 두 단계 절차로 도움말 뷰어에게 요청을 전달합니다. 먼저 특정 도움말 키워드 또는 컨텍스트를 지원할 수 있는지를 뷰어에게 물어본 다음 지원할 수 있는 뷰어에게 도움말 요청을 전달합니다. 둘 이상의 뷰어가 키워드를 지원하면 Man과 Info에 모두 등록된 뷰어를 가진 애플리케이션의 경우와 마찬가지로 Help Manager는 애플리케이션의 사용자가 호출할 도움말 뷰어를 결정할 수 있는 선택 상자를 표시할 수 있습니다. 그 외에는 첫 번째로 응답하는 도움말 시스템을 표시합니다.

도움말 시스템 인터페이스

도움말 시스템은 일련의 인터페이스를 통해 애플리케이션과 도움말 뷰어 간의 통신을 가능하도록 합니다. 이 인터페이스는 모두 HelpIntfs.pas에 정의되어 있으며 Help Manager의 구현도 포함하고 있습니다.

*ICustomHelpViewer*는 입력된 키워드에 기반한 도움말을 표시하는 지원과 특정한 뷰어에서 사용 가능한 모든 도움말을 나열하는 목차로 표시하는 지원을 제공합니다.

*IExtendedHelpViewer*는 숫자 도움말 컨텍스트에 기반한 도움말과 항목 표시를 지원합니다. 대부분의 도움말 시스템에서 항목은 상위 수준의 키워드로 사용됩니다(예를 들어, "IntToStr"는 도움말 시스템에서 키워드일 수 있지만 "String manipulation routines"는 항목 이름이 될 수 있습니다).

*ISpecialWinHelpViewer*는 Windows에서 실행하는 애플리케이션이 받을 수 있지만 쉽게 일반화되지 않는 전문화된 WinHelp 메시지에 응답하는 것을 지원합니다. 일반적으로 Windows 환경에서 실행되는 애플리케이션만 이러한 인터페이스를 구현하는데 이는 비표준 WinHelp 메시지를 광범위하게 사용하는 애플리케이션에만 요구되는 인터페이스입니다.

*IHelpManager*는 애플리케이션의 Help Manager와 통신하고 추가 정보를 요청하는 도움말 뷰어용 메커니즘을 제공합니다. *IHelpManager*는 도움말 뷰어가 자신을 등록하는 시간에 얻어집니다.

*IHelpSystem*은 *TApplication*이 도움말 요청을 도움말 시스템으로 전달하는 메커니즘을 제공합니다. *TApplication*은 애플리케이션을 로드할 때 *IHelpSystem*과 *IHelpManager*를 모두 구현하는 객체의 인스턴스를 얻어서 속성으로 export하는데 이를 통해 애플리케이션의 다른 코드가 적절한 때 도움말을 요청할 수 있습니다.

*IHelpSelector*는 둘 이상의 뷰어가 도움말 요청을 처리할 수 있을 경우에 도움말 시스템이 사용자 인터페이스를 호출하여 어떤 도움말 뷰어를 사용할지 묻고 목차를 표시할 수 있는 메커니즘을 제공합니다. Help Manager 코드가 사용 중인 widget 집합이나 클래스 라이브러리와 상관 없이 동일하도록 이러한 표시 기능을 Help Manager에 직접 구축하지 않습니다.

ICustomHelpViewer 구현

ICustomHelpViewer 인터페이스에는 다음 세 가지 종류의 메소드가 있습니다. 즉, 시스템 수준 정보(예를 들어, 특정한 도움말 요청에 관계되지 않은 정보)를 Help Manager와 통신하는 데 사용하는 메소드, Help Manager가 제공하는 키워드에 기반한 도움말 보여 주기와 관련된 메소드, 목차를 표시하는 메소드가 있습니다.

Help Manager와 통신

*ICustomHelpViewer*는 시스템 정보를 Help Manager와 통신하는 데 사용할 수 있는 네 가지 함수를 제공합니다.

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- *SoftShutDown*

Help Manager는 다음 환경에서 이 함수들을 사용하여 호출합니다.

- *ICustomHelpViewer.GetViewerName: String*은 Help Manager가 뷰어의 이름을 알려고 할 때(예를 들어, 애플리케이션이 모든 등록된 뷰어의 목록을 표시하도록 요청 받았을 경우) 호출됩니다. 이러한 정보는 문자열로 반환되고 논리적으로 정적이어야 합니다(애플리케이션 실행 중에 변경할 수 없습니다.). 멀티바이트 문자 집합은 지원되지 않습니다.
- *ICustomHelpViewer.NotifyID(const ViewerID: Integer)*는 뷰어를 식별하는 고유한 쿠키를 뷰어에게 제공하기 위해 등록 **즉시** 호출됩니다. 이러한 정보는 나중의 사용을 위해 저장해야 하며 Help Manager의 공지에 응답하는 것과 반대로 뷰어가 스스로 종료하는 경우 Help Manager에게 쿠키 식별 기능을 제공하여 Help Manager는 모든 뷰어에 대한 참조를 해제할 수 있습니다. 쿠키를 제공하는 데 실패하거나 잘못된 쿠키를 제공하면 Help Manager는 잘못된 뷰어에 대한 참조를 잠재적으로 해제하게 됩니다.
- *ICustomHelpViewer.ShutDown*은 Help Manager에 의해 호출되어 Manager가 종료 중이고 도움말 뷰어가 할당된 리소스가 해제되어야 한다는 것을 도움말 뷰어에게 알립니다. 모든 리소스 해제를 이 메소드에 맡기는 것이 좋습니다.
- *ICustomHelpViewer.SoftShutDown*은 Help Manager에 의해 호출되어 뷰어를 언로드하지 않고 도움말 시스템의 외부에서 볼 수 있는 창(예를 들어, 도움말 정보를 표시하는 창)을 종료하도록 요청합니다.

Help Manager에 정보 요청

도움말 뷰어는 *IHelpManager* 인터페이스를 통해 Help Manager와 통신하는데 도움말 뷰어가 Help Manager에 등록될 때 인터페이스의 인스턴스는 도움말 뷰어에 반환됩니다. 도움말 뷰어는 *IHelpManager*를 통해 다음 네 가지를 통신할 수 있습니다. 이러한 네 가지에는 현재 활성 컨트롤의 창 핸들에 대한 요청, 현재 활성 컨트롤에 대한 도움말을 포함해야 한다고 Help Manager가 알고 있는 도움말 파일 이름에 대한 요청, 도움말 파일의 경로에 대한 요청과 Help Manager의 요청이 아닌 다른 것에 대한 응답으로 도움말 뷰어가 자체 종료 중이라는 공지가 있습니다.

*IHelpManager.GetHandle: LongInt*는 현재 활성 컨트롤의 핸들을 알아야 하는 경우에 도움말 뷰어에 의해 호출되며 창 핸들이 반환됩니다.

*IHelpManager.GetHelpFile: String*은 현재 활성 컨트롤이 도움말을 포함하는 것으로 알고 있는 도움말 파일의 이름을 알고자 할 때 도움말 뷰어에 의해 호출됩니다.

*IHelpManager.Release*는 도움말 뷰어가 연결 해제 중이라는 것을 Help Manager에게 알리기 위해 호출됩니다. 이것은 *ICustomHelpViewer.ShutDown*을 통한 요청에 대한 응답으로는 절대 호출되지 않고, 예상하지 못한 연결 해제를 Help Manager에게 알리는 데에만 사용됩니다.

키워드 방식 도움말 표시

도움말 요청은 키워드 방식 도움말로서 도움말 뷰어에 요청될 때 뷰어가 특정한 문자열 방식의 도움말을 제공하도록 요청되거나 컨텍스트 방식 도움말로서 요청될 때 뷰어가 특정한 숫자 식별자 방식의 도움말을 제공하도록 요청됩니다. 숫자 도움말 컨텍스트는 WinHelp 시스템을 사용하는 Windows에서 실행 중인 애플리케이션에서 도움말 요청의 기본 형태입니다. 대부분의 Linux 도움말 시스템에서는 수치 도움말 컨텍스트를 인식하지 못하기 때문에 CLX 애플리케이션에서 사용하지 않는 것이 좋습니다. *ICustomHelpViewer* 구현은 키워드 방식 도움말 요청을 지원하는 데 필요하고 *IExtendedHelpViewer* 구현은 컨텍스트 방식 도움말 요청을 지원하는 데 필요합니다.

*ICustomHelpViewer*는 키워드 방식 도움말을 처리하는 세 가지 메소드를 제공합니다.

- *UnderstandsKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
ICustomHelpViewer.UnderstandsKeyword(const HelpString:String):Integer
```

이 구문은 Help Manager에 의해 호출된 세 가지 메소드 중에서 첫 번째로서 뷰어가 문자열에 대한 도움말을 제공하는지 묻기 위해 동일한 문자열을 사용하여 각각의 등록된 도움말을 호출하는데 여기서 뷰어는 도움말 요청에 응답하여 표시할 수 있는 여러 도움말 페이지의 수를 나타내는 정수로 응답할 것으로 예상됩니다. 뷰어는 메소드를 사용하여 이를 결정할 수 있습니다. IDE 내에서 HyperHelp 뷰어는 자체 인덱스를 유지 관리하고 인덱스를 찾습니다. 뷰어가 이 키워드에 대한 도움말을 지원하지 않으면 뷰어는 0을 반환합니다. 음수는 현재 0을 의미하는 것으로 해석되지만 이 행동은 차후 릴리스에서 보장되지 않습니다.

```
ICustomHelpViewer.GetHelpStrings(const HelpString:String):TStringList
```

이 구문은 하나 이상의 뷰어가 그 항목에 대한 도움말을 제공할 수 있는 경우에 Help Manager에 의해 호출됩니다. 뷰어는 *TStringList*를 반환할 것입니다. 반환된 목록에 있는 문자열은 그 키워드에 사용할 수 있는 페이지에 매핑되어야 하지만 그 매핑의 특성은 뷰어로 결정될 수 있습니다. HyperHelp 뷰어의 경우에 문자열 목록은 항상 정확히 하나의 항목(entry)을 포함하지만(HyperHelp가 자체 인덱싱 및 기타 다른 곳에서는 점 없는(pointless) 복제가 될 복제를 제공함), 온라인 설명서 뷰어의 경우에 문자열 목록은 여러 개의 문자열들로 이루어지는데 해당 키워드에 대한 페이지를 포함하는 설명서의 각 섹션당 문자열 하나입니다.

```
ICustomHelpViewer.ShowHelp(const HelpString:String)
```

이 구문은 도움말 뷰어가 특정한 키워드에 대한 도움말을 표시해야 하는 경우에 Help Manager에 의해 호출됩니다. 이 메소드는 이 작업의 마지막 메소드 호출인데 언제나 *CanShowKeyword*가 먼저 호출된 다음에 호출됩니다.

목차 표시

*ICustomHelpViewer*는 목차 표시와 관련된 두 가지 메소드를 제공합니다.

- *CanShowTableOfContents*
- *ShowTableOfContents*

이 작업의 이론은 다음 키워드 도움말 요청 함수의 작업과 유사합니다. 즉 Help Manager는 먼저 *ICustomHelpViewer.CanShowTableOfContents: Boolean*을 호출하여 모든 도움말 뷰어를 쿼리한 다음 *ICustomHelpViewer.ShowTableOfContents*를 호출하여 특정한 도움말 뷰어를 호출합니다.

특정 뷰어는 목차 지원 요청을 거부할 수도 있습니다. 예를 들어, 목차의 개념은 Man 페이지의 작동 방식에 제대로 매핑되지 않기 때문에 이런 현상이 일어날 수도 있습니다. 이와 반대로 HyperHelp 뷰어는 목차를 표시하는 요청을 직접 HyperHelp에 전달함으로써 목차를 지원합니다. 그러나 *ICustomHelpViewer*의 구현이 *CanShowTableOfContents*를 통한 쿼리에 *true*로 응답하고 *ShowTableOfContents*를 통한 요청을 무시하는 것은 타당하지 않습니다.

IExtendedHelpViewer 구현

*ICustomHelpViewer*는 키워드 방식 도움말에 대한 직접적인 지원만 제공합니다. 일부 도움말 시스템(특히 WinHelp)은 애플리케이션에 보이지 않는 내부적인 방식으로 *컨텍스트 ID*로 알려진 숫자를 키워드에 연결시켜서 작동합니다. 이 도움말 시스템은 문자열이 아닌 컨텍스트를 사용하여 도움말 시스템을 호출하는 컨텍스트 방식 도움말을 애플리케이션에서 지원해야 하고 도움말 시스템은 번호 자체를 번역합니다.

CLX로 작성된 애플리케이션은 *ICustomHelpViewer*를 구현하는 객체를 *IExtendedHelpViewer*를 구현하는 객체로 확장하여 컨텍스트 방식 도움말을 요구하는 시스템과 대화할 수 있습니다. 또한 *IExtendedHelpViewer*는 도움말 시스템과의 대화를 지원하여 키워드 검색을 사용하는 대신 상위 수준의 항목으로 직접 이동할 수 있도록 합니다.

*IExtendedHelpViewer*는 네 가지 함수를 사용합니다. 네 가지 함수 중에서 *UnderstandsContext*와 *DisplayHelpByContext*는 컨텍스트 방식 도움말을 지원하는 데 사용되고 나머지 *UnderstandsTopic*과 *DisplayTopic*은 항목을 지원하는 데 사용됩니다.

애플리케이션 사용자가 F1 키를 누르면 Help Manager는 다음을 호출합니다.

```
IExtendedHelpViewer.UnderstandsContext(const ContextID:Integer;
const HelpFileName:String):부울
```

그리고 현재 활성화된 컨트롤은 키워드 방식 도움말이 아닌 컨텍스트 방식 도움말을 지원합니다. *ICustomHelpViewer.CanShowKeyword*에서와 같이 Help Manager는 등록된 모든 도움말 뷰어를 반복하여 쿼리합니다. 그러나 *ICustomHelpViewer.CanShowKeyword*의 경우와 달리 둘 이상의 뷰어가 지정된 컨텍스트를 지원하면 주어진 컨텍스트를 지원하는 첫 번째 등록된 뷰어가 호출됩니다.

다음 구문은

```
IExtendedHelpViewer.DisplayHelpByContext(const ContextID:Integer;
const HelpFileName:String)
```

Help Manager가 등록된 도움말 뷰어를 폴링 (polling)한 후에 호출합니다.

항목 지원 함수는 동일한 방식으로 작동합니다.

```
IExtendedHelpViewer.UnderstandsTopic(const Topic:String): Boolean
```

이 구문은 항목을 지원하는지 묻는 도움말 뷰어를 폴링 (polling)하는 데 사용되고

```
IExtendedHelpViewer.DisplayTopic(const Topic:String)
```

이 구문은 항목에 대한 도움말을 제공할 수 있다고 보고하는 첫 번째 등록된 뷰어를 호출하는 데 사용됩니다.

IHelpSelector 구현

*IHelpSelector*는 *ICustomHelpViewer*의 짝 (companion)입니다. 둘 이상의 등록된 뷰어가 입력된 키워드, 컨텍스트 또는 항목에 대한 지원을 제공하거나 목차를 제공하겠다고 요청하면 Help Manager는 이들 중에서 선택해야 합니다. 컨텍스트 또는 항목의 경우에 언제나 Help Manager는 지원을 제공하겠다고 요청하는 첫 번째 도움말 뷰어를 선택합니다. 키워드나 목차의 경우에 Help Manager는 기본적으로 첫 번째 도움말 뷰어를 선택합니다. 뷰어 선택은 애플리케이션으로 조정할 수 있습니다.

Help Manager의 선택을 무시하고 애플리케이션에서 뷰어를 선택하려면 애플리케이션은 *IHelpSelector* 인터페이스의 구현을 제공하는 클래스를 등록해야 합니다. *IHelpSelector*는 *SelectKeyword*와 *TableOfContents*라는 두 가지 함수를 export합니다. 두 함수는 가능한 키워드 일치 사항이나 목차 제공을 요청하는 뷰어의 이름을 포함하는 *TStrings*를 하나씩 인수로 사용합니다. 구현 프로그램은 선택한 문자열을 나타내는 *TStrings*에서 인덱스를 반환해야 합니다.

참고 문자열이 재정렬되면 Help Manager에서 혼란스러울 수 있으므로 *IHelpSelector*의 구현 프로그램에서 문자열을 재정렬하지 않는 것이 좋습니다. 도움말 시스템은 오직 하나의 HelpSelector만 지원하므로 새 선택기 (selector)가 등록되면 이전에 사용한 기존 선택기는 연결이 끊어집니다.

도움말 시스템 객체 등록

Help Manager가 객체들과 통신하려면 *ICustomHelpViewer*, *IExtendedHelpViewer*, *ISpecialWinHelpViewer*, *IHelpSelector*를 구현하는 객체들을 Help Manager에 등록해야 합니다.

도움말 시스템 객체들을 Help Manager에 등록하려면 다음 작업을 수행해야 합니다.

- 도움말 뷰어 등록
- 도움말 선택기 등록

도움말 뷰어 등록

객체 구현을 포함하는 유닛은 *HelpIntfs*를 사용해야 합니다. 객체의 인스턴스는 구현 유닛의 **var** 섹션에서 선언해야 합니다.

구현 유닛의 초기화 섹션은 인스턴스 변수를 할당하여 *RegisterViewer* 함수로 전달해야 합니다. *RegisterViewer*는 *ICustomHelpViewer*를 인수로 사용하고 *IHelpManager*를 반환하는 *HelpIntfs.pas*에 의해 export된 평면 함수(Flat function)입니다. *IHelpManager*는 차후 사용을 위해 저장되어야 합니다.

도움말 선택기 등록

객체 구현을 포함하는 유닛은 *HelpIntfs*와 *Qforms*를 사용해야 합니다. 객체의 인스턴스는 구현 유닛의 **var** 섹션에서 선언해야 합니다.

구현 유닛의 초기화 섹션은 다음과 같은 전역 Application 객체의 *HelpSystem* 속성을 통해 도움말 선택기를 등록해야 합니다.

```
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
```

이 프로시저는 값을 반환하지 않습니다.

VCL 애플리케이션에서 도움말 사용

다음 단원에서는 VCL 애플리케이션 내에서 도움말을 사용하는 방법에 대해 설명합니다.

- TApplication이 VCL 도움말을 처리하는 방법
- VCL 컨트롤의 도움말 처리 방법
- 도움말 시스템 직접 호출
- IHelpSystem 사용

TApplication이 VCL 도움말을 처리하는 방법

VCL의 *TApplication*은 애플리케이션 코드에 액세스할 수 있는 다음과 같은 네 개의 메소드를 제공합니다.

표 5.5 TApplication의 도움말 메소드

HelpCommand	Windows 도움말 스타일 HELP_COMMAND를 취하고 WinHelp로 전달합니다. 이러한 메커니즘을 통해 전달된 Help 요청은 ISpecialWinHelpViewer의 구현에만 전달됩니다.
HelpContext	컨텍스트 방식 도움말에 대한 요청으로 도움말 시스템을 호출합니다.
HelpKeyword	키워드 방식 도움말에 대한 요청으로 도움말 시스템을 호출합니다.
HelpJump	특정 항목 표시를 요청합니다.

네 개의 함수는 모두 전달 받은 데이터를 취하고 도움말 시스템을 나타내는 *TApplication*의 데이터 멤버를 통해 요청을 전달 (forward) 합니다. 이 데이터 멤버는 *HelpSystem* 속성을 통해 직접 액세스할 수 있습니다.

VCL 컨트롤의 도움말 처리 방법

*TControl*에서 파생되는 모든 컨트롤은 도움말 시스템이 사용하는 *HelpSystem*, *HelpType*, *HelpContext*, *HelpKeyword*라는 속성을 나타냅니다.

HelpType 속성은 컨트롤의 디자이너가 도움말이 키워드 방식 도움말 또는 컨텍스트 도움말을 통해 제공되는 것을 예상하는지 결정하는 열거 타입의 인스턴스를 포함합니다. *HelpType*이 *htKeyword*로 설정될 경우 도움말 시스템은 컨트롤이 키워드 방식 도움말을 사용하는 것으로 예상하고, 도움말 시스템은 *HelpKeyword* 속성의 내용만 봅니다. 이와 반대로 *HelpType*이 *htContext*로 설정될 경우, 도움말 시스템은 컨트롤이 컨텍스트 방식 도움말을 사용하는 것으로 예상하고 *HelpContext* 속성의 내용만 봅니다.

속성 이외에 컨트롤은 단일 메소드인 *InvokeHelp*를 보여 주는데 이 메소드는 요청을 도움말 시스템으로 전달하기 위해 호출됩니다. 이 메소드는 매개변수를 사용하지 않고 전역 Application 객체에서 컨트롤이 지원하는 도움말의 종류에 해당하는 메소드를 호출합니다.

*TWinControl*의 *KeyDown* 메소드가 *InvokeHelp*를 호출하기 때문에 F1 키를 누르면 도움말 메시지가 자동으로 호출됩니다.

CLX 애플리케이션에서 도움말 사용

다음 단원에서는 CLX 애플리케이션 내에서 도움말을 사용하는 방법에 대해 설명합니다.

- TApplication이 CLX 도움말을 처리하는 방법
- CLX 컨트롤의 도움말 처리 방법
- 도움말 시스템 직접 호출
- IHelpSystem 사용

TApplication이 CLX 도움말을 처리하는 방법

CLX의 *TApplication*은 애플리케이션 코드에서 액세스할 수 있는 다음과 같은 두 가지 메소드를 제공합니다.

- *ContextHelp*는 컨텍스트 방식 도움말 요청을 사용하여 도움말 시스템을 호출합니다.
- *KeywordHelp*는 키워드 방식 도움말 요청을 사용하여 도움말 시스템을 호출합니다.

이 두 함수는 전달(pass)되는 컨텍스트 또는 키워드를 인수로 사용하고 도움말 시스템을 나타내는 *TApplication*의 데이터 멤버를 통해 요청을 전달(forward)합니다. 이 데이터 멤버는 읽기 전용 속성인 *HelpSystem*을 통해 직접 액세스할 수 있습니다.

CLX 컨트롤의 도움말 처리 방법

*TControl*에서 파생되는 모든 컨트롤은 도움말 시스템이 사용하는 *HelpType*, *HelpFile*, *HelpContext*, *HelpKeyword*라는 네 가지 속성을 나타냅니다. *HelpFile*은 컨트롤의 도움말이 있는 파일의 이름을 포함하는 것으로 가정되고 도움말이 파일 이름에 대해 신경 쓰지 않는 외부 도움말 시스템(예를 들어, Man 페이지 시스템)에 있는 경우 속성은 비워져 있어야 합니다.

HelpType 속성은 컨트롤의 디자이너가 도움말이 키워드 방식 도움말이나 컨텍스트 방식 도움말을 통해 제공되는 것을 예상하는지 결정하는 열거 타입의 인스턴스를 포함하고 다른 두 가지 속성은 여기에 연결됩니다. *HelpType*이 *htKeyword*로 설정될 경우, 도움말 시스템은 컨트롤이 키워드 방식 도움말을 사용하는 것으로 예상하고 *HelpKeyword* 속성의 내용만 봅니다. 이와 반대로 *HelpType*이 *htContext*로 설정될 경우, 도움말 시스템은 컨트롤이 컨텍스트 방식 도움말을 사용하는 것으로 예상하고 도움말 시스템은 *HelpContext* 속성의 내용만 봅니다.

속성 이외에 컨트롤은 단일 메소드인 *InvokeHelp*를 보여 주는데 이 메소드는 요청을 도움말 시스템으로 전달하기 위해 호출됩니다. 이 메소드는 매개변수를 사용하지 않고 전역 Application 객체에서 컨트롤이 지원하는 도움말의 종류에 해당하는 메소드를 호출합니다.

*TWidgetControl*의 *KeyDown* 메소드가 *InvokeHelp*를 호출하기 때문에 F1 키를 누르면 도움말 메시지는 자동으로 호출됩니다.

도움말 시스템 직접 호출

VCL이나 CLX에 의해 제공되지 않은 추가적인 도움말 시스템 기능에 있어서 *TApplication*은 도움말 시스템에 대해 직접 액세스할 수 있는 읽기 전용 속성을 제공합니다. 이 속성은 인터페이스 *IHelpSystem* 구현의 인스턴스입니다. *IHelpSystem*과 *IHelpManager*는 동일한 객체의 의해 구현되지만 어떤 인터페이스는 애플리케이션이 Help Manager와 통신하는 데 사용되고 어떤 인터페이스는 도움말 뷰어가 Help Manager와 통신하는 데 사용됩니다.

IHelpSystem 사용

*IHelpSystem*을 통해 VCL 또는 CLX 애플리케이션은 다음 세 가지를 수행할 수 있습니다.

- Help Manager에 경로 정보 제공
- 새 도움말 선택기 제공
- Help Manager가 도움말을 표시하도록 요청

여러 외부 도움말 시스템에서 동일한 키워드에 대한 도움말을 제공할 수 있는 경우에는 도움말 선택기 할당을 통해 Help Manager는 의사 결정을 위임(delegate)할 수 있습니다. 자세한 내용은 27 페이지의 "IHelpSelector 구현" 단원을 참조하십시오.

*IHelpSystem*은 Help Manager에게 도움말을 표시하도록 요청하기 위해 다음 네 가지 프로시저와 한 가지 함수를 export합니다.

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

*Hook*은 전적으로 WinHelp 호환성을 위해 고안되었으므로 CLX 애플리케이션에서 사용해서는 안 됩니다. *Hook*을 통해 키워드 방식, 컨텍스트 방식 및 항목 방식 도움말에 대한 요청에 직접적으로 매핑될 수 없는 WM_HELP 메시지를 처리할 수 있습니다. 나머지 각 메소드는 키워드, 컨텍스트 ID를 인수로 사용하고 도움말에 요청하는 항목과 도움말이 찾으리라고 예상되는 도움말 파일을 사용합니다.

일반적으로 항목 방식 도움말을 요청하지 않는 경우 컨트롤의 *InvokeHelp* 메소드를 통해 도움말 요청을 Help Manager에 전달하는 것이 효과적이고 더욱 확실합니다.

IDE 도움말 시스템 사용자 지정

Delphi IDE는 VCL 또는 CLX 애플리케이션과 똑같은 방식으로 다중 도움말 뷰어를 지원합니다. 도움말 요청을 Help Manager에 위임한 다음 등록된 도움말 뷰어에 전달(forward)합니다. IDE는 VCL에서 사용하는 것과 동일한 WinHelpViewer를 이용합니다.

IDE에서의 새 도움말 뷰어 설치 방법은 한 가지를 제외하고 CLX 애플리케이션에서의 설치 방법과 동일합니다. *ICustomHelpViewer*(원하는 경우 *IExtendedHelpViewer*)를 구현하는 객체를 작성하여 도움말 요청을 자신이 선택한 외부 뷰어에 전달하고 *ICustomHelpViewer*를 IDE에 등록합니다.

사용자 지정 도움말 뷰어를 IDE에 등록하려면 다음과 같이 합니다.

- 1 도움말 뷰어를 구현하는 유닛이 HelpIntfs.pas를 포함하는지 확인합니다.
- 2 유닛을 IDE에 등록된 디자인 타임 패키지로 만들고 런타임 패키지를 선택하여 패키지를 만듭니다. (이것은 유닛이 사용하는 Help Manager 인스턴스가 IDE가 사용하는 Help Manager 인스턴스와 동일하다는 것을 확인하는 데 필요합니다.)
- 3 도움말 뷰어가 유닛 내에서 전역 인스턴스로 존재하는지 확인합니다.
- 4 유닛의 초기화 섹션에서 인스턴스가 *RegisterHelpViewer* 함수로 넘겨지는지 확인합니다.

6

애플리케이션 사용자 인터페이스 개발

Delphi에서 컴포넌트 팔레트의 컴포넌트를 선택한 후 폼에 선택한 컴포넌트를 가져다 놓아 사용자 인터페이스(UI)를 디자인합니다. 컴포넌트의 속성을 설정하고 컴포넌트의 이벤트 핸들러를 코딩하여 원하는 동작을 수행합니다.

애플리케이션 동작 제어

TApplication, *TScreen* 및 *TForm*은 프로젝트의 동작을 제어하여 모든 Delphi 애플리케이션의 중요한 요소를 형성하는 클래스입니다. *TApplication* 클래스는 표준 프로그램의 동작을 캡슐화하는 속성 및 메소드를 제공하여 애플리케이션의 기초를 형성합니다. *TScreen*은 런타임에 화면 해상도 및 사용 가능한 디스플레이 글꼴 등의 시스템 특정 정보를 유지할 뿐만 아니라 로드된 폼과 데이터 모듈을 추적하는 데 사용됩니다. *TForm* 클래스의 인스턴스는 애플리케이션 사용자 인터페이스의 구축 블록입니다. 애플리케이션의 창과 대화 상자는 *TForm*을 기반으로 합니다.

메인 폼 사용

*TForm*은 GUI 애플리케이션을 작성하기 위한 핵심 클래스입니다. 기본 프로젝트를 표시하는 Delphi를 열거나 프로젝트를 새로 만들 때 사용자가 UI를 디자인할 수 있도록 폼이 표시됩니다.

프로젝트에서 만들고 저장한 첫 번째 폼은 기본적으로 런타임에 생성되는 첫 번째 폼인 프로젝트의 메인 폼이 됩니다. 프로젝트에 폼을 추가할 때 애플리케이션의 메인 폼으로 다른 폼을 지정할 수 있습니다. 또한 한 폼을 메인 폼으로 지정하면 런타임에 테스트하기가 쉽습니다. 폼 생성 순서를 변경하지 않는 한 메인 폼은 실행되는 애플리케이션에서 표시되는 첫 번째 폼이기 때문입니다.

다음과 같은 방법으로 프로젝트 메인 폼을 변경합니다.

- 1 Project|Options를 선택한 다음 Forms 페이지를 선택합니다.
- 2 Main Form 콤보 상자에서 프로젝트 메인 폼으로 사용할 폼을 선택한 다음 OK를 선택합니다.

이제 애플리케이션을 실행하면 메인 폼으로 선택한 폼이 먼저 나타납니다.

폼 추가

프로젝트에 폼을 추가하려면 File|New Form을 선택합니다. Project Manager(View|Project Manager)에 나열된 모든 프로젝트 폼 및 연결 유닛을 볼 수 있고 View|Forms를 선택하여 폼 목록을 표시할 수 있습니다.

폼 연결

프로젝트에 폼을 추가하면 프로젝트 파일의 폼에 참조가 추가되지만 프로젝트의 다른 유닛에는 추가되지 않습니다. 새로운 폼을 참조하는 코드를 작성하기 전에 참조 폼의 유닛 파일에 있는 폼에 참조를 추가해야 합니다. 이것을 **폼 연결**이라고 합니다.

폼을 연결하는 일반적인 이유는 해당 폼의 컴포넌트에 대한 액세스를 제공하기 위함입니다. 예를 들면, 경우에 따라 폼 연결을 통해 데이터 모듈의 데이터 액세스 컴포넌트에 연결하는 data-aware 컴포넌트가 들어 있는 폼을 사용할 수 있습니다.

다음과 같은 방법으로 하나의 폼을 다른 폼에 연결합니다.

- 1 다른 폼을 참조할 폼을 선택합니다.
- 2 File|Use Unit을 선택합니다.
- 3 참조할 폼의 폼 유닛 이름을 선택합니다.
- 4 OK를 선택합니다.

하나의 폼을 다른 폼에 연결한다는 것은 하나의 폼 유닛의 **uses** 절에 다른 폼 유닛에 대한 참조가 들어 있어 연결된 폼과 해당 컴포넌트가 이제 연결 폼의 유효 범위(scope)에 있음을 의미합니다.

순환 유닛 참조 방지

두 개의 폼이 서로 참조해야 하는 경우 프로그램 컴파일 시 "순환 참조" 오류가 발생할 수 있습니다. 이러한 오류를 방지하려면 다음 중 하나를 수행하십시오.

- 유닛 식별자를 사용하여 두 개의 **uses** 절을 모두 각 유닛 파일의 **implementation** 부분에 둡니다. File|Use Unit 명령을 사용합니다.
- **uses** 절 하나는 **interface** 부분에 두고 다른 하나는 **implementation** 부분에 둡니다. 이 유닛의 **interface** 부분에 다른 폼의 유닛 식별자를 둘 필요는 없습니다.

두 개의 **uses** 절을 모두 각 유닛 파일에 있는 **interface** 부분에 두지 마십시오. 그렇게 할 경우 컴파일 타임에 "순환 참조" 오류가 생깁니다.

메인 폼 숨기기

애플리케이션이 처음 시작될 때 메인 폼이 나타나지 않게 할 수 있습니다. 그렇게 하려면 다음 항목에 설명되어 있는 *Application* 전역 변수를 사용해야 합니다.

다음과 같이 하여 시작할 때 메인 폼을 숨깁니다.

- 1 Project|View Source를 선택하여 메인 프로젝트 파일을 표시합니다.
- 2 Application.CreateForm에 대한 호출 뒤 그리고 Application.Run에 대한 호출 앞에 다음 줄을 추가합니다.

```
Application.ShowMainForm := False;
Form1.Visible := False; { the name of your main form may differ }
```

참고 위에서처럼 런타임에 설정하지 않고 디자인 타임에 Object Inspector를 사용하여 폼의 *Visible* 속성을 *False*로 설정할 수 있습니다.

애플리케이션 레벨에서 작업

TApplication 타입의 *Application* 전역 변수는 모든 VCL 또는 CLX 기반 애플리케이션 안에 포함됩니다. *Application*은 프로그램의 백그라운드에서 발생하는 다양한 함수를 제공할 뿐만 아니라 애플리케이션을 캡슐화합니다. 예를 들어, *Application*은 프로그램 메뉴에서 도움말 파일을 호출하는 방식을 처리합니다. *TApplication* 작동 방식에 대한 이해는 독립형 애플리케이션 개발자보다 컴포넌트 작성자에게 더 중요하며 프로젝트를 만들 때 Project|Options 메뉴의 *Application* 페이지에서 *Application*이 처리하는 옵션을 설정해야 합니다.

또한 *Application*은 애플리케이션 전체에 적용되는 많은 이벤트를 받습니다. 예를 들어, *OnActivate* 이벤트는 응용 프로그램이 처음 시작할 때 작업을 수행할 수 있게 해주고, *OnIdle* 이벤트는 응용 프로그램이 유휴 상태일 때 백그라운드 프로세스를 수행할 수 있게 해주며, *OnMessage* 이벤트는 Windows의 경우 Windows 메시지를 가로챌 수 있게 해주고, *OnEvent* 이벤트는 이벤트를 가로챌 수 있게 해줍니다. IDE를 사용하여 *Application* 전역 변수의 속성 및 이벤트를 조사할 수는 없지만 다른 컴포넌트인 *TApplicationEvents*가 이벤트를 가로채서 IDE를 이용해 이벤트 핸들러를 제공할 수 있게 해줍니다.

화면 처리

*Screen*이라는 *TScreen* 타입의 전역 변수는 프로젝트를 생성할 때 만들어집니다. *Screen*은 애플리케이션이 실행 중인 화면의 상태를 캡슐화합니다. *Screen*에서 실행되는 일반 작업은 다음 내용들을 지정합니다.

- 커서 모양
- 실행 중인 애플리케이션의 창 크기
- 화면 장치에서 사용 가능한 글꼴 목록
- 다중 화면 동작(크로스 플랫폼에서는 사용 불가)

Windows 애플리케이션이 여러 모니터에서 실행되는 경우, *Screen*에서 모니터 목록과 치수를 관리해 주므로 사용자는 사용자 인터페이스 레이아웃을 효과적으로 관리할 수 있습니다.

크로스 플랫폼 프로그래밍에 CLX를 사용하는 경우, 기본 동작은 애플리케이션이 현재 화면 장치에 대한 정보에 기반하는 화면 컴포넌트를 만든 다음 이것을 *Screen*에 할당하는 것입니다.

레이아웃 관리

가장 간단하게 하려면 폼에서 컨트롤을 두는 위치에 따라 사용자 인터페이스의 레이아웃을 제어합니다. 위치 선택은 컨트롤의 *Top*, *Left*, *Width* 및 *Height* 속성에 반영됩니다. 런타임에 이러한 값을 변경하여 폼에서의 컨트롤 위치 및 크기를 변경할 수 있습니다.

그 밖에도 컨트롤은 콘텐츠 또는 컨테이너에 자동으로 조정할 수 있게 하는 다양한 속성을 가집니다. 따라서 작은 부분들이 통일된 전체로 조화를 이루도록 폼을 레이아웃할 수 있습니다.

두 속성은 부모 컨트롤과 관련해서 컨트롤 위치 및 크기를 지정하는 방법에 영향을 줍니다. *Align* 속성을 사용하여 특정 가장자리를 따라서 부모 컨트롤 내에서 컨트롤을 완벽하게 맞출 수 있으며 다른 모든 컨트롤이 정렬된 후 전체 클라이언트 영역을 채울 수 있습니다. 부모 컨트롤의 크기가 조정되면 해당 부모 컨트롤에 정렬된 컨트롤은 자동으로 크기가 조정되며 특정 가장자리에 맞도록 위치가 유지됩니다.

컨트롤의 위치를 부모 컨트롤의 특정 가장자리에 따라 상대적으로 두지만 꼭 가장자리에 맞아야 하거나 전체 가장자리를 따라 실행될 수 있도록 크기를 조정할 필요가 없는 경우에는 *Anchors* 속성을 사용합니다.

컨트롤이 너무 커지거나 너무 작아지지 않게 하려면 *Constraints* 속성을 사용합니다. *Constraints*는 사용자가 컨트롤의 최대 높이, 최소 높이, 최대 너비 및 최소 너비를 지정할 수 있습니다. 이 값들을 사용하여 컨트롤의 높이와 너비의 크기(픽셀)를 제한하십시오. 예를 들어, 컨테이너 객체의 *MinWidth* 및 *MinHeight* constraints 속성을 설정하여 자식 객체가 항상 표시되게 할 수 있습니다.

*Constraints*의 값은 부모/자식 계층을 통해 전달되며 객체에 크기 제한이 있는 정렬된 자식이 포함되기 때문에 객체 크기가 제한될 수 있습니다. *Constraints*는 또한 *ChangeScale* 메소드가 호출될 때 컨트롤이 특정한 크기로 조정되는 것을 방지합니다.

*TControl*은 *TConstrainedResizeEvent* 타입의 보호 이벤트인 *OnConstrainedResize*를 불러들입니다.

```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight,
    MaxWidth, MaxHeight: Integer) of object;
```

이 이벤트는 컨트롤 크기를 변경하려는 시도가 있을 경우 사용자가 크기 제한을 오버라이드할 수 있습니다. 제한 값은 이벤트 핸들러 내에서 변경될 수 있는 var 매개변수로서 전달됩니다. *OnConstrainedResize*는 컨테이너 객체에 대해 게시됩니다 (*TForm*, *TScrollBar*, *TControlBar* 및 *TPanel*). 또한 컴포넌트 작성자는 모든 *TControl*의 자손 클래스에 대해 이 이벤트를 사용하거나 게시할 수 있습니다.

크기를 변경할 수 있는 항목이 있는 컨트롤은 컨트롤 자체 크기를 글꼴에 맞게 조정하거나 포함된 객체에 맞게 조정할 수 있도록 *AutoSize* 속성을 가집니다.

이벤트 공지에 대한 응답

운영 체제는 사용자의 애플리케이션이 실행되는 동안 언제 이벤트가 발생(마우스 클릭, 키 입력 등)했는지 알려 줍니다. VCL과 CLX 객체에서 이벤트 공지가 처리되는 기본 방식은 다르지만 컴포넌트 수준에서 사용자가 이벤트 공지와 관련하여 작업하는 방식은 일반적으로 동일합니다. 컴포넌트는 이벤트와 가장 일반적으로 발생하는 이벤트에 대해 기본 제공된 메소드를 가집니다. 대부분의 경우에는 컴포넌트에 제공된 메소드를 사용할 수 있습니다. 추가로 이벤트 핸들링을 작성해야 하는 경우에는 자신의 이벤트 핸들링을 작성하여 기존 메소드를 오버라이드할 수 있습니다. 자신의 컴포넌트를 작성하는 경우가 아니라면 기본 이벤트 공지 스키마를 변경할 필요가 없습니다.

VCL Windows 전용 애플리케이션을 개발하는 경우에는 Windows가 메시지 기반의 운영 체제라는 사실을 이해해야 합니다. 시스템 메시지는 이 메시지를 이벤트나 이벤트 핸들러에 대한 메시지로 번역하는 메시지 핸들러에 의해 처리됩니다. Windows에서 메시지 자체는 컨트롤에 전달되는 레코드입니다. 예를 들어, 대화 상자에서 마우스 버튼을 클릭하면 Windows가 활성화된 컨트롤로 메시지를 전달하고 해당 컨트롤을 포함하는 애플리케이션은 이 새로운 이벤트에 반응합니다. 클릭이 버튼 위에서 발생한 경우에는 메시지 수령 시에 *OnClick* 이벤트가 활성화될 것입니다. 클릭이 단지 폼 내에서 발생한 경우에는 애플리케이션이 메시지를 무시할 수 있습니다.

Windows에 의해 애플리케이션에 전달되는 레코드 타입은 *TMsg*라고 합니다. Windows는 각각의 메시지에 대한 상수를 미리 정의하며 이 값들은 *TMsg* 레코드의 메시지 필드에 저장됩니다. 이들 각각의 상수는 *wm* 문자로 시작합니다.

사용자가 메시지 핸들링 시스템을 오버라이드하고 자신의 메시지 핸들러를 작성한 경우가 아니라면 VCL은 자동으로 메시지를 처리합니다. 메시지와 메시지 핸들링에 대한 자세한 내용은 46-1 페이지의 "메시지 처리 시스템에 대한 이해", 46-3 페이지의 "메시지 처리 변경" 및 46-5 페이지의 "새 메시지 핸들러 생성"을 참조하십시오.

CLX 크로스 플랫폼 프로그램용: 이벤트가 발생했다는 운영 체제의 공지는 기본 Qt widget 레이어에 전달되어 이벤트로 번역되고 결국은 *HookEvents*에 의해 이벤트 객체로 번역됩니다. *EventFilter*는 CLX 컨트롤이 Qt 마우스 또는 키보드 이벤트를 처리할 필요가 있을 때 자동으로 호출됩니다.

*EventFilter*는 기본 응답 수행에 따라 이벤트 공지에 응답합니다. 일반적으로 여기에는 *OnClick* 이벤트를 발생시키는 *Click* 메소드와 같은 적절한 가상 메소드에 대한 이벤트 디스패칭이 포함됩니다.

CLX 참고 *EventFilter* 메소드를 오버라이드할 경우에는 기본 이벤트 프로세스가 발생할 수 있도록 상속된 메소드를 호출해야 합니다.

폼 사용

Delphi의 IDE에서 폼을 만들면 Delphi는 애플리케이션의 메인 엔트리 포인트에 코드를 포함시켜 메모리에 폼을 자동으로 만듭니다. 일반적으로 이것은 바람직한 동작이므로 변경할 필요가 없습니다. 즉, 메인 창은 프로그램이 실행되는 동안 계속 유지되므로 메인 창에서 폼을 만들 때 기본 Delphi 동작을 변경하지 않습니다.

그러나 프로그램이 실행되는 동안 메모리에서 애플리케이션의 모든 폼을 원하지 않을 수도 있습니다. 즉, 한 번에 애플리케이션의 모든 대화 상자가 메모리에 있는 것을 원하지 않을 경우 대화 상자가 나타나기를 원할 때 동적으로 해당 대화 상자를 만들 수 있습니다.

폼은 모달 폼이거나 모달리스 폼입니다. 모달 폼은 사용자가 다른 폼으로 전환하기 전에 상호 작용해야 하는 폼입니다(예: 사용자 입력을 필요로 하는 대화 상자). 모달리스 폼은 다른 창에 의해 가려지거나 사용자가 닫거나 또는 최소화할 때까지 표시되는 창입니다.

메모리에 상주하는 폼의 제어

기본적으로 Delphi는 다음 코드를 애플리케이션의 메인 엔트리 포인트에 포함하여 메모리에 애플리케이션의 메인 폼을 자동으로 작성합니다.

```
Application.CreateForm(TForm1, Form1);
```

이 함수는 폼과 동일한 이름을 갖는 전역 변수를 만듭니다. 따라서 애플리케이션의 모든 폼은 연결된 전역 변수를 갖습니다. 이 변수는 폼 클래스의 인스턴스에 대한 포인터이고 애플리케이션이 실행되는 동안 폼을 참조하는 데 사용됩니다. 폼의 유닛을 **uses** 절에 포함하는 모든 유닛은 이 변수를 통해 폼에 액세스할 수 있습니다.

프로젝트 유닛에서 이 방법으로 만든 모든 폼은 프로그램이 호출되면 나타나며 애플리케이션이 실행되는 동안 메모리에 있습니다.

자동 생성된 폼 표시

시작할 때 폼을 만들고 나중에 프로그램 실행 중 일정 시간 동안 표시되지 않기를 원하는 경우 폼의 이벤트 핸들러에서 *ShowModal* 메소드를 사용하여 메모리에 이미 로드된 폼을 표시할 수 있습니다.

```
procedure TMainForm.Button1Click(Sender:TObject);
begin
    ResultsForm.ShowModal;
end;
```

이 경우에는 메모리에 이미 폼이 있으므로 다른 인스턴스를 만들거나 해당 인스턴스를 소멸할 필요가 없습니다.

동적으로 폼 생성

항상 모든 애플리케이션이 한 번에 메모리에 있기를 원하지 않을 수도 있습니다. 로드 시 필요한 메모리 양을 줄이려면 폼을 사용해야 할 때만 일부 폼을 만듭니다. 예를 들어, 대화 상자는 사용자가 상호 작용하는 중에만 메모리에 있으면 됩니다.

IDE를 사용하여 실행하는 동안 다른 단계에서 폼을 만들려면 다음과 같이 수행합니다.

- 1 메인 메뉴에서 File|New Form을 선택하여 새로운 폼을 표시합니다.
- 2 Project|Options|Forms 페이지의 자동 생성 폼 목록에서 해당 폼을 제거합니다. 이렇게 하여 폼 호출을 제거합니다. 또는 프로그램의 메인 엔트리 포인트에서 다음 줄을 수동으로 제거할 수도 있습니다.


```
Application.CreateForm(TResultsForm, ResultsForm);
```
- 3 모달리스 폼일 경우 폼의 *Show* 메소드를 사용하고 모달 폼의 경우에는 *ShowModal* 메소드를 사용하여 필요할 때 폼을 호출합니다.

메인 폼의 이벤트 핸들러는 결과 폼의 인스턴스를 만들고 소멸해야 합니다. 결과 폼을 호출하는 방법 중 한 가지로 다음과 같이 전역 변수를 사용합니다. *ResultsForm*이 모달 폼이므로 핸들러는 *ShowModal* 메소드를 사용합니다.

```
procedure TMainForm.Button1Click(Sender:TObject);
begin
  ResultsForm:=TResultForm.Create(self)
try
  ResultsForm.ShowModal;
finally
  ResultsForm.Free;
end;
```

위의 예제에서 **try..finally**의 사용에 주의하십시오. **finally** 절에 *ResultsForm.Free*; 행을 입력함으로써 폼이 예외를 발생시키는 경우라도 폼 메모리가 해제되는 것을 보장할 수 있습니다.

예제의 이벤트 핸들러는 폼을 닫은 후 해당 폼을 삭제하므로 애플리케이션에서 *ResultsForm*을 사용해야 할 경우에는 폼을 다시 생성해야 합니다. *Show*를 사용하여 폼을 표시한 경우 폼이 열려 있는 동안 *Show*가 반환되기 때문에 이벤트 핸들러 내에서 해당 폼을 삭제할 수 없습니다.

참고 생성자를 사용하여 폼을 만드는 경우 해당 폼이 Project|Options|Forms 페이지의 자동 생성 폼 목록에 없는지 반드시 확인해야 합니다. 특히 목록에서 동일한 이름을 갖는 폼을 삭제하지 않고 새로운 폼을 만드는 경우, Delphi가 시작 시 폼을 만들고 해당 이벤트 핸들러가 폼의 새 인스턴스를 만들어 자동 생성된 인스턴스에 대한 참조를 덮어씁니다. 자동 생성된 인스턴스가 여전히 존재하지만 애플리케이션에서 더 이상 해당 인스턴스에 액세스할 수 없습니다. 이벤트 핸들러가 종료되면 전역 변수는 더 이상 유효한 폼을 가리키지 않습니다. use 전역 변수를 사용하려고 시도할 경우 애플리케이션이 중단될 수 있습니다.

창 같은 모달리스 폼 생성

모달리스 폼의 참조 변수는 폼을 사용하는 동안 계속 존재해야 합니다. 이것은 모달리스 폼의 참조 변수가 전역 유효 범위(scope)를 가져야 한다는 것을 의미합니다. 대부분의 경우 폼(폼의 이름 속성과 일치하는 변수 이름)을 만들 때 생성된 전역 참조 변수를 사용합니다. 애플리케이션에서 폼의 추가 인스턴스를 필요로 하는 경우 각 인스턴스에 대해 별도의 전역 변수를 선언합니다.

로컬 변수로 폼 인스턴스 생성

모달 폼의 고유 인스턴스를 안전하게 만들려면 이벤트 핸들러의 로컬 변수를 새 인스턴스에 대한 참조로 사용합니다. 로컬 변수를 사용하는 경우 *ResultsForm*이 자동 생성된 것인지 여부는 중요하지 않습니다. 이벤트 핸들러의 코드는 전역 폼 변수를 참조하지 않습니다. 예를 들면, 다음과 같습니다.

```
procedure TMainForm.Button1Click(Sender:TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

폼의 전역 인스턴스는 이 버전의 이벤트 핸들러에서 사용되지 않습니다.

일반적으로 애플리케이션은 폼의 전역 인스턴스를 사용합니다. 그러나 모달 폼의 새 인스턴스가 필요하고 하나의 함수와 같은 애플리케이션의 제한된 별도 섹션에서 폼을 사용하는 경우 로컬 인스턴스는 폼을 사용하는 가장 안전하고 효율적인 방법입니다.

물론 로컬 변수는 폼을 사용하는 동안 해당 폼이 존재하도록 하는 전역 유효 범위(scope)를 가져야 하기 때문에 모달리스 폼의 이벤트 핸들러에서 로컬 변수를 사용할 수는 없습니다. *Show*는 폼이 열리는 즉시 반환되므로 로컬 변수를 사용한 경우 로컬 변수는 유효 범위(scope)에서 즉시 벗어납니다.

폼에 추가 인수 전달

일반적으로 IDE 안에서 애플리케이션에 폼을 만듭니다. 이렇게 만들면 폼은 만드는 폼의 소유자인 *Owner*라는 인수를 취하는 생성자를 가집니다. (소유자는 호출 애플리케이션 객체 또는 폼 객체입니다.) *Owner*는 *nil* 값을 가질 수 있습니다.

추가 인수를 폼에 전달하려면 이 새로운 생성자를 사용하여 별개의 생성자를 만들고 폼을 인스턴스화합니다. 아래 예제의 폼 클래스는 *whichButton* 별도 인수를 갖는 추가 생성자를 표시합니다. 이 새로운 생성자는 수동으로 폼 클래스에 추가됩니다.

```
TResultsForm = class(TForm)
  ResultsLabel:TLabel
  OKButton:TButton;
  procedure OKButtonClick(Sender:TObject);
private
public
```

```

    constructor CreateWithButton(whichButton:Integer; Owner:TComponent);
end;

```

이것은 *whichButton* 추가 인수를 전달하는 수동으로 코딩된 생성자입니다. 이 생성자는 *whichButton* 매개변수를 사용하여 폼에서 *Label* 컨트롤의 *Caption* 속성을 설정합니다.

```

constructor CreateWithButton(whichButton:Integer; Owner:TComponent);
begin
    inherited Create(Owner);
    case whichButton of
        1: ResultsLabel.Caption := 'You picked the first button.';
        2: ResultsLabel.Caption := 'You picked the second button.';
        3: ResultsLabel.Caption := 'You picked the third button.';
    end;
end;
end;

```

여러 생성자를 사용하여 폼의 인스턴스를 만드는 경우 용도에 가장 적합한 생성자를 선택할 수 있습니다. 예를 들면, 폼 호출에서 버튼에 대한 다음과 같은 *OnClick* 핸들러는 별도 매개변수를 사용하는 *TResultsForm*의 인스턴스를 만듭니다.

```

procedure TMainForm.SecondButtonClick(Sender:TObject);
var
    rf:TResultsForm;
begin
    rf := TResultsForm.CreateWithButton(2, self);
    rf.ShowModal;
    rf.Free;
end;

```

폼에서 데이터 검색

대부분의 실제 애플리케이션들은 여러 폼으로 구성됩니다. 이러한 폼들 사이에 정보를 전달해야 하는 경우가 종종 있습니다. 정보를 받는 폼의 생성자에 대한 매개변수로써 또는 폼 속성에 값을 할당하여 정보를 전달할 수 있습니다. 폼에서 정보를 얻는 방법은 모달 폼인지 모달리스 폼인지에 따라 다릅니다.

모달리스 폼에서 데이터 검색

폼의 *public* 멤버 함수를 호출하거나 폼의 속성을 쿼리하면 모달리스 폼에서 정확한 정보를 쉽게 추출할 수 있습니다. 예를 들어, 애플리케이션에 색상 목록("Red", "Green", "Blue" 및 기타 등등)을 갖는 *ColorListBox*라는 리스트 박스가 들어 있는 *ColorForm*이라는 모달리스 폼이 있다고 가정합니다. *ColorListBox*에서 선택한 색상 이름 문자열은 사용자가 색상을 새로 선택할 때마다 *CurrentColor*라는 속성에 자동으로 저장됩니다. 폼의 클래스 선언은 다음과 같습니다.

```

TColorForm = class(TForm)
    ColorListBox:TListBox;
    procedure ColorListBoxClick(Sender:TObject);
private
    FColor:String;
public

```

```

    property CurColor:String read FColor write FColor;
end;

```

리스트 박스의 *OnClick* 이벤트 핸들러인 *ColorListBoxClick*은 리스트 박스에서 항목을 새로 선택할 때마다 *CurrentColor* 속성 값을 설정합니다. 이벤트 핸들러는 색상 이름이 들어 있는 리스트 박스에서 문자열을 선택하여 *CurrentColor*에 할당합니다. *CurrentColor* 속성은 *SetColor* 설정 함수를 사용하여 *FColor* private 데이터 멤버에 실제 속성 값을 저장합니다.

```

procedure TColorForm.ColorListBoxClick(Sender:TObject);
var
    Index:Integer;
begin
    Index := ColorListBox.ItemIndex;
    if Index >= 0 then
        CurrentColor := ColorListBox.Items[Index]
    else
        CurrentColor := '';
    end;
end;

```

이제 애플리케이션 내의 *ResultsForm*이라는 또 다른 폼에서 *ResultsForm*의 *UpdateButton*이라는 버튼을 클릭할 때마다 *ColorForm*의 현재 선택되어 있는 색상을 알아야 한다고 가정합니다. *UpdateButton*의 *OnClick* 이벤트 핸들러는 다음과 같습니다.

```

procedure TResultForm.UpdateButtonClick(Sender:TObject);
var
    MainColor:String;
begin
    if Assigned(ColorForm) then
        begin
            MainColor := ColorForm.CurrentColor;
            {do something with the string MainColor}
        end;
    end;
end;

```

이벤트 핸들러는 먼저 *Assigned* 함수를 사용하여 *ColorForm*이 존재하는지 확인합니다. 그런 다음 이벤트 핸들러는 *ColorForm*의 *CurrentColor* 속성의 값을 얻습니다.

또는 *ColorForm*이 *GetColor*라는 public 함수를 가지는 경우 다른 폼은 *CurrentColor* 속성(예: *MainColor := ColorForm.GetColor*;)을 사용하지 않고 현재 색상을 얻을 수 있습니다. 실제로는 리스트 박스에서 직접 선택하여 다른 폼에서 *ColorForm*의 현재 선택된 색상을 얻지 못하게 할 방법이 없습니다.

```

with ColorForm.ColorListBox do
    MainColor := Items[ItemIndex];

```

하지만 속성을 사용하면 *ColorForm*에 대한 인터페이스를 매우 단순하게 만들 수 있습니다. 폼이 *ColorForm*에 대해 알아야 하는 것은 *CurrentColor*의 값을 확인하는 것뿐입니다.

모달 폼에서 데이터 검색

모달리스 폼과 마찬가지로 모달 폼에도 다른 폼에서 필요로 하는 정보가 들어 있는 경우가 많습니다. 가장 일반적인 예는 폼 A가 모달 폼 B를 실행하는 것입니다. 폼 B가 닫혀 있으면 폼 A는 폼 A를 처리할 방법을 결정하기 위해서 사용자가 폼 B에서 수행한 작업을 알아야 합니다. 폼 B가 아직 메모리에 있는 경우 위 예의 모달리스 폼에서처럼 속성이나 멤버 함수를 통해 쿼리할 수 있습니다. 그러나 종료 시에 폼 B가 메모리에서 삭제되는 상황은 어떻게 처리해야 할까요? 폼은 명시적인 반환 값을 갖지 않으므로 폼을 소멸하기 전에 해당 폼의 중요 정보를 보존해야 합니다.

예를 들어, 모달 폼으로 디자인된 *ColorForm* 폼의 수정된 버전을 살펴 봅니다. 클래스 선언은 다음과 같습니다.

```
TColorForm = class(TForm)
  ColorListBox: TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender:TObject);
  procedure SelectButtonClick(Sender:TObject);
private
  FColor:Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

폼은 색상 이름 목록을 갖는 *ColorListBox*라는 리스트 박스를 가집니다. *SelectButton*이라는 버튼을 누르면 *ColorListBox*에서 현재 선택된 색상 이름을 확인한 다음 폼을 닫습니다. *CancelButton*은 폼을 닫는 버튼입니다.

사용자 정의 생성자가 *Pointer* 인수를 취하는 클래스에 추가되었습니다. 이 *Pointer*가 *ColorForm*을 실행하는 폼이 알고 있는 문자열을 가리킨다고 가정합니다. 이 생성자의 구현은 다음과 같습니다.

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

생성자는 *FColor* private 데이터 멤버에 대한 포인터를 저장하고 문자열을 빈 문자열에 대해 초기화합니다.

참고 위의 사용자 정의 생성자를 사용하려면 폼을 명시적으로 만들어야 합니다. 애플리케이션을 시작할 때 자동적으로 생성되지 않습니다. 자세한 내용은 6-6 페이지의 "메모리에 상주하는 폼의 제어"를 참조하십시오.

애플리케이션의 리스트 박스에서 색상을 선택하고 *SelectButton*을 눌러 선택 내용을 저장한 다음 폼을 닫습니다. *SelectButton*의 *OnClick* 이벤트 핸들러는 다음과 같습니다.

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
```

```

        String(FColor^) := ColorListBox.Items[ItemIndex];
    end;
    Close;
end;

```

이벤트 핸들러는 생성자에게 전달된 포인터가 참조하는 문자열에 선택한 색상 이름을 저장합니다.

*ColorForm*을 효과적으로 사용하려면 호출하는 폼은 생성자에 기존 문자열에 대한 포인터를 전달해야 합니다. 예를 들어, 클릭한 *ResultsForm*의 *UpdateButton*이라는 버튼에 대한 응답으로 *ResultsForm*이라는 폼에 의해 *ColorForm*이 인스턴스화된다고 가정합니다. 이벤트 핸들러는 다음과 같을 것입니다.

```

procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
    MainColor: String;
begin
    GetColor(Addr(MainColor));
    if MainColor <> '' then
        {do something with the MainColor string}
    else
        {do something else because no color was picked}
    end;

procedure GetColor(PColor:Pointer);
begin
    ColorForm := TColorForm.CreateWithColor(PColor, Self);
    ColorForm.ShowModal;
    ColorForm.Free;
end;

```

*UpdateButtonClick*은 *MainColor*라는 문자열을 만듭니다. *MainColor*의 주소는 *ColorForm*을 만들어 *MainColor*에 대한 포인터를 생성자에 대한 인수로 전달하는 *GetColor* 함수에 전달됩니다. *ColorForm*은 닫는 즉시 삭제되지만 선택된 색상 이름은 해당 색상이 선택된 것처럼 *MainColor*에 여전히 보존됩니다. 그렇지 않을 경우, *MainColor*에는 색상을 선택하지 않고 *ColorForm*을 종료했음을 명백히 나타내는 빈 문자열이 들어 있습니다.

이 예제는 하나의 문자열 변수를 사용하여 모달 폼의 정보를 가집니다. 물론 필요할 경우 더 복잡한 객체를 사용할 수도 있습니다. 변경하거나 선택하지 않고(예: 빈 문자열에 *MainColor* 기본값 설정) 모달 폼을 닫았는지 여부를 호출하는 폼에게 항상 알려야 합니다.

컴포넌트와 컴포넌트 그룹의 재사용

Delphi에서는 컴포넌트를 사용하여 수행했던 작업을 저장하고 재사용하는 여러 가지 방법을 제공합니다.

- *컴포넌트 템플릿*은 컴포넌트 그룹을 구성하고 저장하기 위한 간단하고 빠른 방법을 제공합니다. 6-13 페이지의 "컴포넌트 템플릿 생성 및 사용"을 참조하십시오.

- 폼, 데이터 모듈 및 프로젝트를 *레포지토리*에 저장할 수 있습니다. 레포지토리는 재사용 가능한 요소들의 집중식 데이터베이스를 제공하고 폼을 상속하여 변경 내용을 전파할 수 있습니다. 5-20 페이지의 "Object Repository 사용"을 참조하십시오.
- *프레임*을 컴포넌트 팔레트 또는 레포지토리에 저장할 수 있습니다. 프레임은 폼 상속을 사용하고 폼이나 다른 프레임에 포함(embedded)될 수 있습니다. 6-14 페이지의 "프레임 사용"을 참조하십시오.
- *사용자 지정 컴포넌트*를 생성하는 것은 코드를 재사용하기 위한 가장 복잡한 방법이지만 가장 많은 유연성을 제공합니다. 40장 "컴포넌트 생성 개요"를 참조하십시오.

컴포넌트 템플릿 생성 및 사용

하나 이상의 컴포넌트로 구성된 템플릿을 생성할 수 있습니다. 컴포넌트를 폼에 정렬한 후 폼의 속성을 설정하고 컴포넌트 코드를 작성한 다음 *컴포넌트 템플릿*으로 저장합니다. 나중에 컴포넌트 팔레트에서 템플릿을 선택하여 미리 구성된 컴포넌트를 폼 위에 두기만 하면 모든 연결된 속성과 이벤트 처리 코드가 프로젝트에 동시에 추가됩니다.

일단 템플릿을 폼에 두면 컴포넌트를 직접 폼에 둔 것처럼 각 컴포넌트의 위치를 재지정할 수 있고, 속성을 재설정할 수 있으며, 컴포넌트의 이벤트 핸들러를 만들거나 수정할 수 있습니다.

다음과 같은 방법으로 컴포넌트 팔레트를 만듭니다

- 1 폼에 컴포넌트를 두고 정렬합니다. Object Inspector에서 컴포넌트의 속성 및 이벤트를 설정합니다.
- 2 컴포넌트를 선택합니다. 여러 컴포넌트를 선택하는 가장 쉬운 방법은 모든 컴포넌트 위로 마우스를 끄는 것입니다. 회색 핸들이 선택된 각 컴포넌트의 모서리에 나타납니다.
- 3 Component|Create Component Template을 선택합니다.
- 4 Component Name 편집 상자에서 컴포넌트 템플릿의 이름을 지정합니다. 기본적으로 2 단계에서 선택된 첫 번째 컴포넌트의 종류 다음에 "Template"이라는 단어가 오는 것이 좋습니다. 예를 들어, 레이블을 선택한 후 편집 상자를 선택하면 제안된 이름은 "TLabelTemplate"이 됩니다. 이름을 변경할 수 있지만 기존 컴포넌트 이름과 중복되지 않도록 주의하십시오.
- 5 팔레트 페이지 편집 상자에서 템플릿을 넣으려는 컴포넌트 팔레트 페이지를 지정합니다. 존재하지 않는 페이지를 지정하면 템플릿을 저장할 때 새 페이지가 생성됩니다.
- 6 팔레트 아이콘 아래에서 팔레트의 템플릿을 나타내는 비트맵을 선택합니다. 기본적으로 2 단계에서 선택된 첫 번째 컴포넌트의 컴포넌트 종류에서 사용하는 비트맵을 사용합니다. 다른 비트맵을 찾으려면 Change를 클릭합니다. 선택한 비트맵은 24픽셀 x 24픽셀 이하여야 합니다.
- 7 OK를 클릭합니다.

컴포넌트 팔레트에서 템플릿을 제거하려면 Component|Configure Palette를 선택합니다.

프레임 사용

폼과 마찬가지로 프레임 (*TFrame*)도 다른 컴포넌트의 컨테이너입니다. 프레임은 자동 인스턴스화, 컴포넌트 소멸, 컴포넌트 속성 동기화를 위한 동일한 부모 자식 관계에 대해서 폼과 동일한 소유 관계 메커니즘을 사용합니다.

그러나 어떤 면에서 프레임은 폼보다 사용자 지정된 컴포넌트와 더 유사합니다. 프레임은 쉽게 다시 사용하기 위해 컴포넌트 팔레트에 저장할 수 있으며 폼, 다른 프레임 또는 다른 컨테이너 객체 내에 중첩시킬 수 있습니다. 프레임을 만들고 저장한 후 프레임은 계속 유닛으로 기능하고 프레임에 들어 있는 컴포넌트(다른 프레임 포함)의 변경 내용을 상속합니다. 다른 프레임이나 폼에 포함된 프레임은 자신이 파생된 프레임의 변경 내용을 계속 상속합니다.

프레임은 애플리케이션의 여러 위치에 사용되는 컨트롤 그룹을 구성하는 데 유용합니다. 예를 들어, 여러 폼에서 사용되는 비트맵이 있는 경우 해당 비트맵을 프레임에 넣으면 비트맵의 복사본 하나만이 애플리케이션의 리소스에 있습니다. 또한 프레임이 있는 테이블을 편집하기 위한 편집 필드 집합을 기술하고 테이블에 데이터를 입력할 때마다 해당 편집 필드를 사용할 수 있습니다.

프레임 생성

빈 프레임을 만들려면 File|New Frame 을 선택하거나 File|New 를 선택한 다음 Frame 을 더블 클릭합니다. 그런 다음 새 프레임에 컴포넌트 혹은 다른 프레임을 가져다 놓습니다.

필수적이지는 않지만 프레임을 프로젝트 일부로 저장하는 것이 좋습니다. 폼은 없고 프레임만 들어 있는 프로젝트를 만들려면 File|New Application 을 선택하고 새 폼과 유닛을 저장하지 않고 닫은 다음 File|New Frame 을 선택하고 프로젝트를 저장합니다.

참고 프레임을 저장할 때 *Unit1*, *Project1* 등 기본 이름을 사용하지 않아야 합니다. 이러한 기본 이름은 나중에 프레임을 사용하려고 할 때 충돌을 일으킬 수 있기 때문입니다.

디자인 타임에 View|Forms 를 선택하고 프레임을 선택하면 현재 프로젝트에 있는 프레임을 표시할 수 있습니다. 폼과 데이터 모듈에서 마우스 오른쪽 버튼을 클릭하고 View as Form 또는 View as Text 를 선택하면 폼 디자이너와 프레임의 폼 파일 사이를 토글할 수 있습니다.

컴포넌트 팔레트에 프레임 추가

프레임을 컴포넌트 팔레트에 컴포넌트 템플릿으로 추가합니다. 컴포넌트 팔레트에 프레임을 추가하려면 폼 디자이너에서 프레임을 열고(다른 컴포넌트에 포함된 프레임은 이러한 용도로 사용할 수 없음) 프레임을 마우스 오른쪽 버튼으로 클릭한 다음 Add to Palette 를 선택합니다. Component Template Information 대화 상자가 열리면 이름, 팔레트 페이지 및 새 템플릿의 아이콘을 선택합니다.

프레임 사용 및 수정

애플리케이션에서 프레임을 사용하려면 직접적 또는 간접적으로 프레임을 폼에 두어야 합니다. 폼, 다른 프레임 또는 패널 및 스크롤 상자와 같은 다른 컨테이너 객체에 프레임을 직접 추가할 수 있습니다.

폼 디자이너에는 애플리케이션에 프레임을 추가하는 방법이 두 가지 있습니다.

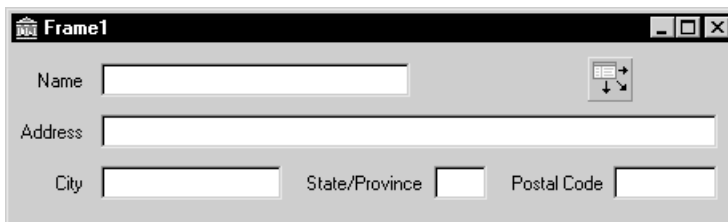
- 컴포넌트 팔레트에서 프레임을 선택하여 폼, 다른 프레임 또는 다른 컨테이너 객체에 가져다 놓습니다. 필요한 경우 폼 디자이너에서 프로젝트에 프레임 유닛 파일을 포함할 권한이 있는지 묻습니다.
- 컴포넌트 팔레트의 Standard 페이지에서 *Frames*를 선택한 후 폼 또는 다른 프레임을 클릭합니다. 프로젝트에 이미 들어 있는 프레임 목록을 보여 주는 대화 상자가 나타나면 프레임을 선택한 다음 OK를 클릭합니다.

폼 또는 다른 컨테이너에 프레임을 가져다 놓으면 Delphi는 선택한 프레임의 자손인 새로운 클래스를 선언합니다. (마찬가지로 프로젝트에 새로운 폼을 추가하면 Delphi는 *TForm*의 자손인 새로운 클래스를 선언합니다.) 이는 원래 (조상) 프레임에 나중에 발생한 변경 내용은 포함된 프레임에 전파되지만 포함된 프레임의 변경 내용은 조상 프레임에 역전파되지 않는다는 것을 의미합니다.

예를 들어, 두 개 이상의 애플리케이션에서 반복적으로 사용하기 위해 데이터 액세스 컴포넌트 및 data-aware 컨트롤을 그룹화하기를 원한다고 가정합니다. 이 작업을 수행하는 한 가지 방법은 컴포넌트를 컴포넌트 템플릿에 모으는 것이지만 템플릿을 사용하기 시작하고 나중에 컨트롤 배열을 바꾸고자 할 경우에는 다시 되돌아가서 해당 템플릿이 위치한 각 프로젝트를 수동으로 변경해야 합니다.

한편 데이터베이스 컴포넌트들을 템플릿 대신 프레임에 놓을 경우에는 원래 프레임에 대한 변경 내용이 프로젝트가 재컴파일될 때 포함된 자손 프레임에 자동으로 전파됩니다. 동시에 원래 프레임이나 다른 포함된 자손 프레임에 영향을 주지 않고 포함된 프레임을 수정할 수 있습니다. 포함된 프레임 수정에 대한 유일한 제한은 컴포넌트를 프레임에 추가할 수 없다는 것입니다.

그림 6.1 data-aware 컨트롤과 데이터 소스 컴포넌트를 갖는 프레임



프레임은 유지를 간편하게 해줄 뿐만 아니라 리소스를 보다 효율적으로 사용할 수 있게 해줍니다. 예를 들어, 애플리케이션에서 비트맵 또는 다른 그래픽을 사용하려면 *TImage* 컨트롤의 *Picture* 속성에 그래픽을 로드할 수 있습니다. 그러나 하나의 애플리케이션에서 동일한 그래픽을 반복적으로 사용하는 경우 폼에 각 *Image* 객체를 두면 폼의 리소스 파일에 그래픽의 또 다른 복사본이 추가됩니다. (이는 *TImage.Picture*를 한 번 설정하고 *Image* 컨트롤을 컴포넌트 템플릿으로 저장하는 경우에도 마찬가지입니다.) 보다 나은

해결책은 *Image* 객체를 프레임에 가져다 놓고 해당 객체에 그래픽을 로드한 후 그래픽이 나타날 프레임을 사용하는 것입니다. 이렇게 하면 크기가 작은 폼 파일이 생기며 원래 프레임의 *Image*를 수정하면 해당되는 모든 위치에서 그래픽을 변경할 수 있는 추가적인 이점이 있습니다.

프레임 공유

다른 개발자와 프레임을 공유하는 두 가지 방법이 있습니다.

- Object Repository에 프레임을 추가합니다.
- 프레임 유닛(.pas) 파일과 폼(.dfm 또는 .xfm) 파일을 배포합니다.

Repository에 프레임을 추가하려면 프레임이 들어 있는 프로젝트를 열고 폼 디자이너에서 마우스 오른쪽 버튼을 클릭한 다음 Add to Repository를 선택합니다. 더 자세한 내용은 5-20 페이지의 "Object Repository 사용"을 참조하십시오.

프레임 유닛 파일과 폼 파일을 다른 개발자에게 보내면 그 개발자는 해당 프레임 유닛 파일과 폼 파일을 열고 컴포넌트 팔레트에 추가할 수 있습니다. 프레임 내에 다른 프레임이 포함되어 있는 경우, 해당 프레임을 프로젝트 일부로 열어야 합니다.

툴바 및 메뉴의 작업 구성

Delphi는 메뉴 및 툴바의 작성, 사용자 정의 및 관리 작업을 간소화해 주는 몇 가지 기능을 제공합니다. 이 기능들을 사용하면 툴바의 버튼을 누르거나 메뉴에서 명령을 선택하거나 또는 아이콘을 가리키고 클릭하는 동작을 통해 애플리케이션 사용자가 초기화할 수 있는 액션의 목록을 구성할 수 있습니다.

대체로 하나 이상의 사용자 인터페이스 요소에 여러 가지 액션이 사용됩니다. 예를 들어 잘라내기, 복사하기 및 붙여넣기 등의 명령은 편집 메뉴와 툴바에 자주 나타납니다. 사용자는 단지 애플리케이션의 여러 UI 요소에서 사용할 수 있도록 액션을 한 번만 추가하면 됩니다.

Windows 플랫폼에서는 액션의 정의 및 그룹화, 다양한 레이아웃의 작성, 그리고 디자인 및 런타임 시 메뉴의 사용자 정의 등을 쉽게 할 수 있는 도구가 제공됩니다. 이런 도구들은 총체적으로 ActionBand 도구라고 하며 사용자가 이 도구들을 사용하여 만드는 메뉴와 툴바는 작업 밴드라고 합니다. 일반적으로는 다음과 같이 ActionBand 사용자 인터페이스를 작성할 수 있습니다.

- 액션 리스트를 작성하여 애플리케이션에서 사용할 수 있는 일련의 작업을 생성합니다(Action Manager, *TActionManager* 사용).
- 애플리케이션에 사용자 인터페이스 요소를 추가합니다(*TActionMainMenuBar* 및 *TActionToolBar*와 같은 ActionBand 컴포넌트 사용).
- Action Manager에서 사용자 인터페이스 요소로 작업을 끌어다 놓습니다.

다음 표는 메뉴 및 툴바 설정과 관련된 용어를 정의한 것입니다.

표 6.1 작업 설정 용어

용어	정의
액션	메뉴 항목을 클릭하는 것과 같은 사용자 행동에 대한 응답. 자주 필요로 하는 많은 표준 작업들이 제공되어 사용자가 애플리케이션에서 있는 그대로 사용할 수 있습니다. 예를 들어 File Open, File SaveAs, File Run 및 File Exit와 같은 파일 기능들이 편집, 포매팅, 검색, 도움말, 대화 상자 및 윈도우 동작 등의 여러 액션과 함께 포함되어 있습니다. 또한 사용자 정의 액션을 프로그래밍하여 액션 리스트와 Action Manager를 사용하여 액션에 액세스할 수 있습니다.
액션 밴드	사용자 정의 가능한 메뉴나 툴바와 관련된 일련의 액션을 위한 컨테이너. 메인 메뉴 및 툴바에 대한 <i>ActionBand</i> 컴포넌트 (<i>TActionMainMenuBar</i> 및 <i>TActionToolBar</i>)는 액션 밴드의 예입니다.
액션 범주	사용자가 액션을 그룹화하고 이를 메뉴나 툴바에 그룹으로 둘 수 있습니다. 표준 작업 범주 중 하나의 예가 Search인데 Find, FindFirst, FindNext 및 Replace 작업이 모두 함께 포함되어 있습니다.
Action 클래스	애플리케이션에서 사용되는 작업을 수행하는 클래스. 모든 표준 작업은 <i>TEditCopy</i> , <i>TEditCut</i> 및 <i>TEditUndo</i> 와 같은 액션 클래스로 정의됩니다. 이런 클래스를 Customize 대화 상자에서 액션 밴드로 끌어다 놓고 사용할 수 있습니다.
액션 클라이언트	이들 대부분은 작업을 초기화하는 공지를 수신하는 메뉴 항목이나 버튼을 나타내는 경우가 많습니다. 클라이언트가 마우스 클릭 같은 사용자 명령을 받으면 연결 액션이 초기화됩니다.
액션 리스트	사용자가 수행한 작업에 대해 애플리케이션에서 취할 수 있는 액션 리스트가 들어 있습니다.
Action Manager	<i>ActionBand</i> 컴포넌트에서 재사용될 수 있는 논리적인 일련의 액션들을 그룹화하고 구성합니다. <i>TActionManager</i> 를 참조하십시오.
메뉴	애플리케이션 사용자가 클릭하여 실행할 수 있는 명령들을 나열합니다. <i>ActionBand</i> 메뉴 클래스 <i>TActionMainMenuBar</i> 를 사용하거나 <i>TMainMenu</i> 또는 <i>TPopupMenu</i> 와 같은 크로스 플랫폼 컴포넌트를 사용하여 메뉴를 작성할 수 있습니다.
대상	액션이 어떤 작업을 수행하는 대상 항목을 나타냅니다. 대상은 보통 메모나 데이터 컨트롤 같은 컨트롤입니다. 모든 액션이 대상을 필요로 하는 것은 아닙니다. 예를 들어, 표준 도움말 액션은 대상을 무시하고 도움말 시스템을 실행합니다.
툴바	클릭했을 때 프로그램이 현재 문서 인쇄와 같은 어떤 액션을 수행하게 하는 보이는 버튼 아이콘의 행을 표시합니다. <i>ActionBand</i> 툴바 컴포넌트 <i>TActionToolBar</i> 를 사용하거나 크로스 플랫폼 컴포넌트 <i>TToolBar</i> 를 사용하여 툴바를 만들 수 있습니다.

크로스 플랫폼 개발의 경우에는 6-23 페이지의 "액션 리스트 사용"을 참조하십시오.

액션(action)의 개념

애플리케이션을 개발함에 따라 다양한 UI 요소에서 사용할 수 있는 일련의 액션들을 생성할 수 있습니다. 사용자는 이를 범주로 구성하여 메뉴에서 액션 집합으로 (Cut, Copy 및 Paste) 또는 동시에 (Tools|Customize) 메뉴에 둘 수 있습니다.

액션은 메뉴 명령이나 툴바 버튼과 같은 하나 이상의 사용자 인터페이스 요소에 대응합니다. 액션에는 다음 두 가지 기능이 있습니다. (1) 컨트롤이 설정 또는 선택되어 있는지와 같은 사용자 인터페이스 요소에 대한 일반적인 속성을 나타내며, (2) 컨트롤이 실행될 때 (예를 들어, 애플리케이션 사용자가 버튼을 클릭하거나 메뉴 항목을 선택할 때) 응답합니다. 사용자는 메뉴, 버튼, 툴바, 컨텍스트 메뉴 등을 통해 애플리케이션에서 사용할 수 있는 액션 리스트를 작성할 수 있습니다.

액션은 다른 컴포넌트와 연결됩니다.

- **Clients:** 하나 이상의 클라이언트가 액션을 사용합니다. 클라이언트는 종종 메뉴 항목 또는 버튼 (예를 들어 *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox*, *TRadioButton* 등)을 나타냅니다. 또한 작업은 *TActionMainMenuBar* 및 *TActionToolBar*와 같은 *ActionBand* 컴포넌트에 존재합니다. 클라이언트가 마우스 클릭과 같은 사용자 명령을 받으면 액션 작업이 초기화됩니다. 일반적으로 클라이언트의 *OnClick* 이벤트는 해당 액션의 *Execute* 이벤트와 연결됩니다.
- **Target:** 액션은 대상(target)에서 작동합니다. 대상은 보통 메모나 데이터 컨트롤과 같은 컨트롤입니다. 컴포넌트 작성자는 모듈 애플리케이션을 보다 많이 만들기 위해 유닛을 디자인하고, 사용하고, 패키징하는 컨트롤 요구에 따른 액션을 만들 수 있습니다. 모든 액션이 대상을 사용하지는 않습니다. 예를 들어, 표준 도움말 액션은 대상을 무시하고 도움말 시스템을 실행합니다.

대상은 컴포넌트일 수도 있습니다. 예를 들어, 데이터 컨트롤은 대상을 연결된 데이터셋으로 변경합니다.

클라이언트는 액션에 영향을 미칩니다. 액션은 클라이언트가 작업을 실행할 때 응답합니다. 액션은 클라이언트에도 영향을 미칩니다. 액션 속성은 클라이언트 속성을 동적으로 업데이트합니다. 예를 들어, 런타임에 액션이 해제되어 있으면(액션의 *Enabled* 속성을 *False*로 설정하여) 해당 액션의 모든 클라이언트가 해제되어 회색으로 나타납니다.

Action Manager 또는 Action List 에디터(액션 리스트 객체 *TActionList*를 더블 클릭하여 표시)를 사용하여 액션을 추가, 삭제 및 재정렬할 수 있습니다. 이러한 액션은 나중에 클라이언트 컨트롤에 연결됩니다.

액션 밴드 설정

액션은 어떤 "레이아웃"(모양이든 위치이든) 정보도 유지하지 않기 때문에 Delphi는 이런 데이터를 저장할 수 있는 액션 밴드를 제공합니다. 작업 밴드는 레이아웃 정보와 일련의 컨트롤을 지정할 수 있는 매카니즘을 제공합니다. 사용자는 툴바나 메뉴와 같은 UI 요소로서 액션을 렌더링할 수 있습니다.

Action Manager(*TActionManager*)를 사용하여 일련의 액션을 구성합니다. 제공된 표준 액션을 사용하거나 자신의 새로운 액션을 생성할 수 있습니다.

그런 다음 액션 밴드를 생성합니다.

- *TActionMainMenuBar*를 사용하여 메인 메뉴를 만듭니다.
- *TActionToolBar*를 사용하여 툴바를 만듭니다.

액션 밴드는 일련의 작업을 보유하고 렌더링하는 컨테이너와 같은 역할을 합니다. 디자인시에 Action Manager 에디터에서 항목을 액션 밴드로 끌어다 놓을 수 있습니다. 런타임 시에는 애플리케이션 사용자도 Action Manager 에디터와 유사한 대화 상자를 사용하여 애플리케이션의 메뉴나 툴바를 사용자 정의할 수 있습니다.

툴바 및 메뉴 생성

참고 이 단원은 Windows 애플리케이션에서 메뉴와 툴바를 만들기 위해 권장되는 방법을 설명합니다. 크로스 플랫폼 개발의 경우에는 *TToolBar* 및 *TMainMenu*와 같은 메뉴 컴포넌트를 사용하여 이를 액션 리스트(*TActionList*)로 구성해야 합니다. 6-23 페이지의 "액션 리스트 설정"을 참조하십시오.

Action Manager를 사용하여 애플리케이션에 포함된 액션에 기초한 툴바 및 메뉴를 자동으로 생성합니다. Action Manager는 표준 작업 및 사용자가 작성한 사용자 정의 액션을 관리합니다. 그런 다음 이들 액션에 따라 UI 요소를 작성하고 액션 밴드를 사용하여 액션 항목을 메뉴 항목이나 툴바의 버튼으로 렌더링합니다.

메뉴, 툴바 및 기타 액션 밴드 작성에 대한 일반적인 절차는 다음과 같은 단계를 포함합니다.

- Action Manager를 폼에 둡니다.
- 액션을 적절한 액션 리스트로 구성하는 Action Manager에 액션을 추가합니다.
- 사용자 인터페이스에 대한 액션 밴드(즉, 메뉴 또는 툴바)를 생성합니다.
- 작업을 애플리케이션 인터페이스에 끌어다 놓습니다.

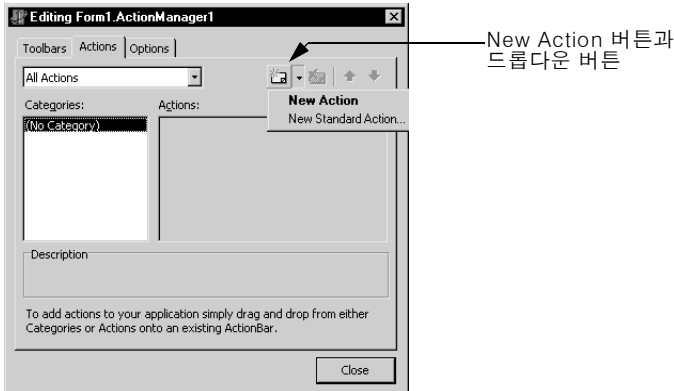
다음 절차는 이 단계들을 더 자세히 설명합니다.

액션 밴드를 사용하여 메뉴 및 툴바를 작성하려면 다음과 같이 합니다.

- 1 컴포넌트 팔레트의 Additional 페이지에서 Action Manager 컴포넌트(*TActionManager*)를 툴바나 메뉴를 작성하려는 폼에 둡니다.
- 2 메뉴나 툴바에 이미지를 두려면 컴포넌트 팔레트의 Win32 페이지에서 ImageList 컴포넌트를 폼에 둡니다. (사용할 이미지를 ImageList에 추가하거나 기존의 이미지를 사용해야 합니다.)
- 3 컴포넌트 팔레트의 Additional 페이지에서 다음 액션 밴드 중 하나 또는 여러 개를 폼에 가져다 놓습니다.
 - *TActionMainMenuBar*(메인 메뉴 디자인용)
 - *TActionToolBar*(툴바 디자인용)
- 4 다음과 같은 방법으로 ImageList를 Action Manager에 연결합니다. Action Manager와 Object Inspector에 포커스를 둔 상태에서 Images 속성의 ImageList 이름을 선택합니다.
- 5 액션을 Action Manager 에디터의 작업 창에 추가합니다.
 - Action Manager를 더블 클릭하여 Action Manager 에디터를 표시합니다.

- New Action 버튼 옆의 드롭다운 화살표를 클릭하고 (Actions 탭의 오른쪽 상단 구석에서 가장 왼쪽에 있는 버튼, 그림 6.2 참조) "New Action..."이나 "New Standard Action..."을 선택합니다. 트리 뷰가 표시됩니다. 하나 이상의 액션이나 액션의 범주를 Action Manager의 액션 창에 추가합니다. Action Manager는 액션을 액션 리스트에 추가합니다.

그림 6.2 Action Manager 에디터



- 6 Action Manager 에디터에서 단일 작업이나 작업 범주를 디자인하고 있는 메뉴나 툴바에 끌어다 놓습니다.

사용자 정의 액션을 추가하려면 New Action 버튼을 클릭하고 액션이 시작되었을 때의 응답 방식을 정의하는 이벤트 핸들러를 작성하여 새로운 *TAction*을 만듭니다. 자세한 내용은 6-24 페이지의 "액션 실행 시 발생하는 이벤트"를 참조하십시오. 액션을 정의한 다음에는 이를 표준 액션에서와 같이 메뉴나 툴바에 끌어다 놓을 수 있습니다.

메뉴, 버튼 및 툴바에 색상, 패턴 또는 그림 추가

Background 및 *BackgroundLayout* 속성을 사용하여 메뉴 항목이나 버튼에 사용할 색상, 패턴 또는 비트맵을 지정할 수 있습니다. 이런 속성들을 사용하여 메뉴의 왼쪽이나 오른쪽에서 실행되는 배너를 설정할 수 있습니다.

액션 클라이언트 객체로부터 하위 항목에 백그라운드와 레이아웃을 할당합니다. 메뉴 항목의 백그라운드를 설정하려면 폼 디자이너에서 항목을 포함하는 메뉴 항목을 누릅니다. 예를 들어, File을 선택하면 File 메뉴에 나타나는 항목의 백그라운드를 변경합니다. Object Inspector의 *Background* 속성에서 색상, 패턴, 비트맵을 할당할 수 있습니다.

BackgroundLayout 속성을 사용하면 요소에 백그라운드를 두는 방식을 설명할 수 있습니다. 색상이나 이미지는 일반적으로 캡션 뒤에 두거나, 항목 영역에 맞도록 늘이거나 또는 한 영역을 덮어쓰도록 작은 사각형으로 바둑판식으로 배열할 수 있습니다.

일반 (blNormal), 늘이기 (blStretch) 또는 바둑판식 배열 (blTile) 백그라운드 항목은 투명한 백그라운드로 렌더링됩니다. 배너를 작성하는 경우 전체 이미지는 항목의 왼쪽 (blLeftBanner)이나 오른쪽 (blRightBanner)에 놓입니다. 이미지는 크기에 맞게 크게 또는 작게 조정되지 않기 때문에 올바른 크기인지 확인해야 합니다.

액션 밴드의 백그라운드를 변경하려면 (즉, 메인 메뉴나 툴바에서) 액션 밴드를 선택하고 액션 밴드 컬렉션 편집기에서 *TActionClientBar*를 선택합니다. *Background* 및 *BackgroundLayout* 속성을 설정하여 전체 툴바나 메뉴에서 사용할 색상이나 패턴 또는 비트맵을 지정할 수 있습니다.

메뉴 및 툴바에 아이콘 추가

메뉴 항목 옆에 아이콘을 추가하거나 툴바의 캡션을 아이콘으로 대체할 수 있습니다. 비트맵이나 아이콘을 *ImageList*를 사용하여 구성합니다.

- 1 컴포넌트 팔레트의 Win32 페이지에서 *ImageList* 컴포넌트를 폼에 둡니다.
- 2 *ImageList*에 사용할 이미지를 추가합니다. *ImageList*를 더블 클릭합니다. Add를 클릭하고 사용할 이미지를 찾고 작업이 끝나면 OK를 클릭합니다. Program Files\Common Files\Borland Shared\Images에는 몇 가지 예제 이미지가 포함되어 있습니다. (버튼 이미지에는 활성 및 비활성 버튼 상태에 따른 두 가지 보기가 포함됩니다.)
- 3 컴포넌트 팔레트의 Additional 페이지에서 다음 액션 밴드 중 하나 또는 여러 개를 폼에 가져다 놓습니다.
 - *TActionMainMenuBar*(메인 메뉴 디자인용)
 - *TActionToolBar*(툴바 디자인용)
- 4 Action Manager에 *ImageList*를 연결합니다. 먼저 Action Manager에 포커스를 둡니다. 그 다음 Object Inspector의 Images 속성에서 *ImageList* 이름을 선택합니다.
- 5 Action Manager 에디터를 사용하여 Action Manager에 액션을 추가합니다. *ImageIndex* 속성을 *ImageList*의 번호로 설정하여 이미지와 액션을 연결할 수 있습니다.
- 6 Action Manager 에디터에서 단일 액션이나 액션 범주를 메뉴나 툴바로 끌어다 놓습니다.
- 7 아이콘만 표시하고 캡션은 표시되지 않는 툴바의 경우: Toolbar 액션 밴드를 선택하고 Items 속성을 더블 클릭합니다. 컬렉션 에디터에서 하나 이상의 항목을 선택하고 Caption 속성을 설정할 수 있습니다.
- 8 이미지가 메뉴나 툴바에 자동으로 나타납니다.

사용자가 사용자 정의할 수 있는 툴바 및 메뉴 생성

Action Manager와 함께 액션 밴드를 사용하여 사용자 정의가 가능한 툴바와 메뉴를 만들 수 있습니다. 런타임 시 애플리케이션 사용자는 Action Manager 에디터와 유사한 사용자 정의 대화 상자를 사용하여 애플리케이션 사용자 인터페이스의 툴바와 메뉴(액션 밴드)를 사용자 정의할 수 있습니다.

애플리케이션 사용자가 애플리케이션에서 액션 밴드를 사용자 정의할 수 있게 하려면 다음과 같이 합니다.

- 1 Action Manager 컴포넌트를 폼에 둡니다.
- 2 액션 밴드 컴포넌트(*TActionMainMenuBar*, *TActionToolBar*)를 둡니다.

3 Action Manager를 더블 클릭하여 Action Manager 에디터를 표시합니다.

- 애플리케이션에서 사용하려는 액션을 추가합니다. 또한 표준 액션 리스트 아래에 표시된 Customize 액션을 추가합니다.
- Dialogs 탭에서 *TCustomizeDlg* 컴포넌트를 폼에 두고 ActionManager 속성을 사용하여 이를 Action Manager와 연결합니다. 사용자가 만든 사용자 정의를 스트리밍할 파일 이름을 지정합니다.
- 액션을 액션 밴드 컴포넌트에 끌어다 놓습니다. (툴바나 메뉴에 Customize 작업을 추가했는지 확인합니다.)

4 애플리케이션을 완료합니다.

애플리케이션을 컴파일하고 실행할 때 사용자는 Action Manager 에디터와 유사한 사용자 정의 대화 상자를 표시하는 Customize 명령에 액세스할 수 있습니다. 사용자는 메뉴 항목을 끌어다 놓을 수 있으며 Action Manager에서 제공한 동일한 액션을 사용하여 투바를 만들 수 있습니다.

작업 밴드의 사용되지 않는 항목 및 범주 숨기기

ActionBands 사용의 이점 중 하나는 사용되지 않는 항목과 범주를 사용자가 숨길 수 있다는 것입니다. 시간이 지남에 따라 액션 밴드는 애플리케이션 사용자를 위해 사용자 정의되어 사용하는 항목만 표시되고 나머지는 보기에서 숨겨집니다. 숨겨진 항목은 사용자가 드롭다운 버튼을 누르면 다시 표시됩니다. 또한 사용자는 사용자 정의 대화 상자에서 사용 기록을 리셋하여 모든 액션 밴드 항목의 표시를 복구할 수 있습니다. 항목 숨기기는 액션 밴드의 기본 작업이지만 이 작업은 개별 항목이나 File 메뉴와 같은 특정 컬렉션의 모든 항목 또는 해당 액션 밴드의 모든 항목의 숨김을 방지하기 위해 변경될 수 있습니다.

Action manager는 연관된 *TActionClientItem*의 *UsageCount* 필드에 저장되어 있는, 사용자가 액션을 호출한 횟수를 추적합니다. Action manager는 세션 번호라는 애플리케이션이 실행된 횟수뿐만 아니라 마지막 액션이 사용된 세션 번호도 기록합니다. *UsageCount* 값은 항목을 숨기기 전에 허용된 최대 세션 번호를 조회하기 위해 사용됩니다. 최대 세션 번호는 현재 세션 번호와 항목의 마지막 사용에 대한 세션 번호의 차이와 비교됩니다. 이 차이가 *PrioritySchedule*에 정의된 번호보다 크면 항목이 숨겨집니다. *PrioritySchedule*의 기본값은 다음 표에 표시되어 있습니다.

표 6.2 Action manager의 *PrioritySchedule* 속성 기본값

액션 밴드 항목이 사용된 세션 번호	마지막 사용 이후에 항목이 숨겨지지 않을 세션 번호
0, 1	3
2	6
3	9
4, 5	12
6-8	17
9-13	23

표 6.2 Action manager의 PrioritySchedule 속성 기본값 (계속)

액션 밴드 항목이 사용된 세션 번호	마지막 사용 이후에 항목이 숨겨지지 않을 세션 번호
14-24	29
25 이상	31

디자인 시에 항목의 숨기기를 해제할 수 있습니다. 특정 액션과 이를 포함하는 모든 컬렉션이 숨겨지지 않게 하려면 액션의 `TActionClientItem` 객체를 찾아서 `UsageCount`를 `-1`로 설정합니다. File 메뉴나 심지어는 메인 메뉴 바와 같은 전체 컬렉션 항목이 숨겨지지 않도록 방지하려면 컬렉션과 연관된 `TActionClients` 객체를 찾아서 `HideUnused` 속성을 `False`로 설정합니다.

액션 리스트 사용

참고 이 단원의 내용은 크로스 플랫폼 개발을 위한 툴바 및 메뉴 설정에 관한 것입니다. Windows 환경용으로 개발하는 데에도 여기서 언급한 방법을 사용할 수 있습니다. 하지만 액션 밴드를 사용하는 것이 더 간단하며 더 많은 옵션을 제공합니다. 액션 리스트는 Action Manager에 의해 자동으로 처리됩니다. 액션 밴드 및 Action Manager 사용에 대한 내용은 6-16 페이지의 "툴바 및 메뉴의 작업 구성"을 참조하십시오.

액션 리스트에는 사용자가 수행한 작업에 대해 애플리케이션에서 취할 수 있는 액션 리스트가 들어 있습니다. 액션 객체를 사용하여 사용자 인터페이스에서 애플리케이션에 의해 실행된 함수를 중앙 집중화합니다. 이렇게 하면 애플리케이션 상태에 따라 액션을 중앙 집중화된 방식으로 설정 및 해제할 수 있을 뿐만 아니라 (예를 들어, 툴바 버튼과 메뉴 항목이 동일한 액션을 수행할 때) 액션을 수행하는 공통적인 코드를 공유할 수 있습니다.

액션 리스트 설정

액션 리스트 설정은 다음과 같은 기본 단계를 이해하고 나면 매우 쉽습니다.

- 액션 리스트를 만듭니다.
- 액션 리스트에 액션을 추가합니다.
- 액션의 속성을 설정합니다.
- 액션을 클라이언트에 첨부합니다.

보다 자세한 단계는 다음과 같습니다.

- 1 폼 또는 데이터 모듈에 `TActionList` 객체를 가져다 놓습니다. (`ActionList`는 컴포넌트 팔레트의 Standard 페이지에 있습니다.)
- 2 `TActionList` 객체를 더블 클릭하여 Action List 에디터를 표시합니다.
 - a 편집기에서 이미 정의된 액션 리스트 중 하나를 사용합니다. 마우스 오른쪽 버튼을 클릭하여 New Standard Action을 선택합니다.

- b 이미 정의된 액션은 Standard Actions Classes 대화 상자에서 Dataset, Edit, Help 및 Window 등의 범주별로 구성됩니다. 액션 리스트에 추가할 모든 표준 액션을 선택한 다음 OK를 클릭합니다.
또는
 - c 사용자 고유의 새 액션을 만듭니다. 마우스 오른쪽 버튼을 클릭하여 New Action을 선택합니다.
- 3 Object Inspector에서 각 액션의 속성을 설정합니다. (설정하는 속성은 액션의 모든 클라이언트에 영향을 미칩니다.)
- Name* 속성은 액션을 식별하고 다른 속성 및 이벤트(*Caption, Checked, Enabled, HelpContext, Hint, ImageIndex, ShortCut, Visible* 및 *Execute*)는 클라이언트 컨트롤의 속성 및 이벤트에 해당합니다. 클라이언트의 해당 속성은 일반적으로 해당 클라이언트 속성과 동일한 이름입니다. 예를 들어, 액션의 *Enabled* 속성은 *TToolButton*의 *Enabled* 속성에 해당합니다. 그러나 액션의 *Checked* 속성은 *TToolButton*의 *Down* 속성에 해당합니다.
- 4 이미 정의된 액션을 사용하는 경우 작업에는 자동으로 발생하는 표준 응답이 들어 있습니다. 사용자 고유의 작업을 만드는 경우 실행 시 작업 응답 방법을 정의하는 이벤트 핸들러를 작성해야 합니다. 자세한 내용은 6-24 페이지의 "액션 실행 시 발생하는 이벤트"를 참조하십시오.
- 5 액션 리스트의 액션을 해당 액션이 필요한 클라이언트에 첨부합니다.
- 폼이나 데이터 모듈에서 버튼이나 메뉴 항목과 같은 컨트롤을 클릭합니다. Object Inspector에서 *Action* 속성은 사용 가능한 액션 리스트를 표시합니다.
 - 원하는 작업을 선택합니다.

TEditDelete 또는 *TDataSetPost*와 같은 표준 액션은 모두 사용자가 예상하는 액션을 수행합니다. 필요한 경우 표준 액션 전체의 사용 방법에 대한 자세한 내용은 온라인 참조 도움말을 참조하십시오. 사용자 고유의 액션을 작성하는 경우에는 작업 실행 시 발생할 일에 대해 자세히 알고 있어야 합니다.

액션 실행 시 발생하는 이벤트

이벤트를 실행하면 일반적인 액션에 대한 일련의 이벤트가 일어납니다. 그런 다음 이벤트가 작업을 처리하지 않으면 또 다른 일련의 이벤트가 일어납니다.

이벤트에 응답

클라이언트 컴포넌트 또는 컨트롤이 선택되어 있거나 작동되어 있으면 응답할 수 있는 액션에 대해 일련의 이벤트가 발생합니다. 예를 들어, 다음 코드는 액션 실행 시 툴바의 가시성을 토글하는 작업의 이벤트 핸들러를 설명합니다.

```
procedure TForm1.Action1Execute(Sender:TObject);
begin
    { Toggle Toolbar1? visibility }
    Toolbar1.Visible := not Toolbar1.Visible;
end;
```

참고 이벤트 및 이벤트 핸들러에 대한 일반적인 내용은 이벤트 및 이벤트 핸들러 작업 3-26 페이지의 "이벤트 및 이벤트 핸들러 작업"을 참조하십시오.

세 가지 다른 레벨인 액션, 액션 리스트, 애플리케이션 중 하나에서 응답하는 이벤트 핸들러를 제공할 수 있습니다. 이 내용은 이미 정의된 표준 액션이 아닌 새로운 일반 액션을 사용할 경우에만 해당합니다. 표준 액션은 이러한 이벤트가 일어날 때 실행되는 기본 제공 동작을 가지고 있으므로 표준 액션을 사용할 경우 이 내용에 대해 염려하지 않아도 됩니다.

이벤트 핸들러가 이벤트에 대해 응답하는 순서는 다음과 같습니다.

- 액션 리스트
- 애플리케이션
- Action

사용자가 클라이언트 컨트롤을 클릭했을 때 Delphi는 액션 리스트나 애플리케이션이 액션을 처리하지 않는 경우 맨 처음에 액션 리스트, 그 다음에 애플리케이션 객체, 그 다음에 해당 액션으로 이어지는 작업의 `Execute` 메소드를 호출합니다. 이에 대해 더 자세히 설명하기 위해서 Delphi는 사용자 액션에 응답하는 방법을 찾을 때 디스패칭 순서를 따릅니다.

1 액션 리스트에 `OnExecute` 이벤트 핸들러를 제공하고 이 이벤트 핸들러가 액션을 처리하면 애플리케이션에서 진행됩니다.

액션 리스트의 이벤트 핸들러는 기본적으로 `False`를 반환하는 `Handled`라는 매개변수를 가집니다. 핸들러가 할당되어 이벤트를 처리하면 `True`가 반환되고 처리 순서가 여기서 끝납니다. 예를 들면, 다음과 같습니다.

```
procedure TForm1.ActionList1Execute(Action: TBasicAction; var Handled: Boolean);
begin
    Handled := True;
end;
```

액션 리스트 이벤트 핸들러에서 `Handled`를 `True`로 설정하지 않으면 처리 과정이 진행됩니다.

2 액션 리스트의 `OnExecute` 이벤트 핸들러를 작성하지 않은 경우 또는 이벤트 핸들러가 액션을 처리하지 않는 경우, 애플리케이션의 `OnActionExecute` 이벤트 핸들러가 실행됩니다. 이벤트 핸들러가 액션을 처리하면 애플리케이션에서 진행됩니다.

애플리케이션의 액션 리스트가 이벤트를 처리하지 못하는 경우 전역 `Application` 객체는 `OnActionExecute` 이벤트를 받습니다. 액션 리스트의 `OnExecute` 이벤트 핸들러와 마찬가지로 `OnActionExecute` 핸들러는 기본적으로 `False`를 반환하는 `Handled`라는 매개변수를 가집니다. 이벤트 핸들러가 할당되어 이벤트를 처리하면 `True`가 반환되고 처리 순서가 여기서 끝납니다. 예를 들면, 다음과 같습니다.

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
    { Prevent execution of all actions in Application }
    Handled := True;
end;
```

3 애플리케이션의 `OnExecute` 이벤트 핸들러가 액션을 처리하지 않으면 액션의 `OnExecute` 이벤트 핸들러가 실행됩니다.

기본 제공 액션을 사용하거나 편집 컨트롤과 같은 특정 대상 클래스에서 작동하는 방법을 인지하는 사용자 고유의 액션을 만들 수 있습니다. 이벤트 핸들러를 어떤 레벨에서도 찾을 수 없으면 애플리케이션은 액션을 실행할 대상을 찾고자 합니다. 애플리케이션이 액션을 실행할 대상을 찾으면 작업이 실행됩니다. 애플리케이션에서 이미 정의된 액션 클래스에 응답할 수 있는 대상을 찾는 방법에 대한 자세한 내용은 다음 단원을 참조하십시오.

액션이 해당 대상을 찾는 방법

6-24 페이지의 "액션 실행 시 발생하는 이벤트"에서는 액션 실행 시 일어나는 실행 주기에 대해 설명합니다. 액션 리스트, 애플리케이션 또는 액션 레벨에서 액션에 응답할 이벤트 핸들러가 할당되어 있지 않으면 애플리케이션은 액션이 적용될 대상 객체를 식별하고자 합니다.

애플리케이션은 다음 순서로 대상을 찾습니다.

- 1 활성화된 컨트롤: 애플리케이션은 먼저 활성화된 컨트롤을 잠재적 대상으로 찾습니다.
- 2 활성화된 폼: 애플리케이션이 활성화된 컨트롤을 찾지 못하는 경우나 활성화된 컨트롤이 대상으로 작용할 수 없는 경우 화면의 *ActiveForm*을 찾습니다.
- 3 폼의 컨트롤: 활성화된 폼이 적절한 대상이 아닌 경우 애플리케이션은 대상의 활성화된 폼에서 다른 컨트롤을 찾습니다.

대상이 없으면 이벤트 실행 시 아무 일도 일어나지 않습니다.

일부 컨트롤은 검색을 확장하여 대상에서 연결된 컴포넌트로 연장할 수 있습니다. 예를 들어, *data-aware* 컨트롤은 연결된 데이터셋 컴포넌트로 연장됩니다. 또한 일부 이미 정의된 액션에서는 File Open 대화 상자와 같은 대상을 사용하지 않습니다.

액션 업데이트

애플리케이션이 유휴 상태일 때 *OnUpdate* 이벤트가 표시되는 컨트롤 또는 메뉴 항목에 연결되어 있는 모든 액션에 대해 발생합니다. 이를 통해 애플리케이션에서 설정, 설정 해제, 선택 또는 선택 해제 등을 위한 중앙 집중화된 코드를 실행할 수 있습니다. 예를 들어, 다음 코드는 툴바가 표시될 때 "선택"되어 있는 액션에 대한 *OnUpdate* 이벤트 핸들러를 설명합니다.

```

procedure TForm1.Action1Update(Sender:TObject);
begin
    { Indicate whether ToolBar1 is currently visible }
    (Sender as TAction).Checked := ToolBar1.Visible;
end;

```

경고 시간을 많이 소요하는 코드를 *OnUpdate* 이벤트 핸들러에 추가하지 마십시오. 이것은 애플리케이션이 유휴 상태일 때마다 실행됩니다. 이벤트 핸들러가 시간을 너무 많이 소요하면 전체 애플리케이션의 성능에 부정적인 영향을 끼칩니다.

이미 정의된 액션 클래스

Action Manager 위에서 마우스 오른쪽 단추를 클릭하여 New Standard Action을 선택하면 이미 정의된 액션을 애플리케이션에 추가할 수 있습니다. New Standard Action Classes 대화 상자가 표시되면서 이미 정의된 액션 클래스 및 연관된 표준 액션들이 나열됩니다. 이들은 Delphi에 포함된 액션들이면서 또 자동으로 액션을 수행하는 객체들입니다. 이미 정의된 액션들은 다음 클래스 안에 구성되어 있습니다.

표 6.3 Action 클래스

클래스	설명
편집	표준 편집 액션: 편집 컨트롤 대상에서 사용합니다. <i>TEditAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 메소드를 오버라이드하여 클립보드를 통해 복사, 잘라내기 및 붙여넣기 작업을 실행하는 자손 클래스의 기본 클래스입니다.
서식	표준 포매팅 액션: 굵게, 기울임, 밑줄, 취소선과 같은 텍스트 서식 옵션을 적용하기 위한 리치 텍스트로 사용합니다. <i>TRichEditAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 및 <i>UpdateTarget</i> 메소드를 오버라이드하여 대상의 서식을 구현하는 자손 클래스의 기본 클래스입니다.
도움말	표준 도움말 액션: 모든 대상에서 사용합니다. <i>THelpAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 메소드를 오버라이드하여 도움말 시스템에 명령을 전달하는 자손 클래스의 기본 클래스입니다.
창	표준 창 액션: MDI 애플리케이션의 대상인 폼에서 사용합니다. <i>TWindowAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 메소드를 오버라이드하여 MDI 자식 폼의 배열, 계단식 배열, 닫기, 바둑판식 배열 및 최소화 작업을 수행하는 자손 클래스의 기본 클래스입니다.
파일	파일 액션: File Open, File Run 또는 File Exit와 같은 파일 작업에서 사용합니다.
검색	검색 액션: 검색 옵션에서 사용합니다. <i>TSearchAction</i> 은 사용자가 편집 컨트롤 검색을 위해 검색 문자열을 입력할 수 있는 모달리스 대화 상자를 표시하는 액션의 일반 동작을 구현합니다.
탭	탭 제어 액션: 마법사의 Prev 버튼과 Next 버튼과 같은 탭 컨트롤의 탭 사이를 이동하는 데 사용합니다.
목록	리스트 컨트롤 액션: 리스트 뷰의 항목 관리를 위해 사용합니다.
대화 상자	대화 상자 액션: 대화 상자 컴포넌트에서 사용합니다. <i>TDialogAction</i> 은 실행 시 대화 상자를 표시하는 액션의 일반 동작을 구현합니다. 각 자손 클래스는 특정 대화 상자를 나타냅니다.
인터넷	인터넷 액션: 인터넷 브라우징, 다운로드 및 전자 우편 발송 등의 기능에 사용합니다.
데이터셋	데이터셋 액션: 데이터셋 컴포넌트 대상에서 사용합니다. <i>TDataSetAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 및 <i>UpdateTarget</i> 메소드를 오버라이드하여 대상의 탐색과 편집 작업을 구현하는 자손 클래스의 기본 클래스입니다. <i>TDataSetAction</i> 은 액션이 해당 데이터셋에서 수행되도록 하는 <i>DataSource</i> 속성을 제공합니다. <i>DataSource</i> 가 nil인 경우 현재 포커스가 맞춰진 data-aware 컨트롤을 사용합니다.
툴	툴: 액션 밴드에 대한 사용자 정의 대화 상자를 자동 표시하기 위한 <i>TCustomizeActionBars</i> 와 같은 추가 도구입니다.

모든 작업 객체는 온라인 참조 도움말의 액션 객체 이름에 기술되어 있습니다. 액션 객체가 작동하는 방법에 대한 자세한 내용은 도움말을 참조하십시오.

액션 컴포넌트 작성

사용자가 직접 이미 정의된 액션 클래스를 만들 수도 있습니다. 사용자가 직접 액션 클래스를 작성할 때 객체의 특정 대상 클래스에서 실행하기 위한 기능을 기본 제공할 수 있습니다. 그런 다음 이미 정의된 액션 클래스를 사용하는 것과 동일한 방법으로 사용자 지정 액션을 사용할 수 있습니다. 즉, 액션이 인식하여 대상 클래스에 자신을 적용할 수 있고 사용자가 클라이언트 컨트롤에 액션을 할당할 수 있을 때 해당 액션은 이벤트 핸들러를 작성할 필요 없이 대상에서 동작합니다.

컴포넌트 작성자는 QStdActns 및 DBActns 유닛의 클래스를 그 고유의 액션 클래스를 파생하여 특정 컨트롤이나 컴포넌트에 관한 동작을 구현하기 위한 예제로 사용할 수 있습니다. 이러한 특수화된 액션 (*TEditAction*, *TWindowAction* 및 기타 등등)의 기본 클래스는 일반적으로 *HandlesTarget*, *UpdateTarget* 및 다른 메소드를 오버라이드하여 객체의 특정 클래스에 대한 액션의 대상을 제한합니다. 자손 클래스는 일반적으로 *ExecuteTarget*을 오버라이드하여 특화된 작업을 수행합니다. 이 메소드들은 다음과 같이 설명되어 있습니다.

메소드	설명
<i>HandlesTarget</i>	사용자가 작업에 연결된 툴바 또는 메뉴 항목과 같은 객체를 호출할 때 자동으로 호출됩니다. <i>HandlesTarget</i> 메소드를 사용하여 "대상(target)"으로서의 <i>Target</i> 매개변수에 의해 지정된 객체에서 이번에 실행하는 데 적절한지 여부를 작업 객체에 나타낼 수 있습니다. 자세한 내용은 6-26 페이지의 "액션이 해당 대상을 찾는 방법"을 참조하십시오.
<i>UpdateTarget</i>	액션이 현재 상황에 따라 자체적으로 업데이트할 수 있도록 애플리케이션이 유휴 상태일 때 자동으로 호출됩니다. <i>OnUpdateAction</i> 대신 사용합니다. 자세한 내용은 6-26 페이지의 "액션 업데이트"를 참조하십시오.
<i>ExecuteTarget</i>	<i>OnExecute</i> 대신에 사용자 액션에 대한 응답으로 액션이 실행될 때 자동으로 호출됩니다(예를 들어, 사용자가 현재 액션에 연결된 메뉴 항목을 선택하거나 툴 버튼을 누를 때). 자세한 내용은 6-24 페이지의 "액션 실행 시 발생하는 이벤트"를 참조하십시오.

액션 등록

사용자 고유의 액션을 작성할 때 액션을 등록하여 Action List 에디터에 액션이 나타나도록 할 수 있습니다. Actnlist 유닛의 전역 루틴을 사용하면 액션을 등록 및 등록 해제할 수 있습니다.

```
procedure RegisterActions(const CategoryName:string; const AClasses:array of TBasicActionClass; Resource:TComponentClass);
```

```
procedure UnRegisterActions(const AClasses:array of TbasicActionClass);
```

*RegisterActions*를 호출하면 등록한 액션이 애플리케이션에서 사용될 수 있도록 Action List 에디터에 나타납니다. 액션을 구성하기 위한 범주 이름뿐만 아니라 기본 속성 값을 제공할 수 있게 해주는 *Resource* 매개변수도 제공할 수 있습니다.

예를 들어, 다음 코드는 IDE에 표준 액션을 등록합니다.

```
{ Standard action registration }
RegisterActions('', [TAction], nil);
RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);
RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

*UnRegisterActions*를 호출하면 해당 액션이 Action List 에디터에 더 이상 나타나지 않습니다.

메뉴 생성 및 관리

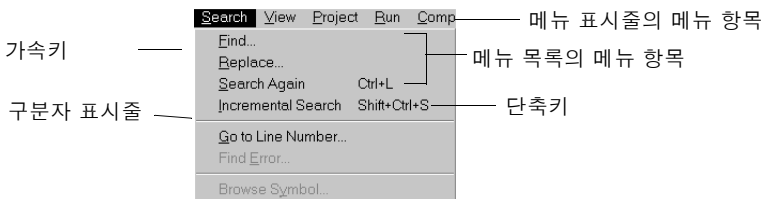
메뉴를 통해 논리적으로 그룹화된 명령을 쉽게 실행할 수 있습니다. 메뉴 디자이너를 사용하면 폼에(이미 디자인된 또는 사용자 지정에 맞게) 메뉴를 쉽게 추가할 수 있습니다. 폼에 메뉴 컴포넌트를 추가하고 메뉴 디자이너를 연 다음 메뉴 디자이너 창에 메뉴 항목을 직접 입력합니다. 메뉴 항목을 추가하거나 삭제할 수 있고 메뉴 항목을 드래그 앤 드롭하여 디자인 타임에 메뉴 항목을 재정렬할 수 있습니다.

런타임에 나타날 모양과 동일하게 디자인이 폼에서 즉시 보이므로 결과를 보기 위해 프로그램을 실행할 필요가 없습니다. 또한 코드에서 사용자에게 자세한 내용이나 옵션을 제공하기 위해 런타임에 메뉴를 변경할 수 있습니다.

이 장에서는 메뉴 디자이너를 사용하여 메뉴 표시줄과 팝업(로컬) 메뉴를 디자인하는 방법에 대해 설명합니다. 또한 디자인 타임 및 런타임에 메뉴를 사용하는 다음 방법을 설명합니다.

- 메뉴 디자이너 열기
- 메뉴 작성
- Object Inspector에서 메뉴 항목 편집
- 메뉴 디자이너 컨텍스트 메뉴 사용
- 메뉴 템플릿 사용
- 메뉴를 템플릿으로 저장
- 메뉴 항목에 이미지 추가

그림 6.3 메뉴 용어

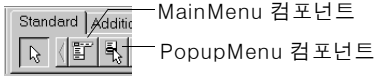


메뉴 항목을 선택했을 때 실행되는 코드에 메뉴 항목을 연결하는 방법에 대한 정보는 3-28 페이지의 "메뉴 이벤트를 이벤트 핸들러에 연결"을 참조하십시오.

메뉴 디자이너 열기

메뉴 디자이너를 사용하여 애플리케이션의 메뉴를 디자인합니다. 메뉴 디자이너를 사용하여 시작하기 전에 먼저 MainMenu 또는 PopupMenu 컴포넌트를 폼에 추가합니다. 두 메뉴 컴포넌트는 모두 컴포넌트 팔레트의 Standard 페이지에 있습니다.

그림 6.4 MainMenu 컴포넌트 및 PopupMenu 컴포넌트



MainMenu 컴포넌트는 폼의 제목 표시줄에 붙여지는 메뉴를 만듭니다. PopupMenu 컴포넌트는 폼에서 마우스 오른쪽 버튼을 클릭하면 나타나는 메뉴를 만듭니다. 팝업 메뉴에는 메뉴 표시줄이 없습니다.

메뉴 디자이너를 열려면 폼에서 메뉴 컴포넌트를 선택한 다음,

- 메뉴 컴포넌트를 더블 클릭합니다.

또는

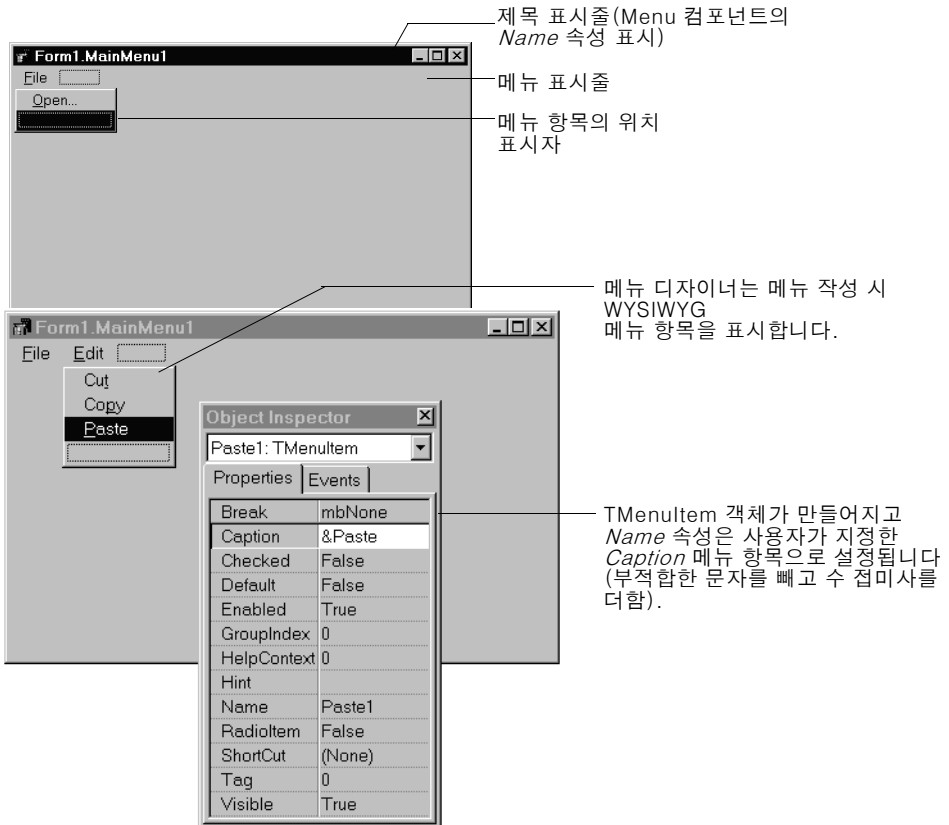
- Object Inspector의 Properties 페이지에서 *Items* 속성을 선택하고 값 열의 [Menu]를 더블 클릭하거나 생략(...) 버튼을 클릭합니다.

메뉴 디자이너는 디자이너에서 선택한 첫 번째 (비어 있는) 메뉴 항목과 Object Inspector에서 선택한 *Caption* 속성과 함께 표시됩니다.

그림 6.5 팝업 메뉴의 메뉴 디자이너



그림 6.6 메인 메뉴의 메뉴 디자이너



메뉴 작성

애플리케이션에 포함할 모든 메뉴들에 대해서 메뉴 컴포넌트를 폼이나 폼들에 추가합니다. 완전히 처음부터 각 메뉴 구조를 만들거나 이미 디자인된 메뉴 템플릿 중 하나에서 시작할 수 있습니다.

이 단원에서는 디자인 타임에 메뉴를 만들기 위한 기본 사항을 설명합니다. 메뉴 템플릿에 대한 자세한 내용은 6-39 페이지의 "메뉴 템플릿 사용"을 참조하십시오.

메뉴 이름 지정

모든 컴포넌트에서 폼에 메뉴 컴포넌트를 추가할 때 Delphi에서는 메뉴 항목에 *MainMenu1* 과 같은 기본 이름을 제공합니다. 사용자는 오브젝트 파스칼 이름 지정 규칙에 따르는 좀더 의미 있는 메뉴 이름을 지정할 수도 있습니다.

Delphi에서 폼의 타입 선언에 메뉴 이름을 추가하면 메뉴 이름이 컴포넌트 목록에 나타납니다.

메뉴 항목 이름 지정

메뉴 컴포넌트와 대조적으로 폼에 메뉴 항목을 추가할 때는 명시적으로 메뉴 항목의 이름을 지정해야 합니다. 다음 두 가지 방법 중 하나로 메뉴 항목 이름을 지정할 수 있습니다.

- *Name* 속성에 값을 직접 입력합니다.
- *Caption* 속성에 값을 먼저 입력하고 Delphi가 캡션의 *Name* 속성을 파생합니다.

예를 들어, 메뉴 항목에 *File*의 *Caption* 속성 값을 제공하는 경우 Delphi는 해당 메뉴 항목에 *File1*의 *Name* 속성을 할당합니다. *Caption* 속성을 입력하기 전에 *Name* 속성을 채우면 Delphi는 값이 입력될 때까지 *Caption* 속성을 비워 둡니다.

참고

Caption 속성에 오브젝트 파스칼 식별자에 유효하지 않은 문자를 입력하면 Delphi는 그에 맞게 *Name* 속성을 수정합니다. 예를 들어, 사용자가 캡션에서 숫자로 시작하는 것을 원하면 Delphi는 *Name* 속성을 파생한 문자에서 숫자가 선행하도록 합니다.

다음 표에서는 표시된 모든 메뉴 항목이 동일한 메뉴 표시줄에 나타난다는 가정 하에 이에 관한 예제를 보여 줍니다.

표 6.4 예제 캡션 및 파생된 이름

컴포넌트 캡션	파생된 이름	설명
&File	File1	앰퍼샌드 (&) 제거
&File(두 번째 나타남)	File2	중복된 항목을 번호로 지정
1234	N12341	선행 문자와 번호 추가
1234(두 번째 나타남)	N12342	파생된 이름을 명확하게 나타내는 숫자 추가

표 6.4 예제 캡션 및 파생된 이름 (계속)

컴포넌트 캡션	파생된 이름	설명
\$@@@#	N1	모든 비정규 문자를 제거하고 선행 문자 및 번호 추가
- (하이픈)	N2	비정규 문자가 있는 캡션이 두 번째 나타날 때 순차적으로 정렬

메뉴 컴포넌트에서 Delphi가 폼 타입 선언에 메뉴 항목 이름을 추가하면 컴포넌트 목록에 추가된 메뉴 항목 이름이 나타납니다.

메뉴 항목 추가, 삽입 및 삭제

다음 절차는 메뉴 구조를 만드는 것과 관련된 기본 작업 수행 방법에 대해 설명합니다. 각 절차는 메뉴 디자이너 창이 열려 있다고 가정합니다.

다음과 같은 방법으로 디자인 타임에 메뉴 항목을 추가합니다.

- 1 메뉴 항목을 만들 위치를 선택합니다.

메뉴 디자이너를 여는 경우 메뉴 표시줄의 첫 번째 위치가 이미 선택되어 있습니다.

- 2 캡션을 입력하기 시작합니다. 또는 Object Inspector에 커서를 두고 값을 입력하여 먼저 *Name* 속성을 입력합니다. 이러한 경우 *Caption* 속성을 다시 선택하고 값을 입력해야 합니다.

- 3 *Enter* 키를 누릅니다.

그러면 메뉴 항목의 다음 위치 표시자가 선택됩니다.

먼저 *Caption* 속성을 입력한 경우 화살표 키를 사용하여 방금 입력한 메뉴 항목으로 돌아갑니다. Delphi는 입력한 캡션 값에 기반하여 *Name* 속성을 채웁니다. (6-32 페이지의 "메뉴 항목 이름 지정" 참조)

- 4 만들려는 새 항목마다 *Name* 및 *Caption* 속성에 값을 계속 입력하거나 *Esc*를 눌러 메뉴 표시줄로 돌아갑니다.

화살표 키를 사용하여 메뉴 표시줄에서 메뉴로 이동한 다음 목록의 항목과 항목 사이를 이동하고 *Enter*를 눌러 작업을 완료합니다. 메뉴 표시줄로 돌아가려면 *Esc*를 누릅니다.

다음과 같은 방법으로 비어 있는 새 메뉴 항목을 삽입합니다.

- 1 메뉴 항목에 커서를 둡니다.

- 2 *Ins*를 누릅니다.

메뉴 표시줄의 선택한 항목 왼쪽 및 메뉴 목록의 선택한 항목 위에 메뉴 항목이 삽입됩니다.

다음과 같은 방법으로 메뉴 항목 또는 명령을 삭제합니다.

- 1 삭제할 메뉴 항목에 커서를 둡니다.

- 2 *Del*을 누릅니다.

참고 메뉴 목록에서 마지막으로 입력된 항목 아래 또는 메뉴 표시줄의 마지막 항목 옆에 나타나는 기본 위치 표시자를 삭제할 수 없습니다. 이 위치 표시자는 런타임에 메뉴에 나타나지 않습니다.

구분자 표시줄 추가

구분자 표시줄은 메뉴 항목 사이에 선을 삽입합니다. 구분자 표시줄을 사용하여 메뉴 목록 내에서 그룹화를 표시하거나 메뉴 목록에서 시각적으로 구분할 수 있습니다.

메뉴 항목을 구분자 표시줄로 만들려면 캡션에 하이픈(-)을 입력합니다.

가속키 및 단축키 지정

가속키를 사용하여 *Alt* 해당 문자를 눌러 키보드에서 메뉴 명령에 액세스할 수 있으며 코드에서 선행 앰퍼샌드에 의해 표시됩니다. 앰퍼샌드 다음에 오는 문자는 메뉴에서 밑줄이 그어져 표시됩니다.

Delphi는 중복된 가속키를 자동으로 확인하고 런타임에 조정합니다. 이렇게 함으로써 런타임에 동적으로 작성한 메뉴에는 중복된 가속키가 들어 있지 않고 모든 메뉴 항목에는 하나의 가속키가 있습니다. 메뉴 항목의 *AutoHotkeys* 속성을 *maManual*로 설정하여 이 자동 확인을 끌 수 있습니다.

다음과 같은 방법으로 가속키를 지정합니다.

- 해당 문자 앞에 앰퍼샌드를 추가합니다.

예를 들어, *S*를 갖는 *Save* 메뉴 명령을 가속키로 추가하려면 *&Save*를 입력합니다.

단축키를 사용하여 메뉴를 직접 사용하지 않고 단축키 조합으로 입력하여 작업을 수행할 수 있습니다.

다음과 같은 방법으로 단축키를 지정합니다.

- *Object Inspector*를 사용하여 *ShortCut* 속성에 값을 입력하거나 드롭다운 목록에서 키 조합을 선택합니다.

이 목록은 사용자가 입력할 수 있는 키 조합들입니다.

추가한 단축키는 메뉴 항목 캡션 옆에 나타납니다.

주의 가속키와 달리 단축키는 중복에 대해 자동으로 검사하지 않습니다. 사용자 자신이 고유성을 확인해야 합니다.

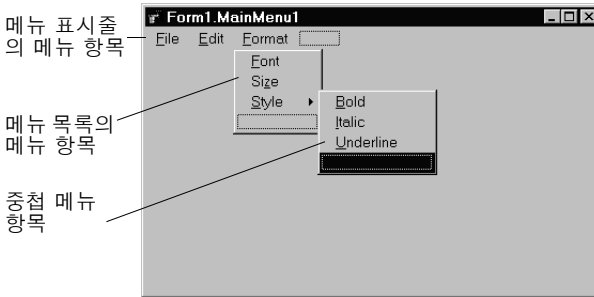
하위 메뉴 생성

많은 애플리케이션에는 메뉴 항목 옆에 드롭다운 목록이 있어 해당 메뉴 항목과 관련된 추가 명령을 제공합니다. 이러한 목록은 메뉴 항목의 오른쪽에 화살표 키로 표시됩니다. Delphi는 메뉴에 만드는 다양한 레벨의 하위 메뉴를 지원합니다.

이런 방법으로 메뉴 구조를 구성하면 화면의 세로 공간을 절약할 수 있습니다. 그러나 최적화된 디자인을 위해서 인터페이스 디자인에서 메뉴 레벨을 둘이나 세 개 이상 사용

하지 않는 것이 좋습니다. 팝업 메뉴에서 하위 메뉴가 있는 경우 하위 메뉴를 하나만 사용하기를 원할 수 있습니다.

그림 6.7 중첩 메뉴 구조



다음과 같은 방법으로 하위 메뉴를 만듭니다.

- 1 하위 메뉴를 만들 메뉴 항목을 선택합니다.
- 2 **Ctrl**+**→**를 눌러서 첫 위치 표시자를 만들거나 오른쪽 단추를 클릭하고 Create Submenu를 선택합니다.
- 3 하위 메뉴 항목의 이름을 입력하거나 기존 메뉴 항목을 이 위치 표시자로 끌어 놓습니다.
- 4 **Enter**를 누르거나 **?**를 눌러서 다음 위치 표시자를 만듭니다.
- 5 하위 메뉴에서 만들 각 항목에 대해 3단계와 4단계를 반복합니다.
- 6 **Esc**를 눌러 이전 메뉴 레벨로 돌아갑니다.

기존 메뉴를 밑으로 내려 하위 메뉴 생성

목록의 메뉴 항목 사이에 메뉴 표시줄 또는 메뉴 템플릿의 메뉴 항목을 삽입하여 하위 메뉴를 만들 수 있습니다. 기존 메뉴 구조로 메뉴를 이동하면 메뉴와 연결된 항목 모두가 해당 메뉴와 함께 이동하여 완전히 그대로 유지된 하위 메뉴를 만듭니다. 이는 하위 메뉴에도 적용됩니다. 기존 하위 메뉴로 메뉴 항목을 이동하면 중첩 레벨이 하나 더 만들어 집니다.

메뉴 항목 이동

디자인 타임에 드래그 앤 드롭으로 메뉴 항목을 이동할 수 있습니다. 메뉴 표시줄에서 메뉴 목록의 다른 위치 또는 완전히 다른 메뉴로 메뉴 항목을 이동할 수 있습니다.

메뉴 이동에 대한 유일한 예외는 계층 구조와 관련된 것입니다. 메뉴 표시줄에서 메뉴 항목을 자신의 메뉴로 내릴 수 없고 메뉴 항목을 자신의 하위 메뉴로 이동할 수 없습니다. 그러나 항목의 원래 위치와 상관 없이 메뉴 항목을 다른 메뉴로 이동할 수 있습니다.

메뉴 항목을 끌어 놓는 동안 커서 모양이 바뀌면서 새 위치에 메뉴 항목을 놓을 수 있는지 여부를 나타냅니다. 메뉴 항목을 이동하면 아래의 모든 항목도 함께 이동합니다.

다음과 같은 방법으로 메뉴 표시줄에서 메뉴 항목을 이동합니다.

- 1 드래그 커서의 화살표 끝이 새 위치를 가리킬 때까지 메뉴 표시줄에서 메뉴 항목을 끕니다.
- 2 마우스 버튼을 놓아 메뉴 항목을 새 위치에 놓습니다.
다음과 같은 방법으로 메뉴 목록으로 메뉴 항목을 이동합니다.
- 1 드래그 커서의 화살표 끝이 새 메뉴를 가리킬 때까지 메뉴 표시줄에서 메뉴 항목을 끕니다.
이렇게 하면 메뉴가 열리므로 메뉴 항목을 새 위치로 끌어 놓을 수 있습니다.
- 2 메뉴 항목을 목록으로 끌어 놓은 다음 마우스 버튼을 놓아 메뉴 항목을 새 위치에 놓습니다.

메뉴 항목에 이미지 추가

이미지는 툴바 이미지와 마찬가지로 메뉴 항목 작업에 glyph 및 이미지를 연결함으로써 메뉴를 찾는 데 도움을 줍니다. 메뉴 항목에 단일 비트맵을 추가하거나 애플리케이션의 이미지를 이미지 목록으로 구성한 다음 이미지 목록의 메뉴에 추가할 수 있습니다. 애플리케이션에서 동일한 크기의 여러 비트맵을 사용하는 경우 이러한 비트맵을 이미지 목록에 넣는 것이 유용합니다.

메뉴 또는 메뉴 항목에 단일 이미지를 추가하려면 이미지의 *Bitmap* 속성이 비트맵 이름을 참조하도록 설정하여 메뉴 또는 메뉴 항목에서 사용합니다.

다음과 같은 방법으로 이미지 목록을 사용하여 메뉴에 이미지를 추가합니다.

- 1 *TMainMenu* 또는 *TPopupMenu* 객체를 폼에 가져다 놓습니다.
- 2 *TImageList* 객체를 폼에 가져다 놓습니다.
- 3 *TImageList* 객체를 더블 클릭하여 *ImageList* 편집기를 엽니다.
- 4 Add를 클릭하여 메뉴에서 사용할 비트맵 또는 비트맵 그룹을 선택합니다. OK를 클릭합니다.
- 5 *TMainMenu* 또는 *TPopupMenu* 객체의 *Images* 속성을 방금 만든 *ImageList*로 설정합니다.
- 6 이전에 설명한 대로 메뉴 항목과 하위 메뉴 항목을 만듭니다.
- 7 Object Inspector에서 이미지를 갖고자 하는 메뉴 항목을 선택하고 *ImageIndex* 속성을 *ImageList*의 해당 이미지 번호로 설정합니다 (*ImageIndex*의 기본값은 -1이며 이 값은 이미지를 나타내지 않음).

참고 메뉴에서 적절하게 표시되도록 16 x 16 픽셀의 이미지를 사용합니다. 다른 크기의 메뉴 이미지를 사용할 수는 있지만 16 x 16 픽셀보다 크거나 작은 이미지를 사용하면 정렬 및 일관성 문제가 발생할 수 있습니다.

메뉴 보기

먼저 프로그램 코드를 실행하지 않고도 디자인 시에 폼에서 메뉴를 볼 수 있습니다. (팝업 메뉴 컴포넌트는 디자인 타임에 폼에 표시되지만 팝업 메뉴 자체는 표시되지 않습니다. 디자인 타임에 팝업 메뉴를 보려면 메뉴 디자이너를 사용합니다.)

다음과 같은 방법으로 메뉴를 봅니다.

- 1 폼이 보이는 경우 폼을 클릭하거나 View 메뉴에서 보고자 하는 메뉴의 폼을 선택합니다.
- 2 폼에 메뉴가 하나 이상 있으면 폼의 *Menu* 속성 드롭다운 목록에서 보고자 하는 메뉴를 선택합니다.

프로그램을 실행할 때와 똑같이 폼에 메뉴가 나타납니다.

Object Inspector에서 메뉴 항목 편집

이 단원에서는 메뉴 항목의 여러 속성을 설정하는 방법에 대해 설명합니다(예를 들면, 메뉴 디자이너를 사용하여 *Name* 및 *Caption* 속성 설정).

또한 이 단원에서는 폼에서 선택한 컴포넌트의 속성을 설정하는 것처럼 Object Inspector에서 직접 *ShortCut* 속성과 같은 메뉴 항목 속성을 설정하는 방법에 대해 설명합니다.

메뉴 디자이너를 사용하여 메뉴 항목을 편집할 때 메뉴 항목의 속성은 Object Inspector에 표시됩니다. Object Inspector로 포커스를 전환한 다음 메뉴 항목 속성을 계속 편집할 수 있습니다. 또는 메뉴 디자이너를 열지 않고 Object Inspector의 컴포넌트 목록에서 메뉴 항목을 선택한 다음 메뉴 항목의 속성을 편집할 수 있습니다.

다음과 같은 방법으로 메뉴 디자이너 창을 닫고 메뉴 항목을 계속 편집합니다.

- 1 Object Inspector의 속성 페이지를 눌러 메뉴 디자이너 창에서 Object Inspector로 포커스를 전환합니다.
- 2 일반적인 방법으로 메뉴 디자이너를 닫습니다.

선택한 메뉴 항목의 속성을 계속 편집할 수 있는 Object Inspector에 포커스가 그대로 유지됩니다. 다른 메뉴 항목을 편집하려면 컴포넌트 목록에서 메뉴 항목을 선택합니다.

메뉴 디자이너 컨텍스트 메뉴 사용

메뉴 디자이너 컨텍스트 메뉴를 통해 가장 많이 사용하는 메뉴 디자이너 명령 및 메뉴 템플릿 옵션에 빠르게 액세스할 수 있습니다. (메뉴 템플릿에 대한 자세한 내용은 6-39 페이지의 "메뉴 템플릿 사용"을 참조하십시오.)

컨텍스트 메뉴를 표시하려면 메뉴 디자이너 창을 마우스 오른쪽 버튼으로 클릭하거나 커서가 메뉴 디자이너 창에 있을 때 *Alt+F10*을 누릅니다.

컨텍스트 메뉴의 명령

다음 표는 메뉴 디자이너 컨텍스트 메뉴의 명령에 대해 요약하여 설명한 것입니다.

표 6.5 메뉴 디자이너 컨텍스트 메뉴 명령

메뉴 명령	액션
Insert	커서의 위쪽 또는 왼쪽에 위치 표시자를 삽입합니다.
Delete	선택한 메뉴 항목 및 해당 하위 항목을 모두 삭제합니다.
Create Submenu	중첩 레벨에 위치 표시자를 만들고 선택한 메뉴 항목 오른쪽에 화살표를 추가합니다.
Select Menu	현재 폼에서 메뉴 목록이 열립니다. 메뉴 이름을 더블 클릭하면 메뉴 디자이너 창이 열립니다.
Save As Template	다음에 재사용할 메뉴를 저장할 수 있는 Save Template 대화 상자가 열립니다.
Insert From Template	재사용할 템플릿을 선택할 수 있는 Insert Template 대화 상자가 열립니다.
Delete Templates	기존 템플릿을 삭제할 수 있는 Delete Templates 대화 상자가 열립니다.
Insert From Resource	현재 폼에서 열 .mnu 파일을 선택할 수 있는 Insert Menu from Resource file 대화 상자가 열립니다.

디자인 타임 시 메뉴 간의 전환

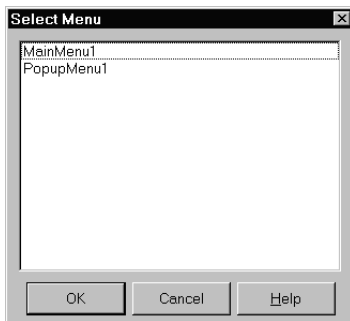
폼에 대한 여러 메뉴를 디자인하는 경우 메뉴 디자이너 컨텍스트 메뉴 또는 Object Inspector를 사용하여 메뉴 간에 쉽게 선택하고 이동할 수 있습니다.

다음과 같은 방법으로 컨텍스트 메뉴를 사용하여 폼과 메뉴 간에 전환합니다.

- 1 메뉴 디자이너에서 마우스 오른쪽 버튼을 클릭한 다음 Select Menu를 선택합니다.

Select Menu 대화 상자가 나타납니다.

그림 6.8 Select Menu 대화 상자



이 대화 상자는 메뉴 디자이너에서 현재 메뉴가 열려 있는 폼에 연결된 모든 메뉴 목록을 나열합니다.

- 2 Select Menu 대화 상자의 목록에서 보거나 편집할 메뉴를 선택합니다.

다음과 같은 방법으로 Object Inspector를 사용하여 폼의 메뉴 간에 전환합니다.

- 1 선택할 메뉴의 폼에 포커스를 맞춥니다.
- 2 컴포넌트 목록에서 편집할 메뉴를 선택합니다.
- 3 Object Inspector의 Properties 페이지에서 이 메뉴에 대한 *Items* 속성을 선택한 다음 생략 버튼을 클릭하거나 Menu를 더블 클릭합니다.

메뉴 템플릿 사용

Delphi는 자주 사용하는 명령이 들어 있는 이미 디자인된 메뉴 또는 메뉴 템플릿을 여러 개 제공합니다. 메뉴를 수정하지 않고 애플리케이션에서 이러한 메뉴를 사용할 수 있고(코드 작성은 제외) 이러한 메뉴를 기본 메뉴로 사용하여 사용자 자신이 원래 디자인한 메뉴에서 작업했던 대로 메뉴를 사용자 지정할 수 있습니다. 메뉴 템플릿에는 이벤트 핸들러 코드가 들어 있지 않습니다.

Delphi와 함께 제공된 메뉴 템플릿은 기본 설치에서 BIN 하위 디렉토리에 저장됩니다. 이 파일들의 확장명은 .DMT(Delphi 메뉴 템플릿)입니다.

메뉴 디자이너를 사용하여 디자인한 메뉴를 템플릿으로 저장할 수도 있습니다. 메뉴를 템플릿으로 저장한 후 이미 디자인된 메뉴에서 작업했던 대로 사용할 수 있습니다. 특정 메뉴 템플릿이 더 이상 필요하지 않을 경우 목록에서 해당 메뉴 템플릿을 삭제할 수 있습니다.

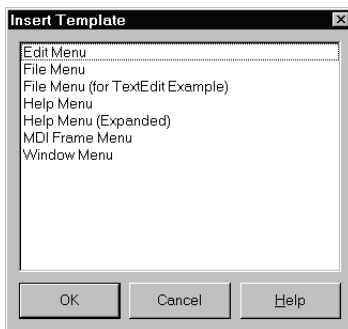
다음과 같은 방법으로 애플리케이션에 메뉴 템플릿을 추가합니다.

- 1 메뉴 디자이너를 마우스 오른쪽 버튼으로 클릭한 다음 Insert From Template을 선택합니다.

템플릿이 없는 경우에는 컨텍스트 메뉴에 Insert From Template 옵션이 흐리게 나타납니다.

Insert Template 대화 상자가 열리면서 사용 가능한 메뉴 템플릿 목록을 표시합니다.

그림 6.9 메뉴에 대한 Insert Template 대화 상자



- 2 삽입할 메뉴 템플릿을 선택한 다음 *Enter* 키를 누르거나 OK를 선택합니다.

그러면 커서가 위치한 폼에 메뉴가 삽입됩니다. 예를 들어, 커서가 목록의 메뉴 항목에 있는 경우 메뉴 템플릿이 선택한 항목 위에 삽입됩니다. 커서가 메뉴 표시줄에 있으면 메뉴 템플릿이 커서의 왼쪽에 삽입됩니다.

다음과 같은 방법으로 메뉴 템플릿을 삭제합니다.

- 1 메뉴 디자이너를 마우스 오른쪽 버튼으로 클릭한 다음 Delete Templates를 선택합니다.

템플릿이 없는 경우 컨텍스트 메뉴에 Delete Templates 옵션이 흐리게 나타납니다.

Delete Templates 대화 상자가 열리면서 사용 가능한 템플릿 목록을 나열합니다.

- 2 삭제할 메뉴 템플릿을 선택한 다음 *Del* 키를 누릅니다.

Delphi는 템플릿 목록과 하드 디스크에서 템플릿을 삭제합니다.

메뉴를 템플릿으로 저장

디자인한 메뉴를 다시 사용할 수 있도록 템플릿으로 저장할 수 있습니다. 메뉴 템플릿을 사용하여 애플리케이션에 일관된 모양을 제공하거나 이러한 메뉴 템플릿을 기본 메뉴 템플릿으로 사용하여 추가로 사용자 지정할 수 있습니다.

사용자가 저장한 메뉴 템플릿은 사용자의 BIN 하위 디렉토리에 .DMT 파일로 저장됩니다.

다음과 같은 방법으로 메뉴를 템플릿으로 저장합니다.

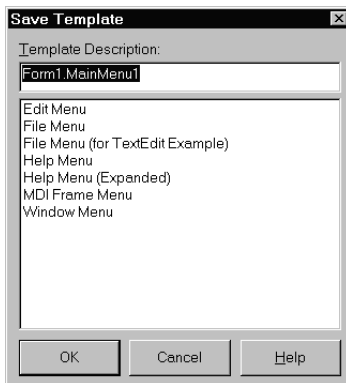
- 1 재사용할 수 있는 메뉴를 디자인합니다.

이 메뉴에는 원하는 수만큼의 항목, 명령 및 하위 메뉴가 많이 들어 있을 수 있습니다. 활성화된 메뉴 디자이너 창의 모든 것이 재사용 가능한 메뉴로 저장됩니다.

- 2 메뉴 디자이너에서 마우스 오른쪽 버튼을 클릭한 다음 Save As Template을 선택합니다.

Save Template 대화 상자가 나타납니다.

그림 6.10 메뉴에 대한 Save Template 대화 상자



- 3 Template Description 편집 상자에 이 메뉴에 대한 간단한 설명을 입력한 다음 OK를 선택합니다.

Save Template 대화 상자가 닫히면서 메뉴 디자인을 저장하고 메뉴 디자이너 창으로 돌아갑니다.

참고 입력한 설명은 Save Template, Insert Template 및 Delete Templates 대화 상자에만 표시됩니다. 이 설명은 메뉴의 *Name* 또는 *Caption* 속성과 관련이 없습니다.

템플릿 메뉴 항목 및 이벤트 핸들러의 이름 지정 규칙

메뉴를 템플릿으로 저장할 때 모든 메뉴가 메뉴 소유자(폼)의 유효 범위(scope) 내에서 고유 이름을 가져야 하므로 Delphi는 메뉴의 *Name* 속성을 저장하지 않습니다. 그러나 메뉴를 사용하여 메뉴를 템플릿으로 새 폼에 삽입할 때 Delphi는 메뉴의 새 이름 및 모든 항목을 생성합니다.

예를 들어, File 메뉴를 템플릿으로 저장한다고 가정합니다. 원래의 메뉴에서 메뉴 이름을 *MyFile*이라고 지정합니다. 이 메뉴를 템플릿으로 새 메뉴에 삽입할 경우 Delphi는 해당 메뉴의 이름을 *File1*이라고 지정합니다. *File1*이라는 이름으로 지정된 기존 메뉴 항목을 갖는 메뉴에 이 메뉴를 삽입하면 Delphi는 이 메뉴의 이름을 *File2*라고 지정합니다.

또한 Delphi는 템플릿으로 저장된 메뉴에 연결된 *OnClick* 이벤트 핸들러를 저장하지 않습니다. 코드를 새 폼에서 적용할 수 있는지 여부를 테스트할 수 있는 방법이 없기 때문입니다. 메뉴 템플릿 항목에 대해 이벤트 핸들러를 새로 생성할 때 Delphi는 이벤트 핸들러 이름을 생성합니다.

메뉴 템플릿의 항목을 폼의 기존 *OnClick* 이벤트 핸들러와 쉽게 연결할 수 있습니다. 자세한 내용은 3-27 페이지의 "이벤트를 기존 이벤트 핸들러에 연결"을 참조하십시오.

런타임 시 메뉴 항목 처리

애플리케이션이 실행 중일 때 기본 메뉴 구조에 메뉴 항목을 추가하여 자세한 내용이나 옵션을 사용자에게 제공하고자 할 경우가 종종 있습니다. 메뉴 항목의 *Add* 또는 *Insert* 메소드를 사용하여 메뉴 항목을 삽입할 수 있고, 메뉴 항목의 *Visible* 속성을 변경하여 메뉴 항목을 숨기거나 표시할 수 있습니다. *Visible* 속성을 통해 메뉴 항목을 메뉴에 표시할지 여부를 결정합니다. 메뉴를 숨기지 않고 흐리게 표시하려면 *Enabled* 속성을 사용합니다.

메뉴 항목의 *Visible* 및 *Enabled* 속성을 사용하는 예제를 보려면 7-10 페이지의 "메뉴 항목 비활성화"를 참조하십시오.

사용자는 다중 문서 인터페이스(MDI) 및 객체 연결과 중첩(OLE) 애플리케이션에서 메뉴 항목을 기존의 메뉴 바에 병합할 수도 있습니다. 다음 단원에서 좀더 자세히 설명합니다.

메뉴 병합

텍스트 에디터 예제 애플리케이션과 같은 MDI 애플리케이션 그리고 OLE 클라이언트 애플리케이션의 경우에는 애플리케이션의 메인 메뉴에서 다른 폼이나 OLE 서버 객체로부터 메뉴 항목을 받을 수 있어야 합니다. 이것을 종종 *메뉴 병합*이라고 합니다. OLE

기술은 Windows 애플리케이션에만 국한되며 크로스 플랫폼 프로그래밍에는 사용할 수 없음을 유의하십시오.

다음 두 가지 속성에 대한 값을 지정하여 메뉴 병합을 준비합니다.

- *Menu*, 폼 속성
- *GroupIndex*, 메뉴의 메뉴 항목 속성

활성 메뉴 지정: Menu 속성

Menu 속성은 폼에 대한 활성 메뉴를 지정합니다. 메뉴 병합 기능은 활성 메뉴에만 적용됩니다. 폼에 하나 이상의 메뉴 컴포넌트가 포함된 경우에는 *Menu* 속성을 코드에서 설정하여 런타임 시 활성 메뉴를 변경할 수 있습니다. 예를 들면 다음과 같습니다.

```
Form1.Menu := SecondMenu;
```

병합된 메뉴 항목의 순서 결정: GroupIndex 속성

GroupIndex 속성은 병합 메뉴 항목이 공유된 메뉴 바에 나타나는 순서를 결정합니다. 병합 메뉴 항목은 메인 메뉴 바의 항목을 대체하거나 삽입될 수 있습니다.

*GroupIndex*의 기본값은 0입니다. *GroupIndex*에 대한 값을 지정할 때는 몇 가지 규칙이 적용됩니다.

- 낮은 숫자가 메뉴의 왼쪽 끝에 먼저 표시됩니다.
예를 들어, File 메뉴와 같이 항상 가장 왼쪽에 표시되는 메뉴에 *GroupIndex* 속성을 0으로 설정합니다. 마찬가지로 Help 메뉴와 같이 항상 가장 오른쪽에 표시되는 메뉴에는 높은 숫자를 설정(반드시 순서대로 설정할 필요는 없음)합니다.
- 메인 메뉴의 항목을 대체하려면 자식 메뉴의 항목에 동일한 *GroupIndex* 값을 제공합니다.
이것은 그룹 지정이나 단일 항목에 적용됩니다. 예를 들어, 메인 폼에 *GroupIndex* 값이 1인 Edit 메뉴 항목이 있는 경우 자식 폼 메뉴에서 한 개 이상의 항목에 대해서도 *GroupIndex* 값에 1을 제공하면 이를 대체할 수 있습니다.
자식 메뉴의 여러 항목에 동일한 *GroupIndex* 값을 제공하면 메인 메뉴로 병합될 때 순서가 변하지 않습니다.
- 메인 메뉴의 항목을 대체하지 않고 항목을 삽입하려면 메인 메뉴 항목의 숫자 범위에 여분을 두고 자식 폼에서 가져온 숫자들을 "끼워 넣습니다."
예를 들어, 메인 메뉴 항목들에 0과 5의 숫자를 지정하고 자식 메뉴에서 1, 2, 3, 4로 지정하여 삽입합니다.

리소스 파일 import하기

다른 애플리케이션에서 작성된 메뉴가 표준 Windows 리소스(.RC) 파일 형식인 경우, Delphi에서 사용할 수 있습니다. 이러한 메뉴를 Delphi 프로젝트로 직접 import함으로써 다른 곳에서 작성한 메뉴를 재작성하는 데 드는 시간과 노력을 절약할 수 있습니다.

다음과 같은 방법으로 기존 .RC 메뉴 파일을 로드합니다.

- 1 메뉴 디자이너에서 메뉴를 표시할 위치에 커서를 둡니다.
import한 메뉴는 디자인 중인 메뉴의 일부일 수도 있고, 그 자체가 전체 메뉴일 수도 있습니다.
- 2 오른쪽 단추를 클릭하여 Insert From Resource를 선택합니다.
Insert Menu From Resource 대화 상자가 나타납니다.
- 3 대화 상자에서 로드할 리소스 파일을 선택하고 OK를 선택합니다.
메뉴 디자이너 창에 메뉴가 나타납니다.

참고 리소스 파일에 하나 이상의 메뉴가 있을 경우, 사용자는 이를 import하기 전에 먼저 각 메뉴를 별도의 리소스 파일로 저장해야 합니다.

툴바 및 쿨바 디자인

*툴바*는 버튼과 다른 컨트롤을 포함하는 일반적으로 폼의 맨 위에 위치하면서 메뉴 표시 줄 아래에 있는 가로로 된 하나의 패널입니다. *쿨바*(또는 리바라고도 함)는 움직이거나 크기 조절이 가능한 밴드에 컨트롤을 표시하는 툴바의 일종입니다. 폼 위에 정렬되는 패널이 여러 개 있을 경우에는 추가되는 순서대로 수직으로 쌓여집니다.

참고 쿨바는 크로스 플랫폼 애플리케이션의 CLX에서는 사용할 수 없습니다.

툴바에 모든 종류의 컨트롤을 놓을 수 있습니다. 버튼은 물론 색상 그리드, 스크롤 막대 및 레이블 등을 놓을 수 있습니다.

여러 가지 방식으로 툴바를 폼에 추가할 수 있습니다.

- 폼에 패널(*TPanel*)을 두고 패널에 컨트롤(전형적으로 스피드 버튼)을 추가합니다.
- *TPanel* 대신 툴바 컴포넌트(*TToolBar*)를 사용하여 컨트롤을 추가합니다. *TToolBar*는 버튼 및 다른 컨트롤을 행에 배치하고 크기와 위치를 자동으로 조정하여 버튼 및 다른 컨트롤을 관리합니다. 툴바에서 툴 버튼(*TToolButton*) 컨트롤을 사용하는 경우 *TToolBar*를 사용하면 버튼을 기능에 따라 쉽게 그룹화하고 다른 표시 옵션을 제공할 수 있습니다.
- 쿨바(*TCoolBar*) 컴포넌트를 사용하여 컨트롤을 추가합니다. 쿨바는 독립적으로 이동이 가능하고 크기 조절이 가능한 밴드에 컨트롤을 표시합니다.

툴바 구현 방법은 애플리케이션에 따라 다릅니다. Panel 컴포넌트를 사용하면 툴바의 룩앤필을 완전히 제어할 수 있다는 이점이 있습니다.

툴바 및 쿨바 컴포넌트를 사용하면 본래의 주어진 Windows 컨트롤을 사용하므로 애플리케이션이 Windows 응용 프로그램의 룩앤필을 갖게 됩니다. 이러한 운영 체제 컨트롤이 나중에 변경되면 애플리케이션도 변경됩니다. 또한 툴바와 쿨바는 Windows의 공통 컴포넌트에 의존하기 때문에 애플리케이션은 COMCTL32.DLL을 필요로 합니다. 툴바와 쿨바는 WinNT 3.51 애플리케이션에서는 지원되지 않습니다.

다음 단원에서는 다음과 같은 작업을 수행하는 방법에 대해 설명합니다.

- 패널 컴포넌트를 사용하여 툴바 및 해당 스피드 버튼 컨트롤 추가
- 툴바 컴포넌트를 사용하여 툴바 및 해당 툴 버튼 컨트롤 추가
- 쿨바 컴포넌트를 사용하여 쿨바 추가
- 클릭 이벤트에 응답
- 숨겨진 툴바 및 쿨바 추가
- 툴바와 쿨바의 숨기기 및 표시

패널 컴포넌트를 사용하여 툴바 추가

다음과 같은 방법으로 패널 컴포넌트를 사용하여 폼에 툴바를 추가합니다.

- 1 컴포넌트 팔레트의 Standard 페이지에 있는 패널 컴포넌트를 폼에 추가합니다.
- 2 패널의 *Align* 속성을 *alTop*으로 설정합니다. 패널이 폼 맨 위에 정렬될 때 창 크기가 변경되어도 패널 높이는 유지되지만 패널 너비는 폼 클라이언트 영역의 전체 너비에 맞춰집니다.
- 3 패널에 스피드 버튼 또는 다른 컨트롤을 추가합니다.

스피드 버튼은 툴바 패널에서 작동하도록 디자인되었습니다. 스피드 버튼에는 일반적으로 캡션은 없고 버튼 기능을 나타내는 *glyph*라는 작은 그래픽만 포함됩니다.

스피드 버튼에는 세 가지 작동 모드가 있습니다. 스피드 버튼은 다음 작동을 수행할 수 있습니다.

- 일반 푸시 버튼처럼 동작
- 클릭하여 켜고 끄기
- 라디오 버튼 집합처럼 동작

툴바에서 스피드 버튼을 구현하려면 다음을 수행하십시오.

- 툴바 패널에 스피드 버튼 추가
- 스피드 버튼의 *glyph* 할당
- 스피드 버튼의 초기 상태 설정
- 스피드 버튼 그룹 생성
- 버튼 토크 허용

패널에 스피드 버튼 추가

툴바 패널에 스피드 버튼을 추가하려면 컴포넌트 팔레트의 Additional 페이지에 있는 스피드 버튼 컴포넌트를 패널에 둡니다.

폼이 아닌 패널은 스피드 버튼을 "소유"하므로 패널을 이동하거나 숨기면 스피드 버튼도 이동하거나 숨겨집니다.

패널의 기본 높이는 41이고 스피드 버튼의 기본 높이는 25입니다. 각 버튼의 *Top* 속성을 8로 설정하면 버튼이 수직 중앙에 위치합니다. 기본 그리드 설정에 의해 스피드 버튼이 해당 수직 위치에 맞춰집니다.

스피드 버튼의 glyph 할당

각 스피드 버튼에는 버튼이 수행하는 작업을 사용자에게 나타내기 위해 *glyph*라는 그래픽 이미지가 필요합니다. 스피드 버튼에 이미지를 하나만 제공하는 경우 버튼은 해당 이미지를 조작하여 버튼이 눌러져 있는지, 선택되어 있는지 또는 사용 불가능한지 여부를 나타냅니다. 사용자가 원할 경우 각 상태에 대해서 별도의 특정 이미지를 제공할 수 있습니다.

런타임에 다른 *glyph*를 할당할 수 있지만 일반적으로 디자인 타임에 *glyph*를 스피드 버튼에 지정합니다.

다음과 같은 방법으로 디자인 타임에 *glyph*를 스피드 버튼에 할당합니다.

- 1 스피드 버튼을 선택합니다.
- 2 Object Inspector에서 *Glyph* 속성을 선택합니다.
- 3 *Glyph* 옆의 값 열을 더블 클릭하여 Picture Editor를 열고 원하는 비트맵을 선택합니다.

스피드 버튼의 초기 상태 설정

사용자는 스피드 버튼의 모양을 통해 버튼의 상태 및 용도에 대해 알 수 있습니다. 스피드 버튼에는 캡션이 없으므로 사용자에게 도움을 줄 수 있는 적절한 시각적 역할이 중요합니다.

표 6.6은 스피드 버튼의 모양을 변경하기 위해 설정할 수 있는 일부 작업을 나열합니다.

표 6.6 스피드 버튼의 모양 설정

스피드 버튼의 모양 변경	툴바의 속성 설정
눌려진 상태로 표시	<i>GroupIndex</i> 속성을 0이 아닌 값으로 설정하고 <i>Down</i> 속성을 <i>True</i> 로 설정합니다.
사용 불가능으로 표시	<i>Enabled</i> 속성을 <i>False</i> 로 설정합니다.
왼쪽 여백 표시	<i>Indent</i> 속성 값을 0보다 큰 값으로 설정합니다.

애플리케이션에 기본 드로잉 툴이 있는 경우 애플리케이션 시작 시 툴바의 해당 버튼이 눌러져 있는지 확인합니다. 그렇게 하려면 *GroupIndex* 속성을 0이 아닌 값으로 설정하고 *Down* 속성을 *True*로 설정합니다.

스피드 버튼 그룹 생성

경우에 따라 일련의 스피드 버튼은 상호 배타적인 선택의 집합을 나타냅니다. 그러한 경우 버튼을 그룹으로 연결하면 그룹 내의 임의 버튼을 눌렀을 때 그룹 내의 다른 버튼이 팝업됩니다.

임의 수의 스피드 버튼을 그룹으로 연결하려면 동일한 수를 각 스피드 버튼의 *GroupIndex* 속성에 할당합니다.

이렇게 하는 가장 쉬운 방법은 그룹에서 원하는 버튼을 모두 선택한 다음 선택한 전체 그룹에서 *GroupIndex*를 고유 값으로 설정하는 것입니다.

버튼 토글 허용

때때로 그룹 내에 있는 버튼 하나를 클릭하여 다른 버튼에는 영향을 주지 않고 눌러진 상태로 바뀌게 하고 다시 한 번 클릭하면 팝업되게 만들고 싶은 경우가 있습니다. 이를 버튼 *토글*이라고 합니다. *AllowAllUp*을 사용하여 토글처럼 동작하는 그룹화된 버튼을 만들 수 있습니다. 그룹화된 버튼을 한 번 클릭하면 눌러지고 다시 클릭하면 팝업됩니다.

그룹화된 스피드 버튼을 토글로 만들려면 *AllowAllUp* 속성을 *True*로 설정합니다.

그룹 내 임의의 스피드 버튼의 *AllowAllUp* 속성을 *True*로 설정하면 그룹 내 모든 버튼에 대해 동일한 속성 값이 자동으로 설정됩니다. 이렇게 하면 그룹이 한 번에 한 버튼만 눌러지지만 동시에 모든 버튼이 팝업될 수 있게 해주는, 일반 그룹처럼 동작하는 그룹을 사용할 수 있습니다.

툴바 컴포넌트를 사용하여 투바 추가

툴바 컴포넌트 (*TToolBar*)는 패널 컴포넌트가 제공하지 않는 버튼 관리 및 표시 기능을 제공합니다. 다음과 같은 방법으로 투바 컴포넌트를 사용하여 폼에 투바를 추가합니다.

- 1 컴포넌트 팔레트의 Win32 페이지에서 폼에 투바 컴포넌트를 추가합니다. 투바는 폼 맨 위에 자동으로 정렬됩니다.
- 2 투바에 투 버튼 또는 다른 컨트롤을 추가합니다.

툴 버튼은 투바 컴포넌트에서 작동하도록 디자인되었습니다. 스피드 버튼과 마찬가지로 투 버튼은 다음과 같은 동작을 수행할 수 있습니다.

- 일반 푸시 버튼처럼 동작
- 눌러서 켜고 끄기
- 라디오 버튼 집합처럼 동작

툴바에서 투 버튼을 구현하려면 다음을 수행하십시오.

- 투 버튼 추가
- 투 버튼에 이미지 할당
- 투 버튼 모양 설정
- 투 버튼 그룹 생성
- 투 버튼 토글 허용

툴 버튼 추가

툴바에 툴 버튼을 추가하려면 툴바를 마우스 오른쪽 버튼으로 클릭한 후 New Button 을 선택합니다.

툴바는 툴 버튼을 "소유"하므로 툴바를 이동하거나 숨기면 툴 버튼도 이동하거나 숨겨 집니다. 또한 툴바의 모든 툴 버튼은 동일한 높이와 너비를 자동으로 유지합니다. 컴포넌트 팔레트의 다른 컨트롤을 툴바에 가져다 놓을 수 있으며 이러한 컨트롤은 동일한 높이를 자동으로 유지합니다. 컨트롤이 툴바의 수평 공간 안에 다 들어 가지 않으면 랩어라운드하거나 새 행에서 시작합니다.

툴 버튼에 이미지 할당

각각의 툴 버튼에는 런타임에 버튼에 나타날 이미지를 결정하는 *ImageIndex* 속성이 있습니다. 툴 버튼에 이미지를 하나만 제공할 경우 버튼은 해당 이미지를 처리하여 버튼이 사용 불가능한지 여부를 나타냅니다. 다음과 같은 방법으로 디자인 타임 시 툴 버튼에 이미지를 할당합니다.

- 1 버튼을 표시하는 툴바를 선택합니다.
- 2 Object Inspector에서 *TImageList* 객체를 툴바의 *Images* 속성에 할당합니다. 이미지 목록은 크기가 같은 아이콘 또는 비트맵의 컬렉션입니다.
- 3 툴 버튼을 선택합니다.
- 4 Object Inspector에서 버튼에 지정할 이미지 목록의 이미지에 해당하는 툴 버튼의 *ImageIndex* 속성에 정수를 할당합니다.

툴 버튼이 사용 불가능하거나 마우스 포인터 아래에 있을 때 별도의 이미지가 해당 툴 버튼에 나타나도록 지정할 수도 있습니다. 그렇게 하려면 툴바의 *DisabledImages* 및 *HotImages* 속성에 별도의 이미지 목록을 할당합니다.

툴 버튼 모양 및 초기 상태 설정

표 6.7은 툴 버튼 모양을 변경하기 위해 설정할 수 있는 일부 동작들을 나열합니다.

표 6.7 툴 버튼 모양 설정

툴 버튼의 모양 변경	툴바의 속성 설정
눌려진 상태로 표시	도구 버튼에서 <i>Style</i> 속성은 <i>tbsCheck로 Down</i> 속성은 <i>True</i> 로 설정합니다.
사용 불가능으로 표시	<i>Enabled</i> 속성을 <i>False</i> 로 설정합니다.
왼쪽 여백 표시	<i>Indent</i> 속성 값을 0보다 큰 값으로 설정합니다.
"팝업" 테두리가 있어 툴바를 알기 쉽게 표시	<i>Flat</i> 속성을 <i>True</i> 로 설정합니다.

참고 *TToolBar*의 *Flat* 속성을 사용하는 경우에는 COMCTL32.DLL의 4.70 또는 이후 버전이 필요합니다.

특정 툴 버튼 다음에 새로운 행의 컨트롤을 놓으려면 행의 마지막에 나타날 툴 버튼을 선택하고 *Wrap* 속성을 *True*로 설정합니다.

툴바의 자동 줄 바꿈 (auto-wrap) 기능을 해제하려면 툐바의 *Wrapable* 속성을 *False* 로 설정합니다.

툴 버튼 그룹 생성

툴 버튼 그룹을 만들려면 연결할 버튼을 선택하고 *Style* 속성을 *tbsCheck*로 설정한 다음 *Grouped* 속성을 *True*로 설정합니다. 그룹화된 툐 버튼을 선택하면 그룹 내의 다른 버튼이 팝업되어 상호 배타적 선택 집합을 나타내는 데 도움이 됩니다.

*Style*은 *tbsCheck*로 설정되고 *Grouped*는 *True*로 설정된 일련의 인접한 툐 버튼은 단일 그룹을 형성합니다. 툐 버튼 그룹을 해제하려면 다음과 같은 버튼을 이용해서 분리합니다.

- *Grouped* 속성이 *False*인 툐 버튼
- *Style* 속성이 *tbsCheck*로 설정되지 않은 툐 버튼. 툐바에 공백이나 구분선을 만들려면 *Style*이 *tbsSeparator* 또는 *tbsDivider*로 설정된 툐 버튼을 추가합니다.
- 툐 버튼 이외의 다른 컨트롤

툴 버튼 토글 허용

*AllowAllUp*을 사용하여 토글처럼 동작하는 그룹화된 툐 버튼을 만듭니다. 그룹화된 버튼을 한 번 클릭하면 눌러지고 또 한 번 클릭하면 팝업됩니다. 그룹화된 툐 버튼을 토글로 만들려면 *AllowAllUp* 속성을 *True*로 설정합니다.

스피드 버튼에서 그룹 내 임의의 툐 버튼의 *AllowAllUp*을 *True*로 설정하면 그룹 내 모든 버튼에 대해 동일한 속성 값이 자동으로 설정됩니다.

쿨바 컴포넌트 추가

참고 *TCoolBar* 컴포넌트는 COMCTL32.DLL의 4.70 이후 버전이 필요하며 CLX에서는 사용할 수 없습니다.

쿨바 컴포넌트 *TCoolBar*(*rebar*라고도 함)는 독립적으로 이동이 가능하며 크기 조절이 가능한 밴드에 창 모양의 컨트롤을 표시합니다. 사용자는 각 밴드의 왼쪽 면에 있는 크기 조정 그림을 끌어서 밴드의 위치를 조정할 수 있습니다.

다음과 같은 방법으로 Windows 애플리케이션의 폼에 쿨바를 추가합니다.

- 1 컴포넌트 팔레트의 Win32 페이지에서 폼에 쿨바 컴포넌트를 추가합니다. 쿨바는 폼 맨 위에 자동으로 정렬됩니다.
- 2 컴포넌트 팔레트에서 창 모양의 컨트롤을 바에 추가합니다.

TWinControl 자손인 VCL 컴포넌트만 창 모양의 컨트롤입니다. 레이블이나 속도 버튼과 같은 그래픽 컨트롤을 쿨바에 추가할 수 있지만 독립된 밴드에는 표시되지 않습니다.

쿨바 모양 설정

쿨바 컴포넌트는 몇 가지 유용한 옵션을 제공합니다. 표 6.8은 도구 버튼의 모양을 변경하기 위해 설정할 수 있는 몇 가지 작업들을 나열한 것입니다.

표 6.8 쿨 버튼 모양 설정

쿨바의 모양 변경	툴바의 속성 설정
포함하는 밴드를 수용하도록 자동으로 크기 조정	<i>AutoSize</i> 속성을 <i>True</i> 로 설정합니다.
밴드의 높이를 동일하게 유지	<i>FixedSize</i> 속성을 <i>True</i> 로 설정합니다.
수평이 아닌 수직으로 방향 지정	<i>Vertical</i> 속성을 <i>True</i> 로 설정합니다. 이것은 <i>FixedSize</i> 속성의 결과를 변경합니다.
런타임 시 밴드의 <i>Text</i> 속성이 표시되지 않도록 방지	<i>ShowText</i> 속성을 <i>False</i> 로 설정합니다. 쿨바의 각 밴드는 자체의 <i>Text</i> 속성을 지닙니다.
바 주위의 경계 제거	<i>BandBorderStyle</i> 을 <i>bsNone</i> 으로 설정합니다.
사용자가 런타임 시 밴드의 경계를 변경하지 못하도록 방지 (사용자는 여전히 밴드를 이동하고 크기를 조정할 수 있습니다.)	<i>FixedOrder</i> 를 <i>True</i> 로 설정합니다.
쿨바에 대한 백그라운드 이미지 생성	<i>Bitmap</i> 속성을 <i>TBitmap</i> 객체로 설정합니다.
밴드 왼쪽에 나타나는 이미지 목록 선택	<i>Images</i> 속성을 <i>TImageList</i> 객체로 설정합니다.

개별 밴드에 이미지를 할당하려면 쿨바를 선택하고 Object Inspector의 *Bands* 속성을 더블 클릭합니다. 그런 다음 밴드를 선택하고 *ImageIndex* 속성에 값을 할당합니다.

클릭 이벤트에 응답

사용자가 투바에서 버튼과 같은 컨트롤을 클릭하는 경우 애플리케이션에서 이벤트 핸들러를 통해 응답하도록 하는 *OnClick* 이벤트를 생성합니다. *OnClick*은 버튼의 기본 이벤트이므로 디자인 타임에 버튼을 더블 클릭하여 이벤트의 뼈대 핸들러를 생성할 수 있습니다. 자세한 내용은 3-26 페이지의 "이벤트 및 이벤트 핸들러 작업" 및 3-26 페이지의 "컴포넌트의 기본 이벤트에 대한 핸들러 생성"을 참조하십시오.

툴 버튼에 메뉴 할당

툴 버튼 (*TToolButton*)을 가지는 투바 (*TToolBar*)를 사용하는 경우 특정 버튼과 메뉴를 연결할 수 있습니다.

- 1 투 버튼을 선택합니다.
- 2 Object Inspector에서 투 버튼의 *DropDownMenu* 속성에 팝업 메뉴 (*TPopupMenu*)를 할당합니다.

메뉴의 *AutoPopup* 속성을 *True*로 설정하는 경우 버튼을 누를 때 메뉴가 자동으로 나타납니다.

숨겨진 툴바 추가

툴바가 항상 보일 필요는 없습니다. 경우에 따라서는 실제로 많은 툴바를 가지고 있어도 사용할 때만 툴바를 보이게 하는 것이 편리합니다. 툴바가 여러 개 있는 폼을 만들지만 툴바의 일부 또는 전부를 숨기는 경우도 있습니다.

다음과 같은 방법으로 숨겨진 툴바를 만듭니다.

- 1 폼에 툴바, 쿨바 또는 패널 컴포넌트를 추가합니다.
- 2 컴포넌트의 *Visible* 속성을 *False*로 설정합니다.

디자인 타임에는 툴바를 볼 수 있고 수정할 수 있을지라도 런타임에는 애플리케이션에서 특별히 툴바를 보이게 할 때까지는 숨겨진 채로 있습니다.

툴바 숨기기 및 표시

경우에 따라 애플리케이션에 툴바가 여러 개 필요하지만 한 번에 모든 툴바를 폼에 나타내고 싶지 않은 경우도 있습니다. 또는 사용자가 툴바를 표시할지 여부를 결정하도록 할 수 있습니다. 모든 컴포넌트에서 런타임에 필요에 따라 툴바를 표시하거나 숨길 수 있습니다.

런타임에 툴바를 숨기거나 표시하려면 툴바의 *Visible* 속성을 각각 *False* 또는 *True*로 설정합니다. 대개 특정 사용자 이벤트에 대응해서 이렇게 설정하거나 애플리케이션의 작동 모드에서 이것을 변경합니다. 이렇게 하려면 전형적으로 각각의 툴바에서 닫기 버튼을 갖도록 합니다. 사용자가 닫기 버튼을 클릭하면 애플리케이션에서 해당 툴바가 숨겨집니다.

툴바를 토글하는 방법을 제공할 수도 있습니다. 다음 예제에서 펜(Pen)의 툴바는 메인 툴바의 버튼에서 토글됩니다. 한 번 클릭할 때마다 버튼이 눌러지거나 해제되므로 *OnClick* 이벤트 핸들러는 버튼이 눌러져 있는지 여부에 따라 Pen 툴바를 표시하거나 숨길 수 있습니다.

```
procedure TForm1.PenButtonClick(Sender:TObject);  
begin  
  PenBar.Visible := PenButton.Down;  
end;
```

데모 프로그램

액션 및 액션 리스트를 사용하는 Windows 애플리케이션 예제를 보려면 Demos\RichEdit를 참조하십시오. 또한 Application 마법사(File|New Project page), MDI Application, SDI Application, 및 Winx Logo Applications는 액션 및 액션 리스트 객체를 사용할 수 있습니다. 크로스 플랫폼 애플리케이션 예제는 Demos\CLX를 참조하십시오.

7

컨트롤 사용

컨트롤은 런타임 시 상호 작용할 수 있는 비주얼 컴포넌트입니다. 이 장에서는 여러 컨트롤에서 일반적으로 사용되는 다양한 기능에 대해 설명합니다.

컨트롤의 드래그 앤 드롭 구현

드래그 앤 드롭은 객체를 처리하는 편리한 방법입니다. 컨트롤 전체를 끌거나 또는 리스트 박스나 트리 뷰와 같은 한 컨트롤의 항목에서 다른 컨트롤로 끌 수 있습니다.

- 드래그 연산 시작
- 드래그된 항목 수용
- 항목 드롭
- 드래그 연산 끝내기
- 드래그 객체를 사용하여 드래그 앤 드롭 사용자 지정
- 드래그 마우스 포인터 변경

드래그 연산 시작

모든 컨트롤에는 드래그 연산이 시작되는 방법을 정하는 *DragMode*라는 속성이 있습니다. *DragMode*가 *dmAutomatic*인 경우, 사용자가 컨트롤 위에서 커서의 마우스를 누를 때 자동으로 드래깅이 시작됩니다. *dmAutomatic*은 일반적인 마우스 동작을 방해할 수 있기 때문에 *DragMode*를 *dmManual*(기본값)로 설정하고 마우스 다운 이벤트를 처리하여 드래깅을 시작할 수 있습니다.

컨트롤을 수동으로 드래깅하려면 컨트롤의 *BeginDrag* 메소드를 호출합니다. *BeginDrag*는 *Immediate*라는 부울 매개변수를 사용하고 옵션으로 *Threshold*라는 정수 매개변수를 사용합니다. *Immediate* 매개변수의 값으로 *True*를 전달하는 경우, 드래깅은 즉시 시작됩니다. *False*를 전달하는 경우, 사용자가 *Threshold*에서 지정된 픽셀만큼 마우스를 움직여야 드래깅이 시작됩니다. 다음과 같은 호출은

```
BeginDrag (False)
```

컨트롤이 드래그 연산을 시작하지 않고 마우스 클릭을 받아들일 수 있게 해줍니다.

*BeginDrag*를 호출하기 전에 마우스 다운 이벤트의 매개변수를 테스트하여 어떤 버튼을 사용자가 눌렀는지 확인하는 것과 같은 드래깅 시작 여부에 관한 다른 조건을 부여할 수 있습니다. 예를 들어, 다음 코드는 마우스 왼쪽 버튼이 눌러졌을 경우에만 드래그 연산을 초기화하여 파일 리스트 박스의 마우스 다운 이벤트를 처리합니다.

```

procedure TFMForm.FileListBox1MouseDown(Sender:TObject;
  Button:TMouseButton; Shift:TShiftState; X, Y:Integer);
begin
  if Button = mbLeft then { drag only if left button pressed }
    with Sender as TFileListBox do { treat Sender as TFileListBox }
      begin
        if ItemAtPos(Point(X, Y), True) >= 0 then { is there an item here? }
          BeginDrag(False); { if so, drag it }
        end;
      end;
end;

```

드래그된 항목 수용

사용자가 컨트롤 위로 어떤 항목을 드래그할 때 그 컨트롤은 *OnDragOver* 이벤트를 받습니다. 이 때 사용자가 그 항목을 드롭한다면 컨트롤은 이 항목을 받아들일 수 있는지 여부를 나타내야 합니다. 컨트롤이 드래그된 항목을 받아들일 수 있는지 나타내기 위해 드래그 커서의 모양이 변합니다. 컨트롤 위로 드래그된 항목을 받아들여려면 컨트롤의 *OnDragOver* 이벤트에 이벤트 핸들러를 연결합니다.

drag-over 이벤트에는 *Accept*라는 매개변수가 있으므로 항목을 받아들여려면 이벤트 핸들러에서 이 매개변수의 값을 *True*로 설정합니다. *Accept*가 *True*인 경우, 애플리케이션에서 드래그 앤 드롭 이벤트를 컨트롤에 보냅니다.

drag-over 이벤트에는 이벤트 핸들러가 드롭을 받아들일지 여부를 정하는 데 사용할 수 있는 드래깅의 소스 및 마우스 커서의 현재 위치와 같은 매개변수도 있습니다. 다음 예제에서 디렉토리 트리 뷰는 파일 리스트 박스에서 온 드래그된 항목만 받아들입니다.

```

procedure TFMForm.DirectoryOutline1DragOver(Sender, Source:TObject; X,
  Y: Integer; State:TDragState; var Accept:Boolean);
begin
  if Source is TFileListBox then
    Accept := True
  else
    Accept := False;
  end;

```

항목 드롭

컨트롤이 드래그된 항목을 받아들일 수 있다고 나타나는 경우, 드롭된 항목을 처리해야 합니다. 드롭되는 항목을 처리하려면 드롭을 받아들이는 컨트롤의 *OnDragDrop* 이벤트에 이벤트 핸들러를 연결합니다. drag-over 이벤트와 마찬가지로 드래그 앤 드롭 이벤트는 드래그된 항목의 소스 및 받아들이는 컨트롤 위에 있는 마우스 커서의 좌표를 나타냅니다. 마우스 커서의 좌표를 나타내는 매개변수로 항목이 드래그되는 동안의 경로를

모니터할 수 있습니다. 예를 들어, 커서 좌표를 사용하면 드래그 항목이 전달될 때 컴포넌트의 색상을 변경할 수도 있습니다.

다음 예제에서 파일 리스트 박스로부터 드래그된 항목을 받아들이는 디렉토리 트리 뷰는 현재 위치에서 드롭되는 디렉토리로 파일을 옮깁니다.

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileListBox1.FileName, Items[GetItem(X, Y)].FullPath);
    end;
end;
```

드래그 연산 끝내기

드래그 연산은 항목이 성공적으로 드롭되거나 또는 항목을 받아들일 수 없는 컨트롤 위에서 마우스 버튼을 놓았을 때 끝납니다. 이 시점에서 항목이 드래그된 컨트롤에 `end-drag` 이벤트가 보내집니다. 컨트롤에서 항목이 드래그되었을 때 이 컨트롤이 응답할 수 있게 하려면 컨트롤의 `OnEndDrag` 이벤트에 이벤트 핸들러를 연결합니다.

`OnEndDrag` 이벤트의 가장 중요한 매개변수인 `Target`은 어떤 컨트롤이 드롭을 받아들이는지 나타냅니다. `Target`이 `nil`인 경우, 어떤 컨트롤도 드래그된 항목을 받아들이지 않는다는 것을 의미합니다. `OnEndDrag` 이벤트에는 받아들이는 컨트롤의 좌표도 포함되어 있습니다.

이 예제에서 파일 리스트 박스는 파일 목록을 새로 고침으로써 `end-drag` 이벤트를 처리합니다.

```
procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileListBox1.Update;
end;
```

드래그 객체를 사용하여 드래그 앤 드롭 사용자 지정

`TDragObject`의 자손을 사용해서 객체의 드래그 앤 드롭 동작을 사용자 지정할 수 있습니다. 표준 `drag-over` 및 드래그 앤 드롭 이벤트는 드래그된 항목의 소스 및 받아들이는 컨트롤 위에 있는 마우스 커서의 좌표를 나타냅니다. 추가적인 상태 정보를 얻으려면 `TDragObject` 또는 `TDragObjectEx`에서 사용자 지정 드래그 객체를 파생시키고 파생된 객체의 가상 메소드를 오버라이드합니다. `OnStartDrag` 이벤트에서 사용자 지정 드래그 객체를 만듭니다.

일반적으로 `drag-over` 및 드래그 앤 드롭 이벤트의 소스 매개변수는 드래그 연산을 시작하는 컨트롤입니다. 만일 다른 종류의 컨트롤이 동일한 종류의 데이터를 포함하는 연산을 시작할 수 있다면 소스는 각각의 컨트롤을 지원할 필요가 있습니다. 그러나 `TDragObject`의 자손을 사용하는 경우, 소스는 드래그 객체 자체입니다. 각 컨트롤이 `OnStartDrag` 이벤트에서 동일한 종류의 드래그 객체를 만든다면 그 대상(`target`)은 한 종류의 객체만 처리하면 됩니다. 컨트롤과 달리 `drag-over` 이벤트 및 드래그 앤 드롭

이벤트는 *IsDragObject* 함수를 호출하여 소스가 드래그 객체인지 아닌지 구별할 수 있습니다.

*TDragObjectEx*의 자손은 자동으로 메모리 해제가 되는 반면 *TDragObject*의 자손은 그렇지 않습니다. 사용자가 명시적으로 메모리를 해제하지 않는 *TDragObject* 자손이 있는 경우, 메모리 손실을 방지하는 대신 자손 객체를 변경하여 *TDragObjectEx*로부터 파생시킬 수 있습니다.

객체 드래그를 통해 애플리케이션의 메인 실행 파일에서 구현된 폼과 DLL을 사용하여 구현된 폼 사이에, 또는 다른 DLL을 사용하여 구현된 폼들 사이에 항목을 드래그합니다.

드래그 마우스 포인터 변경

소스 컴포넌트의 *DragCursor* 속성을 설정하여 드래그 연산을 수행하는 동안 마우스 포인터의 모양을 사용자 지정할 수 있습니다(VCL에만 해당).

컨트롤에서 드래그 앤 드롭 구현

참고 VCL에서는 드래그 앤 도킹 속성을 사용할 수 있으나 CLX에서는 사용할 수 없습니다.

*TWinControl*의 자손은 도킹 공간으로 동작할 수 있고 *TControl*의 자손은 도킹 공간에 도킹되는 자식 윈도우로 동작할 수 있습니다. 예를 들어, 폼 윈도우의 왼쪽 가장 자리에 도킹 공간을 제공하려면 폼의 왼쪽 가장 자리에 패널을 배치하여 패널을 도킹 공간으로 만듭니다. 도킹 가능한 컨트롤을 패널에 드래그 앤 드롭하면 이 컨트롤은 패널의 자식 컨트롤이 됩니다.

- 윈도우 컨트롤을 도킹 공간으로 만들기
- 컨트롤을 도킹 가능한 자식 컨트롤로 만들기
- 자식 컨트롤의 도킹 방법 제어
- 자식 컨트롤이 도킹 해제되는 방법 제어
- 자식 컨트롤이 드래그 앤 드롭 연산에 응답하는 방법 제어

윈도우 컨트롤을 도킹 공간으로 만들기

참고 VCL에서는 드래그 앤 도킹 속성을 사용할 수 있으나 CLX에서는 사용할 수 없습니다. 다음과 같은 방법으로 윈도우 컨트롤을 도킹 공간으로 만듭니다.

- 1 *DockSite* 속성을 *True*로 설정합니다.
- 2 도킹 공간 객체가 도킹된 클라이언트를 포함하고 있을 경우에만 나타나게 하려면 *AutoSize* 속성을 *True*로 설정합니다. *AutoSize*가 *True*인 경우, 도킹 공간은 자식 컨트롤이 도킹될 때까지 크기가 0으로 지정됩니다. 그런 다음 자식 컨트롤이 도킹되면 자식 컨트롤의 크기에 맞게 크기가 조정됩니다.

컨트롤을 도킹 가능한 자식 컨트롤로 만들기

참고 VCL에서는 드래그 앤 도킹 속성을 사용할 수 있으나 CLX에서는 사용할 수 없습니다. 다음과 같은 방법으로 컨트롤을 도킹 가능한 자식 컨트롤로 만듭니다.

- 1 *DragKind* 속성을 *dkDock*으로 설정합니다. *DragKind*가 *dkDock*인 경우, 컨트롤을 드래그해서 새로운 도킹 공간으로 이동하거나 컨트롤의 도킹을 해제하면 이동식 창 (floating window)이 됩니다. *DragKind*가 *dkDrag*(기본값)인 경우, 컨트롤을 드래그하면 *OnDragOver*, *OnEndDrag* 및 *OnDragDrop* 이벤트로 구현되어야 하는 드래그 앤 드롭 연산이 시작됩니다.
- 2 *DragMode*를 *dmAutomatic*으로 설정합니다. *DragMode*가 *dmAutomatic*인 경우, 사용자가 마우스로 컨트롤을 드래그하기 시작할 때 *DragKind*의 값에 따라 드래그 앤 드롭 또는 도킹은 자동으로 초기화됩니다. *DragMode*가 *dmManual*인 경우, *BeginDrag* 메소드를 호출하여 드래그 앤 도킹 또는 드래그 앤 드롭 연산을 시작할 수 있습니다.
- 3 컨트롤의 도킹을 해제하고 이동식 창이 될 때 그 컨트롤을 포함하는 *TWinControl* 자손을 나타내도록 하려면 *FloatingDockSiteClass* 속성을 설정합니다. 마우스를 놓은 다음 컨트롤이 도킹 공간에 있지 않을 때 이 클래스의 윈도우 컨트롤이 동적으로 생성되고 생성된 윈도우 컨트롤은 도킹 가능한 자식의 부모가 됩니다. 도킹 가능한 자식 컨트롤이 *TWinControl*의 자손이면 테두리 및 제목 표시줄을 얻기 위해 폼을 지정하려고 할지라도 그 컨트롤을 포함하는 별도의 이동식 도킹 공간을 만들 필요는 없습니다. 동적인 컨테이너 윈도우를 생략하기 위해 *FloatingDockSiteClass*를 컨트롤과 동일한 클래스로 설정하면 부모가 없는 이동식 창이 됩니다.

자식 컨트롤의 도킹 방법 제어

참고 VCL에서는 드래그 앤 도킹 속성을 사용할 수 있으나 CLX에서는 사용할 수 없습니다.

도킹 공간은 그 위에서 마우스를 놓으면 자동적으로 자식 컨트롤을 받아들입니다. 대부분의 컨트롤의 경우, 첫 번째 자식 컨트롤은 클라이언트 영역을 채우면서 도킹되고 두 번째 자식 컨트롤은 클라이언트 영역을 분할하면서 도킹됩니다. Page 컨트롤은 자식 컨트롤을 새로운 탭 시트에 도킹합니다(자식 컨트롤이 다른 페이지 컨트롤인 경우에는 탭 시트 안에 합쳐집니다.).

도킹 공간에서 자식 컨트롤이 도킹되는 방법을 더 제어할 때 세 개의 이벤트를 사용할 수 있습니다.

```
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var InfluenceRect: TRect; var CanDock: Boolean) of object;
```

*OnGetSiteInfo*는 도킹 공간에서 컨트롤 위로 도킹 가능한 자식 컨트롤을 드래그할 때 발생합니다. 이는 공간이 *DockClient* 매개변수에서 지정된 컨트롤을 자식 컨트롤로 받아들일지 여부를 나타내고 자식 컨트롤로 받아들인다면 자식 컨트롤이 도킹될 공간이 어디인지를 나타냅니다. *OnGetSiteInfo*가 발생할 때 *InfluenceRect*는 도킹 공간의 화면 좌표로 초기화되고 *CanDock*은 *True*로 초기화됩니다. *InfluenceRect*를 변경하여

좀더 제한된 도킹 영역을 만들 수 있고 *CanDock*을 *False*로 설정하여 자식 컨트롤이 도킹되는 것을 거부할 수 있습니다.

```
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender:TObject; Source: TDragDockObject; X, Y: Integer; State:
TDragState; var Accept:Boolean) of object;
```

*OnDockOver*는 사용자가 컨트롤 위에 도킹 가능한 자식 컨트롤을 드래그할 때 도킹 공간에서 발생합니다. 이는 드래그 앤 드롭 연산의 *OnDragOver* 이벤트와 유사합니다. *Accept* 매개변수를 설정해서 자식 컨트롤이 도킹 가능하다는 것을 나타낼 수 있습니다. 도킹 가능한 컨트롤이 *OnGetSiteInfo* 이벤트 핸들러에 의해 거부된 경우(컨트롤이 잘못된 타입일 경우), *OnDockOver*는 발생하지 않습니다.

```
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender:TObject; Source: TDragDockObject; X, Y: Integer) of
object;
```

*OnDockDrop*은 사용자가 컨트롤 위로 도킹 가능한 자식 컨트롤을 드래그할 때 도킹 공간에서 발생합니다. 이는 드래그 앤 드롭 연산에서의 *OnDragDrop* 이벤트와 유사합니다. 이 이벤트를 이용해서 컨트롤을 자식 컨트롤로 받아들이는 데 필요한 일들을 처리합니다. *Source* 매개변수에 의해 지정된 *TDockObject*의 *Control* 속성을 이용해서 자식 컨트롤에 액세스할 수 있습니다.

자식 컨트롤이 도킹 해제되는 방법 제어

참고 VCL에서는 드래그 앤 도킹 속성을 사용할 수 있으나 CLX에서는 사용할 수 없습니다.

자식 컨트롤이 드래그되고 *DragMode* 속성 값이 *dmAutomatic*인 경우, 도킹 공간은 자식 컨트롤이 자동으로 도킹 해제될 수 있게 합니다. 자식 컨트롤이 드래그되어 빠져 나갈 때 도킹 공간은 *OnUnDock* 이벤트 핸들러에서 대응 행동을 할 수 있고 도킹 해제를 막을 수도 있습니다.

```
property OnUnDock: TUnDockEvent;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow:Boolean) of
object;
```

Client 매개변수는 도킹 해제를 시도하는 자식 컨트롤을 나타내고 *Allow* 매개변수는 도킹 공간(*Sender*)이 도킹 해제를 거부할 수 있게 해줍니다. *OnUnDock* 이벤트 핸들러 구현 시 다른 자식 컨트롤이 현재 도킹되어 있는지 여부를 아는 것이 유용할 수 있습니다. 이 정보는 *TControl*의 인덱스된 배열인 읽기 전용 *DockClients* 속성에서 알아낼 수 있습니다. 도킹된 클라이언트의 수는 읽기 전용 *DockClientCount* 속성으로 알 수 있습니다.

자식 컨트롤이 드래그 앤 드롭 연산에 응답하는 방법 제어

참고 VCL에서는 드래그 앤 도킹 속성을 사용할 수 있으나 CLX에서는 사용할 수 없습니다.

도킹 가능한 자식 컨트롤은 드래그 앤 드롭 연산 중에 발생하는 두 개의 이벤트를 갖습니다. *OnStartDock*은 드래그 앤 드롭 연산의 *OnStartDrag* 이벤트와 유사하며 도킹 가능한 자식 컨트롤이 사용자 지정 드래그 객체를 생성할 수 있게 합니다. *OnEndDrag*와 마찬가지로 *OnEndDock*은 드래깅이 끝날 때 발생합니다.

컨트롤에서 텍스트 사용

다음 단원에서는 서식있는 편집 (rich edit) 컨트롤과 메모 컨트롤의 다양한 기능을 사용하는 방법을 설명합니다. 이 기능 중 일부는 편집 컨트롤에도 적용됩니다.

- 텍스트 정렬 설정
- 런타임 시 스크롤 막대 추가
- 클립보드 객체 추가
- 텍스트 선택
- 모든 텍스트 선택
- 텍스트 잘라내기, 복사 및 붙여넣기
- 선택한 텍스트 삭제
- 메뉴 항목 사용 불가능
- 팝업 메뉴 제공
- OnPopup 이벤트 처리

텍스트 정렬 설정

서식있는 편집 (rich edit) 컴포넌트 또는 메모 컴포넌트에서 텍스트를 왼쪽 정렬, 오른쪽 정렬 또는 가운데 정렬할 수 있습니다. 텍스트 정렬을 변경하려면 편집 컴포넌트의 *Alignment* 속성을 설정합니다. 정렬은 *WordWrap* 속성이 *True*일 때에만 적용되며 금칙 처리가 해제되어 있으면 정렬할 여백이 없게 됩니다.

예를 들어, 다음 코드는 OnClick 이벤트 핸들러를 Character|Left 메뉴 항목에 연결하고 동일한 이벤트 핸들러를 Character 메뉴의 Right 및 Center 메뉴 항목에 연결시킵니다.

```
procedure TEditForm.AlignClick(Sender:TObject);
begin
  Left1.Checked := False; { clear all three checks }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True; { check the item clicked }
  with Editor do { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;
```

런타임 시 스크롤 막대 추가

필요한 경우 서식있는 편집(rich edit) 컴포넌트와 메모 컴포넌트에 수평 또는 수직 스크롤 막대를 포함시킬 수 있습니다. 금칙 처리가 활성화된 경우 컴포넌트에는 수직 스크롤 막대만 필요합니다. 사용자가 금칙 처리를 해제하면 해당 텍스트가 편집기의 오른쪽에 제한을 받지 않으므로 컴포넌트에 수평 스크롤 막대가 필요할 수도 있습니다.

런타임 시 스크롤 막대를 추가하려면 다음과 같이 합니다.

- 1 텍스트가 오른쪽 여백을 초과하는지 확인합니다. 대부분의 경우 이것은 금칙 처리의 사용 여부 확인을 의미합니다. 또한 텍스트 행이 실제로 컨트롤의 폭을 초과하는지 여부를 확인할 수도 있습니다.
- 2 서식있는 편집(rich edit) 컴포넌트 또는 메모 컴포넌트의 *ScrollBars* 속성에 스크롤 막대를 포함할지 여부를 설정합니다.

다음 예제에서는 Character|WordWrap 메뉴 항목에 *OnClick* 이벤트 핸들러를 연결합니다.

```

procedure TEditForm.WordWrap1Click(Sender:TObject);
begin
  with DataSet do
    begin
      WordWrap := not WordWrap; { toggle word-wrapping }
      if WordWrap then
        ScrollBars := ssVertical { wrapped requires only vertical }
      else
        ScrollBars := ssBoth; { unwrapped might need both }
        WordWrap1.Checked := WordWrap; { check menu item to match property }
      end;
    end;
  end;

```

서식있는 편집(rich edit) 컴포넌트와 메모 컴포넌트는 약간 다른 방식으로 스크롤 막대를 처리합니다. 서식있는 편집(rich edit) 컴포넌트는 텍스트가 컴포넌트의 영역 안에 모두 들어 간다면 스크롤 막대를 숨길 수 있습니다. 메모 컴포넌트는 스크롤 막대가 활성화되어 있으면 항상 스크롤 막대를 나타냅니다.

클립보드 객체 추가

텍스트를 다루는 대부분의 애플리케이션은 다른 애플리케이션의 문서를 비롯해서 선택한 텍스트를 문서 사이에서 이동하는 방법을 제공합니다. Delphi의 *Clipboard* 객체는 클립보드(예: Windows 클립보드)를 캡슐화하고 텍스트 잘라내기, 복사, 붙여넣기 기능(및 그래픽을 포함한 다른 형식)에 대한 메소드를 포함합니다. *Clipboard* 객체는 *Clipbrd* 유닛에 선언됩니다.

다음과 같은 방법으로 객체를 애플리케이션에 *Clipboard*를 추가합니다.

- 1 클립보드를 사용할 유닛을 선택합니다.
- 2 **implementation** 예약어를 찾습니다.
- 3 **implementation** 아래의 **uses** 절에 *Clipbrd*를 추가합니다.

- **implementation** 부분에 이미 **uses** 절이 있으면 그 끝에 *Clipbrd*를 추가합니다.
- **uses** 절이 없으면 다음을 추가합니다.

```
uses Clipbrd;
```

예를 들어, 자식 윈도우가 있는 애플리케이션에서 유닛의 **implementation** 부분의 **uses** 절은 다음과 같습니다.

```
uses
  MDIFrame, Clipbrd;
```

텍스트 선택

클립보드에 텍스트를 보내기 전에 해당 텍스트가 선택되어야 합니다. 선택한 텍스트의 선택 표시 기능은 편집 컴포넌트에 내장되어 있습니다. 사용자가 텍스트를 선택하면 선택 표시되어 나타납니다.

표 7.1은 선택한 텍스트를 처리하는 데 일반적으로 사용하는 속성을 나열한 것입니다.

표 7.1 선택한 텍스트의 속성

속성	설명
<i>SelectText</i>	컴포넌트 내에서 선택된 텍스트를 나타내는 문자열이 들어 있습니다.
<i>SelectLength</i>	선택한 문자열의 길이가 들어 있습니다.
<i>SelectStart</i>	문자열의 시작 위치가 들어 있습니다.

모든 텍스트 선택

SelectAll 메소드는 서식있는 편집 (rich edit) 컴포넌트 또는 메모 컴포넌트의 전체 내용을 선택합니다. 이 메소드는 컴포넌트에서 컴포넌트의 내용이 보이는 영역을 벗어날 때 특히 유용합니다. 대부분의 다른 경우에는 사용자가 키 입력이나 마우스를 끌어 텍스트를 선택합니다.

서식있는 편집 (rich edit) 컴포넌트 또는 메모 컴포넌트의 전체 내용을 선택하려면 *RichEdit1* 컨트롤의 *SelectAll* 메소드를 호출합니다.

예를 들면, 다음과 같습니다.

```
procedure TMainForm.SelectAll(Sender:TObject);
begin
  RichEdit1.SelectAll; { select all text in RichEdit }
end;
```

텍스트 잘라내기, 복사 및 붙여넣기

Clipbrd 유닛을 사용하는 애플리케이션은 클립보드를 통해 텍스트, 그래픽 및 객체를 잘라내고, 복사하고, 붙여넣을 수 있습니다. 표준 텍스트 처리 컨트롤을 캡슐화하는 편집 컴포넌트에는 모두 클립보드를 사용하는 데 필요한 메소드가 들어 있습니다. (클립보드에서 그래픽을 사용하는 방법에 대한 자세한 내용은 8-21 페이지의 "클립보드에서 그래픽 사용"을 참조하십시오.)

클립보드로 텍스트를 잘라내고, 복사하거나 붙여넣으려면 편집 컴포넌트의 *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 각각 호출합니다.

예를 들어, 다음 코드는 Edit|Cut, Edit|Copy 및 Edit|Paste 명령의 *OnClick* 이벤트에 이벤트 핸들러를 각각 연결합니다.

```
procedure TEditForm.CutToClipboard(Sender:TObject);
begin
    Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender:TObject);
begin
    Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender:TObject);
begin
    Editor.PasteFromClipboard;
end;
```

선택한 텍스트 삭제

선택한 텍스트를 클립보드에서 잘라내지 않고 편집 컴포넌트에서 삭제할 수 있습니다. 삭제하려면 *ClearSelection* 메소드를 호출합니다. 예를 들어 Edit 메뉴에 Delete 항목이 있을 경우 다음과 같이 코드를 작성합니다.

```
procedure TEditForm.Delete(Sender:TObject);
begin
    RichEdit1.ClearSelection;
end;
```

메뉴 항목 비활성화

종종 메뉴에서 메뉴 명령을 제거하지 않고 비활성화하는 것이 유용할 때가 있습니다. 예를 들어 텍스트 편집기에서 텍스트를 선택하지 않으면 Cut, Copy 및 Delete 명령을 적용할 수 없습니다. 메뉴 항목을 활성화 또는 비활성화하는 적절한 시기는 사용자가 메뉴를 선택할 때입니다. 메뉴 항목을 비활성화하려면 해당 *Enabled* 속성을 *False*로 설정합니다.

다음 예제에서 이벤트 핸들러는 자식 폼 메뉴 모음의 Edit 항목에서 *OnClick* 이벤트에 연결됩니다. *RichEdit1*에 선택한 텍스트가 있는지 여부에 따라 Edit 메뉴의 Cut, Copy 및 Delete 메뉴 항목에 *Enabled*를 설정합니다. Paste 명령은 클립보드에 텍스트가 있는지 여부에 따라 활성화 또는 비활성화됩니다.

```

procedure TEditForm.Edit1Click(Sender:TObject);
var
    HasSelection:Boolean; { declare a temporary variable }
begin
    Pastel.Enabled := Clipboard.HasFormat(CF_TEXT); {enable or disable the Paste
                                                    menu item}
    HasSelection := Editor.SelLength > 0; { True if text is selected }
    Cut1.Enabled := HasSelection; { enable menu items if HasSelection is True }
    Copy1.Enabled := HasSelection;
    Delete1.Enabled := HasSelection;
end;

```

클립보드의 *HasFormat* 메소드는 클립보드가 특정 형식의 객체, 텍스트 또는 이미지를 포함하는지 여부에 따라 부울 값을 반환합니다. *CF_TEXT* 매개변수를 갖는 *HasFormat* 을 호출하여 클립보드가 텍스트를 포함하는지 여부에 따라 Paste 메뉴 항목이 활성화 되거나 비활성화됩니다.

클립보드에서 그래픽을 사용하는 것에 대한 자세한 내용은 8장 "그래픽 및 멀티미디어 작업"에 있습니다.

팝업 메뉴 제공

팝업, 로컬, 메뉴는 애플리케이션의 사용하기 쉬운 일반적인 기능입니다. 이 기능을 사용하면 애플리케이션 작업 영역에서 마우스 오른쪽 버튼을 클릭하여 마우스 움직임 최소화할 수 있으므로 자주 사용하는 명령의 목록에 액세스할 수 있습니다.

예를 들어, 텍스트 편집기 애플리케이션에서 Cut, Copy 및 Paste 편집 명령을 반복하는 팝업 메뉴를 추가할 수 있습니다. 이 팝업 메뉴 항목에서 Edit 메뉴의 해당 항목과 동일한 이벤트 핸들러를 사용할 수 있습니다. 해당 메뉴 항목에 이미 단축키가 있으므로 팝업 메뉴에 대한 가속키나 단축키를 만들 필요가 없습니다.

폼의 *PopupMenu* 속성에서 사용자는 폼에서 항목을 마우스 오른쪽 버튼으로 클릭할 때 표시할 팝업 메뉴를 지정합니다. 각 컨트롤에도 특정 컨트롤에 대해 사용자 지정 메뉴를 허용하는 폼의 속성을 오버라이드할 수 있는 *PopupMenu* 속성이 있습니다.

다음과 같은 방법으로 폼에 팝업 메뉴를 추가합니다.

- 1 폼에 팝업 메뉴 컴포넌트를 놓습니다.
- 2 메뉴 디자이너를 사용하여 팝업 메뉴의 항목을 정의합니다.
- 3 메뉴를 팝업 메뉴 컴포넌트의 이름에 표시하는 폼이나 컨트롤의 *PopupMenu* 속성을 설정합니다.
- 4 팝업 메뉴 항목의 *OnClick* 이벤트에 핸들러를 연결합니다.

OnPopup 이벤트 처리

일반 메뉴에서 항목을 사용 가능 또는 불가능하게 하는 것처럼 메뉴를 표시하기 전에 팝업 메뉴 항목을 조정할 수 있습니다. 7-10 페이지의 "메뉴 항목 비활성화"에 설명된 것처럼 일반 메뉴에서, 메뉴 상단의 항목에서 *OnClick* 이벤트를 처리할 수 있습니다.

그러나 팝업 메뉴의 경우 최상위 레벨 메뉴 모음이 없으므로 팝업 메뉴 명령을 준비하려면 메뉴 컴포넌트 자체에서 이벤트를 처리해야 합니다. *OnPopup*이라는 팝업 메뉴 컴포넌트는 팝업만을 처리하기 위한 이벤트를 제공합니다.

다음과 같은 방법으로 메뉴 항목을 표시하기 전에 팝업 메뉴의 항목을 조정합니다.

- 1 팝업 메뉴 컴포넌트를 선택합니다.
- 2 해당 *OnPopup* 이벤트에 이벤트 핸들러를 연결합니다.
- 3 이벤트 핸들러에 메뉴 항목을 활성화, 비활성화, 숨기거나 보이게 하는 코드를 작성합니다.

다음 코드에서는 앞의 7-10 페이지의 "메뉴 항목 비활성화"에 설명되었던 *Edit1Click* 이벤트 핸들러가 팝업 메뉴 컴포넌트의 *OnPopup* 이벤트에 연결됩니다. 한 줄의 코드가 팝업 메뉴에 있는 각 항목에 대한 *Edit1Click*에 추가됩니다.

```
procedure TEditForm.Edit1Click(Sender:TObject);
var
    HasSelection:Boolean;
begin
    Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
    Paste2.Enabled := Paste1.Enabled;{Add this line}
    HasSelection := Editor.SelLength <> 0;
    Cut1.Enabled := HasSelection;
    Cut2.Enabled := HasSelection;{Add this line}
    Copy1.Enabled := HasSelection;
    Copy2.Enabled := HasSelection;{Add this line}
    Delete1.Enabled := HasSelection;
end;
```

컨트롤에 그래픽 추가

몇몇 컨트롤에서는 컨트롤이 렌더링되는 방법을 사용자 지정할 수 있습니다. 이러한 컨트롤에는 리스트 박스, 콤보 박스, 메뉴, 헤더, 탭 컨트롤, 리스트 뷰, 상태 표시줄, 트리 뷰 및 툴바 등이 있습니다. 컨트롤이나 해당 항목을 그리는 표준 메소드를 사용하는 대신 컨트롤의 소유자(일반적으로는 폼)가 런타임 시 컨트롤이나 항목을 그립니다. *owner-draw* 컨트롤은 항목에 대한 텍스트 대신 또는 텍스트에 추가해서 그래픽을 제공하는 데 가장 일반적으로 사용됩니다. 메뉴에 이미지를 추가하기 위한 *owner-draw* 사용 방법에 대한 자세한 내용은 6-36 페이지의 "메뉴 항목에 이미지 추가"를 참조하십시오.

모든 *owner-draw* 컨트롤은 항목 목록을 포함합니다. 일반적으로 이 목록들은 텍스트로 표시되는 문자열 목록이거나 텍스트로 표시되는 문자열을 포함하는 객체의 목록입니다.

니다. 객체를 목록 각각의 항목과 연결해서 항목을 그릴 때 해당 객체를 쉽게 사용할 수 있습니다.

일반적으로 Delphi에서 owner-draw 컨트롤을 만들려면 다음과 같은 단계를 따릅니다.

- 1 컨트롤이 owner-draw 항목임을 표시
- 2 문자열 목록에 그래픽 객체 추가
- 3 Owner-draw 항목 그리기

컨트롤이 owner-draw 항목임을 표시

컨트롤 그리기를 사용자 지정하려면 색칠해야 할 때 컨트롤의 이미지를 렌더링하는 이벤트 핸들러를 제공해야 합니다. 일부 컨트롤은 이러한 이벤트를 자동으로 받습니다. 예를 들어 리스트 뷰, 트리 뷰 및 톨바는 모두 사용자가 속성을 설정할 필요 없이 그리는 과정 중 여러 단계에서 이벤트를 받습니다. 이러한 이벤트는 "OnCustomDraw" 또는 "OnAdvancedCustomDraw"와 같은 이름을 갖습니다.

그러나 다른 컨트롤은 owner-draw 이벤트를 받기 전에 사용자가 속성을 설정해야 합니다. 리스트 박스, 콤보 박스, 헤더 컨트롤 및 상태 표시줄에는 *Style*이라는 속성이 있습니다. *Style*에서는 컨트롤에 "표준" 스타일이라고 하는 default drawing을 사용할지 owner drawing을 사용할지 여부를 결정합니다. 그리드는 *DefaultDrawing*이라는 속성을 사용하여 default drawing을 사용 가능 또는 사용 불가능하게 합니다. 리스트 뷰 및 탭 컨트롤은 default drawing을 사용 가능 또는 사용 불가능하게 하는 *OwnerDraw*라는 속성을 갖습니다.

리스트 박스와 콤보 박스에는 표 7.2가 설명하는 것처럼 *fixed*와 *variable*이라는 추가 owner-draw 스타일이 있습니다. 텍스트가 들어 있는 항목의 크기는 다양하지만 다른 컨트롤은 항상 고정되어 있으며 각 항목의 크기는 컨트롤을 그리기 전에 결정됩니다.

표 7.2 Fixed 와 variable owner-draw 스타일

Owner-draw 스타일	의미	예
Fixed	각 항목은 <i>ItemHeight</i> 속성에서 결정된 동일한 높이를 갖습니다.	<i>lbOwnerDrawFixed</i> , <i>csOwnerDrawFixed</i>
Variable	각 항목은 런타임 시 데이터에 의해 결정된 다른 높이를 가질 수도 있습니다.	<i>lbOwnerDrawVariable</i> , <i>csOwnerDrawVariable</i>

문자열 목록에 그래픽 객체 추가

모든 문자열 목록에는 해당 문자열 목록 외에 객체 목록을 포함하는 기능이 있습니다.

예를 들어, 파일 관리자 애플리케이션에서 드라이브 문자와 함께 드라이브의 형식을 나타내는 비트맵을 추가할 수 있습니다. 그렇게 하려면 애플리케이션에 비트맵 이미지를 추가한 후 다음 단원에 설명되어 있는 대로 이 이미지들을 문자열 목록의 적당한 위치에 복사합니다.

애플리케이션에 이미지 추가

이미지 컨트롤은 비트맵과 같은 그래픽 이미지가 들어 있는 논비주얼 (nonvisual) 컨트롤입니다. 이미지 컨트롤을 사용하면 폼에 그래픽 이미지를 표시할 수 있습니다. 또한 애플리케이션에서 사용할 숨겨진 이미지를 포착하는 데에도 사용할 수 있습니다. 예를 들어 다음과 같이 숨겨진 이미지 컨트롤에 owner-draw 컨트롤의 비트맵을 저장할 수 있습니다.

- 1 메인 폼에 이미지 컨트롤을 추가합니다.
- 2 *Name* 속성을 설정합니다.
- 3 각 이미지 컨트롤의 *Visible* 속성을 *False*로 설정합니다.
- 4 Object Inspector에서 Picture Editor를 사용하여 각 이미지의 *Picture* 속성을 원하는 비트맵으로 설정합니다.

애플리케이션을 실행하면 이미지 컨트롤은 보이지 않습니다.

문자열 목록에 이미지 추가

애플리케이션에 그래픽 이미지가 있다면 그래픽 이미지를 문자열 목록의 문자열에 연결할 수 있습니다. 객체를 문자열로 추가하는 동시에 기존 문자열에 객체를 연결할 수 있습니다. 더 좋은 방법은 필요한 데이터를 모두 사용할 수 있다면 객체와 문자열을 동시에 추가하는 것입니다.

다음 예제는 이미지를 문자열 목록에 추가하는 방법을 보여 줍니다. 이는 파일 관리자에 애플리케이션의 일부분입니다. 여기서 각각의 유효한 드라이브 문자와 함께 각 드라이브의 타입을 나타내는 비트맵을 추가합니다. 다음은 *OnCreate* 이벤트 핸들러입니다.

```

procedure TFMForm.FormCreate(Sender:TObject);
var
    Drive:Char;
    AddedIndex:Integer;
begin
    for Drive := 'A' to 'Z' do { iterate through all possible drives }
    begin
        case GetDriveType(Drive + ':/') of { positive values mean valid drives }
            DRIVE_REMOVABLE: {add a tab }
                AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
            DRIVE_FIXED: {add a tab }
                AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
            DRIVE_REMOTE: {add a tab }
                AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
        end;
        if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then { current drive? }
            DriveTabSet.TabIndex := AddedIndex; { then make that current tab }
        end;
    end;

```

Owner-draw 항목 그리기

속성을 설정하거나 사용자 지정 그리기 이벤트 핸들러를 제공하여 컨트롤이 owner-draw임을 나타내는 경우, 컨트롤은 더 이상 화면 상에 그려지지 않습니다. 그 대신 운영 체제는 컨트롤 안에 가시적 항목의 이벤트를 생성합니다. 애플리케이션이 항목을 그리기 위한 이벤트를 처리합니다.

owner-draw 컨트롤에 항목을 그리려면 컨트롤에 있는 각각의 가시적 항목에 대해 다음을 수행합니다. 모든 항목에 대해 하나의 이벤트 핸들러를 사용합니다.

1 필요하다면 항목의 크기를 조정합니다.

동일한 크기의 항목(예를 들어, *IsOwnerDrawFixed*의 리스트 박스 스타일)은 크기를 조정할 필요가 없습니다.

2 항목을 그립니다.

Owner-draw 항목 크기 조정

애플리케이션이 variable owner-draw 컨트롤에 있는 각 항목을 그리기 전에 운영 체제는 measure-item 이벤트를 생성합니다. measure-item 이벤트에서는 컨트롤에 나타나는 항목의 위치를 애플리케이션에서 표시합니다.

Delphi는 항목의 크기를 결정합니다. 일반적으로는 현재 글꼴로도 항목의 텍스트를 표시하기에는 충분합니다. 애플리케이션은 이벤트를 처리하고 선택된 사각형을 변경할 수 있습니다. 예를 들어 항목의 텍스트를 비트맵으로 대체하려는 경우 사각형을 비트맵의 크기로 변경합니다. 비트맵과 텍스트를 모두 나타내려면 사각형을 충분한 크기로 조정합니다.

owner-draw 항목의 크기를 변경하려면 owner-draw 컨트롤 내의 measure-item 이벤트에 이벤트 핸들러를 연결합니다. 컨트롤에 따라 이벤트 이름은 달라집니다. 리스트 박스와 콤보 박스는 *OnMeasureItem*을 사용합니다. 그리드에는 measure-item 이벤트가 없습니다.

크기 변경 이벤트에는 항목 인덱스 번호와 해당 항목의 크기라는 두 개의 중요한 매개변수가 있습니다. 크기는 조정이 가능합니다. 애플리케이션에서 높이를 작거나 크게 할 수 있습니다. 연속되는 항목의 위치는 선행 항목의 크기에 따라 다릅니다.

예를 들어 variable owner-draw 리스트 박스에서 애플리케이션이 첫 항목의 높이를 5픽셀로 설정하면 두 번째 항목은 위로부터 6픽셀 아래에서 시작합니다. 리스트 박스와 콤보 박스에서 애플리케이션이 변경할 수 있는 유일한 것은 항목의 높이입니다. 항목의 폭은 항상 컨트롤의 폭입니다.

Owner-draw 그리드는 그릴 때에는 셀의 크기를 변경할 수 없습니다. 각 행과 열의 크기는 *ColWidths*와 *RowHeights* 속성을 이용하여 그리기 전에 설정됩니다.

owner-draw 리스트 박스의 *OnMeasureItem* 이벤트에 연결된 다음 코드는 연결된 비트맵에 맞추기 위해 각 목록 항목의 높이를 늘립니다.

```

procedure TFMForm.DriveTabSetMeasureTab(Sender:TObject; Index:Integer;
var TabWidth:Integer); { note that TabWidth is a var parameter}
var
    BitmapWidth:Integer;
begin
    BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
    { increase tab width by the width of the associated bitmap plus two }
    Inc(TabWidth, 2 + BitmapWidth);
end;

```

참고 문자열 목록의 *Objects* 속성에서 항목을 타입 변환해야 합니다. *Objects*는 *TObject* 타입의 속성으로 모든 형식의 객체를 포함할 수 있습니다. 배열에서 객체를 검색할 때 객체를 항목의 실제 타입으로 타입 변환해야 합니다.

Owner-draw 항목 그리기

애플리케이션이 owner-draw 컨트롤을 그리거나 다시 그려야 할 필요가 있을 때 Windows는 각 컨트롤의 가시적 항목의 draw-item 이벤트를 생성합니다. 컨트롤에 따라 항목은 전체 또는 하위 항목으로 해당 항목의 그리기 이벤트를 받을 수도 있습니다.

Owner-draw 컨트롤에서 각 항목을 그리려면 해당 컨트롤의 draw-item 이벤트에 이벤트 핸들러를 연결합니다.

일반적으로 Owner drawing의 이벤트 이름은 다음 중 하나로 시작합니다.

- *OnDrawItem*이나 *OnDrawCell*과 같은 *OnDraw*
- *OnCustomDrawItem*과 같은 *OnCustomDraw*
- *OnAdvancedCustomDrawItem*과 같은 *OnAdvancedCustomDraw*

Draw-item 이벤트에는 그릴 항목을 식별하는 매개변수, 그려 넣을 사각형 및 항목에 포커스가 있는지 여부와 같은 항목의 상태에 관한 몇 가지 정보가 들어 있습니다. 애플리케이션은 주어진 사각형 내에서 적절한 항목을 렌더링하여 각 이벤트를 처리합니다.

예를 들어 다음 코드는 각 문자열과 연결된 비트맵이 있는 리스트 박스에 항목을 그리는 방법을 표시합니다. 다음과 같이 리스트 박스의 *OnDrawItem* 이벤트에 이 핸들러를 연결합니다.

```

procedure TFMForm.DriveTabSetDrawTab(Sender:TObject; TabCanvas:TCanvas;
    R:TRect; Index:Integer; Selected:Boolean);
var
    Bitmap:TBitmap;
begin
    Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
    with Canvas do
        begin
            Draw(R.Left, R.Top + 4, Bitmap); { draw bitmap }
            TextOut(R.Left + 2 + Bitmap.Width, { position text }
                R.Top + 2, DriveTabSet.Tabs[Index]); { and draw it to the right of the
                bitmap}
        end;
    end;

```

8

그래픽 및 멀티미디어 작업

그래픽과 멀티미디어 요소로 애플리케이션을 꾸밀 수 있습니다. Delphi는 이러한 기능을 애플리케이션에 도입할 수 있는 다양한 방법을 제공합니다. 그래픽 요소를 추가하려면 디자인 타임에 미리 그려 놓은 그림을 삽입하거나 그래픽 컨트롤을 사용하여 만들거나 런타임 시 동적으로 그릴 수 있습니다. 멀티미디어 기능을 추가하기 위해 Delphi는 오디오와 비디오 클립을 재생할 수 있는 특수한 컴포넌트를 포함합니다. 크로스 플랫폼 프로그래밍에는 멀티미디어 컴포넌트를 사용할 수 없다는 것에 유의하십시오.

그래픽 프로그래밍 개요

Graphics 유닛에 정의된 VCL 그래픽 컴포넌트는 Windows GDI(Graphics Device Interface)를 캡슐화하여 Windows 애플리케이션에 그래픽을 쉽게 추가할 수 있습니다. QGraphics 유닛에 정의된 CLX 그래픽 컴포넌트는 크로스 플랫폼 애플리케이션에 그래픽을 추가하기 위한 Qt 그래픽 widget을 캡슐화합니다.

Delphi 애플리케이션에서 그래픽을 그리려면 객체 위에 직접 그리지 않고 객체의 *캔버스*에 그립니다. 캔버스는 객체의 속성이면서 그 자체가 객체이기도 합니다. 캔버스 객체의 주요 이점은 리소스를 효과적으로 처리하고 장치 컨텍스트를 관리하므로 프로그램은 사용자가 화면, 프린터 또는 비트맵이나 메타파일(CLX의 드로잉)의 어느 곳에 그림을 그리는지 상관 없이 동일한 메소드를 사용할 수 있다는 것입니다. 캔버스는 런타임 시에만 사용할 수 있으므로 코드를 작성하면 캔버스에서 모든 작업을 할 수 있습니다.

VCL 참고 *TCanvas*는 Windows 장치 컨텍스트에 대한 래퍼(wrapper) 리소스 관리자이기 때문에 사용자는 또한 캔버스의 모든 Windows GDI 함수를 사용할 수 있습니다. 캔버스의 *Handle* 속성은 장치 컨텍스트 핸들입니다.

CLX 참고 *TCanvas*는 Qt painter에 대한 래퍼(wrapper) 리소스 관리자입니다. 캔버스의 *Handle* 속성은 Qt painter 객체의 인스턴스에 대한 타입이 지정된 포인터입니다. 이 인스턴스 포인터를 드러내면 painter 객체에 대한 인스턴스 포인터를 필요로 하는 낮은 Qt 그래픽 라이브러리 기능을 사용할 수 있습니다.

애플리케이션에 그래픽 이미지가 나타나는 방식은 그리는 캔버스의 객체 형식에 따라 다릅니다. 컨트롤의 캔버스에 직접 그리면 그림은 즉시 표시됩니다. 그러나 *TBitmap* 캔버스와 같은 오프스크린 이미지에서 그리면 경우는 이미지는 컨트롤이 비트맵에서 컨트롤의 캔버스로 복사되어야 표시됩니다. 즉, 비트맵을 그려 이미지 컨트롤에 할당할 때 이미지는 컨트롤이 해당 *OnPaint* 메시지 (VCL) 또는 이벤트 (CLX) 를 처리할 수 있을 때에만 나타납니다.

그래픽 작업을 사용하다 보면 *그리기 (Drawing)*와 *색칠 (Painting)*이라는 용어가 자주 나옵니다.

- 그리기는 코드를 가지고 선이나 도형과 같은 특정 단일 그래픽 요소를 만드는 것입니다. 코드에서 캔버스의 그리기 메소드를 호출하면 객체에게 캔버스의 특정 위치에 특정 그래픽을 그릴 것을 지시합니다.
- 색칠은 객체의 전체 모양을 만드는 것입니다. 색칠은 일반적으로 그리기를 포함합니다. 즉, *OnPaint* 이벤트의 응답으로 객체는 일반적으로 일부 그래픽을 그립니다. 예를 들어, 편집 상자는 사각형을 그려 자신을 색칠한 다음 내부에 일부 텍스트를 그립니다. 반면 도형 컨트롤은 단일 그래픽을 그려서 자신을 색칠합니다.

이 장의 처음에 나오는 예제는 다양한 그래픽을 그리는 방법을 보여 주지만 이러한 작업은 *OnPaint* 이벤트에 대한 응답을 수행하는 것입니다. 후반부에는 다른 이벤트에 응답하여 같은 종류의 그리기를 수행하는 방법을 보여 줍니다.

화면 새로 고침

간혹 운영 체제에서 화면 상의 객체 모습을 새로 고쳐야 한다고 결정하여 VCL 이 *OnPaint* 이벤트로 라우팅되는 Windows의 WM_PAINT 메시지를 생성합니다. (크로스 플랫폼 개발에서 CLX를 사용하는 경우, CLX가 *OnPaint* 이벤트로 라우팅되는 페인트 이벤트가 생성됩니다.) 해당 객체에 대한 *OnPaint* 이벤트 핸들러를 작성한 경우, 이 핸들러는 *Refresh* 메소드를 사용할 때 호출됩니다. 폼의 *OnPaint* 이벤트 핸들러에 생성된 기본 이름은 *FormPaint*입니다. 경우에 따라 *Refresh* 메소드를 사용하여 컴포넌트나 폼을 새로 고칠 수도 있습니다. 예를 들어, 폼의 *OnResize* 이벤트 핸들러에서 *Refresh*를 호출하여 모든 그래픽을 다시 표시할 수 있으며 또는 VCL을 사용하는 경우에는 폼에 백그라운드를 그릴 수 있습니다.

일부 운영 체제에서는 유효하지 않은 창의 클라이언트 영역 다시 그리기를 자동으로 처리하지만 Windows는 그렇지 않습니다. Windows 운영 체제에서는 화면에 그려진 모든 것은 영구적입니다. 예를 들면, 창을 드래그하는 동안 폼이나 컨트롤이 일시적으로 흐려지면 해당 폼이나 컨트롤이 다시 노출될 때 흐려진 영역을 다시 그려야 합니다. WM_PAINT 메시지에 대한 자세한 내용은 Windows 온라인 도움말을 참조하십시오.

TImage 컨트롤을 사용하여 폼에 그래픽 이미지를 표시하는 경우 *TImage*에 들어 있는 그래픽의 색칠 및 새로 고침은 CLX에 의해 자동으로 처리됩니다. *Picture* 속성은 *TImage*가 표시하는 실제 비트맵, 그리기 또는 다른 그래픽 객체를 지정합니다. 또한 *Proportional* 속성을 설정하면 이미지가 왜곡되지 않게 이미지 컨트롤에 전체를 표시할 수 있습니다. *TImage*의 그리기를 통해 영구적인 이미지를 만듭니다. 따라서 포함된 이미지를 다시 그릴 필요가 없습니다. 이와 대조적으로 *TPaintBox*의 캔버스는 화면 장치 (VCL) 나 painter (CLX)에 직접 매핑되어 *PaintBox*의 캔버스에 그려진 모든 그림은

일시적으로 그려진 것입니다. 폼 자체를 비롯한 거의 모든 컨트롤 또한 일시적으로 그려진 것입니다. 따라서 해당 생성자에 있는 *TPaintBox* 상에서 그리거나 색칠하는 경우 클라이언트 영역이 바뀔 때마다 이미지가 다시 칠해지도록 *OnPaint* 이벤트 핸들러에 해당 코드를 추가해야 합니다.

그래픽 객체 형식

VCL/CLX는 표 8.1에 표시된 대로 그래픽 객체를 제공합니다. 이러한 객체는 8-10 페이지의 "캔버스 메소드를 사용하여 그래픽 객체 그리기"에 설명되어 있는 캔버스에 그릴 수 있고 8-19 페이지의 "그래픽 파일 로드 및 저장"에서 설명한 대로 그래픽 파일로 로드 및 저장할 수 있는 메소드를 가집니다.

표 8.1 그래픽 객체 형식

객체	설명
그림	모든 그래픽 이미지를 갖기 위해 사용합니다. 추가 그래픽 파일 형식을 추가하려면 <i>Picture Register</i> 메소드를 사용합니다. 이 메소드를 사용하면 이미지 컨트롤에서 이미지를 표시하는 것과 같이 임의의 파일을 처리할 수 있습니다.
비트맵	이미지를 작성, 처리(크기 변경, 스크롤, 회전 및 색칠) 및 디스크에 파일로 저장하는 데 사용하는 강력한 그래픽 객체입니다. 비트맵의 사본을 만드는 것은 이미지가 아닌 <i>처리</i> 가 복사되는 것이므로 빠릅니다.
클립보드	애플리케이션에서 또는 애플리케이션으로 잘라내거나 복사하거나 붙여넣은 텍스트나 그래픽의 컨테이너를 나타냅니다. 클립보드를 사용하면 적절한 형식에 따라 데이터를 얻고 검색하고, 참조 카운팅을 처리하며, 클립보드를 열거나 닫고, 클립보드 내의 객체 형식을 관리하고 처리할 수 있습니다.
아이콘	아이콘 파일에서 로드된 값을 나타냅니다(::ICO 파일).
메타파일(VCL 전용) 드로잉(CLX 전용)	이미지의 실제 비트맵 픽셀이 들어 있는 것이 아니라 이미지를 생성하는 데 필요한 작업을 기록하는 파일이 들어 있습니다. 메타파일이나 드로잉은 이미지의 상세 정보를 잃지 않고 크기를 조정할 수 있으며 특히 프린터와 같은 고해상도 장치에서는 비트맵보다 훨씬 적은 메모리를 필요로 하기도 합니다. 하지만 메타파일과 드로잉은 비트맵만큼 빨리 표시되지 않습니다. 성능보다 유연성과 정밀성이 더 중요할 때는 메타파일이나 드로잉을 사용하십시오.

캔버스의 일반적인 속성과 메소드

표 8.2는 캔버스 객체에서 공통적으로 사용되는 속성을 나열한 것입니다. 속성과 메소드의 전체 목록은 온라인 도움말의 *TCanvas* 컴포넌트를 참조하십시오.

표 8.2 캔버스 객체의 일반적인 속성

속성	설명
Font	이미지 위에 텍스트를 작성할 때 사용할 글꼴을 지정합니다. TFont 객체의 속성을 설정하여 글꼴, 글꼴 색, 크기 및 스타일을 지정합니다.
Brush	캔버스가 그래픽 모양과 배경을 채우는 데 사용하는 색상과 패턴을 결정합니다. TBrush 객체의 속성을 설정하여 캔버스의 공간을 채울 때 사용할 색상, 패턴 또는 비트맵을 지정합니다.
Pen	캔버스가 선과 윤곽 모양을 그리는 데 사용할 펜의 종류를 지정합니다. TPen 객체의 속성을 설정하여 펜의 색상, 스타일, 너비 및 모드를 지정합니다.
PenPos	펜의 현재 그리기 위치를 지정합니다.
Pixels	현재 ClipRect 내의 픽셀 영역 색상을 지정합니다.

이 속성은 8-5 페이지의 "캔버스 객체의 속성 사용"에 더 자세히 설명되어 있습니다.

표 8.3은 사용 가능한 메소드의 목록입니다.

표 8.3 캔버스 객체의 일반적인 속성

메소드	설명
Arc	지정된 사각형으로 경계를 이루는 타원의 둘레를 따라 이미지에 원호를 그립니다.
Chord	선과 타원의 교차점으로 표시되는 닫힌 그림을 그립니다.
CopyRect	다른 캔버스에서 현재 캔버스로 이미지의 부분을 복사합니다.
Draw	좌표(X, Y)에 의해 지정된 위치의 캔버스에 Graphic 매개변수에 의해 지정된 그래픽 객체를 렌더링합니다.
Ellipse	캔버스에 경계를 이루는 사각형에 의해 정의된 타원을 그립니다.
FillRect	현재 브러시를 사용하여 캔버스에 지정된 사각형을 채웁니다.
FloodFill(VCL 전용)	현재 브러시를 사용하여 캔버스 영역을 채웁니다.
FrameRect	경계를 그리기 위해 캔버스의 Brush를 사용하여 사각형을 그립니다.
LineTo	캔버스에 PenPos에서 X와 Y로 지정된 지점까지 선을 그리고 펜 위치를 (X, Y) 지점으로 설정합니다.
MoveTo	현재 그리기 위치를 (X,Y) 지점으로 변경합니다.
Pie	캔버스의 (X1, Y1)과 (X2, Y2) 사각형에 의해 경계를 이루는 파이 모양의 타원 색션을 그립니다.
Polygon	마지막 지점에서 첫 번째 지점까지 선을 그려서 통과 지점을 연결하고 모양을 닫는 일련의 선을 캔버스에 그립니다.
Polyline	현재 펜으로 캔버스 위에 Points에 통과한 각 지점을 연결시키는 일련의 선을 그립니다.

표 8.3 캔버스 객체의 일반적인 속성 (계속)

메소드	설명
Rectangle	왼쪽 위 모서리 (X1, Y1) 점과 우측 아래 모서리의 (X2, Y2) 점으로 이루어진 사각형을 캔버스에 그립니다. 펜을 사용하여 <i>Rectangle</i> 로 상자를 그리고 브러시로 상자를 채웁니다.
RoundRect	캔버스에 모서리가 둥근 사각형을 그립니다.
StretchDraw	이미지가 지정된 사각형에 맞도록 캔버스에 그래픽을 그립니다. 그래픽 이미지의 크기나 가로 세로 비율을 알맞게 변경해야 할 수도 있습니다.
TextHeight, TextWidth	현재 글꼴로 된 문자열의 높이와 너비를 각각 반환합니다. 높이에는 줄 간격이 포함됩니다.
TextOut	(X,Y) 지점에서 시작하는 문자열을 캔버스에 작성하고 PenPos를 문자열의 끝으로 업데이트합니다.
TextRect	영역 내에 문자열을 작성하며 영역 밖으로 벗어난 문자열은 나타나지 않습니다.

이 메소드는 8-10 페이지의 "캔버스 메소드를 사용하여 그래픽 객체 그리기"에 더 자세히 설명되어 있습니다.

캔버스 객체의 속성 사용

캔버스 객체를 사용하여 선을 그리는 펜, 모양을 채우는 브러시, 텍스트를 작성하는 글꼴, 이미지를 나타내는 픽셀의 배열 속성을 설정할 수 있습니다.

이 단원에서는 다음을 설명합니다.

- 펜 사용
- 브러시 사용
- 픽셀 읽기 및 설정

펜 사용

캔버스 컨트롤의 *Pen* 속성은 모양의 윤곽으로 그려진 선을 비롯하여 선이 나타나는 방법을 제어합니다. 직선을 그리는 것은 두 지점 간에 있는 픽셀 그룹을 변경하는 것입니다.

펜에는 사용자가 변경할 수 있는 *Color*, *Width*, *Style* 및 *Mode*와 같은 네 가지 속성이 있습니다.

- *Color* 속성: 펜 색상 변경
- *Width* 속성: 펜 너비 변경
- *Style* 속성: 펜 스타일 변경
- *Mode* 속성: 펜 모드 변경

이 속성 값으로 펜이 선의 픽셀을 변경하는 방법을 결정합니다. 기본적으로 모든 펜은 1 픽셀의 너비, 실선 스타일 및 캔버스에 이미 존재하는 모든 것을 오버라이드하는 copy 라는 모드를 가지고 검정으로 시작합니다.

*TPenRecall*을 사용하여 펜의 속성을 빠르게 저장 및 복원할 수 있습니다.

펜 색상 변경

런타임 시 다른 *Color* 속성을 설정하는 것처럼 색상을 설정할 수 있습니다. 펜 색상으로 다른 선과 다각선뿐만 아니라 도형의 경계로 그려진 선을 비롯하여 펜이 그리는 선의 색상을 결정합니다. 펜 색상을 변경하려면 펜의 *Color* 속성에 값을 할당합니다.

사용자가 펜의 새 색상을 선택하도록 하려면 펜의 툴바에 색상 그리드를 넣습니다. 색상 그리드를 사용하여 전경 색과 배경색을 모두 설정할 수 있습니다. 그리드가 없는 펜 스타일의 경우 선분 간의 틈에 그려진 배경 색을 고려해야 합니다. 배경 색은 Brush 색상 속성에 있습니다.

사용자는 그리드를 클릭하여 새 색상을 선택하므로 다음 코드는 *OnClick* 이벤트에 응답하여 펜 색상을 변경합니다.

```
procedure TForm1.PenColorClick(Sender:TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

펜 너비 변경

펜 너비는 펜이 그리는 선의 두께를 픽셀로 결정합니다.

참고 두께가 1 이상인 경우, Windows 95/98은 펜의 *Style* 속성 값과는 상관 없이 항상 실선을 그립니다.

펜 너비를 변경하려면 펜의 *Width* 속성에 숫자로 값을 할당합니다.

펜의 너비 값을 설정하기 위해 펜의 툴바에 스크롤 막대를 넣는다고 가정합니다. 그리고 사용자에게 피드백을 제공하기 위해 스크롤 막대 옆에 있는 레이블을 업데이트한다고 가정합니다. 스크롤 막대의 위치를 사용하여 펜 너비를 결정하면 위치가 변경될 때마다 펜 너비를 업데이트합니다.

스크롤 막대의 *OnChange* 이벤트를 처리하는 방법은 다음과 같습니다.

```
procedure TForm1.PenWidthChange(Sender:TObject);
begin
  Canvas.Pen.Width := PenWidth.Position;{ set the pen width directly }
  PenSize.Caption := IntToStr(PenWidth.Position);{ convert to string for caption }
end;
```

펜 스타일 변경

펜의 *Style* 속성을 통해 실선, 점쇄선, 점선 등을 설정할 수 있습니다.

VCL 참고 Windows에서 배포를 위한 크로스 플랫폼 애플리케이션을 개발하는 경우, Windows 95/98은 사용자가 지정하는 스타일에 관계 없이 1픽셀이 넘는 점괵선이나 점선을 지원하지 않고 그보다 너비가 넓은 모든 펜은 실선으로 처리합니다.

펜 속성을 설정하는 작업은 이벤트를 처리하기 위해 서로 다른 컨트롤이 동일한 이벤트 핸들러를 공유하도록 하는 경우 이상적입니다. 어느 컨트롤이 실제로 이벤트를 갖는지 알려면 *Sender* 매개변수를 확인합니다.

펜의 톨바에 여섯 개의 펜 스타일 버튼에 대한 하나의 클릭 이벤트 핸들러를 만들려면 다음을 수행하십시오.

- 1 여섯 개의 펜 스타일 버튼을 모두 선택하고 Object Inspector | Events | *OnClick* 이벤트를 선택한 다음 Handler 열에 *SetPenStyle*을 입력합니다.

Delphi는 *SetPenStyle*이라고 하는 빈 클릭 이벤트 핸들러를 생성하여 이를 모든 6개 버튼의 *OnClick* 이벤트에 연결합니다.

- 2 클릭 이벤트를 보낸 컨트롤인 *Sender* 값에 따라 다른 펜 스타일을 설정하여 클릭 이벤트 핸들러를 다음과 같은 방법으로 채웁니다.

```
procedure TForm1.SetPenStyle(Sender:TObject);
begin
  with Canvas.Pen do
  begin
    if Sender = SolidPen then Style := psSolid
    else if Sender = DashPen then Style := psDash
    else if Sender = DotPen then Style := psDot
    else if Sender = DashDotPen then Style := psDashDot
    else if Sender = DashDotDotPen then Style := psDashDotDot
    else if Sender = ClearPen then Style := psClear;
  end;
end;
```

펜 모드 변경

펜의 *Mode* 속성을 사용하여 펜 색상과 캔버스 색상을 결합하기 위해 다양한 방법을 지정할 수 있습니다. 예를 들어, 펜이 항상 검은색이거나 캔버스 배경 색의 반대 색이 되거나 펜 색상의 반대 색이 되도록 설정할 수 있습니다. 자세한 내용은 온라인 도움말의 *TPen*을 참조하십시오.

펜 위치 파악

펜이 다음 선을 그리기 시작하는 위치인 현재 그리기 위치를 펜 위치라고 합니다. 캔버스는 펜 위치를 *PenPos* 속성에 저장합니다. 펜 위치는 선 그리기에만 영향을 미치며 도형과 텍스트의 경우 필요한 모든 좌표를 사용자가 입력합니다.

펜 위치를 설정하려면 캔버스의 *MoveTo* 메소드를 호출합니다. 예를 들어, 다음 코드를 사용하여 펜 위치를 캔버스의 왼쪽 위 모서리로 이동합니다.

```
Canvas.MoveTo(0, 0);
```

참고 *LineTo* 메소드로 선을 그려도 현재 위치가 선의 끝점으로 이동합니다.

브러시 사용

캔버스의 *Brush* 속성은 도형의 내부를 포함하여 영역을 채우는 방법을 제어합니다. 브러시로 영역을 채우는 것은 지정된 방법으로 인접한 많은 픽셀을 변경하는 방법입니다. 브러시에는 사용자가 처리할 수 있는 다음과 같은 세 가지 속성이 있습니다.

- Color 속성: 채우기 색상 변경
- Style 속성: 브러시 스타일 변경
- Bitmap 속성: 비트맵을 브러시 패턴으로 사용

이 속성 값으로 캔버스가 도형이나 다른 영역을 채우는 방법을 결정합니다. 기본적으로 모든 브러시는 흰색의 실선 스타일로 시작하며 패턴 비트맵을 사용하지 않습니다.

*TBrushRecall*을 사용하여 브러시의 속성을 빠르게 저장 및 복원할 수 있습니다.

브러시 색상 변경

브러시 색상은 캔버스가 도형을 채우는 데 사용할 색상을 결정합니다. 채우기 색상을 변경하려면 브러시의 *Color* 속성에 값을 할당합니다. 브러시는 텍스트의 배경 색과 선 그리기에 사용되므로 일반적으로 배경 색 속성을 설정합니다.

브러시 툴바의 색상 그리드를 클릭하면, 펜 색상과 동일한 방법으로 브러시 색상을 설정할 수 있습니다(8-6 페이지의 "펜 색상 변경" 참고).

```
procedure TForm1.BrushColorClick(Sender:TObject);
begin
    Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

브러시 스타일 변경

브러시 스타일은 캔버스가 도형을 채우는 데 사용하는 패턴을 결정합니다. 브러시 스타일을 사용하여 브러시 색상과 캔버스에 이미 있는 색상을 결합하기 위해 다양한 방법을 지정할 수 있습니다. 이미 정의된 스타일에는 단색, 무색 및 다양한 선과 해칭 패턴이 있습니다.

브러시 스타일을 변경하려면 해당 *Style* 속성을 *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross* 또는 *bsDiagCross* 등 이미 정의된 값 중 하나로 설정합니다.

이 예제에서는 8개의 브러시 스타일 버튼 집합에 대한 클릭 이벤트 핸들러를 공유하여 브러시 스타일을 설정합니다. 8개의 버튼을 모두 선택하고 Object Inspector|Events|*OnClick*으로 설정한 다음 *OnClick* 핸들러의 이름을 *SetBrushStyle*로 지정합니다. 핸들러 코드는 다음과 같습니다.

```
procedure TForm1.SetBrushStyle(Sender:TObject);
begin
    with Canvas.Brush do
    begin
        if Sender = SolidBrush then Style := bsSolid
        else if Sender = ClearBrush then Style := bsClear
```

```

else if Sender = HorizontalBrush then Style := bsHorizontal
else if Sender = VerticalBrush then Style := bsVertical
else if Sender = FDiagonalBrush then Style := bsFDiagonal
else if Sender = BDiagonalBrush then Style := bsBDiagonal
else if Sender = CrossBrush then Style := bsCross
else if Sender = DiagonalCrossBrush then Style := bsDiagCross;
end;
end;

```

브러시의 비트맵 속성 설정

브러시의 *Bitmap* 속성을 사용하여 도형 및 다른 영역을 채우기 위한 패턴으로 사용하는 브러시의 비트맵 이미지를 지정합니다.

다음 예제는 파일에서 비트맵을 로드하고 비트맵을 Form1 Canvas의 Brush에 할당합니다.

```

var
  Bitmap:TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end;

```

참고 브러시는 자신의 *Bitmap* 속성에 할당된 비트맵 객체의 소유권을 가지고 있다고 가정하지 않습니다. 브러시를 사용하는 동안에는 비트맵 객체가 유효한 상태로 있는지 확인해야 하고 나중에 비트맵 객체를 해제해야 합니다.

픽셀 읽기 및 설정

모든 캔버스에는 인덱스화된 *Pixels* 속성이 있어서 캔버스에 이미지를 만드는 색깔이 칠해진 각각의 점을 나타냅니다. *Pixels*에 직접 액세스할 필요는 거의 없으며 이 속성은 픽셀의 색상을 찾거나 설정하는 것과 같은 작은 동작을 편리하게 수행하는 경우에만 사용됩니다.

참고 각각의 픽셀을 설정하고 가져오는 일은 지역 단위의 그래픽 작업을 수행하는 것보다 속도가 수천 배 느립니다. Pixel 배열 속성을 사용하여 일반 배열의 이미지 픽셀에 액세스하지 마십시오. 이미지 픽셀에 대한 고성능의 액세스는 *TBitmap.ScanLine* 속성을 참조하십시오.

캔버스 메소드를 사용하여 그래픽 객체 그리기

이 단원에서는 일반적인 일부 메소드를 사용하여 그래픽 객체를 그리는 방법을 설명합니다. 다음과 같은 내용을 다룹니다.

- 선 및 다각선 그리기
- 도형 그리기
- 모서리가 둥근 사각형 그리기
- 다각형 그리기

선 및 다각선 그리기

캔버스에서는 직선과 다각선을 그릴 수 있습니다. 직선은 두 점을 연결하는 픽셀의 선입니다. 다각선은 끝과 끝이 연결된 일련의 직선입니다. 캔버스에서는 펜을 사용하여 모든 선을 그립니다.

선 그리기

캔버스에서 직선을 그리려면 캔버스의 *LineTo* 메소드를 사용합니다.

*LineTo*는 현재의 펜 위치에서 사용자가 지정하는 펜 위치까지 선을 그리고 선의 끝점을 현재 위치로 나타냅니다. 캔버스에서는 펜을 사용하여 선을 그립니다.

예를 들어, 다음 메소드는 폼을 색칠할 때마다 폼 전반에 교차된 대각선을 그립니다.

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  with Canvas do
  begin
    MoveTo(0, 0);
    LineTo(ClientWidth, ClientHeight);
    MoveTo(0, ClientHeight);
    LineTo(ClientWidth, 0);
  end;
end;
```

다각선 그리기

개별적인 선 외에도 캔버스에서는 연결된 선분의 그룹인 다각선을 그릴 수 있습니다.

캔버스에 다각선을 그리려면 캔버스의 *Polyline* 메소드를 호출합니다.

Polyline 메소드에 전달된 매개변수는 점의 배열입니다. 다각선이 첫 번째 점에서 *MoveTo*를 수행하고 각 연속된 점에서는 *LineTo*를 수행하는 경우를 생각해 볼 수 있습니다. 여러 선을 그리는 경우 *Polyline* 메소드를 사용하면 *MoveTo* 메소드와 *LineTo* 메소드를 사용하는 것보다 더 빨리 그립니다. 그 이유는 많은 호출 오버헤드를 제거하기 때문입니다.

예를 들어, 다음 메소드는 폼에 마름모꼴을 그립니다.

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  with Canvas do
    Polyline([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
  end;
```

위의 예제는 개방형 배열 매개변수를 즉시 만드는 Delphi의 기능을 이용합니다. 어떤 점의 배열이라도 전달할 수 있지만 빠르게 배열을 구성할 수 있는 쉬운 방법은 해당 요소를 괄호 안에 넣고 전체를 매개변수로 전달하는 것입니다. 자세한 내용은 온라인 도움말을 참조하십시오.

도형 그리기

캔버스에는 다른 종류의 도형을 그리는 데 사용하는 메소드가 있습니다. 캔버스에서는 펜으로 도형의 윤곽을 그린 다음 브러시로 내부를 채웁니다. 도형의 테두리를 형성하는 선은 현재 *Pen* 객체에 의해 제어됩니다.

이 단원에서는 다음과 같은 내용을 다룹니다.

- 사각형 및 타원 그리기
- 모서리가 둥근 사각형 그리기
- 다각형 그리기

사각형 및 타원 그리기

캔버스에 사각형이나 타원을 그리려면 경계를 이루는 사각형의 좌표를 전달하는 캔버스의 *Rectangle* 메소드나 *Ellipse* 메소드를 호출합니다.

Rectangle 메소드는 경계를 이루는 사각형을 그리고 *Ellipse* 메소드는 사각형의 모든 면에 닿는 타원을 그립니다.

다음 메소드는 폼의 왼쪽 위 사분면을 채우는 사각형을 그린 다음 동일한 영역에 타원을 그립니다.

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
  Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

모서리가 둥근 사각형 그리기

캔버스에 모서리가 둥근 사각형을 그리려면 캔버스의 *RoundRect* 메소드를 호출합니다.

*RoundRect*에 전달된 첫 번째 네 개의 매개변수는 *Rectangle* 메소드나 *Ellipse* 메소드의 경우처럼 경계를 이루는 사각형입니다. *RoundRect*는 둥근 모서리를 그리는 방법을 나타내는 매개변수를 두 개 더 가집니다.

예를 들어, 다음 메소드는 지름이 10픽셀인 원처럼 모서리를 둥글게 하면서 폼의 왼쪽 위 사분면에 모서리가 둥근 사각형을 그립니다.

```

procedure TForm1.FormPaint(Sender:TObject);
begin
  Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;

```

다각형 그리기

캔버스에 여러 개의 면을 갖는 다각형을 그리려면 캔버스의 *Polygon* 메소드를 호출합니다.

Polygon 메소드는 점의 배열을 자신의 유일한 매개변수로 갖고 펜으로 점을 연결한 다음 마지막 점을 첫 번째 점과 연결하여 다각형을 닫습니다. 선을 그린 후에는 *Polygon* 메소드는 브러시를 사용하여 다각형 내부의 영역을 채웁니다.

예를 들어, 다음 코드는 폼의 왼쪽 아래 중간 부분에 직각 삼각형을 그립니다.

```

procedure TForm1.FormPaint(Sender:TObject);
begin
  Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
    Point(ClientWidth, ClientHeight)]);
end;

```

애플리케이션에서 여러 드로잉 객체 처리

다양한 그리기 메소드(사각형, 도형, 선 등)는 일반적으로 툴바와 버튼 패널에서 사용할 수 있습니다. 애플리케이션은 원하는 드로잉 객체를 설정하기 위해 스피드 버튼 클릭에 응답할 수 있습니다. 이 단원에서는 다음과 같은 방법을 설명합니다.

- 사용할 드로잉 툴 파악
- 스피드 버튼으로 툴 변경
- 드로잉 툴 사용

사용할 드로잉 툴 파악

그래픽 프로그램은 사용자가 언제든지 사용하려는 드로잉 툴(선, 사각형, 타원 또는 모서리가 둥근 사각형)의 종류를 파악해야 합니다. 각 툴의 종류마다 번호를 할당할 수 있지만 그럴 경우 각 번호가 어느 툴에 해당하는지 기억해야 합니다. 각 번호에 기억을 돕는 상수 이름을 할당하면 쉽게 구별할 수 있지만 코드는 어떤 번호가 해당 범위 내에 있고 알맞은 타입인지 구별하지 못합니다. 다행스럽게도 오브젝트 파스칼은 이러한 두 가지 부족함을 처리할 방법을 제공합니다. 바로 열거 타입을 선언하는 방법입니다.

열거 타입은 상수에 순차적으로 값을 할당하는 빠른 방법입니다. 이 방법 또한 타입 선언이므로 오브젝트 파스칼의 타입 검사를 사용하여 이러한 특정 값만 할당하도록 할 수 있습니다.

열거 타입을 선언하려면 예약어 `type` 다음에 타입 식별자, 등호 및 타입 내의 값 식별자를 괄호에 넣고 쉼표로 구분합니다.

예를 들어, 다음 코드는 그래픽 애플리케이션에서 사용 가능한 각 드로잉 툴의 열거 타입을 선언합니다.

```

type
    TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);

```

규칙에 의해 타입 식별자는 *T*로 시작하고 열거 타입을 구성하는 것과 같은 유사한 상수 그룹은 두 글자의 접두사로 시작합니다. 예를 들어, 드로잉 툴(drawing tool)에서는 *dt*와 같이 두 글자의 접두사로 시작합니다.

TDrawingTool 타입의 선언은 상수 그룹을 선언하는 것과 같습니다.

```

const
    dtLine = 0;
    dtRectangle = 1;
    dtEllipse = 2;
    dtRoundRect = 3;

```

주요 차이점은 열거 타입을 선언함으로써 상수에 값만 부여하는 것이 아니라 오브젝트 파스칼의 타입 검사를 사용하여 많은 오류를 방지해 줄 타입도 부여한다는 것입니다. TdrawingTool 타입의 변수는 dtLine..dtRoundRect 상수 중 오직 하나에만 할당될 수 있습니다. 다른 숫자(0..3의 범위 중 하나)를 할당하려고 시도하면 컴파일 시 오류가 발생합니다.

다음 코드에서 폼에 추가된 필드는 폼의 드로잉 툴을 확보합니다.

```

type
    TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
    TForm1 = class(TForm)
        ...{ method declarations }
    public
        Drawing: Boolean;
        Origin, MovePt: TPoint;
        DrawingTool: TDrawingTool;{ field to hold current tool }
    end;

```

스피드 버튼으로 툴 변경

각 드로잉 툴은 연결된 *OnClick* 이벤트 핸들러가 필요합니다. 애플리케이션에 선, 사각형, 타원 및 모서리가 둥근 사각형과 같은 각 네 개의 드로잉 툴에 툴바 버튼이 있다고 가정합니다. 다음 이벤트 핸들러를 네 개의 드로잉 툴 버튼의 *OnClick* 이벤트에 첨부하고 *DrawingTool*을 각각에 적합한 값으로 설정합니다.

```

procedure TForm1.LineButtonClick(Sender: TObject);{ LineButton }
begin
    DrawingTool := dtLine;
end;

procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
    DrawingTool := dtRectangle;
end;

procedure TForm1.EllipseButtonClick(Sender: TObject);{ EllipseButton }
begin
    DrawingTool := dtEllipse;
end;

procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }

```

```
begin
  DrawingTool := dtRoundRect;
end;
```

드로잉 툴 사용

이제 어느 툴을 사용해야 하는지 구별할 수 있으므로 다른 도형을 그리는 방법을 나타내야 합니다. 그리기를 수행하는 유일한 메소드는 마우스 이동과 마우스 업 핸들러이며 어떤 툴을 선택하든지 그리기 코드만 선을 그립니다.

다른 드로잉 툴을 사용하려면 코드는 선택한 툴을 기반으로 그리는 방법을 지정해야 합니다. 각 툴의 이벤트 핸들러에 이 지침을 추가합니다.

이 단원에서는 다음을 설명합니다.

- 도형 그리기
- 이벤트 핸들러 간에 코드 공유

도형 그리기

도형을 그리는 것은 선을 그리는 것만큼 쉽습니다. 각 도형마다 하나의 문장을 취하고 좌표만 알면 됩니다.

네 개의 툴 모두에 대해 도형을 그리기 위한 *OnMouseUp* 이벤트 핸들러를 다시 작성하는 방법은 다음과 같습니다.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button TMouseButton; Shift:TShiftState;
                             X,Y: Integer);

begin
  case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
        Canvas.LineTo(X, Y);
      end;
    dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
    dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
    dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                                   (Origin.X - X) div 2, (Origin.Y - Y) div 2);

  end;
  Drawing := False;
end;
```

물론 도형을 그리려면 *OnMouseMove* 핸들러를 다음과 같이 업데이트해야 합니다.

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
    begin
      Canvas.Pen.Mode := pmNotXor;
      case DrawingTool of
        dtLine:begin
          Canvas.MoveTo(Origin.X, Origin.Y);
          Canvas.LineTo(MovePt.X, MovePt.Y);
```

```

        Canvas.MoveTo(Origin.X, Origin.Y);
        Canvas.LineTo(X, Y);
    end;
dtRectangle: begin
    Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
    Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
end;
dtEllipse: begin
    Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
    Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
end;
dtRoundRect: begin
    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
        (Origin.X - X) div 2, (Origin.Y - Y) div 2);
    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
        (Origin.X - X) div 2, (Origin.Y - Y) div 2);
end;
end;
MovePt := Point(X, Y);
end;
Canvas.Pen.Mode := pmCopy;
end;

```

일반적으로 위 예제에 있는 모든 반복적인 코드들은 별도의 루틴에 있습니다. 다음 단원은 모든 마우스 이벤트 핸들러에서 호출할 수 있는 단일 루틴의 모든 도형 그리기 코드를 보여 줍니다.

이벤트 핸들러 간의 코드 공유

대다수의 이벤트 핸들러가 동일한 코드를 사용할 경우 모든 이벤트 핸들러가 공유할 수 있는 루틴으로 반복 코드를 이동시키면 애플리케이션을 더 효율적으로 만들 수 있습니다.

다음과 같은 방법으로 폼에 메소드를 추가합니다.

- 1 폼 객체에 메소드 선언을 추가합니다.

폼 객체의 선언 마지막에 있는 **public**이나 **private** 부분에 선언을 추가할 수 있습니다. 코드가 단순히 일부 이벤트 처리의 세부 사항을 공유하는 경우 공유 메소드를 **private**으로 하는 것이 가장 안전할 것입니다.

- 2 폼 유닛의 구현 부분에 메소드 구현을 작성합니다.

메소드 구현의 헤더는 동일한 순서의 동일한 매개변수를 가진 선언과 정확히 일치해야 합니다.

다음 코드는 *DrawShape*라는 폼에 메소드를 추가하고 각 핸들러에서 메소드를 호출합니다. 먼저 *DrawShape*의 선언이 폼 객체의 선언에 다음과 같은 방법으로 추가됩니다.

```

type
    TForm1 = class(TForm)
        ...{ fields and methods declared here}
    public
        { Public declarations }
        procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
    end;

```

그런 다음 *DrawShape*의 구현을 유닛의 implementation 부분에 다음과 같이 작성합니다.

```

implementation
{$R *.FRM}
...{ other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
    begin
      Pen.Mode := AMode;
      case DrawingTool of
        dtLine:
          begin
            MoveTo(TopLeft.X, TopLeft.Y);
            LineTo(BottomRight.X, BottomRight.Y);
          end;
        dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
        dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
        dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
          (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
      end;
    end;
end;

```

다른 이벤트 핸들러는 *DrawShape*를 호출하기 위해 수정됩니다.

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy);{ draw the final shape }
  Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
    begin
      DrawShape(Origin, MovePt, pmNotXor);{ erase the previous shape }
      MovePt := Point(X, Y);{ record the current point }
      DrawShape(Origin, MovePt, pmNotXor);{ draw the current shape }
    end;
end;

```

그래픽 위에 그리기

애플리케이션의 그래픽 객체를 처리하기 위한 컴포넌트가 필요하지는 않습니다. 화면에 아무 것도 그리지 않고도 그래픽 객체를 생성하고, 그리고, 저장하고 소멸시킬 수 있습니다. 사실 애플리케이션에서 폼에 직접 그리지는 경우는 거의 없습니다. 대개는 애플리케이션이 그래픽에 작동한 다음 이미지 컨트롤 컴포넌트를 사용하여 폼에 그래픽을 표시합니다.

애플리케이션의 그리기 기능 대신 이미지 컨트롤에서 그래픽을 처리하면 인쇄 기능, 클립보드 사용 및 그래픽 객체 저장과 불러오기 동작을 쉽게 추가할 수 있습니다. 그래픽 객체는 비트맵 파일, 드로잉, 아이콘 또는 jpeg 그래픽과 같은 설치되어 있는 다른 그래픽 클래스일 수 있습니다.

참고 *TBitmap* 캔버스와 같은 오프스크린 이미지에 그리는 것이므로 컨트롤이 비트맵으로부터 컨트롤의 캔버스에 복사해야 이미지가 표시됩니다. 즉, 비트맵을 그려서 이미지 컨트롤에 할당하게 되면 컨트롤이 그리기 메시지를 처리해야 이미지가 나타납니다. 하지만 컨트롤의 캔버스 속성에 직접 그리면 그림 객체가 즉시 표시됩니다.

스크롤 가능한 그래픽 만들기

그래픽은 폼과 크기가 같지 않아도 됩니다. 더 크거나 작을 수 있습니다. 폼에 스크롤 상자 컨트롤을 추가하고 내부에 그래픽 이미지를 두면 폼보다 더 크거나 심지어 화면보다도 큰 그래픽을 표시할 수 있습니다. 스크롤 가능한 그래픽을 추가하려면 먼저 *TScrollBar* 컴포넌트를 추가한 다음 이미지 컨트롤을 추가합니다.

이미지 컨트롤 추가

이미지 컨트롤은 비트맵 객체를 표시할 수 있는 컨테이너 컴포넌트입니다. 이미지 컨트롤을 사용하면 항상 표시될 필요가 없거나 애플리케이션에서 다른 그림을 생성하기 위해 필요한 비트맵을 가질 수 있습니다.

참고 7-12 페이지의 "컨트롤에 그래픽 추가"에서는 컨트롤에서 그래픽을 사용하는 방법을 보여 줍니다.

컨트롤 두기

이미지 컨트롤은 폼의 아무 곳이나 둘 수 있습니다. 그림에 자신의 크기를 맞추는 이미지 컨트롤 기능을 이용할 경우, 왼쪽 위 모서리만 설정하면 됩니다. 이미지 컨트롤이 비트맵에서 보이지 않는 표시자인 경우에는 논비주얼(nonvisual) 컴포넌트와 마찬가지로 아무 곳이나 둘 수 있습니다.

폼의 클라이언트 영역에 이미 정렬된 스크롤 상자에 이미지 컨트롤을 가져다 놓는 경우 스크롤 상자가 이미지의 그림 중 오프스크린 부분에 액세스하는 데 필요한 모든 스크롤 막대를 추가합니다. 그런 다음 이미지 컨트롤의 속성을 설정합니다.

초기 비트맵 크기 설정

어떤 이미지 컨트롤을 두게 되면 단지 컨테이너에 불과합니다. 그러나 디자인 타임에 이미지 컨트롤의 *Picture* 속성을 설정하여 정적 그래픽을 포함할 수는 있습니다. 그리고 8-19 페이지의 "그래픽 파일 로드 및 저장"에서 설명한 대로 컨트롤은 런타임 시 파일에서 컨트롤 그림을 로드할 수도 있습니다.

다음과 같은 방법으로 애플리케이션이 시작할 때 빈 비트맵을 만듭니다.

- 1 이미지가 들어 있는 폼의 *OnCreate* 이벤트에 핸들러를 첨부합니다.
- 2 비트맵 객체를 만들고 비트맵 객체를 이미지 컨트롤의 *Picture.Graphic* 속성에 할당합니다.

이 예제에서 이미지는 애플리케이션의 메인 폼인 *Form1*에 있으므로 코드는 다음과 같은 방법으로 *Form1*의 *OnCreate* 이벤트에 핸들러를 첨부합니다.

```

procedure TForm1.FormCreate(Sender: TObject);
var
    Bitmap:TBitmap;{ temporary variable to hold the bitmap }
begin
    Bitmap := TBitmap.Create;{ construct the bitmap object }
    Bitmap.Width := 200;{ assign the initial width... }
    Bitmap.Height := 200;{ ...and the initial height }
    Image.Picture.Graphic := Bitmap;{ assign the bitmap to the image control }
    Bitmap.Free; {We are done with the bitmap, so free it }
end;

```

그림의 *Graphic* 속성에 비트맵을 할당하면 그림 객체에 비트맵이 복사됩니다. 그러나 그림 객체는 비트맵의 소유권을 갖지 않으므로 할당한 다음 해제해야 합니다.

지금 애플리케이션을 실행하는 경우 폼의 클라이언트 영역에 비트맵을 나타내는 흰색 영역이 나타납니다. 클라이언트 영역에 전체 이미지가 표시되지 않도록 창의 크기를 조정하는 경우 이미지의 나머지 부분을 표시할 수 있도록 스크롤 상자가 스크롤 막대를 자동으로 표시하는 것을 볼 수 있습니다. 그러나 이미지 위에서 그리려는 경우 현재 이미지와 스크롤 상자 뒤에 있는 폼에서 애플리케이션의 그리기 작업이 아직 진행 중이므로 그래픽이 나타나지 않습니다.

비트맵에 그리기

비트맵에서 그리려면 이미지 컨트롤의 캔버스를 사용하고 이미지 컨트롤의 해당 이벤트에 마우스 이벤트 핸들러를 첨부합니다. 일반적으로 영역 동작(채우기, 사각형, 다각선 등)을 사용합니다. 영역 동작은 빠르고 효율적인 그리기 방법입니다.

각 픽셀에 액세스해야 할 때 이미지를 그리는 효과적인 방법은 비트맵 *ScanLine* 속성을 사용하는 것입니다. 일반적인 목적으로 사용하는 경우 비트맵 픽셀 형식을 24비트로 설정한 다음 *ScanLine*에서 반환된 포인터를 RGB 배열로 처리합니다. 그렇지 않으면 *ScanLine* 속성의 원시 형식을 알아야 합니다. 이 예제는 *ScanLine*을 사용하여 픽셀을 한 번에 한 줄씩 얻는 방법을 보여 줍니다.

```

procedure TForm1.Button1Click(Sender: TObject);
// This example shows drawing directly to the Bitmap
var
    x,y : integer;
    Bitmap: TBitmap;
    P : PByteArray;
begin
    Bitmap := TBitmap.create;
    try
        Bitmap.LoadFromFile('C:\Program Files\Borland\Delphi 4\Images\Splash\256color\
factory.bmp');
        for y := 0 to Bitmap.height -1 do
            begin
                P := Bitmap.ScanLine[y];
                for x := 0 to Bitmap.width -1 do
                    P[x] := y;
            end;
    end;

```



```

canvas.draw(0,0,Bitmap);
finally
  Bitmap.free;
end;
end;

```

그래픽 파일 로드 및 저장

하나의 애플리케이션이 실행되는 동안에만 존재하는 그래픽 이미지는 가치가 매우 제한적입니다. 경우에 따라 동일한 그림을 항상 사용하려고 하거나 만들어진 그림을 나중에 사용하기 위해 저장하고자 할 수 있습니다. 이미지 컴포넌트를 사용하면 파일에서 쉽게 그림을 로드하고 다시 저장할 수 있습니다.

그래픽 이미지를 로드, 저장, 및 교체하는 데 사용하는 컴포넌트는 비트맵 파일, 메타파일, glyph 등 많은 그래픽 형식을 지원합니다. 또한 CLX 컴포넌트는 설치 가능한 그래픽 클래스를 지원합니다.

그래픽 파일을 로드하고 저장하는 방법은 다른 파일과 유사하며 다음 단원에 설명되어 있습니다.

- 파일에서 그림 로드
- 그림을 파일에 저장
- 그림 교체

파일에서 그림 로드

애플리케이션에서 그림을 수정해야 하거나 애플리케이션 외부에 그림을 저장하려는 경우 파일에서 그림을 로드하는 기능을 제공해야 합니다. 그러면 다른 사용자나 애플리케이션에서 그림을 수정할 수 있습니다.

이미지 컨트롤로 그래픽 파일을 로드하려면 이미지 컨트롤의 *Picture* 객체의 *LoadFromFile* 메소드를 호출합니다.

다음 코드는 그림 파일 열기 대화 상자에서 파일 이름을 선택한 다음 해당 파일을 *Image*라는 이름의 이미지 컨트롤로 해당 파일을 로드합니다.

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    begin
      CurrentFile := OpenPictureDialog1.FileName;
      Image.Picture.LoadFromFile(CurrentFile);
    end;
end;

```

그림을 파일에 저장

그림 객체는 여러 가지 형태로 그래픽을 로드 및 저장할 수 있으며 그림 객체가 그래픽을 로드하고 저장할 수 있도록 그래픽 파일 형식을 직접 만들고 등록할 수 있습니다.

파일에 이미지 컨트롤의 내용을 저장하려면 이미지 컨트롤의 *Picture* 객체의 *SaveToFile* 메소드를 호출합니다.

SaveToFile 메소드에는 저장할 파일 이름이 필요합니다. 그림을 새로 만든 경우 파일 이름이 없거나 사용자가 기존 그림을 다른 파일에 저장할 수도 있습니다. 어느 경우든지 다음 단원에 나와 있는 대로 사용자는 애플리케이션에 저장하기 전에 파일 이름을 부여해야 합니다.

File|Save 및 File|Save As 메뉴 항목에 첨부된 다음 이벤트 핸들러 쌍은 차례대로 이름이 있는 파일의 재저장, 이름이 없는 파일의 저장 그리고 기존 파일을 새 이름으로 저장하는 기능을 처리합니다.

```

procedure TForm1.Button1Click(Sender:TObject);
begin
    if CurrentFile <> '' then
        Image.Picture.SaveToFile(CurrentFile){ save if already named }
    else SaveAs1Click(Sender);{ otherwise get a name }
end;
procedure TForm1.Button1Click(Sender:TObject);
begin
    if SaveDialog1.Execute then{ get a file name }
    begin
        CurrentFile := SaveDialog1.FileName;{ save the user-specified name }
        Save1Click(Sender);{ then save normally }
    end;
end;

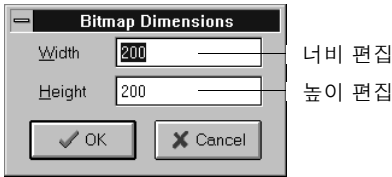
```

그림 교체

이미지 컨트롤의 그림은 언제든지 교체할 수 있습니다. 이미 그래픽이 있는 그림에 새 그래픽을 할당하면 기존 그래픽이 새 그래픽으로 바뀝니다.

이미지 컨트롤에서 그림을 변경하려면 이미지 컨트롤의 *Picture* 객체에 새 그래픽을 할당합니다.

새 그래픽을 만드는 것은 처음으로 그래픽을 만드는 것과 동일한 과정을 거치지만(8-17 페이지의 "초기 비트맵 크기 설정" 참조) 초기 그래픽에 사용된 기본 크기와 다른 새 그래픽에 사용할 크기를 선택하는 방법도 제공해야 합니다. 크기 선택 옵션을 제공하는 간단한 방법은 그림 8.1에서와 같은 대화상자를 표시하는 것입니다.

그림 8.1 BMPDlg 유닛의 Bitmap-dimension 대화상자.

이 대화 상자는 *GraphEx* 프로젝트(EXAMPLES\DOC\GRAPHEX 디렉토리)에 포함되어 있는 *BMPDlg* 유닛 안에 만들어집니다.

프로젝트에서 대화 상자를 사용하여 메인 폼의 유닛에 있는 *uses* 절에 대화 상자를 추가합니다. 그런 다음 File|New 메뉴 항목의 *OnClick* 이벤트에 이벤트 핸들러를 첨부할 수 있습니다. 예제는 다음과 같습니다.

```

procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap:TBitmap;{ temporary variable for the new bitmap }
begin
  with Canvas do
    begin
      ActiveControl := WidthEdit;{ make sure focus is on width field }
      WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ use current dimensions... }
      HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...as default }
      if ShowModal <> idCancel then{ continue if user doesn't cancel dialog box }
        begin
          Bitmap := TBitmap.Create;{ create fresh bitmap object }
          Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
          Bitmap.Height := StrToInt(HeightEdit.Text);{ use specified height }
          Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
          CurrentFile := '';{ indicate unnamed file }
          Bitmap.Free;
        end;
    end;
  end;

```

참고 그림 객체의 *Graphic* 속성에 새 비트맵을 할당하면 그림 객체는 새 그래픽을 복사하지 만 소유권을 갖지는 않습니다. 그림 객체는 자신의 내부 그래픽 객체를 유지합니다. 이로 인해 이전의 코드는 비트맵 객체를 할당한 다음 해제합니다.

클립보드에서 그래픽 사용

애플리케이션 내에 그래픽을 복사하고 붙여넣거나 다른 애플리케이션과 그래픽을 교환하는 데 Windows 클립보드를 사용할 수 있습니다. VCL의 클립보드 객체를 사용하면 그래픽을 포함하는 다른 종류의 정보를 쉽게 처리할 수 있습니다.

애플리케이션에서 클립보드 객체를 사용하려면 클립보드 데이터를 액세스해야 하는 모든 유닛의 *uses* 절에 Clipbrd(CLX의 QClipbrd) 유닛을 추가해야 합니다

크로스 플랫폼 애플리케이션의 경우 CLX를 사용하여 클립보드에 저장된 데이터는 연관된 *TStream* 객체의 mime 타입으로 저장됩니다. CLX는 다음의 이미 정의된 mime 소스와 mime 타입 문자열 상수를 다음 CLX 객체에 제공합니다.

- TBitmap = 'image/delphi.bitmap'
- TComponent = 'application/delphi.component'
- TPicture = 'image/delphi.picture'
- TDrawing = 'image/delphi.drawing'

클립보드에 그래픽 복사

클립보드에 이미지 컨트롤의 콘텐츠를 포함하는 모든 그림을 복사할 수 있습니다. 클립보드에 저장되면 그림은 모든 애플리케이션에서 사용할 수 있습니다.

클립보드에 그림을 복사하려면 *Assign* 메소드를 사용하여 클립보드 객체에 그림을 할당합니다.

다음 코드는 Edit|Copy 메뉴 항목을 클릭해서 *Image*라고 명명된 이미지 컨트롤에서 클립보드로 그림을 복사하는 방법을 보여 줍니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
    Clipboard.Assign(Image.Picture)
end.
```

클립보드로 그림 잘라내기

클립보드로 그림을 잘라내는 것은 복사하는 것과 완전히 똑같은 방법이지만 잘라내기의 경우 원본에서 그래픽을 지워 줍니다.

그림에서 그래픽을 잘라내어 클립보드에 넣으려면 먼저 클립보드에 그래픽을 복사하고 원본을 지웁니다.

대부분의 경우 잘라내기의 유일한 문제점은 원본 이미지가 지워졌다는 것을 보여 주는 방법에 있습니다. 다음과 같이 코드가 Edit|Cut 메뉴 항목의 *OnClick* 이벤트에 이벤트 핸들러를 첨부하는 경우에서처럼 해당 영역을 하얗게 설정하는 것이 일반적인 해결 방법입니다.

```
procedure TForm1.Cut1Click(Sender:TObject);
var
    ARect:TRect;
begin
    Copy1Click(Sender){ copy picture to clipboard }
    with Image.Canvas do
        begin
            CopyMode := cmWhiteness;{ copy everything as white }
            ARect := Rect(0, 0, Image.Width, Image.Height);{ get bitmap rectangle }
            CopyRect(ARect, Image.Canvas, ARect);{ copy bitmap over itself }
            CopyMode := cmSrcCopy;{ restore normal mode }
        end;
    end;
```

클립보드에서 그림 붙여넣기

클립보드에 비트맵 그래픽이 들어 있는 경우 이미지 컨트롤과 폼의 표면을 비롯한 모든 이미지 객체에 클립보드를 붙여넣을 수 있습니다.

다음과 같은 방법으로 클립보드에서 그래픽을 붙여넣습니다.

- 1 클립보드에 그래픽이 들어 있는지 보려면 클립보드의 *HasFormat* 메소드 (VCL을 사용하는 경우) 또는 *Provides* 메소드 (CLX를 사용하는 경우)를 호출하십시오.

HasFormat (또는 CLX의 *Provides*)은 부울 함수입니다. *Provides*는 클립보드에 매개변수에 지정된 형식의 항목이 들어 있는 경우 *True*를 반환합니다. Windows 플랫폼에서 그래픽을 테스트하려면 *CF_BITMAP*을 전달합니다. 크로스 플랫폼 애플리케이션의 경우에는 *SDelphiBitmap*을 전달합니다.

- 2 대상에 클립보드를 할당합니다.

다음 코드는 Edit|Paste 메뉴 항목을 클릭해서 클립보드에서 이미지 컨트롤로 그림을 붙여넣는 방법을 보여 줍니다.

```
procedure TForm1.PasteButtonClick(Sender:TObject);
var
  Bitmap:TBitmap;
begin
  if Clipboard.HasFormat(CF_BITMAP) then { is there a bitmap on the Windows clipboard? }
  begin
    Image1.Picture.Bitmap.Assign(Clipboard);
  end;
end;
```

크로스 플랫폼용 CLX 개발의 동일한 예제는 다음과 같습니다.

```
procedure TForm1.PasteButtonClick(Sender:TObject);
var
  Bitmap:TBitmap;
begin
  if Clipboard.Provides(SDelphiBitmap) then { is there a bitmap on the clipboard? }
  begin
    Image1.Picture.Bitmap.Assign(Clipboard);
  end;
end;
```

클립보드의 그래픽은 이 애플리케이션에 가져오거나 Microsoft Paint와 같은 다른 애플리케이션에서 복사할 수 있습니다. 클립보드에 지원하는 형식이 없을 때에는 붙여넣기 메뉴를 사용할 수 없으므로 이러한 경우에서 클립보드 형식은 확인할 필요가 없습니다.

양단 묶음(Rubber banding) 예제

이 예제에서는 사용자가 런타임 시 그래픽을 그릴 때 마우스 이동을 추적하는 그래픽 애플리케이션에서의 "양단 묶음" 효과를 구현하는 것에 대한 자세한 내용을 설명합니다. 이 단원의 예제 코드는 Demos\DOC\Graphex 디렉토리에 있는 예제 애플리케이션에서 가져온 것입니다. 애플리케이션에서는 클릭하고 드래그하여 창의 캔버스에 선과 도형을

그립니다. 즉, 마우스 버튼을 누르면 그리기를 시작하고 버튼을 놓으면 그리기를 끝냅니다.

처음에는 예제 코드에서 메인 폼의 표면에 그리는 방법을 보여 줍니다. 이후의 예제에서는 비트맵에 그리는 것을 보여 줍니다.

다음 항목에서는 해당 예제를 설명합니다.

- 마우스에 응답
- 마우스 동작을 추적하기 위해 폼 객체에 필드 추가
- 선 그리기 작업 정제

마우스에 응답

애플리케이션은 마우스 버튼 다운, 마우스 이동 및 마우스 버튼 업 등의 마우스 동작에 응답할 수 있습니다. 또한 모달 대화 상자에서 *Enter*를 누르는 것과 같이 키 입력으로 만들 수 있는 클릭(한 번에 누르기 및 놓기 가능)에 응답할 수 있습니다.

이 단원에서는 다음과 같은 내용을 다룹니다.

- 마우스 이벤트의 내용
- 마우스 다운 동작에 응답
- 마우스 업 동작에 응답
- 마우스 이동에 응답

마우스 이벤트의 내용

VCL에는 *OnMouseDown* 이벤트, *OnMouseMove* 이벤트 및 *OnMouseUp* 이벤트 등 세 가지의 마우스 이벤트가 있습니다.

애플리케이션은 마우스 동작을 발견하면 다섯 개의 매개변수를 전달해서 해당 이벤트에 대해 사용자가 정의한 이벤트 핸들러를 호출합니다. 이벤트에 대한 응답을 사용자 지정하려면 이 매개변수의 정보를 사용하십시오. 다섯 개의 매개변수는 다음과 같습니다.

표 8.4 마우스 이벤트 매개변수

매개변수	의미
<i>Sender</i>	마우스 동작을 발견한 객체입니다.
<i>Button</i>	다음의 <i>mbLeft</i> , <i>mbMiddle</i> 또는 <i>mbRight</i> 중 어느 마우스 버튼과 관계 있는지 나타냅니다.
<i>Shift</i>	마우스 동작 시 <i>Alt</i> , <i>Ctrl</i> 및 <i>Shift</i> 키의 상태를 나타냅니다.
<i>X</i> , <i>Y</i>	이벤트가 발생한 좌표입니다.

대부분 마우스 이벤트 핸들러에 반환된 좌표가 필요하지만 경우에 따라 어떤 마우스 버튼이 이벤트를 유발했는지 알려면 *Button*을 확인해야 합니다.

참고 Delphi는 어느 마우스 버튼이 눌러졌는지 결정할 때 Microsoft Windows와 동일한 기준을 사용합니다. 따라서 사용자가 기본적인 "첫 번째"와 "두 번째" 마우스 버튼을 바꾸어 오른쪽 마우스 버튼이 첫 번째 버튼이 되게 할 경우 첫 번째(오른쪽) 버튼은 *Button* 매개변수 값으로서 *mbLeft*로 기록됩니다.

마우스 다운 동작에 응답

사용자가 마우스의 버튼을 누를 때마다 *OnMouseDown* 이벤트는 포인터가 위치한 객체로 전달됩니다. 그러면 객체는 이벤트에 응답할 수 있습니다.

마우스 다운 동작에 응답하려면 *OnMouseDown* 이벤트에 이벤트 핸들러를 첨부합니다.

VCL은 다음과 같은 방법으로 폼에 마우스 다운 이벤트의 빈 핸들러를 생성합니다.

```
procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
end;
```

마우스 다운 동작에 응답

다음 코드를 사용하면 마우스로 클릭한 폼의 해당 위치에 'Here!'라는 문자열을 표시합니다.

```
procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.TextOut(X, Y, 'Here!');{ write text at (X, Y) }
end;
```

애플리케이션이 실행되면 폼에 마우스 커서로 마우스 버튼을 눌러 클릭한 지점에 'Here!' 문자열이 나타나게 할 수 있습니다. 다음 코드는 사용자가 버튼을 누른 좌표에 현재 그리기 위치를 설정합니다.

```
procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.MoveTo(X, Y);{ set pen position }
end;
```

이제 마우스 버튼을 누르면 펜 위치가 설정되고 선의 시작점이 설정됩니다. 버튼을 놓는 지점까지 선을 그리려면 마우스 업 이벤트에 응답해야 합니다.

마우스 업 동작에 응답

OnMouseUp 이벤트는 마우스 버튼을 놓을 때마다 발생합니다. 버튼을 누를 때 마우스 커서가 위치한 객체로 이벤트가 전달되지만 버튼을 놓을 때의 커서가 놓인 위치가 동일한 객체일 필요는 없습니다. 이러한 방식을 사용하면 폼의 테두리를 넘어 선을 그릴 수 있습니다.

마우스 업 동작에 응답하려면 *OnMouseUp* 이벤트의 핸들러를 정의해야 합니다.

다음은 마우스 버튼을 놓는 지점까지 선을 그리는 간단한 *OnMouseUp* 이벤트 핸들러입니다.

```

procedure TForm1.FormMouseUp(Sender:TObject; Button:TMouseButton;
    Shift:TShiftState; X, Y:Integer);
begin
    Canvas.LineTo(X, Y);{ draw line from PenPos to (X, Y) }
end;
    
```

이 코드를 통해 사용자는 클릭, 끌기 및 놓기를 사용하여 선을 그릴 수 있습니다. 이러한 경우 버튼을 놓으면 선을 볼 수 있습니다.

마우스 이동에 응답

OnMouseMove 이벤트는 마우스를 이동할 때 주기적으로 발생합니다. 이벤트는 버튼을 누를 때 마우스 포인터 아래에 있던 객체로 전달됩니다. 이를 통해 마우스가 움직이는 동안 일시적인 선을 그림으로써 사용자에게 중간 피드백을 줄 수 있습니다.

마우스 이동에 응답하려면 *OnMouseMove* 이벤트의 이벤트 핸들러를 정의해야 합니다. 다음 예제에서는 마우스 이동 이벤트를 사용하여 사용자가 마우스 버튼을 누르고 있는 동안 폼에 중간 도형을 그려서 사용자에게 일부 피드백을 제공합니다. *OnMouseMove* 이벤트 핸들러는 *OnMouseMove* 이벤트가 발생한 위치의 폼에 선을 그립니다.

```

procedure TForm1.FormMouseMove(Sender:TObject;Button:TMouseButton;
    Shift:TShiftState; X, Y:Integer);
begin
    Canvas.LineTo(X, Y);{ draw line to current position }
end;
    
```

위의 코드를 사용하여 마우스를 폼 위로 이동하면 마우스 버튼을 누르기 전에도 마우스를 따라 그려지게 됩니다.

마우스 이동 이벤트는 마우스 버튼을 누르지 않았을 때에도 발생합니다.

눌려진 마우스 버튼이 있는지 추적하려면 폼 객체에 객체 필드를 추가해야 합니다.

마우스 동작을 추적하기 위해 폼 객체에 필드 추가

마우스 버튼이 눌려졌는지 추적하려면 폼 객체에 객체 필드를 추가해야 합니다. 폼에 컴포넌트를 추가할 때 Delphi는 해당 필드 이름으로 컴포넌트를 참조할 수 있도록 폼 객체에 해당 컴포넌트를 나타내는 필드를 추가합니다. 폼 유닛의 헤더 파일에 타입 선언을 편집하여 폼에 직접 만든 필드를 추가할 수도 있습니다.

다음 예제를 보면 폼은 사용자가 마우스 버튼을 눌렀는지 여부를 추적해야 합니다. 그렇게 하려면 폼에서 부울 필드를 추가하고 사용자가 마우스 버튼을 누를 때 해당 값을 설정합니다.

객체에 필드를 추가하려면 선언 아래의 **public** 지시어 뒤에 필드 식별자와 타입을 지정하는 객체 타입 정의를 편집합니다.

Delphi는 **public** 지시어 앞의 모든 선언을 "소유"합니다. 그러므로 public 지시어 앞에 컨트롤을 나타내는 필드와 이벤트에 응답하는 메소드를 넣습니다.

다음 코드는 폼 객체의 선언에서 부울 타입의 *Drawing*이라는 필드를 폼에 제공합니다. 또한 *Origin* 지점과 *TPoint* 타입의 *MovePt*를 저장하는 두 개의 필드도 추가합니다.


```

type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender:TObject; Button:TMouseButton;
      Shift:TShiftState; X, Y:Integer);
    procedure FormMouseDown(Sender:TObject; Button:TMouseButton;
      Shift:TShiftState; X, Y:Integer);
    procedure TForm1.FormMouseMove(Sender:TObject; Button:TMouseButton;
      Shift:TShiftState; X, Y:Integer);
  public
    Drawing:Boolean;{ field to track whether button was pressed }
    Origin, MovePt:TPoint;{ fields to store points }
  end;

```

그릴 것인지 여부를 추적하는 *Drawing* 필드가 있으면 다음과 같은 방법으로 사용자가 마우스 버튼을 누를 때 *True*로 설정하고 놓을 때 *False*로 설정합니다.

```

procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Drawing := True;{ set the Drawing flag }
  Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.LineTo(X, Y);
  Drawing := False;{ clear the Drawing flag }
end;

```

그런 다음 다음과 같은 방법으로 *OnMouseMove* 이벤트 핸들러를 수정하면 *Drawing* 이 *True*일 때에만 그릴 수 있습니다.

```

procedure TForm1.FormMouseMove(Sender:TObject;Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  if Drawing then{ only draw if Drawing flag is set }
    Canvas.LineTo(X, Y);
end;

```

이렇게 하면 마우스 다운과 마우스 업 이벤트 간에만 그릴 수 있지만 여전히 직선 대신 마우스 이동을 추적하는 점선이 나타납니다.

그러나 문제는 마우스를 이동할 때마다 마우스 이동 이벤트 핸들러가 펜 위치를 이동시키는 *LineTo*를 호출하여 사용자가 마우스 버튼을 놓을 때까지 직선이 시작하는 지점을 잃게 되는 것입니다.

선 그리기 다듬기

다양한 지점을 추적하는 필드를 적절히 사용하면 애플리케이션의 선 그리기를 다듬을 수 있습니다.

원래 지점 추적

선을 그릴 때 선이 *Origin* 필드와 함께 시작하는 지점을 추적합니다.

*Origin*은 마우스 다운 이벤트가 발생하는 지점으로 설정되어야만 마우스 업 이벤트 핸들러가 *Origin*을 사용하여 다음 코드처럼 선의 시작 위치를 지정할 수 있습니다.

```

procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
  Origin := Point(X, Y);{ record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
  Canvas.LineTo(X, Y);
  Drawing := False;
end;

```

이렇게 변경하면 애플리케이션에서는 마지막 선을 다시 그리지만 중간 동작을 그리지 않고 애플리케이션은 "양단 묶음"을 지원하지 않습니다.

이동 추적

이 예제의 문제점은 현재 작성된 *OnMouseMove* 이벤트 핸들러가 원래 위치가 아니라 마지막 *마우스 위치*에서 현재의 마우스 위치로 선을 그린다는 것입니다. 원래 지점으로 그리기 위치를 이동한 다음 현재 위치에서 그림으로써 이 문제를 해결할 수 있습니다.

```

procedure TForm1.FormMouseMove(Sender:TObject;Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  if Drawing then
  begin
    Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
    Canvas.LineTo(X, Y);
  end;
end;

```

위의 코드는 현재 마우스 위치를 추적하지만 중간 선이 사라지지 않으므로 마지막 선을 거의 볼 수 없습니다. 예제에서는 다음 선을 그리기 전에 이전의 선이 있던 위치를 계속 추적하여 선을 하나씩 지워야 합니다. *MovePt* 필드를 통해 선을 지우는 작업을 할 수 있습니다.

*MovePt*는 각 중간 선의 끝점으로 설정되어야 하므로 *MovePt*와 *Origin*을 사용하여 다음에 선이 그려질 때 해당 선을 지웁니다.

```

procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
  Origin := Point(X, Y);

```

```

MovePt := Point(X, Y);{ keep track of where this move was }
end;
procedure TForm1.FormMouseMove(Sender:TObject;Button:TMouseButton;
Shift:TShiftState; X, Y:Integer);
begin
  if Drawing then
    begin
      Canvas.Pen.Mode := pmNotXor;{ use XOR mode to draw/erase }
      Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
      Canvas.LineTo(MovePt.X, MovePt.Y);{ erase the old line }
      Canvas.MoveTo(Origin.X, Origin.Y);{ start at origin again }
      Canvas.LineTo(X, Y);{ draw the new line }
    end;
      MovePt := Point(X, Y);{ record point for next move }
      Canvas.Pen.Mode := pmCopy;
    end;

```

이제 선을 그릴 때 "양단 묶음" 효과가 나타납니다. 펜의 모드를 *pmNotXor*로 변경하면 선을 배경 픽셀과 조합할 수 있습니다. 선을 지울 때 픽셀을 이전의 상태로 설정하게 됩니다. 선을 그린 후 펜 모드를 기본값인 *pmCopy*로 다시 변경하여 마우스 버튼을 놓으면 펜은 마지막 그리기 작업을 준비합니다.

멀티미디어 작업

Delphi에서는 애플리케이션에 멀티미디어 컴포넌트를 추가할 수 있습니다. 이렇게 하려면 Win32 페이지의 *TAnimate* 컴포넌트를 사용하거나 컴포넌트 팔레트의 System 페이지에서 *TMediaPlayer* 컴포넌트를 사용할 수 있습니다. 애플리케이션에 소리가 없는 비디오 클립을 추가하려는 경우에는 애니메이션 컴포넌트를 사용합니다. 오디오 또는 비디오 클립을 애플리케이션에 추가하려는 경우에는 미디어 플레이어 컴포넌트를 사용합니다.

TAnimate 및 *TMediaPlayer* 컴포넌트에 대한 자세한 내용은 VCL 온라인 도움말을 참조하십시오.

이 단원에서는 다음 항목들을 설명합니다.

- 애플리케이션에 소리가 없는 비디오 클립 추가
- 애플리케이션에 오디오 및/또는 비디오 클립 추가

애플리케이션에 소리가 없는 비디오 클립 추가

Delphi의 애니메이션 컨트롤을 통해 사용자는 애플리케이션에 소리가 없는 비디오 클립을 추가할 수 있습니다.

다음과 같은 방법으로 애플리케이션에 소리가 없는 비디오 클립을 추가합니다.

- 1 컴포넌트 팔레트의 Win32 페이지에서 *Animate* 아이콘을 더블 클릭합니다. 그러면 비디오 클립을 표시하려는 폼 창에 애니메이션 컨트롤이 생깁니다.

- Object Inspector를 사용하여 *Name* 속성을 선택하고 애니메이션 컨트롤의 새로운 이름을 입력합니다. 애니메이션 컨트롤을 호출할 때 이 이름을 사용합니다. (Delphi 식별자의 이름을 지정할 때는 표준 규칙을 따르십시오.)

디자인 타임 속성을 설정하거나 이벤트 핸들러를 생성할 때에는 항상 Object Inspector를 직접 사용합니다.

- 다음 중 하나를 수행합니다.

- Common AVI* 속성을 선택하고 드롭다운 목록에서 사용 가능한 AVI 중 하나를 선택합니다.
- FileName* 속성을 선택하여 생략(...) 버튼을 클릭하고 사용할 수 있는 로컬 또는 네트워크 디렉토리에서 AVI 파일을 선택하여 Open AVI 대화 상자에서 Open을 클릭합니다.
- ResName* 또는 *ResID* 속성을 사용하여 AVI의 리소스를 선택합니다. *ResHandle*을 사용하여 *ResName* 또는 *ResID*로 구분되는 리소스를 가진 모듈을 지정합니다.

그러면 AVI 파일이 메모리에 로드됩니다. *Active* 속성이나 *Play* 메소드를 사용하여 재생되기 전에 AVI 클립의 첫 프레임을 화면에 표시하려면 *Open* 속성을 *True*로 설정합니다.

- AVI 클립을 재생하려는 횟수를 *Repetitions* 속성에 설정합니다. 값을 0으로 두면 *Stop* 메소드가 호출될 때까지 계속 반복됩니다.
- 애니메이션 컨트롤 설정에 다른 사항들을 변경합니다. 예를 들어, 애니메이션 컨트롤이 열릴 때 표시되는 첫 프레임을 변경하려는 경우에는 *StartFrame* 속성을 원하는 프레임 값으로 설정합니다.
- 드롭다운 목록을 사용하여 *Active* 속성을 *True*로 설정하거나 이벤트 핸들러를 작성하여 런타임 시 특정 이벤트가 발생할 때 AVI 클립을 실행합니다. 예를 들어, 버튼 객체를 클릭할 때 AVI 클립을 활성화하려면 이를 지정하는 버튼의 *OnClick* 이벤트를 작성합니다. 또한 AVI 재생 시간을 지정하기 위해 *Play* 메소드를 호출할 수도 있습니다.

참고

*Active*를 *True*로 설정한 후에 폼이나 폼의 컴포넌트를 변경한 경우에는 *Active* 속성이 *False*로 되므로 다시 *True*로 설정해야 합니다. 런타임 전이나 후에 설정하십시오.

소리가 없는 비디오 클립 추가 예제

애플리케이션이 시작할 때 첫 화면으로서 애니메이션 로고를 표시하려고 한다고 가정합니다. 로고 재생이 끝나면 화면이 사라집니다.

이 예제를 실행하려면 새로운 프로젝트를 작성하여 Unit1.pas 파일을 Frmlogo.pas로 저장하고 Project1.dpr 파일을 Logo.dpr로 저장합니다. 그런 다음 단계를 따릅니다.

- 컴포넌트 팔레트의 Win32 페이지에서 Animate 아이콘을 더블 클릭합니다.
- Object Inspector를 사용하여 Name 속성을 *Logo1*로 설정합니다.

- 3 FileName 속성을 선택하고 생략(...) 버튼을 클릭하여 ..\Demos\Coolstuff 디렉토리에서 cool.avi 파일을 선택합니다. 그런 다음 Open AVI 대화 상자에서 Open을 클릭합니다.
그러면 cool.avi 파일이 메모리에 로드됩니다.
- 4 애니메이션 컨트롤 상자를 누르고 폼의 오른쪽 위로 끌어서 위치를 정합니다.
- 5 Repetitions 속성을 5로 설정합니다.
- 6 폼을 눌러 포커스를 두고 Name 속성을 *LogoForm1*로 설정하고 Caption 속성을 *Logo Window*로 설정합니다. 이제 폼의 높이를 줄여 애니메이션 컨트롤이 폼의 중앙에 오도록 합니다.
- 7 폼의 *OnActivate* 이벤트를 더블 클릭하여 런타임 시 폼이 포커스를 가졌을 때 AVI 클립으로 실행되도록 다음과 같이 코드를 작성합니다.

```
Logol.Active := True;
```
- 8 컴포넌트 팔레트의 Standard 페이지의 Label 아이콘을 더블 클릭합니다. Caption 속성을 선택하고 *Welcome to Cool Images 4.0*을 입력합니다. 이제 Font 속성을 선택하고 생략(...) 버튼을 클릭하여 Font Style을 Bold로, Size를 18, Color는 Navy로 Font 대화 상자에서 선택하고 OK를 클릭합니다. 레이블 컨트롤을 클릭하고 끌어서 폼의 중앙에 둡니다.
- 9 애니메이션 컨트롤을 클릭하여 다시 포커스를 둡니다. *OnStop* 이벤트를 더블 클릭하여 AVI 파일이 멈출 때 폼을 닫으려면 다음 코드를 작성합니다.

```
LogoForm1.Close;
```
- 10 Run|Run을 선택하여 애니메이션 로고 창을 실행합니다.

애플리케이션에 오디오 및/또는 비디오 클립 추가

Delphi의 미디어 플레이어 컴포넌트에서는 사용자가 애플리케이션에 오디오 및/또는 비디오 클립을 추가할 수 있습니다. 이 컴포넌트는 미디어 장치를 열고 미디어 장치에서 사용되는 오디오 및/또는 비디오 클립의 재생, 멈춤, 일시 정지, 기록 등을 수행합니다. 미디어 장치는 하드웨어나 소프트웨어일 수 있습니다.

참고 오디오 및 비디오 클립 지원은 크로스 플랫폼 프로그래밍에는 제공되지 않습니다.

다음과 같은 방법으로 애플리케이션에 오디오 및/또는 비디오 클립을 추가합니다.

- 1 컴포넌트 팔레트의 System 페이지에 있는 미디어 플레이어 아이콘을 더블 클릭합니다. 그러면 미디어 기능을 사용하려는 폼 창에 미디어 플레이어 컨트롤이 자동으로 생깁니다.
- 2 Object Inspector를 사용하여 *Name* 속성을 선택하고 미디어 플레이어 컨트롤의 새로운 이름을 입력합니다. 이 이름은 이 미디어 플레이어 컨트롤을 호출할 때 사용됩니다. (Delphi 식별자 이름 지정에 대한 표준 규칙을 따르십시오.)
디자인 타임 속성을 설정하거나 이벤트 핸들러를 생성할 때에는 항상 Object Inspector를 직접 사용합니다.

- 3 *DeviceType* 속성을 선택한 다음 *AutoOpen* 속성이나 *Open* 메소드를 사용하여 적절한 장치 타입을 선택합니다. (*DeviceType*이 *dtAutoSelect*이면 *FileName* 속성에 지정한 미디어 파일의 파일 확장자에 따라 장치 타입이 선택됩니다.) 장치 타입과 기능에 대한 자세한 내용은 표 8.5를 참조하십시오.
- 4 장치가 파일에 미디어를 저장하는 경우에는 *FileName* 속성을 사용하여 미디어 파일의 이름을 지정합니다. *FileName* 속성을 선택하여 생략(...) 버튼을 클릭하고 사용 가능한 로컬 또는 네트워크 디렉토리에서 미디어 파일을 선택하고 Open 대화 상자의 Open을 누릅니다. 그렇지 않으면 런타임 시 선택된 미디어 장치에 대한 미디어가 저장된 하드웨어(디스크, 카세트, 등)를 삽입합니다.
- 5 *Filtered* 속성을 *True*로 설정합니다. 이러한 방식으로 미디어 플레이어는 미디어 플레이어 컨트롤이 들어 있는 폼이 런타임 시 작성될 때 지정된 장치를 자동으로 엽니다. *AutoOpen*이 *False*인 경우에는 장치를 *Open* 메소드에 대한 호출로 열어야 합니다.
- 6 *AutoEnable* 속성을 *True*로 설정하여 런타임 시 필요에 따라 미디어 플레이어 버튼을 활성화 또는 비활성화하거나 *EnabledButtons* 속성을 더블 클릭하여 활성화하거나 비활성화하려는지에 따라 각 버튼을 *True* 또는 *False*로 설정합니다.
멀티미디어 장치는 사용자가 미디어 플레이어 컴포넌트의 해당 버튼을 누를 때 재생, 일시 정지, 멈춤 등의 작업을 수행합니다. 장치는 각 버튼(Play, Pause, Stop, Next, Previous 등)에 해당하는 메소드에 의해 조정될 수도 있습니다.
- 7 미디어 플레이어 컨트롤을 클릭하고 폼의 적당한 위치로 옮겨가거나 *Align* 속성을 선택하고 드롭다운 목록에서 적절한 정렬 위치를 선택하여 폼에 미디어 플레이어 컨트롤 바를 둡니다.
런타임 시 미디어 플레이어를 보이지 않게 하려면 *Visible* 속성을 *False*로 설정하고 적절한 메소드 (*Play*, *Pause*, *Stop*, *Next*, *Previous*, *Step*, *Back*, *Start Recording*, *Eject*)를 호출하여 장치를 제어합니다.
- 8 미디어 플레이어 컨트롤에 대한 다른 사항을 설정합니다. 예를 들어, 미디어에 디스플레이 창이 필요한 경우에는 미디어를 디스플레이하는 컨트롤에 대한 *Display* 속성을 설정합니다. 장치가 여러 트랙을 사용하는 경우에는 원하는 트랙으로 *Tracks* 속성을 설정합니다.

표 8.5 멀티미디어 장치 타입 및 기능

장치 타입	사용할 소프트웨어/ 하드웨어	재생	트랙 사용	디스플레이 창 사용
dtAVIVideo	Windows용 AVI 비디오 플레이어	AVI 비디오 파일	아니오	예
dtCDAudio	Windows 용 CD 오디오 플레이어 이어나 CD 오디오 플레이어	CD 오디오 디스크	예	아니오
dtDAT	디지털 오디오 테이프 플레이어	디지털 오디오 테이프	예	아니오
dtDigitalVideo	Windows용 디지털 비디오 플레이어	AVI, MPG, MOV 파일	아니오	예
dtMMMovie	MM 무비 플레이어	MM 필름	아니오	예
dtOverlay	오버레이 장치	아날로그 비디오	아니오	예

표 8.5 멀티미디어 장치 타입 및 기능 (계속)

장치 타입	사용할 소프트웨어/ 하드웨어	재생	트랙 사용	디스플레이 창 사용
dtScanner	이미지 스캐너	재생하지 않음 (Record 시 이미지 스캔)	아니오	아니오
dtSequencer	Windows용 MIDI 시퀀서	MIDI 파일	예	아니오
dtVCR	비디오 카세트 레코더	비디오 카세트	아니오	예
dtWaveAudio	Windows용 웨이브 오디오 플레이어	WAV 파일	아니오	아니오

오디오 및/또는 비디오 클립 추가 예제(VCL 전용)

이 예제는 Delphi용 멀티미디어 광고의 AVI 비디오 클립을 실행합니다. 이 예제를 실행하려면 새로운 프로젝트를 작성하여 Unit1.pas 파일을 FrmAd.pas로 저장하고 Project1.dpr 파일을 DelphiAd.dpr로 저장합니다. 그리고 다음 작업들을 수행합니다.

- 1 컴포넌트 팔레트의 System 페이지에 있는 미디어 플레이어 아이콘을 더블 클릭합니다.
- 2 Object Inspector를 사용하여 미디어 플레이어의 Name 속성을 *VideoPlayer1*로 설정합니다.
- 3 DeviceType 속성을 선택하고 드롭다운 목록에서 dtAVIVideo를 선택합니다.
- 4 FileName 속성을 선택하고 생략(...)버튼을 클릭하여 ..\Demos\Coolstuff 디렉토리에서 speedis.avi 파일을 선택합니다. Open 대화 상자에서 Open을 누릅니다.
- 5 AutoOpen 속성은 *True*로, Visible 속성은 *False*로 설정합니다.
- 6 컴포넌트 팔레트의 Win32 페이지에서 Animate 아이콘을 더블 클릭합니다. AutoSize 속성은 *False*로, Height 속성은 *175*, Width 속성은 *200*으로 설정합니다. 애니메이션 컨트롤을 클릭하고 끌어서 폼의 왼쪽 위 모서리에 둡니다.
- 7 미디어 플레이어를 눌러서 다시 포커스를 둡니다. Display 속성을 선택하고 드롭다운 목록에서 Animate1을 선택합니다.
- 8 폼을 눌러서 다시 포커스를 두고 Name 속성을 선택하고 *Delphi_Ad*를 입력합니다. 이제 애니메이션 컨트롤의 크기에 맞게 다시 폼의 크기를 조정합니다.
- 9 폼의 *OnActivate* 이벤트를 더블 클릭하여 폼이 포커스를 받았을 때 AVI 비디오를 실행하도록 다음 코드를 작성합니다.

```
VideoPlayer1.Play;
```

- 10 Run|Run을 선택하여 AVI 비디오를 실행합니다.

9

다중 스레드 애플리케이션 작성

Delphi는 다중 스레드 애플리케이션을 더욱 쉽게 작성할 수 있도록 여러 가지 객체를 제공합니다. 다중 스레드 애플리케이션은 실행을 동시에 할 수 있는 여러 개의 경로가 들어 있는 애플리케이션입니다. 다중 스레드를 사용하는 데에는 세심한 주의를 요하지만 다음과 같은 작업을 수행하여 프로그램을 향상시킬 수 있습니다.

- **병목 현상 방지.** 스레드가 하나인 경우에는 디스크의 파일 액세스, 다른 컴퓨터와의 통신 또는 멀티미디어 콘텐츠 표시 등의 느린 프로세스를 기다리는 동안 프로그램에서 모든 실행을 중지해야 합니다. CPU는 프로세스가 완료될 때까지 유휴 상태로 있습니다. 그러나 다중 스레드를 사용하면 애플리케이션에서 하나의 스레드가 느린 프로세스의 결과를 기다리는 동안 별도의 스레드에서 다른 프로세스를 계속 실행할 수 있습니다.
- **프로그램 동작 조직화.** 경우에 따라 프로그램의 동작을 독립적으로 기능하는 여러 개의 병렬 프로세스로 조직화할 수 있습니다. 스레드를 사용하면 이러한 병렬 프로세스들에 대해 단일 코드 섹션을 동시에 실행할 수 있습니다. 보다 중요한 작업에 CPU 시간을 부여할 수 있도록 스레드를 사용하여 다양한 프로그램 작업에 우선 순위를 할당하십시오.
- **멀티프로세스.** 프로그램을 실행하는 시스템에 다중 프로세서가 있으면 작업을 여러 스레드로 나누고 별도의 프로세서에서 동시에 실행시켜 성능을 향상시킬 수 있습니다.

참고 기본적인 하드웨어가 멀티프로세싱을 지원하더라도 모든 운영 체제가 진정한 멀티프로세싱을 구현하는 것은 아닙니다. 예를 들어, 기본적인 하드웨어가 멀티프로세싱을 지원하더라도 Windows 9x는 멀티프로세싱을 구현하는 것처럼 보여질 뿐입니다.

스레드 객체 정의

대부분의 애플리케이션에서 스레드 객체를 사용하면 애플리케이션에 실행 스레드를 나타낼 수 있습니다. 스레드 객체는 가장 일반적으로 필요한 스레드 사용을 캡슐화하여 다중 스레드 애플리케이션 작성을 단순화합니다.

참고 스레드 객체를 통해 보안 속성이나 스레드의 스택 크기를 제어할 수는 없습니다. 보안 속성이나 스레드의 스택 크기를 제어해야 할 경우 *BeginThread* 함수를 사용해야 합니다. *BeginThread*를 사용하더라도 9-7 페이지의 "스레드 조정"에 설명되어 있는 일부 스레드 동기화 객체 및 메소드에서 이점을 얻을 수 있습니다. *BeginThread* 사용에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

애플리케이션에서 스레드 객체를 사용하려면 *TThread*의 새 자손을 만들어야 합니다. *TThread*의 자손을 만들려면 메인 메뉴에서 File|New를 선택합니다. 새 객체 대화 상자에서 Thread Object를 선택합니다. 새 스레드 객체의 클래스 이름을 입력하라는 메시지가 나타납니다. 이름을 입력하면 Delphi는 새 유닛 파일을 만들어 스레드를 구현합니다.

참고 클래스 이름을 필요로 하는 대부분의 IDE 대화 상자와는 달리 New Thread Object 대화 상자는 사용자가 입력한 클래스 이름 앞에 "T"를 자동으로 붙이지 않습니다.

자동으로 생성된 유닛 파일에는 새 스레드 객체의 뼈대 코드가 들어 있습니다. 스레드의 이름을 *TMyThread*로 지정한 경우 다음과 같이 나타납니다.

```
unit Unit1;
interface
uses
  Classes
type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Place thread code here }
end;
end.
```

Execute 메소드에 코드를 채워야 합니다. 그 단계에 대해서는 다음 단원에서 설명합니다.

스레드 초기화

새 스레드 클래스의 초기화 코드를 작성하려면 Create 메소드를 오버라이드해야 합니다. 스레드 클래스의 선언에 새 생성자를 추가하고 선언을 구현하는 초기화 코드를 작성합니다. 여기서 스레드의 기본 우선 순위를 할당하고 스레드가 실행 종료될 때 자동으로 해제되어야 할지 여부를 나타낼 수 있습니다.

기본 우선 순위 할당

우선 순위는 운영 체제가 애플리케이션의 모든 스레드 중에서 CPU 시간의 일정을 정할 때 스레드가 갖는 기본 설정 수를 나타냅니다. 시간적으로 중요한 작업을 처리할 때에는

높은 우선 순위의 스레드를 사용하고 다른 작업을 수행할 때에는 낮은 우선 순위의 스레드를 사용합니다. 스레드 객체의 우선 순위를 나타내려면 *Priority* 속성을 설정하십시오.

Windows 애플리케이션을 작성하면 *Priority* 값은 표 9.1에서 설명한 대로 7포인트 범위까지 내려갑니다.

표 9.1 스레드 우선 순위

값	우선 순위
tpIdle	이 스레드는 시스템이 유휴 상태일 때에만 실행됩니다. Windows는 <i>tpIdle</i> 속성을 가진 스레드를 실행할 때 다른 스레드를 방해하지 않습니다.
tpLowest	이 스레드의 우선 순위는 정상보다 2포인트 아래입니다.
tpLower	이 스레드의 우선 순위는 정상보다 1포인트 아래입니다.
tpNormal	이 스레드는 정상 우선 순위를 가집니다.
tpHigher	이 스레드의 우선 순위는 정상보다 1포인트 위입니다.
tpHighest	이 스레드의 우선 순위는 정상보다 2포인트 위입니다.
tpTimeCritical	이 스레드는 가장 높은 우선 순위를 가집니다.

참고 크로스 플랫폼 애플리케이션 작성 시 Windows와 Linux의 우선 순위를 할당하기 위해서는 별도의 코드를 사용해야 합니다. Linux에서 *우선 순위*는 루트에 의해서만 변경될 수 있는 스레드 정책에 따른 숫자 값입니다. 자세한 내용은 *TThread*의 CLX 버전 및 *우선 순위* 온라인 도움말을 참조하십시오.

경고 CPU를 많이 사용하는 작업의 스레드 우선 순위를 높이면 애플리케이션의 다른 스레드를 사용하지 못하는 "기아" 상태가 될 수 있습니다. 외부 이벤트를 기다리는 데 대부분의 시간을 소비하는 스레드에만 높은 우선 순위를 적용합니다.

다음 코드는 다른 애플리케이션의 성능을 방해하지 않아야 할 백그라운드 작업을 수행하는 경우의 우선 순위가 낮은 스레드 생성자를 보여 줍니다.

```

constructor TMyThread.Create(CreateSuspended:Boolean);
begin
    inherited Create(CreateSuspended);
    Priority := tpIdle;
end;

```

스레드 해제 시기 표시

일반적으로 스레드가 작업을 마치면 간단히 해제됩니다. 이러한 경우 스레드 객체가 스스로 해제되도록 하는 것이 가장 쉽습니다. 스스로 해제되게 하려면 *FreeOnTerminate* 속성을 *True*로 설정합니다.

그러나 하나의 스레드 종료 시기를 다른 스레드 종료 시기와 일치시켜야 하는 경우가 있습니다. 예를 들어, 다른 스레드에서 동작을 수행하기 전에 값을 반환할 스레드를 기다릴 경우가 있습니다. 이렇게 두 개의 스레드 종료 작업을 일치시키기 위해 두 번째 스레드가 반환 값을 받은 다음 첫 번째 스레드를 해제하려고 합니다. 이러한 상황에서는 *FreeOnTerminate*를 *False*로 설정한 다음 두 번째 스레드에서 첫 번째 스레드를 명시적으로 해제함으로써 처리할 수 있습니다.

스레드 함수 작성

Execute 메소드가 스레드 함수입니다. 동일한 프로세스 공간을 공유하는 것을 제외하고는 스레드 함수가 애플리케이션에 의해 실행되는 프로그램으로 생각할 수 있습니다. 스레드 함수를 작성하는 것은 애플리케이션에서 다른 스레드에 의해 사용되는 메모리를 겹쳐 쓰지 않아야 하므로 별도의 프로그램을 작성하는 것보다 다소 까다롭습니다. 반면에 스레드가 다른 스레드와 동일한 프로세스 공간을 공유하므로 공유 메모리를 사용하여 스레드 간에 통신할 수 있습니다.

메인 VCL/CLX 스레드 사용

VCL이나 CLX 객체 계층에서 객체를 사용할 때 해당 속성과 메소드는 스레드에 대한 안전을 보장받지 못합니다. 즉, 속성에 액세스하거나 메소드를 실행하면 다른 스레드 동작에서 보호되지 않는 메모리를 사용하는 일부 동작을 수행할 수 있습니다. 이러한 이유에서 VCL과 CLX 객체에 액세스하기 위해 메인 스레드를 별도로 둡니다. 이 스레드가 바로 사용자 애플리케이션의 컴포넌트에서 받는 모든 Windows 메시지를 처리하는 스레드입니다.

모든 객체가 단일 스레드에서 해당 속성에 액세스하고 메소드를 실행할 경우에는 사용자의 객체들이 서로 방해하는 것을 걱정하지 않아도 됩니다. 메인 스레드를 사용하려면 필요한 동작을 수행하는 별도의 루틴을 만드십시오. 스레드의 *Synchronize* 메소드 내에서 별도의 루틴을 호출합니다. 예를 들면, 다음과 같습니다.

```
procedure TMyThread.PushTheButton;
begin
    Button1.Click;
end;
:
procedure TMyThread.Execute;
begin
    :
    Synchronize(PushTheButton);
    :
end;
```

*Synchronize*는 메인 스레드가 메시지 루프로 들어올 때까지 기다린 다음 전달된 메소드를 실행합니다.

참고 *Synchronize*는 메시지 루프를 사용하므로 콘솔 애플리케이션에서는 작동하지 않습니다. 콘솔 애플리케이션에서 VCL이나 CLX 객체에 대한 액세스를 보호하려면 임계 구역과 같은 다른 메커니즘을 사용해야 합니다.

항상 메인 스레드를 사용할 필요는 없습니다. 일부 객체에는 thread-aware 기능이 있습니다. 객체의 메소드가 스레드에 대해 안전하다는 것을 알고 *Synchronize* 메소드를 사용하지 않으면 VCL이나 CLX 스레드가 메시지 루프로 들어올 때까지 기다릴 필요가 없으므로 성능이 향상됩니다. 다음과 같은 상황에서는 *Synchronize* 메소드를 사용할 필요가 없습니다.

- 데이터 액세스 컴포넌트가 다음과 같이 스레드에 대해 안전한 경우, 즉 BDE 활성 데이터셋인 경우, 각각의 스레드는 해당 데이터베이스 세션 컴포넌트를 가져야 합니다.

이 때 스레드에 대해 안전하지 않은 Microsoft 라이브러리를 사용하여 구축한 Access 드라이버를 사용하는 경우는 예외입니다. dbDirect의 경우, 타사의 클라이언트 라이브러리가 스레드에 대해 안전하면 dbDirect 컴포넌트는 스레드에 대해 안전합니다. ADO와 InterbaseExpress 컴포넌트는 스레드에 대해 안전합니다.

데이터 액세스 컴포넌트를 사용할 때 *Synchronize* 메소드의 data-aware 컨트롤을 포함하는 모든 호출을 연결해야 합니다. 그러므로 예를 들어 데이터 소스 객체의 *DataSet* 속성을 설정하여 데이터셋에 데이터 컨트롤을 연결하는 호출을 동기화해야 하지만 동기화하여 데이터셋 필드의 데이터에 액세스할 필요는 없습니다.

BDE 사용이 가능한 애플리케이션에서 스레드와 함께 데이터베이스 세션을 사용하는 데 대한 자세한 내용은 20-29 페이지의 "다중 세션 관리"를 참조하십시오.

- VisualCLX 객체가 스레드에 대해 안전하지 않은 경우.
- DataCLX 객체가 스레드에 대해 안전한 경우.
- 그래픽 객체가 스레드에 대해 안전한 경우. *TFont*, *TPen*, *TBrush*, *TBitmap*, *TMetafile* (VCL 전용), *TDrawing* (CLX 전용) 또는 *TIcon*에 액세스하기 위해 메인 VCL이나 CLX 스레드를 사용할 필요는 없습니다. 캔버스 객체는 *TFont*, *TPen*, *TBrush*, *TBitmap*, *TDrawing*, *TIcon* 등을 잠금으로써 *Synchronize* 메소드 외부에서 사용할 수 있습니다 (9-7 페이지의 "객체 잠금" 참조).
- 목록 객체가 스레드에 대해 안전하지 않지만 *TList* 대신 스레드에 대해 안전한 버전인 *TThreadList*를 사용할 수 있는 경우.

백그라운드 스레드가 자신의 실행을 메인 스레드와 동기화할 수 있도록 애플리케이션의 메인 스레드 내에서 *CheckSynchronize* 루틴을 정기적으로 호출하십시오. *CheckSynchronize*를 호출할 최적의 조건은 *OnIdle* 이벤트 핸들러에서 애플리케이션이 유휴 상태일 때입니다. 이렇게 하면 백그라운드 스레드에서 메소드를 호출하는 것이 안전합니다.

스레드 로컬 변수 사용

Execute 메소드와 이 메소드가 호출하는 모든 루틴은 오브젝트 파스칼 루틴과 마찬가지로 자신의 로컬 변수를 가집니다. 또한 이 루틴은 모든 전역 변수에도 액세스할 수 있습니다. 사실 전역 변수는 스레드 간의 통신을 위한 강력한 메커니즘을 제공합니다.

그러나 경우에 따라 해당 스레드에서 실행되는 모든 루틴에 대한 전역 변수이면서 동일한 스레드 클래스의 다른 인스턴스와는 공유되지 않는 변수를 사용하고자 할 수도 있습니다. 스레드 로컬 변수를 선언하면 이와 같이 수행할 수 있습니다. **threadvar** 섹션에 변수를 선언하여 변수를 스레드 로컬로 만듭니다. 예를 들면 다음과 같습니다.

```
threadvar
  x :integer;
```

애플리케이션의 각 스레드에 개별적이지만 각 스레드 내에서는 전역인 정수형 변수를 선언합니다.

threadvar 섹션은 전역 변수에서만 사용할 수 있습니다. Pointer 및 Function 변수는 스레드 변수가 될 수 없습니다. 긴 문자열 같은 copy-on-write 구문을 사용하는 타입도 스레드 변수로 사용할 수 없습니다.

다른 스레드로 종료 확인

스레드는 *Execute* 메소드가 호출될 때 실행을 시작하고(9-11 페이지의 "스레드 객체 실행" 참조) *Execute*가 완료될 때까지 계속됩니다. 이는 스레드가 특정 작업을 수행한 다음 완료되면 중지한다는 모델을 보여 줍니다. 그러나 경우에 따라서 어떤 애플리케이션은 일부 외부 기준을 충족할 때까지 실행되는 스레드를 필요로 합니다.

Terminated 속성을 확인함으로써 스레드가 실행을 완료해야 할 시간이라는 것을 다른 스레드를 통해 나타낼 수 있습니다. 다른 스레드는 사용자의 스레드를 종료시키려고 할 때 *Terminate* 메소드를 호출합니다. *Terminate*는 사용자 스레드의 *Terminated* 속성을 *True*로 설정합니다. *Terminated* 속성 확인 및 *Terminated* 속성에 대한 응답을 수행하여 *Terminate* 메소드를 구현하는 것은 *Execute* 메소드에 달려 있습니다. 다음 예제는 이러한 방법 중 하나를 보여 줍니다.

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

스레드 함수에서의 예외 처리

Execute 메소드는 스레드에서 발생하는 모든 예외를 감지해야 합니다. 스레드 함수에서 예외를 감지하지 못할 경우 애플리케이션에서 액세스 위반이 발생할 수 있습니다. 애플리케이션을 개발할 때 IDE가 예외를 감지하므로 위의 설명이 명확하지 않을 수 있으나 디버거 외부에서 애플리케이션을 실행하면 예외로 인해 런타임 오류가 발생하고 애플리케이션의 실행이 중지됩니다.

스레드 함수 내부에서 발생하는 예외를 감지하려면 *Execute* 메소드의 구현에 **try...except** 블록을 다음과 같은 방법으로 추가합니다.

```
procedure TMyThread.Execute;
begin
  try
    while not Terminated do
      PerformSomeTask;
    except
      { do something with exceptions }
    end;
end;
```

지우기 코드 작성

스레드 실행이 완료되었을 때 이를 지우는 코드를 하나로 통일할 수 있습니다. 스레드가 종료되기 전에 *OnTerminate* 이벤트가 발생합니다. *OnTerminate* 이벤트 핸들러에 지우기 코드를 넣어 *Execute* 메소드 앞에 어떤 실행 경로가 오더라도 항상 실행되게 하십시오.

OnTerminate 이벤트 핸들러는 스레드에서 부분적으로는 실행되지 않습니다. 대신 애플리케이션에 있는 메인 VCL이나 CLX 스레드의 컨텍스트에서 실행됩니다. 이것은 다음과 같은 두 가지 의미를 가집니다.

- 메인 VCL이나 CLX 스레드 값을 원하지 않는 한 *OnTerminate* 이벤트 핸들러에서는 어떤 스레드 로컬 변수도 사용할 수 없습니다.
- 다른 스레드와의 충돌에 대한 걱정 없이 *OnTerminate* 이벤트 핸들러에서 모든 컴포넌트와 VCL 또는 CLX 객체에 안전하게 액세스할 수 있습니다.

메인 VCL이나 CLX 스레드에 대한 자세한 내용은 9-4 페이지의 "메인 VCL/CLX 스레드 사용"을 참조하십시오.

스레드 조정

스레드가 실행될 때 실행되는 코드를 작성할 경우 동시에 실행될 수 있는 다른 스레드의 동작을 고려해야 합니다. 특히 두 개의 스레드가 동일한 전역 객체나 변수를 동시에 사용하지 못하도록 주의를 기울여야 합니다. 뿐만 아니라 하나의 스레드 코드는 다른 스레드에 의해 수행되는 작업 결과에 따라 달라질 수 있습니다.

동시 액세스 피하기

전역 객체나 변수에 액세스할 때 다른 스레드와의 충돌을 피하려면 스레드 코드가 작업을 완료할 때까지 다른 스레드의 실행을 막아야 합니다. 이 때 실행 중인 다른 스레드를 불필요하게 막지 않도록 주의해야 합니다. 그렇게 할 경우 성능이 심하게 저하되어 다중 스레드를 사용하여 얻을 수 있는 대부분의 이점을 얻지 못할 수 있습니다.

객체 잠금

일부 객체에는 다른 스레드가 실행되어 해당 객체 인스턴스를 사용하지 못하게 하는 잠금 기능이 기본 제공되어 있습니다.

예를 들어, *TCanvas*나 자손과 같은 캔버스 객체에는 *Unlock* 메소드가 호출되기 전까지 다른 스레드가 캔버스에 액세스하지 못하도록 막아 주는 *Lock* 메소드가 있습니다.

또한 VCL과 CLX 모두 *TThreadList*와 같이 스레드에 대한 안전한 목록 객체를 가집니다. *TThreadList.LockList*를 호출하면 *UnlockList* 메소드가 호출될 때까지 실행 중인 다른 스레드가 목록을 사용하지 못하도록 막으면서 목록 객체를 반환합니다. *TCanvas.Lock*이나 *TThreadList.LockList*를 호출하면 안전하게 중첩됩니다. 마지막 잠금 호출이 동일한 스레드 내의 해당 잠금 해제 호출과 일치해야만 이 잠금이 해제됩니다.

임계 구역(Critical sections) 사용

객체가 기본 제공된 잠금 기능을 제공하지 않으면 임계 구역을 사용할 수 있습니다. 임계 구역은 한 번에 하나의 스레드만 들어갈 수 있게 하는 문과 같습니다. 임계 구역을 사용하려면 *TCriticalSection*의 전역 인스턴스를 작성합니다. *TCriticalSection*에는 다른 스레드가 구역을 실행하지 못하게 막는 *Acquire*과 블록을 제거하는 *Release*라는 두 개의 메소드가 있습니다.

각 임계 구역은 보호하고자 하는 전역 메모리와 연결됩니다. 해당 전역 메모리에 액세스하는 모든 스레드는 먼저 *Acquire* 메소드를 사용하여 다른 스레드가 *Acquire* 메소드를 사용하지 못하도록 해야 합니다. 완료되면 스레드는 다른 스레드에서 *Acquire*를 호출하여 전역 메모리에 액세스할 수 있도록 *Release* 메소드를 호출합니다.

경고 임계 구역은 모든 스레드가 임계 구역을 사용하여 연결된 전역 메모리에 액세스할 때만 작동합니다. 임계 구역을 무시하고 *Acquire*를 호출하지 않은 채 전역 메모리에 액세스하는 스레드는 동시 액세스 문제를 유발할 수 있습니다.

예를 들어, 전역 변수인 X와 Y에 대한 액세스를 막는 임계 구역 전역 변수인 *LockXY*를 갖고 있는 애플리케이션의 경우를 고려해 보십시오. X나 Y를 사용하는 모든 스레드는 다음과 같은 임계 구역에 대한 호출과 함께 사용하는 경우처럼 에워싸야 합니다.

```
LockXY.Acquire; { lock out other threads }
try
    Y := sin(X);
finally
    LockXY.Release;
end;
```

다중 읽기 배타적 쓰기 동기화 장치(Multi-read exclusive-write synchronizer) 사용

임계 구역을 사용하여 전역 메모리를 보호할 경우 한 번에 단 하나의 스레드만 해당 메모리를 사용할 수 있습니다. 특히 가끔씩 읽어야 하지만 거의 쓰지 않는 객체나 변수가 있는 경우에는 필요 이상의 보호가 될 수도 있습니다. 그렇지만 다른 스레드가 메모리에 쓰지 않는 한 동일한 메모리를 동시에 읽는 다중 스레드에는 위험은 없습니다.

갖고 있는 일부 전역 메모리가 가끔 읽히지만 스레드에서 간혹 쓰기도 할 경우에는 *TMultiReadExclusiveWriteSynchronizer*를 사용하여 보호할 수 있습니다. 이 객체는 임계 구역처럼 동작하지만 다른 스레드가 쓰고 있지만 읽는다면 다중 스레드를 통해 자신이 보호하는 메모리를 읽을 수 있습니다. 스레드는 *TMultiReadExclusiveWriteSynchronizer*에서 보호하는 메모리를 배타적으로 액세스할 수 있어야 합니다.

다중 읽기 배타적 쓰기 동기화 장치를 사용하려면 보호하려는 전역 메모리와 연결된 *TMultiReadExclusiveWriteSynchronizer*의 전역 인스턴스를 만듭니다. 이 메모리에서 읽히는 모든 스레드는 먼저 *BeginRead* 메소드를 호출해야 합니다. 그러면 *BeginRead*에서는 다른 스레드가 메모리에 쓰지 못하게 합니다. 스레드가 보호된 메모리 읽기를 완료하면 *EndRead* 메소드를 호출합니다. 보호된 메모리에 쓰는 모든 스레드는 먼저 *BeginWrite*를 호출해야 합니다. 그러면 *BeginWrite*에서는 다른 스레드가 메모리를 읽거나 쓰지 못하게 합니다. 스레드가 보호된 메모리에 쓰기를 완료하면 메모리를 읽기 위해 기다린 스레드가 시작할 수 있도록 *EndWrite* 메소드를 호출합니다.

경고 다중 읽기 배타적 쓰기 동기화 장치는 임계 구역처럼 모든 스레드가 이 동기화 장치를 사용하여 연결된 전역 메모리에 액세스하는 경우에만 작동합니다. 동기화 장치를 무시하고 *BeginRead*나 *BeginWrite*를 호출하지 않은 채 전역 메모리에 액세스하는 스레드는 동시 액세스 문제를 유발합니다.

메모리 공유에 이용되는 그 밖의 기술

VCL이나 CLX에서 객체를 사용하는 경우, 메인 스레드를 사용하여 코드를 실행할 수 있습니다. 메인 스레드를 사용하면 객체가 다른 스레드의 VCL이나 CLX 객체에서 사용하는 모든 메모리에 간접적으로 액세스하지 못하게 합니다. 메인 스레드에 대한 자세한 내용은 9-4 페이지의 "메인 VCL/CLX 스레드 사용"을 참조하십시오.

전역 메모리를 다중 스레드로 공유할 필요가 없을 경우 전역 변수 대신 스레드 로컬 변수 사용을 고려해 보십시오. 스레드 로컬 변수를 사용하면 사용자의 스레드가 다른 스레드를 기다리거나 잠글 필요가 없습니다. 스레드 로컬 변수에 대한 자세한 내용은 9-5 페이지의 "스레드 로컬 변수 사용"을 참조하십시오.

다른 스레드 기다리기

사용자의 스레드가 다른 스레드의 작업이 완료되기를 기다려야 하는 경우 스레드에게 일시적으로 실행을 중지하도록 알릴 수 있습니다. 다른 스레드가 실행을 완전히 완료할 때까지 기다리거나 다른 스레드가 작업 완료를 신호할 때까지 기다릴 수 있습니다.

스레드의 실행 완료 대기

다른 스레드가 실행 완료되기를 기다리려면 해당 다른 스레드의 *WaitFor* 메소드를 사용합니다. *WaitFor*는 자신의 *Execute* 메소드를 완료하거나 예외로 인한 종료를 통해 다른 스레드가 종료되면 반환됩니다. 예를 들어, 다음 코드는 목록에 있는 객체에 액세스하기 전에 스레드 목록 객체를 다른 스레드가 채울 때까지 기다립니다.

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
    end;
    ThreadList1.UnlockList;
  end;
end;
```

이전의 예제에서 보면 *WaitFor* 메소드가 목록이 성공적으로 채워졌음을 표시할 때만 목록 항목을 액세스할 수 있었습니다. 이 반환 값은 기다린 스레드의 *Execute* 메소드에 의해 할당되어야 합니다. 그러나 *WaitFor*를 호출하는 스레드가 *Execute*를 호출하는 코드가 아닌 스레드 실행 결과를 알고자 하므로 *Execute* 메소드는 값을 반환하지 않습니다. 그 대신 *Execute* 메소드는 *ReturnValue* 속성을 설정합니다. *WaitFor* 메소드는 다른 스레드에 의해 호출될 때 *ReturnValue*를 반환합니다. 반환 값은 정수입니다. 애플리케이션이 그 의미를 결정합니다.

작업 완료 대기

간혹 특정 스레드의 실행 완료를 기다리는 대신 스레드의 일부 작업이 완료되는 것을 기다려야 할 경우도 있습니다. 그렇게 하려면 이벤트 객체를 사용합니다. 이벤트 객체 (*TEvent*)는 모든 스레드에 보이는 신호처럼 동작할 수 있도록 전역 유효 범위(scope)로 만들어야 합니다.

어떤 스레드에서 다른 스레드가 의존하는 작업을 완료할 때에는 `TEvent.SetEvent`를 호출합니다. `SetEvent`는 신호를 켜서 검사하는 다른 스레드에 작업이 완료되었음을 알려 줍니다. 신호를 끄려면 `ResetEvent` 메소드를 사용합니다.

예를 들어, 단일 스레드가 아닌 여러 스레드가 실행이 완료되는 것을 기다려야 하는 상황을 생각해 보십시오. 어떤 스레드가 마지막으로 완료될지 모르기 때문에 단순히 스레드 하나에 `WaitFor` 메소드를 사용할 수는 없습니다. 그 대신 각 스레드가 완료될 때 카운터를 증가시킬 수 있으며 이벤트를 설정하여 마지막 스레드가 모든 작업이 완료되었다는 신호를 보내게 할 수 있습니다.

다음 코드는 완료되어야 하는 모든 스레드의 마지막 `OnTerminate` 이벤트 핸들러를 보여 줍니다. `CounterGuard`는 다중 스레드가 동시에 카운터를 사용하지 못하도록 막는 임계 구역 전역 객체입니다. `Counter`는 완료된 스레드의 수를 계산하는 전역 변수입니다.

```
procedure TDataModule.TaskThreadTerminate(Sender:TObject);
begin
  :
  CounterGuard.Acquire; { obtain a lock on the counter }
  Dec(Counter); { decrement the global counter variable }
  if Counter = 0 then
    Event1.SetEvent; { signal if this is the last thread }
    CounterGuard.Release; { release the lock on the counter }
  :
end;
```

메인 스레드는 `Counter` 변수를 초기화하고 작업 스레드를 실행하며 `WaitFor` 메소드를 호출하여 모든 작업이 완료되었음을 나타내는 신호를 기다립니다. `WaitFor`는 신호가 설정될 때까지 지정된 시간 동안 기다리고 표 9.2의 값 중 하나를 반환합니다.

표 9.2 WaitFor 반환 값

값	의미
<code>wrSignaled</code>	이벤트 신호가 설정됩니다.
<code>wrTimeout</code>	신호가 설정되지 않은 상태로 지정된 시간이 경과합니다.
<code>wrAbandoned</code>	시간 초과 기간이 경과하기 전에 이벤트 객체가 소멸됩니다.
<code>wrError</code>	대기 중 오류가 발생합니다.

다음 코드는 메인 스레드가 작업 스레드를 실행하는 방법과 모두 완료되었을 때 다시 시작하는 방법을 보여 줍니다.

```
Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
  TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{ now continue with the main thread.All task threads have finished }
```

참고 지정된 시간 후 이벤트 대기 중지를 원하지 않는 경우 `INFINITE`의 매개변수 값인 `WaitFor` 메소드를 전달합니다. `INFINITE`를 사용할 때에는 예상되는 신호가 수신되지 않는 경우 스레드가 정지되므로 주의해야 합니다.

스레드 객체 실행

Execute 메소드를 부여하여 스레드 클래스를 구현하면 애플리케이션에서 스레드 클래스를 사용하여 *Execute* 메소드에서 코드를 실행할 수 있습니다. 스레드를 사용하려면 먼저 스레드 클래스의 인스턴스를 만듭니다. 즉시 실행되는 스레드 인스턴스를 만들거나 *Resume* 메소드를 호출할 때만 시작되도록 정지된 상태에서 스레드를 작성할 수 있습니다. 즉시 시작하도록 스레드를 작성하려면 생성자의 *CreateSuspended* 매개변수를 *False*로 설정합니다. 예를 들어, 다음 줄을 작성하여 스레드를 작성하고 실행을 시작합니다.

```
SecondProcess := TMyThread.Create(false); {create and run the thread }
```

경고 애플리케이션에 너무 많은 스레드를 작성하지 마십시오. 다중 스레드를 관리하는 오버헤드의 성능에 영향을 미칠 수 있습니다. 단일 프로세서 시스템의 프로세스당 16개의 스레드까지만 사용할 것을 권장합니다. 이러한 제한은 대부분의 스레드가 외부 이벤트를 기다리는 것을 가정하는 것입니다. 모든 스레드가 활성화되었을 경우에는 더 적게 사용합니다.

동일한 스레드 형식의 다중 인스턴스를 작성하여 병렬 코드를 실행할 수 있습니다. 예를 들어, 각 스레드에서 예상된 응답을 수행하게 하여 일부 사용자 동작에 대한 응답으로 새 스레드 인스턴스를 실행할 수 있습니다.

기본 우선 순위 오버라이드

스레드가 수신해야 하는 CPU 시간의 양이 스레드 작업에 암시된 경우 생성자에 해당 우선 순위를 설정해 둡니다. 이 내용은 9-2 페이지의 "스레드 초기화"에 설명되어 있습니다. 그러나 스레드 실행 시기에 따라 스레드 우선 순위가 달라지는 경우 정지된 상태에서 스레드를 작성하고 우선 순위를 설정한 다음 다음과 같은 방법으로 스레드 실행을 시작합니다.

```
SecondProcess := TMyThread.Create(True); { create but don't run }
SecondProcess.Priority := tpLower; { set the priority lower than normal }
SecondProcess.Resume; { now run the thread }
```

참고 크로스 플랫폼 애플리케이션 작성 시 Windows와 Linux의 우선 순위를 할당하기 위해서는 별도의 코드를 사용해야 합니다. Linux에서 *우선 순위*는 루트에 의해서만 변경될 수 있는 스레드 정책에 따른 숫자 값입니다. 자세한 내용은 *TThread*의 CLX 버전 및 *우선 순위* 온라인 도움말을 참조하십시오.

스레드 시작 및 중지

스레드는 실행을 완료하기 전에 얼마든지 시작되고 중지될 수 있습니다. 스레드를 일시적으로 중지하려면 스레드의 *Suspend* 메소드를 호출합니다. 스레드를 다시 시작하는 것이 안전하면 스레드의 *Resume* 메소드를 호출합니다. *Suspend*는 내부 카운터를 증가시켜 *Suspend*와 *Resume*에 대한 호출을 중첩시킬 수 있습니다. 스레드는 모든 중지자 *Resume*에 대한 호출과 일치할 경우 다시 실행됩니다.

Terminate 메소드를 호출하여 스레드가 미리 실행 종료되도록 요청할 수 있습니다. *Terminate*는 스레드의 *Terminated* 속성을 *True*로 설정합니다. *Execute* 메소드를 적절히 구현했을 경우 *Execute*는 *Terminated* 속성을 정기적으로 확인하고 *Terminated*가 *True*일 때 실행을 중지합니다.

다중 스레드 애플리케이션 디버깅

다중 스레드 애플리케이션 디버깅할 경우 동시에 실행되는 모든 스레드의 상태를 추적하거나 심지어 브레이크포인트에서 중지할 때 어떤 스레드가 실행 중인지 확인하는 것이 혼동될 수 있습니다. Thread Status 상자를 사용하면 애플리케이션의 모든 스레드를 추적하고 처리할 수 있습니다. Thread Status 상자를 표시하려면 메인 메뉴에서 View | Threads를 선택합니다.

브레이크포인트, 예외, 일시 정지 등의 디버그 이벤트가 발생하면 Thread Status 뷰는 각 스레드의 상태를 나타냅니다. Thread Status 상자를 마우스 오른쪽 버튼으로 클릭하여 해당 소스 위치를 찾는 명령에 액세스하거나 다른 스레드를 현재 사용하는 것으로 만듭니다. 스레드가 현재 사용 중인 것으로 표시되면 해당 스레드에 따라 다음 단계나 실행 작업이 결정됩니다.

Thread Status 상자는 스레드 ID로 애플리케이션의 모든 실행 스레드를 나열합니다. 스레드 객체를 사용할 경우 스레드 ID는 *ThreadID* 속성 값입니다. 스레드 객체를 사용하지 않을 경우 각 스레드의 스레드 ID는 *BeginThread*에 대한 호출에 의해 반환됩니다.

Thread Status 상자에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

10

크로스 플랫폼 개발을 위한 CLX 사용

Delphi를 이용하여 Windows 및 Linux 운영 체제 모두에서 실행되는 크로스 플랫폼 32비트 애플리케이션을 개발할 수 있습니다. 기존 Windows 애플리케이션을 수정하거나 권장하는 플랫폼 독립적인 코드 작성 방법을 따라 새 애플리케이션을 만들 수 있습니다. Kylix는 Linux에서 애플리케이션을 컴파일하고 개발할 수 있는 Borland의 Linux용 Delphi 소프트웨어입니다. Linux와 Windows에서 애플리케이션을 개발하고 배포하려면 Delphi는 물론 Kylix도 사용해야 합니다.

이 장에서는 Linux에서 컴파일을 할 수 있도록 Delphi 애플리케이션을 변경하는 방법을 설명하며 Windows와 Linux에서의 애플리케이션 개발의 차이에 관한 정보도 제공합니다. 또한 서로 다른 두 환경 사이에서 이식 가능한 코드 작성에 대한 지침도 제공합니다.

참고 운영 체제 특정 API 호출이 없는 CLX를 사용하여 개발한 대부분의 애플리케이션은 Linux와 Windows 플랫폼 모두에서 실행됩니다. 이러한 애플리케이션은 실행할 플랫폼에서 컴파일해야 합니다.

크로스 플랫폼 애플리케이션 만들기

Delphi 애플리케이션을 만드는 것처럼 크로스 플랫폼 애플리케이션을 쉽게 만들 수 있습니다. 애플리케이션이 완벽한 크로스 플랫폼이 되게 하려면 CLX 비주얼 컴포넌트를 사용해야 하며 운영 체제 특정 API를 사용하지 않아야 합니다. (크로스 플랫폼 애플리케이션 작성에 대한 정보는 10-17 페이지의 "포팅 가능한 코드 작성"을 참조하십시오.)

다음과 같은 방법으로 크로스 플랫폼 애플리케이션을 만듭니다.

- 1 IDE에서 File|New|CLX application을 선택합니다.
컴포넌트 팔레트가 CLX 애플리케이션에서 사용할 수 있는 컴포넌트를 보여 줍니다.

참고 일부 Windows 전용 논비주얼(nonvisual) 컴포넌트는 CLX 애플리케이션에서 사용할 수 있습니다. 컴포넌트 팔레트에는 Windows CLX 애플리케이션에서만 동작하는 기능을 가진 ADO, BDE, System, DataSnap, InterBase, Site Express, FastNet, QReport, COM+, BizSnap 및 Servers 탭이 있습니다. Linux에서 애플리케이션을 컴파일하려는 경우에는 이 탭의 컴포넌트를 사용하거나 \$IFDEF를 사용하여 코드의 이 부분을 Windows 전용으로 표시하지 마십시오.

- 2 IDE에서 애플리케이션을 개발합니다. 애플리케이션에서 CLX 컴포넌트만 사용해야 합니다.
- 3 애플리케이션을 실행할 각 플랫폼에서 애플리케이션을 컴파일하고 테스트합니다. 추가로 변경해야 할 곳이 없는지 알아보기 위해서 모든 오류 메시지를 검토합니다.

Kylix로 애플리케이션을 이동할 때에는 프로젝트 옵션을 재설정해야 합니다. 그 이유는 프로젝트 옵션을 저장하는 .dof 파일이 Kylix에서 재생성되고 기본 옵션 집합이 있는 .kof를 호출하기 때문입니다. Ctrl+O+O를 입력하여 애플리케이션에 많은 컴파일러 옵션을 저장할 수도 있습니다. 옵션은 현재 열린 파일의 첫 부분에 있습니다.

크로스 플랫폼 애플리케이션의 폼 파일은 dfm이 아닌 xfm 확장자를 갖습니다. CLX 컴포넌트를 사용하는 크로스 플랫폼의 폼과 VCL 컴포넌트를 사용하는 폼을 구별하기 위한 것입니다. xfm 폼 파일은 Windows나 Linux 모두에서 사용되지만 dfm 폼은 Windows에서만 사용됩니다.

Delphi 대신 Kylix에서 크로스 플랫폼 애플리케이션 개발을 시작할 수도 있습니다.

- 1 Kylix를 사용하여 Linux에서 애플리케이션을 개발하고 컴파일하고 테스트합니다.
- 2 애플리케이션 소스 파일을 Windows로 이동합니다.
- 3 프로젝트 옵션을 재설정합니다.
- 4 Delphi를 사용하여 애플리케이션을 Windows에서 다시 컴파일합니다.

플랫폼 독립 데이터베이스나 인터넷 애플리케이션을 작성하는 방법에 대한 자세한 내용은 10-23 페이지의 "크로스 플랫폼 데이터베이스 애플리케이션"과 10-30 페이지의 "크로스 플랫폼 인터넷 애플리케이션"을 참조하십시오.

VCL 애플리케이션을 CLX로 포팅

Windows 환경을 위해 작성된 Delphi 애플리케이션이 있는 경우, 이를 크로스 플랫폼으로 만들 수 있습니다. 포팅의 난이도는 애플리케이션의 특성 및 복잡성, Windows 종속 정도에 따라 다릅니다.

다음 단원에서는 Windows와 Linux 환경 간의 주요한 차이점 몇 가지를 설명하고 애플리케이션 포팅 입문에 대한 지침을 제공합니다.

포팅 기법

다음은 애플리케이션을 한 플랫폼에서 다른 플랫폼으로 포팅하기 위해 이용할 수 있는 여러 가지 방법들입니다.

표 10.1 포팅 기법

기술	설명
플랫폼 특정 포팅	운영 체제와 기본으로 사용한 API를 대상으로 함
크로스 플랫폼 포팅	크로스 플랫폼 API를 대상으로 함
Windows 에뮬레이션	코드는 그대로 두고 코드에서 사용하는 API를 포팅함

플랫폼 특정 포팅

플랫폼 특정 포팅은 시간이 걸리고 비용이 많이 들며 주로 해당 플랫폼에서만 애플리케이션을 사용할 수 있습니다. 플랫폼 특정 포팅은 특히 유지 관리가 어려운 다른 코드를 기준으로 만듭니다. 하지만 각 포팅은 특정 운영 체제를 위해 설계된 것으로 플랫폼 특정 기능을 활용할 수 있습니다. 따라서 애플리케이션이 대체로 보다 빠르게 실행됩니다.

크로스 플랫폼 포팅

크로스 플랫폼 포팅은 일반적으로 가장 최신 기법을 제공하며 포팅된 애플리케이션은 다중 플랫폼을 대상으로 합니다. 실제로 크로스 플랫폼 애플리케이션 개발에 포함되는 작업의 양은 기존 코드에 따라 결정됩니다. 코드가 플랫폼 독립성을 염두에 두지 않고 개발되었다면 플랫폼 독립적인 "로직"과 플랫폼 독립적인 "구현"이 같이 섞이는 경우가 있을 수 있습니다.

크로스 플랫폼 접근법은 비즈니스 로직이 플랫폼 독립적인 관점에서 개발되므로 더 선호됩니다. 일부 서비스는 모든 플랫폼에서 같아 보이지만 개별적으로는 특정한 구현을 가진 내부 인터페이스를 통해 추출됩니다. 그 한 가지 예가 Delphi의 런타임 라이브러리입니다. 두 플랫폼에서는 인터페이스가 매우 유사하지만 구현 내용은 크게 다릅니다. 크로스 플랫폼 부분을 분리한 후 그 위에 특정 서비스를 구현해야 합니다. 결국 이 접근법은 폭넓게 공유되는 소스 기반과 개선된 애플리케이션 아키텍처로 인한 유지 비용 감소를 통해 가장 비용이 적게 드는 솔루션입니다.

Windows 에뮬레이션 포팅

Windows 에뮬레이션은 가장 복잡한 방법이며 비용이 많이 들 수 있지만 결과물인 Linux 애플리케이션이 기존 Windows 애플리케이션과는 가장 유사하게 보입니다. 이 접근법은 Linux에 Windows 기능을 구현하는 것을 포함합니다. 엔지니어링의 관점에서 볼 때 이 솔루션은 유지 관리가 가장 어렵습니다.

Windows API를 에뮬레이트하려는 경우 Windows 또는 Linux에만 적용되는 코드의 섹션을 표시하기 위해 `$IFDEF`를 사용하면 두 개의 섹션으로 구분할 수 있습니다.

애플리케이션 포팅

하지만 Windows와 Linux 모두에서 실행하고자 하는 애플리케이션을 포팅하는 경우 코드를 수정하거나 **\$IFDEF**를 사용하여 Windows 또는 Linux에만 적용되는 코드의 섹션을 표시합니다.

VCL 애플리케이션을 CLX로 포팅하려면 다음과 같은 일반적인 절차를 따릅니다.

- 1 Delphi에서 변경할 애플리케이션이 들어 있는 프로젝트를 엽니다.
- 2 .dfm 파일들을 동일한 이름의 .xfm 파일로 복사합니다 (예를 들어, unit1.dfm을 unit1.xfm으로 이름 변경). 유닛 파일에서 .dfm 파일에 대한 참조를 {\$R *.dfm}에서 {\$R *.xfm}으로 이름 변경 (또는 **\$IFDEF**) 합니다. (.xfm 파일은 Kylix와 Delphi 모두에서 동작합니다.)

예를 들면 다음과 같은 **implementation** 섹션의 폼 참조를

```
{$R *.dfm}에서
```

다음 내용으로 변경합니다

```
{$R *.xfm}
```

- 3 모든 **uses** 절을 변경 (또는 **\$IFDEF**) 하여 CLX의 정확한 유닛을 참조하게 합니다. (자세한 내용은 10-10 페이지의 "CLX와 VCL 유닛 비교" 참조)

예를 들어 Windows 애플리케이션에서 다음 **uses** 절을

```
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
```

다음과 같이 CLX 애플리케이션을 위한 절로 변경합니다.

```
uses Windows, Messages, SysUtils, Variants, Classes, QForms, QControls, QStdCtrls;
```

- 4 프로젝트를 저장하고 다시 엽니다. 이제 컴포넌트 팔레트에 CLX 애플리케이션에서 사용할 수 있는 컴포넌트가 표시됩니다.

참고

일부 Windows 전용 논비주얼 (nonvisual) 컴포넌트는 CLX 애플리케이션에서 사용될 수 있습니다. 컴포넌트 팔레트에는 Windows CLX 애플리케이션에서만 동작하는 기능을 가진 ADO, BDE, System, DataSnap, InterBase, Internet Express, Site Express, FastNet, QReport, COM+, Web Services 및 Servers 탭이 있습니다. Linux에서 애플리케이션을 컴파일하려는 경우에는 이 탭의 컴포넌트를 사용하거나 **\$IFDEF**를 사용하여 코드의 이 부분을 Windows 전용으로 표시하지 마십시오.

- 5 Windows 종속성을 필요로 하지 않는 모든 코드를 다시 작성하여 플랫폼 독립적으로 만듭니다. 런타임 라이브러리 루틴과 상수를 사용하여 이 작업을 수행합니다. (자세한 내용은 10-17 페이지의 "포팅 가능한 코드 작성" 참조)
- 6 Linux에서 다른 기능에 대해 동등한 기능을 찾습니다. **\$IFDEF**를 가능한 적게 사용하여 Windows 특정 정보를 구분합니다. (자세한 내용은 10-18 페이지의 "조건 지시어 사용" 참조)

예를 들어 소스 파일에서 다음과 같이 플랫폼 특정 코드를 **\$IFDEF**할 수 있습니다.

```
[$IFDEF MSWINDOWS]
```



```

IniFile.LoadFromFile('c:\x.txt');
[ENDIF]

[IFDEF LINUX]
IniFile.LoadFromFile('/home/name/x.txt');
[ENDIF]

```

7 모든 프로젝트 파일에서 경로에 대한 참조를 검색합니다.

- Linux의 경로명은 슬래시 (/)를 구분자로 사용하며 (예를 들어, /usr/lib) Linux 시스템에서는 파일들이 다른 디렉토리에 위치할 수 있습니다. SysUtils에서 PathDelim 상수를 사용하여 시스템에 맞는 경로 구분자를 지정합니다. Linux 상의 파일에 대한 정확한 위치를 정합니다.
- 드라이브 문자에 대한 참조(예를 들어, C:\)와 문자열에서 두 번째 위치에 있는 콜론을 찾아서 드라이브 문자를 검색하는 코드를 변경합니다. SysUtils에서 DriveDelim 상수를 사용하여 시스템에 맞게 위치를 지정합니다.
- 다중 경로를 지정하는 곳에서 경로 분리자를 세미콜론 (;)에서 콜론(:)으로 변경합니다. SysUtils에서 PathSep 상수를 사용하여 시스템에 맞는 경로 분리자를 지정합니다.
- Linux에서는 파일 이름의 대소문자를 구별하므로 애플리케이션이 파일 이름의 대소문자를 변경하거나 대소문자 중 하나인 것으로 가정하지 않았는지 확인합니다.

8 애플리케이션을 컴파일, 텍스트 및 디버그합니다.

다음과 같은 방법으로 애플리케이션을 Linux로 이전합니다.

- 1 Delphi Windows 애플리케이션 소스 파일과 기타 프로젝트 관련 파일을 Linux 컴퓨터로 이동합니다. (프로그램을 Linux와 Windows 두 플랫폼에서 모두 실행할 경우 Linux와 Windows 간에 소스 파일을 공유할 수 있습니다. 또는 ASCII 모드를 사용하는 ftp와 같은 도구를 사용하여 파일을 전송할 수도 있습니다.)

소스 파일에는 유닛 파일(.pas 파일), 프로젝트 파일(.dpr 파일) 및 모든 패키지 파일(.dpk 파일)이 포함됩니다. 프로젝트 관련 파일에는 폼 파일(.xfm 파일), 리소스 파일(.res 파일) 및 프로젝트 옵션 파일(Kylix의 .dof 파일, .kof 파일로 변경됨)이 포함됩니다. IDE를 사용하지 않고 명령줄에서만 애플리케이션을 컴파일하려면 구성 파일(Kylix의 .cfg 파일, .conf로 변경됨)이 필요합니다.

- 2 Kylix에서 프로젝트를 엽니다. 사용 중인 Windows 특정 기능에 대한 경고가 나타납니다.
- 3 Kylix를 사용하여 프로젝트를 컴파일합니다. 추가로 변경해야 할 곳이 없는지 알아보기 위해서 모든 오류 메시지를 검토합니다.

CLX와 VCL 비교

Kylix는 Visual Component Library(VCL) 대신 Cross Platform(CLX)용 Borland Component Library를 사용합니다. VCL 내에서 많은 컨트롤이 Windows 컨트롤에 액세스하기 위한 쉬운 방법을 제공합니다. 이와 마찬가지로 CLX는 Qt 공유 라이브러리에서 Qt widget(window와 gadget을 합한 것)에 대한 액세스를 제공합니다. Delphi는 CLX와 VCL을 모두 포함합니다.

CLX는 VCL과 매우 유사합니다. 대부분의 컴포넌트 이름이 동일하며 많은 속성들이 동일한 이름을 가집니다. 또한 VCL뿐만 아니라 CLX도 Windows에서 사용 가능하게 될 것입니다(사용 가능한지 알아보려면 Delphi의 최신 버전을 확인합니다).

CLX 컴포넌트는 다음과 같이 그룹화될 수 있습니다.

표 10.2 CLX 부분

부분	설명
VisualCLX	윈시 크로스 플랫폼 GUI 컴포넌트와 그래픽. 이 부분의 컴포넌트는 Linux와 Windows에서 서로 다를 수 있습니다.
DataCLX	클라이언트 데이터 액세스 컴포넌트. 이 부분의 컴포넌트는 클라이언트 데이터셋에 기반한 로컬, 클라이언트/서버 및 n계층의 하위 집합입니다. 코드는 Linux와 Windows에서 동일합니다.
NetCLX	Apache DSO와 CGI Web Broker를 포함한 인터넷 컴포넌트. Linux와 Windows에서 동일합니다.
RTL	구현되고 Classes.pas를 포함하는 런타임 라이브러리. 코드는 Linux와 Windows에서 동일합니다.

VisualCLX의 widget은 Windows 컨트롤을 대체합니다. CLX에서 *TWidgetControl*은 VCL의 *TWinControl*을 대체합니다. *TScrollingWidget*과 같은 다른 컴포넌트는 해당 이름을 가집니다. 하지만 *TWinControl*을 *TWidgetControl*로 변경할 필요는 없습니다. 예를 들어 다음과 같은 타입 선언의 경우,

```
TWinControl = TWidgetControl;
```

QControls.pas 소스 파일에 사용하여 소스 코드의 공유를 간소화합니다. *TWidgetControl*과 해당 자손들은 모두 Qt 객체의 참조인 *Handle* 속성과 이벤트 메커니즘을 처리하는 Hook 객체의 참조인 *Hooks* 속성을 갖습니다.

일부 클래스의 유닛 이름과 위치가 CLX에서는 다릅니다. CLX에서 존재하지 않는 유닛에 대한 참조를 제거하고 이름을 CLX 유닛으로 변경하려면 **uses** 절을 수정해야 합니다. (대부분의 프로젝트 파일과 대부분 유닛의 인터페이스 섹션에는 **uses** 절이 있습니다. 유닛의 구현 섹션도 자체 **uses** 절을 가집니다.)

CLX가 다르게 하는 동작

많은 CLX가 VCL과 일관되도록 구현되지만 일부 기능은 다르게 구현됩니다. 이 단원에서는 크로스 플랫폼 애플리케이션을 작성할 경우 나타나는 CLX와 VCL 구현 간의 일부 차이점에 대한 개요를 제공합니다.

룩앤필(Look and feel)

Linux의 비주얼 환경은 Windows와 약간 다르게 보입니다. 대화 상자의 모양이 어떤 윈도우 관리자(예를 들어, KDE나 Gnome)를 사용하는가에 따라 다를 수 있습니다.

스타일

애플리케이션 전반의 "스타일"을 *OwnerDraw* 속성에 추가하여 사용할 수 있습니다. *TApplication.Style* 속성을 사용하여 애플리케이션 그래픽 요소의 록엔필을 지정할 수 있습니다. 스타일을 사용하여 widget이나 애플리케이션이 완전히 새로운 모습을 갖출 수 있습니다. Linux에서 *owner-draw*를 사용할 수 있지만 스타일 사용을 권합니다.

Variants

시스템에 있던 모든 가변 타입 배열/*SafeArray* 코드는 이제는 다음 두 개의 새로운 유닛입니다.

- Variants.pas
- VarUtils.pas

운영 체제 의존 코드는 이제 *VarUtils.pas* 파일에 격리되어 있고 *Variants.pas*가 필요로 하는 모든 일반 버전도 포함합니다. Windows로부터 CLX 애플리케이션으로의 호출을 포함하는 VCL 애플리케이션을 변환하는 경우 이 호출을 *VarUtils.pas*로의 호출로 대체해야 합니다.

가변 타입을 사용하려면 **uses** 절에서 Variants 유닛을 포함해야 합니다.

*VarIsEmpty*는 *varEmpty*에 대해 간단한 검사를 하여 가변이 지워졌는지 확인하며 Linux에서는 *VarIsClear* 함수를 사용하여 가변을 지웁니다.

사용자 지정 가변 타입 데이터 핸들러

가변에 대해 사용자 지정 데이터 타입을 정의할 수 있습니다. 타입이 가변으로 할당될 때 연산자가 오버로드되게 합니다. 새 가변 타입을 만들려면 *TCustomVariantType* 클래스의 자손으로 새로운 가변 타입을 인스턴스화합니다.

예를 들어, *VarCmplx.pas*를 보면 이 유닛은 사용자 지정 가변 타입을 통해 복잡한 수학적 지원을 구현합니다. 이 유닛은 더하기, 빼기, 곱하기, 나누기(정수 나누기는 아님), 부정과 같은 가변 연산을 지원합니다. 또한 *SmallInt*, 정수, *Single*, *Double*, 통화, 날짜, 부울, 바이트, *OleStr* 및 문자열의 변환을 처리합니다. 모든 실수 타입/순서 타입 변환에서는 복잡한 값의 허수 부분을 버립니다.

Registry

Linux에서는 구성 정보를 저장하기 위한 레지스트리는 사용하지 않습니다. 레지스트리 대신 텍스트 구성 파일과 환경 변수를 사용합니다. Linux의 시스템 구성 파일이 가끔 */etc/hosts*처럼 */etc*에 위치하기도 합니다. 다른 사용자 프로파일은 bash 셸 설정이나 X 프로그램의 기본값을 설정하는 데 사용하는 *.XDefaults*를 보유하는 점(.)이 앞에 오는 숨김 파일(예: *.bashrc*)에 위치합니다.

레지스트리 의존 코드는 애플리케이션과 동일한 디렉토리에 저장하는 대신 로컬 구성 텍스트 파일을 사용하여 변경할 수 있습니다. 로컬 구성 파일로 모든 결과를 내보내는 것을 제외하고 모든 레지스트리 함수를 포함한 유닛을 작성하는 것이 레지스트리에 대한 이전의 의존성을 처리하는 한 방법입니다.

Linux 상의 전역 위치에 정보를 입력하기 위해 루트 디렉토리에 전역 구성 파일을 저장할 수 있습니다. 이는 사용자의 애플리케이션이 동일한 구성 파일에 액세스할 수 있게 합니다. 하지만 파일 권한과 액세스 권한이 올바르게 설정되었는지 확인해야 합니다.

크로스 플랫폼 애플리케이션에서 ini 파일도 사용할 수 있습니다. 그러나 CLX에서는 *TRegIniFile* 대신 *TMemIniFile*을 사용해야 합니다.

그 밖의 차이점

CLX 구현에서도 애플리케이션이 동작하는 방법에 영향을 끼치는 몇 가지 다른 차이점이 있습니다. 이 단원에서는 이러한 차이점에 대해 설명합니다.

*ToggleButton*은 Enter 키로 토글되지 않습니다. Enter 키를 눌러도 Kylix에서는 Delphi에서처럼 클릭 이벤트가 발생하지 않습니다.

*TColorDialog*는 설정할 *TColorDialog.Options* 속성이 없습니다. 따라서 색상 선택 대화 상자의 모양 및 기능을 사용자 지정할 수 없습니다. 또한 *TColorDialog*가 항상 모달은 아닙니다. Kylix에서 애플리케이션의 제목 표시줄을 모달 대화 상자로 조작할 수 있습니다(즉, 색상 대화 상자의 부모 폼을 선택하고 색상 대화 상자가 열려 있는 동안 최대화하는 등의 작업을 할 수 있습니다.).

런타임에 콤보 박스는 Delphi에서와 다르게 Kylix에서 동작합니다. Kylix(Delphi는 아님)에서 콤보 박스의 편집 필드에 텍스트를 입력하고 Enter 키를 눌러서 드롭다운에 항목을 추가할 수 있습니다. *InsertMode*를 *ciNone*으로 설정하여 이 기능을 끌 수 있습니다. 콤보 박스의 목록에 빈(문자열이 아닌) 항목을 추가할 수도 있습니다. 또한 아래쪽 화살표를 계속 누르면 콤보 박스 목록의 마지막 항목에서 계속 스크롤됩니다. 위에서부터 다시 반복됩니다.

*TCustomEdit*는 *Undo*, *ClearUndo* 또는 *CanUndo*를 구현하지 않습니다. 따라서 프로그램에서 편집을 취소할 방법이 없습니다. 그러나 애플리케이션 사용자는 런타임에 편집 상자를 마우스 오른쪽 단추로 클릭하고 Undo 명령을 선택하여 편집 상자(*TEdit*)의 편집 내용을 취소할 수 있습니다.

Windows에서 Enter 키에 대한 *OnKeyDown* 이벤트나 *KeyUp* 이벤트의 키 값은 13입니다. Linux에서 이 값은 4100입니다. Enter 키에 대한 13과 같은 키에 대해 하드 코딩된 숫자 값을 검사하는 경우에는 Delphi 애플리케이션을 Kylix로 포팅할 때 이 값을 변경해야 합니다.

이외에도 다른 점이 있습니다. 모든 CLX 객체에 대한 자세한 내용은 CLX 온라인 문서를 참조하고 참조하려는 코드를 보려면 소스 코드가 들어 있는 Delphi 버전에서 `\Delphi\Source\VCL\CLX`를 참조하십시오.

CLX에 없는 내용

VCL 대신 CLX를 사용하는 경우 많은 객체들이 동일합니다. 하지만 이 객체들은 속성, 메소드 또는 이벤트와 같은 일부 기능이 없을 수 있습니다. 다음과 같은 일반적인 기능이 CLX에는 없습니다.

- 오른쪽에서 왼쪽 텍스트 입력이나 출력을 위한 양방향 속성 (*BidiMode*)
- 공용 컨트롤의 일반 베벨 속성 (일부 객체에는 아직도 베벨 속성이 있음)
- 도킹 속성 및 메소드
- Win3.1 탭 및 *Ctl3D*와 같은 컴포넌트에 대한 역호환 기능
- *DragCursor*와 *DragKind*(끌어서 놓기는 포함되지 않음)

포팅되지 않는 기능

Delphi에서 지원되는 일부 Windows 특정 기능은 Linux 환경으로 직접 이전되지 않을 수 있습니다. COM, ActiveX, OLE, BDE 및 ADO와 같은 기능은 Windows 기술에 종속되므로 Kylix에서는 사용할 수 없습니다. 다음 표는 두 플랫폼 간에 다르게 나타나는 기능들과 Kylix에 해당 기능이 있는 경우 그 기능들을 나열한 것입니다.

표 10.3 변경되었거나 다른 기능

Delphi/Windows 기능	Kylix/Linux 기능
ADO 컴포넌트	일반 데이터베이스 컴포넌트
Automation Servers	사용 불가
BDE	dbExpress 및 일반 데이터베이스 컴포넌트
COM+ 컴포넌트 (ActiveX 포함)	사용 불가
DataSnap	아직 사용할 수 없음
FastNet	사용 불가
Internet Express	아직 사용할 수 없음
레거시 컴포넌트 (예: Win 3.1 컴포넌트 팔레트 탭의 항목)	사용 불가
MAPI (Messaging Application Programming Interface)에는 Windows 메시징 기능의 표준 라이브러리가 있음	SMTP/POP3로 전자 우편 메시지를 보내고 받고 저장할 수 있음
Quick Reports	사용 불가
Web Services (SOAP)	아직 사용할 수 없음
WebSnap	아직 사용할 수 없음
Windows API 호출	CLX 메소드, Qt 호출, libc 호출 또는 다른 시스템 라이브러리에 대한 호출
Windows 메시징	Qt 이벤트
Winsock	BSD 소켓

Linux에서 Windows DLL에 해당하는 것은 공유 객체 라이브러리 (.so 파일)로서 위치 독립 코드 (PIC)를 포함합니다. 다음과 같은 결과가 나타납니다.

- 메모리의 절대 주소를 참조하는 변수 (**absolute** 지시어 사용)는 허용되지 않습니다.
- 외부 함수에 대한 전역 메모리 참조와 호출은 호출 간에 보존되어야 하는 EBX 레지스터에 상대적으로 만들어집니다.

Kylix 또는 Delphi는 정확한 코드를 생성하므로 어셈블러를 사용할 경우 외부 함수에 대한 전역 메모리 참조와 호출만 신경 쓰면 됩니다. 자세한 내용은 10-21 페이지의 "인라인 어셈블러 코드 포함"을 참조하십시오.

Kylix 라이브러리 모듈과 패키지는 .so 파일을 사용하여 구현합니다.

CLX와 VCL 유닛 비교

VCL 또는 CLX의 모든 객체는 유닛 파일인 .pas 소스 파일에 정의되어 있습니다. 예를 들어, 시스템 유닛에는 *TObject*의 구현이 있으며 클래스스 유닛은 기본 *TComponent* 클래스를 정의합니다. 폼에 객체를 갖다 놓거나 애플리케이션 내에서 객체를 사용하면 해당 유닛의 이름이 **uses** 절에 추가되어 컴파일러에게 프로젝트에 어느 유닛을 연결해야 하는지 알려 줍니다.

이 단원에는 CLX와 VCL에 모두 들어 있는 유닛, CLX에만 있는 유닛 및 VCL에만 있는 유닛을 나열한 표가 있습니다.

다음 표는 VCL과 CLX에 모두 들어 있는 유닛을 나열한 것입니다.

표 10.4 VCL와 CLX의 유닛

VCL 유닛	CLX 유닛
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Classes	Classes
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts, Qconsts 및 RTLConsts
Contnrs	Contnrs
Controls	QControls
DateUtils	DateUtils
DB	DB
DBActns	QDBActns
DBClient	DBClient
DBCommon	DBCommon
DBConnAdmin	DBConnAdmin
DBConsts	DBConsts
DBCtrls	QDBCtrls
DBGrids	QDBGrids
DBLocal	DBLocal
DBLocalS	DBLocalS
DBLogDlg	DBLogDlg
DBXpress	DBXpress
Dialogs	QDialogs

표 10.4 VCL와 CLX의 유닛 (계속)

VCL 유닛	CLX 유닛
DSIntf	DSIntf
ExtCtrls	QExtCtrls
FMTBCD	FMTBCD
Forms	QForms
Graphics	QGraphics
Grids	QGrids
HelpIntfs	HelpIntfs
ImgList	QImgList
IniFiles	IniFiles
Mask	QMask
MaskUtils	MaskUtils
Masks	Masks
Math	Math
Menus	QMenus
Midas	Midas
MidConst	MidConst
Printers	QPrinters
Provider	Provider
Qt	Qt
Search	QSearch
Sockets	Sockets
StdActns	QStdActns
StdCtrls	QStdCtrls
SqlConst	SqlConst
SqlExpr	SqlExpr
SqlTimSt	SqlTimSt
SyncObjs	SyncObjs
SysConst	SysConst
SysInit	SysInit
System	System
SysUtils	SysUtils
Types	Types와 QTypes
TypInfo	TypInfo
Variants	Variants
VarUtils	VarUtils

다음 유닛은 CLX에는 있지만 VCL에는 없습니다.

표 10.5 CLX에는 있고 VCL에는 없는 유닛

유닛	설명
DirSel	디렉토리 선택
QStyle	GUI 룩앤필

다음 Windows VCL 유닛은 ADO, COM 및 BDE와 같은 Linux에는 없는 Windows 특정 기능이므로 CLX에 포함되어 있지 않습니다. 이 유닛들이 포함되지 않은 이유들이 그 옆에 나열되어 있습니다.

표 10.6 VCL에만 있는 유닛

유닛	포함되지 않은 이유
ADOCnst	ADO 기능 없음
ADODB	ADO 기능 없음
AppEvnts	TApplicationEvent 객체 없음
AxCtrls	COM 기능 없음
BdeConst	BDE 기능 없음
ComStrs	COM 기능 없음
ConvUtils	Delphi 6의 새로운 기능
CorbaCon	CORBA 기능 없음
CorbaStd	CORBA 기능 없음
CorbaVCL	CORBA 기능 없음
CtlPanel	Windows 제어판 지원 안함
DataBkr	나중에 upsell에 포함될 수도 있음
DBCGrids	BDE 기능 없음
DBExcept	BDE 기능 없음
DBInpReq	BDE 기능 없음
DBLookup	폐기됨
DbOleCtl	COM 기능 없음
DBPWDlg	BDE 기능 없음
DBTables	BDE 기능 없음
DdeMan	ADO 기능 없음
DRTable	BDE 기능 없음
ExtActns	Delphi 6의 새로운 기능
ExtDlgs	그림 대화 상자 없음
FileCtrl	폐기됨
ListActns	Delphi 6의 새로운 기능
MConnect	COM 기능 없음
Messages	Windows 특정 영역
MidasCon	폐기됨
MPlayer	Windows 특정 미디어 플레이어

표 10.6 VCL에만 있는 유닛 (계속)

유닛	포함되지 않은 이유
Mtsobj	COM 기능 없음
MtsRdm	COM 기능 없음
Mtx	COM 기능 없음
mxConsts	COM 기능 없음
ObjBrkr	나중에 upsell에 포함될 수도 있음
OleConstMay	COM 기능 없음
OleCtnrs	COM 기능 없음
OleCtrls	COM 기능 없음
OLEDB	COM 기능 없음
OleServer	COM 기능 없음
Outline	폐기됨
Registry	Windows 특정 레지스트리 지원
ScktCnst	소켓에 의해 교체됨
ScktComp	소켓에 의해 교체됨
SConnect	지원되지 않는 연결 프로토콜
StdConvS	Delphi 6의 새로운 기능
SvcMgr	NT 서비스 지원
Tabnotbk	폐기됨
Tabs	폐기됨
ToolWin	도킹 기능 없음
VarCmplx	Delphi 6의 새로운 기능
VarConv	Delphi 6의 새로운 기능
VCLCom	COM 기능 없음
WebConst	Windows 특정 상수
Windows	Windows 특정 (API)

CLX 객체 생성자의 차이점

폼 디자이너에서 암시적으로 폼에 해당 객체를 넣거나 또는 코드에서 객체의 *Create* 메소드를 명시적으로 사용하여 CLX 객체를 만들면 기본적으로 사용하는 연결된 widget의 인스턴스도 생성됩니다. widget의 인스턴스는 이 CLX 객체가 소유합니다. *Free* 메소드를 사용하여 CLX 객체를 삭제하거나 CLX 객체의 부모 컨테이너에 의해 CLX가 자동으로 삭제되는 경우에는 기본적으로 사용하는 widget도 삭제됩니다. 이 기능은 Windows 애플리케이션의 VCL에서 볼 수 있는 것과 동일한 유형입니다.

`QWidget_Create()`와 같은 Qt 인터페이스 라이브러리로 호출함으로써 코드에서 명시적으로 CLX 객체를 생성하는 경우에는 CLX 객체가 소유하지 않는 Qt widget의 인스턴스를 생성합니다. 이렇게 하면 생성 중에 사용하기 위해 기존 Qt widget의 인스턴스가 CLX 객체로 전달됩니다. 이 CLX 객체는 자신에게 넘겨진 Qt widget을 소유하지 않습니다. 따라서 이러한 방법으로 객체를 만든 후 *Free* 메소드를 호출하면 CLX 객체만

소멸되고 기본으로 사용하는 Qt widget 인스턴스는 소멸되지 않습니다. 이것이 VCL과 다른 점입니다.

일부 CLX 객체는 *OwnHandle* 메소드를 사용하여 기본으로 사용하는 widget의 소유권을 추정해 볼 수 있습니다. *OwnHandle*을 호출한 후 CLX 객체를 삭제하면 기본으로 사용하는 widget도 소멸됩니다.

Windows와 Linux 간의 소스 파일 공유

애플리케이션이 Windows와 Linux 모두에서 실행되게 하려면 두 운영 체제에서 액세스할 수 있도록 소스 파일을 공유하면 됩니다. 두 컴퓨터에서 액세스할 수 있는 서버에 소스 파일을 저장하거나 Linux 컴퓨터의 Samba를 사용하여 Linux와 Windows용 Microsoft 네트워크 파일을 공유하여 파일에 대한 액세스를 제공하는 등 소스 파일을 공유하는 방법은 여러 가지가 있습니다. Linux에서 소스를 보관하도록 하고 Linux에 공유 드라이브를 만들 수 있습니다. 또는 Windows에 소스를 두고 공유를 만들어 Linux 컴퓨터가 액세스할 수 있게 할 수 있습니다.

VCL과 CLX 모두에서 지원하는 객체를 사용하면 Kylix에서 파일을 계속적으로 개발하고 컴파일할 수 있습니다. 완료되면 Linux와 Windows 모두에서 컴파일할 수 있습니다.

Kylix에서는 폼 파일(Delphi의 .dfm 파일)이 .xpm 파일입니다. Delphi나 Kylix에서 새로운 CLX 애플리케이션을 만드는 경우, .dfm 대신 .xpm이 생성됩니다. 크로스 플랫폼 폼 애플리케이션을 작성하려는 경우에는 Delphi와 Kylix 모두에서 .xpm을 사용할 수 있습니다.

Windows와 Linux 간의 환경적 차이점

여기서 크로스 플랫폼이란 Windows와 Linux 운영 체제 모두에서 실제로 변경을 하지 않고 실행할 수 있는 애플리케이션을 의미합니다. 다음 표는 Linux와 Windows 운영 환경 간의 차이점을 나열한 것입니다.

표 10.7 Linux와 Windows 운영 환경의 차이점

차이점	설명
파일 이름 대소문자 구별	Linux에서 대문자는 소문자와 <i>다릅니다</i> . Test.txt 파일은 test.txt와 <i>다릅니다</i> . Linux에서는 파일 이름의 대소문자에 많은 주의를 기울여야 합니다.
줄 끝 문자	Windows에서 텍스트 행은 CR/LF(즉 ASCII 13 + ASCII 10)로 종료되지만 Linux에서는 LF로 종료됩니다. Kylix의 코드 편집기가 이 차이점을 처리하기는 하지만 Windows에서 코드를 import할 때 이러한 차이점을 인식하고 있어야 합니다.
파일 끝 문자	DOS와 Windows에서 #26(Ctrl-Z) 문자 값은 해당 문자 뒤에 데이터가 있더라도 텍스트 파일의 끝으로 취급됩니다. Linux에서는 특별한 파일 끝 문자가 없으며 텍스트 데이터는 파일의 끝에서 끝납니다.

표 10.7 Linux와 Windows 운영 환경의 차이점 (계속)

차이점	설명
배치 파일/셸 스크립트	Linux에서 .bat 파일에 해당하는 것은 셸 스크립트입니다. 스크립트는 명령어를 포함하는 텍스트 파일로 저장되어 <code>chmod +x <scriptfile></code> 명령으로 실행 가능하게 됩니다. 스크립트를 실행하려면 해당 이름을 입력합니다. (스크립트 언어는 Linux에서 사용하는 셸에 따라 다릅니다. Bash를 일반적으로 사용합니다.)
명령 확인	DOS나 Windows에서는 파일이나 폴더를 삭제하려는 경우 "파일을 삭제하시겠습니까?"라고 확인을 합니다. Linux는 일반적으로 묻지 않고 바로 실행합니다. 그러면 실수로 파일이나 전체 파일 시스템을 삭제해버릴 수 있습니다. 파일을 다른 매체에 백업하지 않는 한 Linux에서는 삭제를 취소할 방법이 없습니다.
명령 피드백	Linux에서는 명령이 성공하면 상태 메시지 없이 명령 프롬프트를 다시 표시합니다.
명령 스위치	DOS에서 슬래시 (/)나 대시 (-)를 사용하는 대신 Linux에서는 대시 (-)를 사용하여 명령 스위치를 표시하거나 이중 대시 (--)를 사용하여 여러 문자 옵션을 표시합니다.
구성 파일	Windows에서 구성은 레지스트리나 <code>autoexec.bat</code> 와 같은 파일에서 이루어집니다. Linux에서 구성 파일은 점 (.)으로 시작하는 숨김 파일로 만들어집니다. 많은 구성 파일들은 /etc 디렉토리와 사용자의 홈 디렉토리에 있습니다. Linux도 <code>LD_LIBRARY_PATH</code> (라이브러리의 경로를 찾음)와 같은 환경 변수를 사용합니다. 기타 중요한 환경 변수는 다음과 같습니다. HOME 사용자의 홈 디렉토리(/home/sam) TERM 터미널 유형(xterm, vt100, console) SHELL 사용자 셸의 경로(/bin/bash) USER 사용자의 로그인 이름(sfuller) PATH 프로그램을 검색하는 경로 이 환경 변수들은 셸이나 <code>.bashrc</code> 와 같은 rc 파일에 지정되어 있습니다.
DLL	Linux에서는 공유 객체 파일(.so)을 사용합니다. Windows에서 이러한 파일들은 동적 연결 라이브러리(DLL)입니다.
드라이브 문자	Linux에는 드라이브 문자가 없습니다. Linux의 경로명을 예를 들면 /lib/security입니다. 런타임 라이브러리에서 <code>DriveDelim</code> 을 참조하십시오.
예외	운영 체제 예외는 Linux에서 시그널이라고 합니다.
실행 파일	Linux의 실행 파일은 확장자가 없습니다. Windows의 실행 파일은 exe 확장자를 갖습니다.
파일 이름 확장자	Linux는 파일 이름 확장자를 사용하여 파일 형식을 식별하거나 응용 프로그램과 연결하지 않습니다.

표 10.7 Linux와 Windows 운영 환경의 차이점 (계속)

차이점	설명
파일 권한	Linux에서는 파일(및 디렉토리)에 파일 소유자, 그룹 및 기타에 대해 읽고 쓰고 실행할 권한이 할당됩니다. 예를 들어, 다음과 같습니다. -rwxr-xr-x는 왼쪽에서 오른쪽으로 다음과 같은 의미입니다. -은 파일 형식(- = 일반 파일, d = 디렉토리, l = 링크)이고 rwx는 파일 소유자의 권한(읽기, 쓰기, 실행)이며 r-x는 파일 소유자 그룹의 권한(읽기, 실행)이고 r-x는 다른 모든 사용자의 권한(읽기, 실행)입니다. 루트 사용자(superuser)는 이 권한을 무시할 수 있습니다. 애플리케이션이 올바른 사용자에게서 실행되고 필요 파일에 적절한 액세스 권한이 있는지 확인해야 합니다.
Make 유틸리티	Borland의 make 유틸리티는 Linux 플랫폼에서는 사용할 수 없습니다. 그 대신 Linux에 있는 GNU make 유틸리티를 사용하면 됩니다.
멀티태스킹	Linux는 멀티태스킹을 완벽하게 지원합니다. 동시에 여러 프로그램(Linux에서는 프로세스라고 부름)을 실행할 수 있습니다. 명령 뒤에 &를 사용하면 프로세스를 백그라운드로 실행하고 또 그대로 계속하여 작업할 수도 있습니다. Linux에서는 또한 여러 세션을 지원합니다.
경로명	DOS에서 역슬래시(\)를 사용하는 곳에 Linux는 슬래시(/)를 사용합니다. PathDelim 상수를 사용하여 플랫폼에 맞는 문자를 지정할 수 있습니다. 런타임 라이브러리에서 PathDelim을 참조하십시오.
검색 경로	프로그램 실행 시 Windows는 항상 현재 디렉토리를 먼저 찾은 다음 PATH 환경 변수를 찾습니다. Linux는 현재 디렉토리를 찾지 않고 PATH에 나열된 디렉토리만 검색합니다. 현재 디렉토리에서 프로그램을 실행하려면 그 앞에 ./를 써주어야 합니다. PATH를 수정하여 검색할 첫 경로로서 ./를 포함할 수도 있습니다.
검색 경로 구분자	Windows에서는 검색 경로 구분자로 세미콜론을 사용합니다. Linux에서는 콜론을 사용합니다. 런타임 라이브러리에서 PathDelim을 참조하십시오.
심볼릭 링크	Linux에서 심볼릭 링크는 디스크의 다른 파일을 가리키는 특별한 파일입니다. 애플리케이션의 주요 파일을 가리키는 전역 bin 디렉토리에 심볼릭 링크를 넣으면 시스템 검색 경로를 수정할 필요가 없습니다. 심볼릭 링크는 ln(link) 명령으로 만듭니다. Windows에는 GUI 데스크탑의 단축키가 들어 있습니다. 명령줄에서 프로그램을 사용하도록 하기 위해 Windows 설치 프로그램은 일반적으로 시스템 검색 경로를 수정합니다.

Linux의 디렉토리 구조

Linux에서는 디렉토리가 다릅니다. 파일 시스템 상의 어느 곳에서든지 모든 파일 또는 장치를 마운트할 수 있습니다.

참고 Windows 경로명에서 역슬래시를 사용하는 것과 반대로 Linux 경로명에서는 슬래시를 사용합니다. 첫 번째 슬래시는 루트 디렉토리를 의미합니다.

다음은 Linux에서 일반적으로 사용되는 디렉토리입니다.

표 10.8 공통된 Linux 디렉토리

디렉토리	내용
/	전체 Linux 파일 시스템의 루트 또는 최상위 디렉토리
/root	루트 파일 시스템인 Superuser의 홈 디렉토리
/bin	명령, 유틸리티
/sbin	시스템 유틸리티
/dev	파일로 보여지는 장치
/lib	라이브러리
/home/username	사용자의 로그인 이름이 username인 곳에서 사용자가 소유한 파일
/opt	옵션
/boot	시스템 시동 시 호출되는 커널
/etc	구성 파일
/usr	애플리케이션, 프로그램. 보통 /usr/spool, /usr/man, /usr/include, /usr/local과 같은 디렉토리 포함
/mnt	CD나 플로피 디스크 드라이브와 같은 시스템에 마운트되는 기타 매체
var	로그, 메시지, 스펴 파일
/proc	가상 파일 시스템 및 시스템 통계 보고
/tmp	임시 파일

참고 Linux는 배포판에 따라 종종 다른 위치에 파일을 둡니다. Red Hat 배포판에서는 /bin에 유틸리티 프로그램이 있지만 Debian 배포판에서는 /usr/local/bin에 있습니다.

UNIX/Linux 계층적 파일 시스템의 구성에 대한 더 자세한 내용 및 *Filesystem Hierarchy Standard*를 읽으려면 www.pathname.com을 참조하십시오.

포팅 가능한 코드 작성

Windows 및 Linux에서 실행 가능한 크로스 플랫폼 애플리케이션을 작성하는 경우 다른 조건에서 컴파일하는 코드를 작성할 수 있습니다. 조건부 컴파일을 사용하면 Windows 코드를 유지하면서도 Linux 운영 체제와의 차이점을 수용하도록 할 수 있습니다.

Windows와 Linux 간에 쉽게 포팅할 수 있는 애플리케이션을 만들려면 반드시 다음 사항을 염두에 두어야 합니다.

- 플랫폼 특정(Win32 또는 Linux) API에 대한 호출을 줄이거나 분리시키고 대신 CLX 메소드를 사용합니다.
- 애플리케이션 내에서 구성되는 Windows 메시징(PostMessage, SendMessage)을 제거합니다.
- *TRegIniFile* 대신 *TMemIniFile*을 사용합니다.
- 파일과 디렉토리 이름의 대소문자 구분에 유의합니다.

- 모든 외부 어셈블러 TASM 코드를 포팅합니다. GNU 어셈블러 "as"는 TASM 구문을 지원하지 않습니다. (10-21 페이지의 "인라인 어셈블러 코드 포함" 참조)

플랫폼 독립적인 런타임 라이브러리 루틴을 사용하고 시스템, SysUtils 및 기타 런타임 라이브러리 유닛의 상수를 사용하는 코드를 작성하도록 하십시오. 예를 들면, PathDelim 상수를 사용하여 '/'versus'\ 플랫폼 차이로 코드를 구분합니다.

두 플랫폼에서 멀티바이트 문자를 사용하는 것이 또 다른 방법입니다. Windows 코드는 오래 전부터 멀티바이트 문자당 2바이트를 사용해 왔습니다. Linux에서 멀티바이트 문자 인코딩은 문자당 더 많은 바이트(UTF-8에 대해 최고 6바이트까지)를 가질 수 있습니다. 두 플랫폼 모두 SysUtils의 StrNextChar 함수를 사용하여 조정할 수 있습니다. 다음과 같은 기존의 Windows 코드가 있다고 가정해 보십시오.

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    inc(p);
  inc(p);
end;
```

위의 코드를 플랫폼 독립적인 코드로 대체할 수 있습니다.

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    p := StrNextChar(p)
  else
    inc(p);
end;
```

이 예는 플랫폼 포팅이 가능하고 2바이트보다 긴 멀티바이트 문자를 지원하지만 멀티바이트가 아닌 로케일에 대해서는 프로시저 호출의 성능 비용을 고려하지 않습니다.

런타임 라이브러리 함수를 사용해도 해결되지 않으면 플랫폼 특정 코드를 루틴에 하나로 넣어 분리하거나 서브루틴으로 넣습니다. 소스 코드의 가독성과 이식성을 유지하려면 **\$IFDEF** 블록의 수를 제한하도록 합니다. 조건 심볼 WIN32는 Linux에는 정의되어 있지 않습니다. 조건 심볼 LINUX는 소스 코드가 Linux 플랫폼용으로 컴파일되고 있는 것으로 정의되어 있습니다.

조건 지시어 사용

\$IFDEF 컴파일러 지시어를 사용하는 것은 Windows 및 Linux 플랫폼용으로 코드를 조건화하기에 적합한 방법입니다. 하지만 **\$IFDEF**로는 소스 코드를 이해하고 유지하기가 어렵기 때문에 **\$IFDEF**를 언제 사용하는 것이 적당한지 이해해야 합니다. **\$IFDEF** 사용을 고려할 때 가장 중요한 문제는 "이 코드에 왜 **\$IFDEF**가 필요한가?"와 "**\$IFDEF**를 쓰지 않고도 이 코드를 작성할 수 있는가?"입니다.

크로스 플랫폼 애플리케이션 내에서 **\$IFDEF**를 사용하려면 다음 지침을 따릅니다.

- **\$IFDEF**는 반드시 필요한 경우가 아니면 사용하지 않도록 합니다. 소스 파일에 있는 **\$IFDEF**는 소스 코드가 컴파일될 때 계산됩니다. C/C++와 달리 Delphi는 프로젝트를 컴파일하는 데 유닛 소스(헤더 파일)를 필요로 하지 않습니다. 모든 소스 코드를

완전히 다시 빌드하는 것은 대부분의 Delphi 프로젝트에서 일반적인 작업이 아닙니다.

- 패키지 파일(.dpk)에서는 **\$IFDEF**를 사용하지 마십시오. 소스 파일에만 사용해야 합니다. 컴포넌트 작성자는 크로스 플랫폼 개발 시 **\$IFDEF**를 이용하여 하나의 패키지를 만드는 것이 아니라 두 개의 디자인 타임 패키지를 만들어야 합니다.
- 일반적으로는 **\$IFDEF** MSWINDOWS를 사용하여 WIN32를 포함한 Windows 플랫폼에 대해 검사합니다. **\$IFDEF** WIN32는 32비트와 64비트 Windows 같은 특정 Windows 플랫폼 간의 구분을 위해 사용합니다. WIN64에서 사용하지 않는 것이 확실하지 않다면 코드를 WIN32로 국한시키지 마십시오.
- **\$IFNDEF**와 같은 부정 검사는 반드시 필요한 경우 외에는 하지 않습니다. **\$IFNDEF** LINUX는 **\$IFDEF** MSWINDOWS와 같지 않습니다.
- **\$IFNDEF/\$ELSE** 조합을 피하십시오. 그 대신 가독성을 높이려면 긍정 검사(**\$IFDEF**)를 사용합니다.
- 플랫폼에 따라 다른 **\$IFDEF**에는 **\$ELSE** 절을 사용하지 않습니다. **\$IFDEF** LINUX/**\$ELSE** 또는 **\$IFDEF** MSWINDOWS/**\$ELSE** 대신 LINUX용의 분리된 **\$IFDEF** 블록과 MSWINDOWS 특정 코드를 사용합니다.

예를 들어 이전 코드에는 다음과 같은 내용이 있을 수 있습니다.

```
{IFDEF WIN32}
  (32-bit Windows code)
{ELSE}
  (16-bit Windows code)  ///!By mistake, Linux could fall into this code.
{ENDIF}
```

\$IFDEF의 모든 이식성이 없는 코드의 경우 플랫폼이 **\$ELSE** 절에 의해 런타임 시 알 수 없는 이유로 실패하는 것보다 소스 코드가 컴파일되지 않는 것이 낫습니다. 컴파일 실패가 런타임 실패보다 찾기가 더 쉽습니다.

- 복잡한 검사에는 **\$IF** 구문을 사용합니다. 중첩된 **\$IFDEF**를 **\$IF** 지시어의 부울 표현식으로 바꿉니다. **\$IF** 지시어는 **\$ENDIF**가 아닌 **\$IFEND**를 사용해서 종료해야 합니다. 이전 컴파일러로부터 새로운 **\$IF** 구문을 숨기려면 **\$IF** 표현식을 **\$IFDEF** 안에 둡니다.

모든 조건 지시어가 온라인 도움말에 있습니다. 자세한 내용은 도움말의 "Conditional Compilation" 항목을 참조하십시오.

조건 지시어 종료

\$IFEND 지시어를 사용하여 **\$IF**와 **\$ELSEIF** 조건부 지시어를 종료합니다. **\$IFDEF**/**\$ENDIF**를 사용하는 대신 이렇게 하면 **\$IF/\$IFEND** 블록이 이전 컴파일러로부터 숨겨집니다. 그러면 이전 컴파일러가 **\$IFEND** 지시어를 인식하지 못합니다. **\$IF**는 **\$IFEND**로만 종료할 수 있습니다. 이전의 지시어(**\$IFDEF**, **\$IFNDEF**, **\$IFOPT**)는 **\$ENDIF**로만 종료할 수 있습니다.

참고 **\$IFDEF/\$ENDIF** 안에 **\$IF**를 중첩할 경우 **\$ELSE**를 **\$IF**와 함께 사용하지 마십시오. 이전의 컴파일러는 **\$ELSE**를 보고 **\$IFDEF**의 일부로 생각하여 컴파일러 오류를 만듭니다. **\$IF**가 먼저 받아들여지면 **\$ELSEIF**는 받아들여지지 않고 이전 컴파일러는

\$ELSEIF를 모르기 때문에 이러한 상황에서는 **{ \$ELSE }** 대신 **{ \$ELSEIF True }**를 사용할 수 있습니다. 코드를 여러 다른 버전에서 코드가 실행되는 것을 원하는 협력 업체와 애플리케이션 개발자들에게는 역호환성을 위해 **\$IF**를 숨기는 것이 1차적인 문제가 됩니다.

\$ELSEIF는 **\$ELSE**와 **\$IF**의 조합입니다. **\$ELSEIF** 지시어를 통해 여러 조건부 블록 중 하나만 받아들이게 작성할 수 있습니다. 예를 들면, 다음과 같습니다.

```
{ $IFDEF doit }
  do_doit
{ $ELSEIF RTLVersion >= 14 }
  goforit
{ $ELSEIF somestring = 'yes' }
  beep
{ $ELSE }
  last chance
{ $IFEND }
```

위의 네 가지 경우 중 하나만 받아들입니다. 처음 세 개의 조건이 true가 아닌 경우 **\$ELSE** 절이 받아들여집니다. **\$ELSEIF**는 **\$IFEND**로 종료해야 합니다. **\$ELSEIF**는 **\$ELSE** 뒤에 나타날 수 없습니다. 조건은 일반적인 **\$IF...\$ELSE**처럼 위에서 아래로 값이 구해집니다. 예제에서 doit이 정의되지 않고 RTLVersion은 15이며 somestring = 'yes'인 경우 조건이 둘 다 True일지라도 "beep" 블록이 아닌 "goforit" 블록만 받아들여 집니다.

\$ENDIF를 사용하여 **\$IFDEF**를 종료하는 것을 잊을 경우에는 컴파일러가 소스 파일의 끝에 다음 오류 메시지를 보고합니다.

```
Missing ENDIF
```

소스 파일에 **\$IF/\$IFDEF** 지시어가 여러 개 있는 경우에는 어느 것이 문제를 일으키는지 알아내기 어려울 수도 있습니다. Kylix 또는 Delphi는 일치하지 않는 **\$ENDIF/\$IFEND**가 있는 마지막 **\$IF/\$IFDEF** 컴파일러 지시어의 소스 행에 다음과 같은 오류 메시지를 보고합니다.

```
Unterminated conditional directive
```

해당 위치에서 문제 찾기를 시작합니다.

메시지 나타내기

\$MESSAGE 컴파일러 지시어는 컴파일러가 하듯이 소스 코드에서 힌트, 경고 및 오류를 나타내게 할 수 있습니다.

```
{ $MESSAGE HINT|WARN|ERROR|FATAL 'text string' }
```

메시지 유형은 옵션입니다. 메시지 유형이 지정되지 않은 경우 기본값은 HINT입니다. 텍스트 문자열이 필요하며 작은 따옴표 안에 포함되어야 합니다.

예제:

```
{ $MESSAGE 'Boo!' } 힌트를 표시합니다.
```

```
{ $Message Hint 'Feed the cats' } 힌트를 표시합니다.
```


{**\$Message** Warn 'Looks like rain.'} 경고를 표시합니다.

{**\$Message** Error 'Not implemented'} 오류를 표시하고 컴파일을 계속합니다.

{**\$Message** Fatal 'Bang.Yer dead.'} 오류를 표시하고 컴파일러를 종료합니다.

인라인 어셈블리 코드 포함

Windows 애플리케이션에 인라인 어셈블리 코드를 포함하면 Linux 의 위치 독립 코드 (PIC) 요구 조건 때문에 Linux에서 같은 코드를 사용하지 못할 수도 있습니다. Linux 공유 객체 라이브러리 (DLL에 해당됨)는 모든 코드를 수정하지 않고 메모리에 재배치할 수 있어야 합니다. 이는 전역 변수 또는 기타 절대 주소를 사용하는 루틴이나 외부 함수를 호출하는 인라인 어셈블리 루틴에 1차적인 영향을 줍니다.

오브젝트 파스칼 코드만 포함하는 유닛의 경우 컴파일러는 필요할 때 자동으로 PIC를 생성합니다. PIC 유닛은 .dcu가 아닌 .dpu 확장자를 가집니다. 모든 파스칼 유닛 소스 파일은 PIC와 PIC가 아닌 두 가지 형식으로 컴파일하는 것이 좋습니다. -p 컴파일러 스위치를 사용하여 PIC를 생성합니다. 미리 컴파일되어 있는 유닛은 두 가지 형식으로 사용할 수 있습니다.

실행 파일이나 공유 라이브러리 중 어느 것으로 컴파일하는지에 따라 어셈블리 루틴을 다르게 코딩하려는 경우 {**\$IFDEF** PIC}를 사용하여 어셈블리 코드를 두 버전으로 나눕니다. 또는 이 문제를 피하기 위해 오브젝트 파스칼에서 루틴을 다시 작성할 수도 있습니다.

다음은 인라인 어셈블리 코드에 대한 PIC 규칙입니다.

- Linux에서 전역 오프셋 테이블 또는 GOT로 불리는 현재 모듈의 기본 주소 포인터가 EBX 레지스터에 들어 있으므로 PIC에서는 모든 메모리 참조가 이 레지스터와 관련되어야 합니다. 따라서 다음과 같이 하지 않는 대신

```
MOV EAX,GlobalVar
```

다음을 사용합니다

```
MOV EAX,[EBX].GlobalVar
```

- PIC에서는 Win32에서와 같이 호출 간에 EBX 레지스터를 어셈블리 코드에 남겨 두어야 하고 Win32에서와 달리 외부 함수를 호출하기 전에 EBX 레지스터를 복원해야 합니다.
- PIC 코드는 기본 실행 파일에서 작동되기는 하지만 성능이 떨어지고 더 많은 코드를 생성합니다. 공유 객체에서는 선택의 여지가 없지만 실행 파일에서는 가장 높은 수준의 성능을 바랄 것입니다.

메시지와 시스템 이벤트

메시지 루프와 이벤트는 Linux와 CLX에서 다르게 수행되지만 컴포넌트 작성에 1차적인 영향을 줍니다. 대부분의 컴포넌트와 속성 편집기는 쉽게 포팅됩니다. 클래스 상의 *TObject.Dispatch*와 메시지 메소드 구문이 Linux에서 제대로 수행되는 반면, Linux 환경에서 운영 체제 공지는 메시지보다 시스템 이벤트를 사용하여 처리됩니다.

이벤트 핸들러를 크로스 플랫폼 애플리케이션에 작성하려면 Windows 메시지에 응답하는 대신 표 10.9에 설명된 메소드 중 하나를 오버라이드하여 사용자 지정 메시지를 작성할 수 있습니다. 오버라이드 시 상속된 메소드를 호출하면 모든 기본 프로세스 (default process)를 계속 발생시킬 수 있습니다.

표 10.9 시스템 이벤트에 응답하기 위한 TWidgetControl protected 메소드

메소드	설명
<i>ChangeBounds</i>	<i>TWidgetControl</i> 의 크기가 변경될 때 사용됩니다. Windows의 WM_SIZE 또는 WM_MOVE와 대체로 유사합니다. Qt는 클라이언트 영역에 기반하여 widget의 "geometry"를 설정하며 VCL은 Qt가 프레임으로 참조하는 것을 포함하는 전체 컨트롤 크기를 사용합니다.
<i>ChangeScale</i>	컨트롤 크기 변경 시 자동으로 호출됩니다. 폼의 크기와 다른 화면 해상도에 대한 모든 컨트롤 크기 또는 글꼴 크기를 변경하는 데 사용됩니다. <i>ChangeScale</i> 은 컨트롤의 Top, Left, Width 및 Height 속성을 수정하므로 컨트롤과 컨트롤의 자식의 크기는 물론 위치를 변경합니다.
<i>ColorChanged</i>	컨트롤의 색상이 변경되었을 때 호출됩니다.
<i>CursorChanged</i>	커서 모양이 바뀔 때 호출됩니다. 마우스 커서가 이 widget 위에 있을 때 마우스 커서의 모양을 가집니다.
<i>EnabledChanged</i>	애플리케이션이 창이나 컨트롤의 활성화 상태를 변경할 때 호출됩니다.
<i>FontChanged</i>	글꼴 리소스의 컬렉션이 변경될 때 호출됩니다. widget의 글꼴을 설정하고 모든 자식에게 변경 내용을 알려 줍니다. WM_FONTCHANGE 메시지와 대체로 유사합니다.
<i>PaletteChanged</i>	시스템 팔레트가 변경될 때 호출됩니다.
<i>ShowHintChanged</i>	도움말 힌트가 표시되거나 컨트롤에 숨겨졌을 때 호출됩니다.
<i>StyleChanged</i>	창이나 컨트롤의 GUI 스타일이 변경되었을 때 호출됩니다.
<i>TabStopChanged</i>	폼 상의 탭 순서가 변경되었을 때 호출됩니다.
<i>VisibleChanged</i>	컨트롤이 숨겨지거나 보여질 때 호출됩니다.
<i>WidgetDestroyed</i>	컨트롤을 기본으로 사용하는 widget이 소멸될 때 호출됩니다.

Qt는 C++ 툴킷이므로 모든 widget은 C++ 객체입니다. CLX는 오브젝트 파스칼로 작성되는데 오브젝트 파스칼은 C++ 객체와 직접적으로 상호 작용하지 않습니다. 또한 Qt는 여러 곳에서 다중 상속을 사용합니다. 그러므로 Delphi에는 인터페이스 레이어가 들어 있어 모든 Qt 클래스를 일련의 직접적인 C 함수로 변환합니다. 그런 다음 레이어들은 Linux에서는 공유 객체로, Windows에서는 DLL로 랩됩니다.

모든 *TWidgetControl*은 *CreateWidget*, *InitWidget*과 거의 항상 오버라이드해야 하는 *HookEvents* 가상 메소드를 가집니다. *CreateWidget*은 Qt widget을 만들고 Handle을 FHandle private field 변수로 할당합니다. *InitWidget*은 widget이 생성된 후 호출되고 Handle이 유효하게 됩니다.

Delphi CLX에서 일부 속성 할당은 Create 생성자에서 *InitWidget*으로 옮겨졌습니다. 그래서 실제로 필요할 때까지 Qt 객체의 생성을 지연시킵니다. 예를 들어, *Color*라는 이름의 속성이 있다고 가정합니다. *SetColor*에서 Qt 핸들이 있는지 보려면 *HandleAllocated*에서 확인할 수 있습니다. Handle이 할당되어 있으면 Qt를 적절히 호출하여 색상을 설정할 수 있습니다. 할당되지 않았으면 값을 private field 변수에 저장하고 *InitWidget*에서 속성을 설정합니다.

Linux는 Widget과 System이라는 두 가지 유형의 이벤트를 지원합니다. *HookEvents*는 Qt 객체와 통신하는 특별한 훅 객체에 CLX 컨트롤 이벤트 메소드를 훅(hook)하는 가상 메소드입니다. 훅 객체는 메소드 포인터의 집합입니다. Kylix에서 시스템 이벤트는 *WndProc*에 기본으로 대체되는 *EventHandler*를 거칩니다.

Linux에서의 프로그래밍 차이점

Linux *wchar_t* *widechar*는 문자당 32비트입니다. 오브젝트 파스칼 *widechar*가 지원하는 16비트 유니코드 표준은 Linux와 GNU 라이브러리가 지원하는 32비트 UCS 표준의 서브셋입니다. 파스칼 *widechar* 데이터는 *wchar_t*로 OS 함수에 전달되기 전에 문자당 32비트로 확장되어야 합니다.

Linux에서 와이드 문자열은 긴 문자열처럼 카운트되는 참조입니다(Windows에서는 아님).

Linux에서의 멀티바이트 처리는 다릅니다. Windows에서 멀티바이트 문자(MBCS)는 1바이트와 2바이트 *char* 코드로 표시됩니다. Linux에서는 1에서 6바이트로 표시됩니다.

*AnsiStrings*는 멀티바이트 문자 구조를 지닐 수 있지만 사용자의 로케일 설정에 따릅니다. 일본어, 중국어, 히브리어 및 아라비아어와 같은 멀티바이트 문자의 Linux 인코딩은 동일한 로케일에 대한 Windows 인코딩과 호환되지 않습니다. 멀티바이트는 포팅할 수 없지만 유니코드는 포팅이 가능합니다.

Linux에서는 절대 주소에 변수를 사용할 수 없습니다. *var X: Integer absolute \$1234;* 구문은 PIC에서 지원되지 않으며 Delphi에서 허용되지 않습니다.

크로스 플랫폼 데이터베이스 애플리케이션

Windows에서 Delphi는 데이터베이스 정보 액세스 방법에 대한 여러 선택 사항을 제공합니다. ADO, Borland Database Engine(BDE) 및 InterBase Express 사용이 들어 있습니다. 하지만 Kylix에서는 이 세 가지의 선택 사항을 사용할 수 없습니다. 그 대신 Delphi 버전 6과 함께 출시되어 Windows에서도 사용할 수 있는 새로운 크로스 플랫폼 데이터 액세스 기술인 *dbExpress*를 사용할 수 있습니다.

*dbExpress*가 Linux에서도 실행되도록 데이터베이스 애플리케이션을 포팅하기 전에 *dbExpress*의 사용 방법과 사용자가 이전에 사용하던 데이터 액세스 메커니즘 사용 방법 사이의 차이점을 이해해야 합니다. 그 차이는 여러 수준에서 일어납니다.

- 가장 낮은 수준에는 애플리케이션과 데이터베이스 서버 간에 통신하는 레이어가 있습니다. 이 레이어는 ADO, BDE 또는 InterBase 클라이언트 소프트웨어일 수 있습니다. 이 레이어는 동적 SQL 프로세싱을 위한 경량급(lightweight) 드라이버 집합인 *dbExpress*로 대체됩니다.
- 낮은 수준의 데이터 액세스는 데이터 모듈이나 폼에 추가하는 컴포넌트 집합에 랩됩니다. 이 컴포넌트에는 데이터베이스 서버로의 연결을 나타내는 데이터베이스 연결 컴포넌트와 서버로부터 페치(fetch)된 데이터를 나타내는 데이터셋이 있습니다. 중요한 차이점이 있기는 하지만 *dbExpress* 커서의 단방향성 때문에 이 레벨에서는 그

차이가 적습니다. 그 이유는 데이터베이스 연결 컴포넌트처럼 데이터셋이 공통 조상을 공유하기 때문입니다.

- 사용자 인터페이스 레벨에는 차이가 거의 없습니다. CLX data-aware 컨트롤은 해당 Windows 컨트롤에 가능한 유사하게 디자인되었습니다. 사용자 인터페이스 레벨에서의 주요 차이점은 캐시된 업데이트의 사용을 적용하기 위해 변경이 필요할 때 나타납니다.

기존의 데이터베이스 애플리케이션을 *dbExpress*로 포팅하는 방법에 대한 자세한 내용은 10-26 페이지의 "데이터베이스 애플리케이션을 Linux로 포팅"을 참조합니다. 새 *dbExpress* 애플리케이션 디자인에 대한 자세한 내용은 14장 "데이터베이스 애플리케이션 디자인"을 참조합니다.

dbExpress 차이점

Linux에서 *dbExpress*는 데이터베이스 서버와의 통신을 관리합니다. *dbExpress*는 공통 인터페이스를 구현하는 경량급 (lightweight) 드라이버의 집합으로 구성되어 있습니다. 드라이버는 모두 사용자 애플리케이션에 연결되어야 하는 공유 객체 (.so 파일)입니다. *dbExpress*는 크로스 플랫폼용으로 디자인되었으므로 Windows에서도 동적 연결 라이브러리 (.dll) 집합으로 사용할 수 있습니다.

모든 데이터 액세스 레이어와 마찬가지로 *dbExpress*는 데이터베이스 협력 업체가 제공하는 클라이언트측 소프트웨어를 필요로 합니다. 또한 데이터베이스 특정 드라이버와 두 개의 구성 파일 *dbxconnection*과 *dbxdriver*를 사용합니다. 주요한 Borland Database Engine 라이브러리 (*Idapi32.dll*)와 데이터베이스 특정 드라이버 및 많은 기타 지원 라이브러리를 필요로 하는 BDE 등의 경우보다는 훨씬 적은 것입니다.

다음은 *dbExpress*와 애플리케이션을 포팅하기 위해 필요한 기타 데이터 액세스 레이어 간의 차이점입니다.

- *dbExpress*는 원격 데이터베이스로의 보다 간단하고 빠른 경로를 가능하게 합니다. 그 결과로 간단하고 빠른 데이터 액세스에서 눈에 띄게 향상된 성능을 기대할 수 있습니다.
- *dbExpress*는 쿼리와 내장 프로시저를 처리할 수 있지만 개방형 테이블의 개념은 지원하지 않습니다.
- *dbExpress*는 단방향 커서만 반환합니다.
- *dbExpress*는 INSERT, DELETE 또는 UPDATE 쿼리를 실행하는 기능 이외에 기본 제공된 업데이트 지원이 없습니다.
- *dbExpress*는 메타데이터 캐싱을 하지 않으며 디자인 타임 메타데이터 액세스 인터페이스는 코어 데이터 액세스 인터페이스를 사용하여 구현됩니다.
- *dbExpress*는 사용자가 요청한 쿼리만 실행하므로 다른 쿼리를 끌어들이지 않고 데이터베이스 액세스를 최적화합니다.
- *dbExpress*는 레코드 버퍼나 레코드 버퍼 블록을 내부적으로 관리합니다. 이것이 레코드 버퍼에 사용되는 메모리를 클라이언트가 할당해야 하는 BDE와 다른 점입니다.

- *dbExpress*는 SQL 기반이 아닌(예: Paradox, dBase 또는 FoxPro) 로컬 테이블은 지원하지 않습니다.
- *dbExpress* 드라이버는 InterBase, Oracle, DB2 및 MySQL을 지원합니다. 다른 데이터베이스 서버를 사용하고 있을 경우에는 이 데이터베이스들 중 하나로 데이터를 포팅하고 사용 중인 데이터베이스 서버용 *dbExpress* 드라이버를 작성하거나 사용자의 데이터베이스 서버용으로 사용할 외부 *dbExpress* 드라이버를 구해야 합니다.

컴포넌트 수준 차이점

dbExpress 애플리케이션을 작성할 때에는 기존의 데이터베이스 애플리케이션에 사용되는 것과 다른 데이터 액세스 컴포넌트 집합이 필요합니다. *dbExpress* 컴포넌트는 다른 데이터 액세스 컴포넌트 (*TDataSet* 및 *TCustomConnection*)와 동일한 기본 클래스를 공유하며 이는 속성, 메소드 및 이벤트가 기존 애플리케이션에 사용되는 컴포넌트와 동일하다는 것을 의미합니다.

표 10.10은 Windows의 InterBase Express, BDE 및 ADO에서 사용되는 중요한 데이터베이스 컴포넌트를 나열하며 Linux와 크로스 플랫폼 애플리케이션에서 사용되는 유사한 *dbExpress* 컴포넌트를 보여 줍니다.

표 10.10 유사한 데이터 액세스 컴포넌트

InterBase Express 컴포넌트	BDE 컴포넌트	ADO 컴포넌트	dbExpress 컴포넌트
<i>TIBDatabase</i>	<i>TDatabase</i>	<i>TADOConnection</i>	<i>TSQLConnection</i>
<i>TIBTable</i>	<i>TTable</i>	<i>TADOTable</i>	<i>TSQLTable</i>
<i>TIBQuery</i>	<i>TQuery</i>	<i>TADOQuery</i>	<i>TSQLQuery</i>
<i>TIBStoredProc</i>	<i>TStoredProc</i>	<i>TADOStoredProc</i>	<i>TSQLStoredProc</i>
<i>TIBDataSet</i>		<i>TADODataSet</i>	<i>TSQLDataSet</i>

하지만 *dbExpress* 데이터셋 (*TSQLTable*, *TSQLQuery*, *TSQLStoredProc* 및 *TSQLDataSet*)은 편집을 지원하지 않고 포워드(forward) 탐색만 허용하기 때문에 자신과 유사한 대상보다 더 제한적입니다. *dbExpress* 데이터셋과 기타 Windows에서 사용 가능한 기타 데이터셋의 차이점에 대한 자세한 내용은 22장 "단방향 데이터셋 사용"을 참조하십시오.

편집 및 탐색 지원이 없으므로 대부분의 *dbExpress* 애플리케이션은 *dbExpress* 데이터셋과 직접 연동하지 않습니다. 오히려 *dbExpress* 데이터셋을 메모리에 레코드를 버퍼하고 편집과 탐색 지원을 제공하는 클라이언트 데이터셋에 연결합니다. 이 아키텍처에 대한 자세한 내용은 14-6 페이지의 "데이터베이스 아키텍처"를 참조하십시오.

참고 아주 간단한 애플리케이션의 경우 클라이언트 데이터셋에 연결된 *dbExpress* 데이터셋 대신 *TSQLClientDataSet*을 사용할 수 있습니다. 이는 포팅하려는 애플리케이션의 데이터셋과 포팅된 애플리케이션의 데이터셋 간에 일대일로 대응되기 때문에 간단하다는 이점이 있지만 *dbExpress* 데이터셋을 클라이언트 데이터셋에 명시적으로 연결하는 것보다는 유연하지 못합니다. 대부분의 애플리케이션에서는 *TClientDataSet* 컴포넌트에 연결된 *dbExpress* 데이터셋을 사용하는 것이 좋습니다.

사용자 인터페이스 수준 차이점

CLX data-aware 컨트롤은 해당 Windows 컨트롤과 최대한 유사하게 디자인되어 있습니다. 그 결과 데이터베이스 애플리케이션의 사용자 인터페이스 부분을 포팅하면 Windows 애플리케이션을 CLX로 포팅하는 것 외에 추가로 고려할 사항이 거의 없습니다.

사용자 인터페이스 레벨의 주요 차이점은 *dbExpress* 데이터셋이나 클라이언트 데이터셋이 데이터를 제공하는 방법에 있습니다.

dbExpress 데이터셋만 사용하는 경우 편집을 지원하지 않고 포워드 탐색만 지원한다는 사실에 맞추어 사용자 인터페이스를 조정해야 합니다. 따라서 사용자가 이전의 레코드로 이동하는 것을 허용하는 컨트롤을 제거해야 할 수도 있습니다. *dbExpress* 데이터셋은 데이터를 버퍼하지 않으므로 데이터를 data-aware 그리드에 표시할 수 없습니다. 한 번에 데이터 하나만 표시할 수 있습니다.

dbExpress 데이터셋을 클라이언트 데이터셋에 연결했을 경우 편집 및 탐색과 관련된 사용자 인터페이스 요소는 계속 작동합니다. 클라이언트 데이터셋에 다시 연결하기만 하면 됩니다. 이 경우에 제일 먼저 고려해야 할 것은 데이터베이스에 업데이트를 쓰는 방법을 처리하는 것입니다. 기본적으로 Windows의 대부분의 데이터셋은 포스트될 때 (예를 들어 사용자가 새 레코드로 이동할 때) 자동으로 데이터베이스 서버에 업데이트를 씁니다. 반면 클라이언트 데이터셋은 항상 메모리에 업데이트를 캐시합니다. 이러한 차이점을 조정하는 방법에 대한 내용은 10-28 페이지의 "dbExpress 애플리케이션에서 데이터 업데이트"를 참조합니다.

데이터베이스 애플리케이션을 Linux로 포팅

데이터베이스 애플리케이션을 *dbExpress*로 포팅하면 Windows와 Linux 모두에서 실행되는 크로스 플랫폼 애플리케이션을 만들 수 있습니다. 포팅 기법이 다르기 때문에 포팅 과정에는 애플리케이션을 변경하는 것이 포함됩니다. 포팅의 난이성은 애플리케이션의 유형, 복잡한 정도 및 필요 조건 등에 따라 다릅니다. ADO와 같은 Windows 특정 기술을 많이 사용하는 애플리케이션의 포팅은 Delphi 데이터베이스 기술을 사용하는 애플리케이션을 포팅하는 것보다 더 어렵습니다.

다음과 같은 일반적인 절차를 따라 Windows/VCL 데이터베이스 애플리케이션을 Kylix/CLX로 포팅합니다.

- 1 데이터베이스 데이터를 저장할 곳을 생각합니다. *dbExpress*는 Oracle, Interbase, DB2 및 MySQL용 드라이버를 제공합니다. 데이터는 이들 SQL 서버 중 하나에 두어야 합니다.

일부 Delphi 버전에는 로컬 데이터베이스 데이터를 Paradox, dBase, 및 FoxPro와 같은 플랫폼에서부터 지원하는 플랫폼 중 하나로 이동시키는 데 사용하는 Data Pump 유틸리티가 들어 있습니다. (이 유틸리티의 사용 방법은 Program Files\Common Files\Borland\Shared\BDE에 있는 datapump.hlp 파일을 참조하십시오.)

- 2 사용자 인터페이스 폼과 데이터셋 및 연결 컴포넌트를 가진 데이터 모듈을 분리하지 않았을 경우 포팅을 시작하기 전에 분리해도 됩니다. 이렇게 하면 데이터 모듈에 완전

히 새로운 컴포넌트 집합을 필요로 하는 애플리케이션의 부분을 분리할 수 있습니다. 그런 다음 사용자 인터페이스를 나타내는 폼을 다른 애플리케이션과 마찬가지로 포팅할 수 있습니다. 자세한 내용은 10-4 페이지의 "애플리케이션 포팅"을 참조하십시오.

나머지 단계에서는 데이터셋과 연결 컴포넌트가 자신의 데이터 모듈에서 분리되어 있는 것으로 가정합니다.

- 3 데이터셋의 CLX 버전과 연결 컴포넌트를 보유할 새로운 데이터 모듈을 만듭니다.
- 4 원본 애플리케이션의 각 데이터셋의 경우 *dbExpress* 데이터셋, *TDataSetProvider* 컴포넌트 및 *TClientDataSet* 컴포넌트를 추가합니다. 표 10.10에서 해당되는 것을 사용하여 사용할 *dbExpress* 데이터셋을 결정합니다. 컴포넌트에 이름을 부여합니다.
 - *TClientDataSet* 컴포넌트의 *ProviderName* 속성을 *TDataSetProvider* 컴포넌트의 이름으로 설정합니다.
 - *TDataSetProvider* 컴포넌트의 *DataSet* 속성을 *dbExpress* 데이터셋으로 설정합니다.
 - 원본 데이터셋으로 참조되는 데이터 소스 컴포넌트의 *DataSet* 속성을 변경하여 클라이언트 데이터셋을 바로 참조할 수 있게 합니다.
- 5 새 데이터셋이 원본 데이터셋과 일치하도록 속성을 설정합니다.
 - 원본 데이터셋이 *TTable*, *TADOTable* 또는 *TIBTable* 컴포넌트였을 경우 새 *TSQLTable*의 *TableName* 속성을 원본 데이터셋의 *TableName*으로 설정합니다. 또한 마스터/디테일 관계를 설정하거나 인덱스를 지정하는 데 사용한 모든 속성을 복사합니다. 범위와 필터를 지정하는 속성은 새로운 *TSQLTable* 컴포넌트 보다는 클라이언트 데이터셋에 설정해야 합니다.
 - 원본 데이터셋이 *TQuery*, *TADOQuery* 또는 *TIBQuery* 컴포넌트였을 경우 새 *TSQLQuery* 컴포넌트의 *SQL* 속성을 원본 데이터셋의 *SQL* 속성으로 설정합니다. 새 *TSQLQuery*의 *Params* 속성을 원본 데이터셋의 *Params*나 *Parameters* 속성과 일치하도록 설정합니다. 마스터/디테일 관계 성립을 위해 *DataSource* 속성을 설정했다면 이 속성도 복사합니다.
 - 원본 데이터셋이 *TStoredProc*, *TADOStoredProc* 또는 *TIBStoredProc* 컴포넌트였으면 새 *TSQLStoredProc* 컴포넌트의 *StoredProcName*을 원본 데이터셋의 *StoredProcName*이나 *ProcedureName* 속성으로 설정합니다. 새 *TSQLStoredProc*의 *Params* 속성을 원본 데이터셋의 *Params*나 *Parameters* 속성 값과 일치하도록 설정합니다.
- 6 원본 애플리케이션에 있는 모든 데이터베이스 연결 컴포넌트(*TDatabase*, *TIBDatabase* 또는 *TADOConnection*)의 경우 새 데이터 모듈에 *TSQLConnection* 컴포넌트를 추가합니다. 또한 연결 컴포넌트 없이 연결한(예를 들어 ADO 데이터셋의 *ConnectionString* 속성을 사용하거나 BDE 데이터셋의 *DatabaseName* 속성을 BDE 앨리어스로 설정함으로써) 모든 데이터베이스 서버의 *TSQLConnection* 컴포넌트도 추가해야 합니다.
- 7 4 단계의 각 *dbExpress* 데이터셋마다 해당 *SQLConnection* 속성을 적절한 데이터베이스 연결에 해당하는 *TSQLConnection* 컴포넌트로 설정합니다.

8 각 *TSQLConnection* 컴포넌트마다 데이터베이스 연결에 필요한 정보를 지정합니다. 그렇게 하려면 *TSQLConnection* 컴포넌트를 더블 클릭하여 Connection Editor를 표시하고 매개변수 값을 설정하여 적절한 설정을 나타냅니다. 1 단계의 새로운 데이터베이스 서버로 데이터를 전송했으면 새 서버에 맞는 설정을 지정합니다. 이전과 동일한 서버를 사용할 예정이면 원본 연결 컴포넌트에서 이 정보의 일부를 찾아볼 수 있습니다.

- 원본 애플리케이션이 *TDatabase*를 사용했을 경우에는 *Params*와 *TransIsolation* 속성에 나타나는 정보를 전송해야 합니다.
- 원본 애플리케이션이 *TADOConnection*을 사용했을 경우에는 *ConnectionString*과 *IsolationLevel* 속성에 나타나는 정보를 전송해야 합니다.
- 원본 애플리케이션이 *TIBDatabase*를 사용했을 경우에는 *DatabaseName*과 *Params* 속성에 나타나는 정보를 전송해야 합니다.
- 원본 연결 컴포넌트가 없을 경우 BDE 앨리어스와 관련되었거나 데이터셋의 *ConnectionString* 속성에 나타난 정보를 전송해야 합니다.

이 매개변수 집합을 새로운 연결 이름으로 저장할 수 있습니다. 이 과정에 대한 자세한 내용은 17-2 페이지의 "연결 제어"를 참조하십시오.

dbExpress 애플리케이션에서 데이터 업데이트

dbExpress 애플리케이션은 클라이언트 데이터셋을 사용하여 편집을 지원합니다. 편집 내용을 클라이언트 데이터셋에 포스트하면 변경 사항이 클라이언트 데이터셋의 데이터 메모리 스냅샷에는 기록되지만 데이터베이스 서버에는 자동으로 기록되지 않습니다. 원본 애플리케이션이 업데이트 캐시를 위해 클라이언트 데이터셋을 사용했으면 Linux에서의 편집 지원을 위해 변경할 필요가 없습니다. 그러나 Windows의 대부분의 데이터셋의 기본 동작에 의존하여 레코드를 포스트할 때 데이터베이스 서버에 편집 내용을 기록했을 경우 클라이언트 데이터셋을 사용하기 위해 변경해야 합니다.

이전에 업데이트를 캐시하지 않은 애플리케이션을 변환하는 방법에는 두 가지가 있습니다.

- 각 업데이트된 레코드가 포스트되자마자 데이터베이스 서버에 적용하기 위한 코드를 작성하여 Windows에서 데이터셋의 동작을 흉내낼 수 있습니다. 그러려면 업데이트를 데이터베이스 서버에 적용하는 *AfterPost* 이벤트 핸들러가 있는 클라이언트 데이터셋을 다음과 같이 사용합니다.

```
procedure TForm1.ClientDataSet1AfterPost(DataSet:TDataSet);
begin
  with DataSet as TClientDataSet do
    ApplyUpdates(1);
end;
```

- 캐시된 업데이트를 다루기 위해 사용자 인터페이스를 조정할 수 있습니다. 이 접근법은 네트워크 트래픽의 양을 줄이고 트랜잭션 시간을 최소화하는 등의 이점이 있습니다. 하지만 캐시된 업데이트 사용으로 전환하면 해당 업데이트를 데이터베이스 서버에 다시 적용할 시기를 정해야 하고 사용자 인터페이스를 변경하여 업데이트된 애플리케이션을 사용자들이 초기화하게 하거나 사용자의 편집 내용이 데이터베이스에

기록됐는지 여부에 대한 피드백을 제공해야 할 것입니다. 또한 사용자가 레코드를 포스트할 때 업데이트 오류가 감지되지 않으므로 사용자에게 이러한 오류를 보고하는 방법을 변경하여 어떤 업데이트로 문제를 일으켰는지와 어떤 유형의 문제가 일어났는지를 사용자가 알 수 있게 합니다.

원본 애플리케이션이 업데이트를 캐시하기 위해 BDE나 ADO가 제공하는 지원을 사용했을 경우 코드를 수정하여 클라이언트 데이터셋 사용으로 전환해야 합니다. 다음 표는 BDE와 ADO 데이터셋에서 캐시된 업데이트를 지원하는 속성, 이벤트 및 메소드, *TClientDataSet*의 해당 속성, 메소드 및 이벤트를 나열한 것입니다.

표 10.11 캐시된 업데이트를 위한 속성, 메소드 및 이벤트

BDE 데이터셋 (또는 TDatabase)	ADO 데이터셋	TclientDataSet	용도
<i>CachedUpdates</i>	<i>LockType</i>	필요없음, 클라이언트 데이터셋은 항상 업데이트를 캐시함	캐시된 업데이트가 효력이 있는지 결정합니다.
지원하지 않음	<i>CursorType</i>	지원하지 않음	서버의 변경으로 인해 데이터셋이 분리된 방법을 지정합니다.
<i>UpdatesPending</i>	지원하지 않음	<i>ChangeCount</i>	로컬 캐시가 데이터베이스에 적용하는 데 필요한 업데이트된 레코드를 포함하는지 나타냅니다.
<i>UpdateRecordTypes</i>	<i>FilterGroup</i>	<i>StatusFilter</i>	업데이트된 레코드의 종류를 나타내어 캐시된 업데이트를 적용할 때 보이게 합니다.
<i>UpdateStatus</i>	<i>RecordStatus</i>	<i>UpdateStatus</i>	레코드가 변경되지 않았는지, 수정되었는지, 삽입되었는지 또는 삭제되었는지 나타냅니다.
<i>OnUpdateError</i>	지원하지 않음	<i>OnReconcileError</i>	레코드별로 업데이트 오류를 처리하기 위한 이벤트입니다.
<i>ApplyUpdates</i> (데이터셋이나 데이터베이스에서)	<i>UpdateBatch</i>	<i>ApplyUpdates</i>	레코드를 로컬 캐시의 데이터베이스에 적용합니다.
<i>CancelUpdates</i>	<i>CancelUpdates</i> or <i>CancelBatch</i>	<i>CancelUpdates</i>	대기 중인 업데이트를 적용하지 않고 로컬 캐시에서 제거합니다.
<i>CommitUpdates</i>	자동 처리됨	<i>Reconcile</i>	성공적으로 업데이트된 애플리케이션 다음에 이어지는 업데이트 캐시를 지웁니다.
<i>FetchAll</i>	지원하지 않음	<i>GetNextPacket</i> (및 <i>PacketRecords</i>)	데이터베이스 레코드를 편집 및 업데이트할 목적으로 로컬 캐시에 복사합니다.
<i>RevertRecord</i>	<i>CancelBatch</i>	<i>RevertRecord</i>	업데이트가 아직 적용되지 않았으면 현재 레코드에 대한 업데이트를 취소합니다.

크로스 플랫폼 인터넷 애플리케이션

인터넷 애플리케이션은 클라이언트를 서버에 연결하기 위해 표준 인터넷 프로토콜을 사용하는 클라이언트/서버 애플리케이션입니다. 클라이언트/서버 통신용 표준 인터넷 프로토콜을 사용하므로 애플리케이션을 크로스 플랫폼으로 만들 수 있습니다. 예를 들어 인터넷 애플리케이션의 서버측 프로그램은 컴퓨터의 웹 서버 소프트웨어를 통해 클라이언트와 통신합니다. 서버 애플리케이션은 일반적으로 Linux 또는 Windows용으로 작성되지만 크로스 플랫폼이 될 수 있습니다. 클라이언트는 두 플랫폼 중 어느 하나에 들 수 있습니다.

Delphi나 Kylix를 사용하면 Linux에 배치하기 위한 CGI나 Apache 애플리케이션 등의 웹 서버 애플리케이션을 작성할 수 있습니다. Windows에서는 Microsoft Server DLL (ISAPI), Netscape Server DLL (NSAPI) 및 Windows CGI 애플리케이션과 같은 다른 유형의 웹 서버를 만들 수 있습니다. 직접적인 CGI 애플리케이션과 WebBroker를 사용하는 일부 애플리케이션만 Windows와 Linux 모두에서 실행됩니다.

인터넷 애플리케이션을 Linux로 포팅

크로스 플랫폼으로 만들려는 인터넷 애플리케이션을 사용하고 있는 경우 웹 서버 애플리케이션을 포팅할 것인지 아니면 Linux에서 새 애플리케이션을 작성할 것인지를 고려해야 합니다. 웹 서버 작성에 대한 내용은 27장 "인터넷 애플리케이션 생성"을 참조하십시오. 애플리케이션이 WebBroker를 사용하고 WebBroker 인터페이스에 쓰지만 원시 API 호출을 사용하지 않는 경우에는 Linux로 포팅하는 것만큼 어렵지는 않습니다.

애플리케이션이 ISAPI, NSAPI, Windows CGI 또는 기타 웹 API를 사용하는 경우에는 포팅하기가 더 어렵습니다. 소스 파일을 통해 검색하고 이 API 호출을 Apache (Apache API용 함수 프로토타입에 대해서는 인터넷 디렉토리에서 httpd.pas 참조)나 CGI 호출로 변환해야 합니다. 10-2 페이지의 "VCL 애플리케이션을 CLX로 포팅"에 설명되어 있는 다른 모든 내용들도 변경해야 합니다.

11

패키지와 컴포넌트 사용

*패키지*는 Delphi 애플리케이션이나 IDE 또는 두 가지 모두에서 사용하는 특수한 동적 연결 라이브러리입니다. *런타임 패키지*는 사용자가 애플리케이션을 실행할 때 기능을 제공합니다. *디자인 타임 패키지*는 IDE에 컴포넌트를 설치하고 사용자 지정 컴포넌트에 대한 특수 속성 편집기를 생성하는 데 사용됩니다. 단일 패키지가 디자인 타임과 런타임 모두 사용할 수 있고 디자인 타임 패키지는 종종 런타임 패키지를 호출하여 작동합니다. 패키지를 다른 DLL과 구별하기 위해 패키지 라이브러리를 .bpl(Borland package library) 확장자로 끝나는 파일에 저장합니다.

다른 런타임 라이브러리처럼 패키지는 애플리케이션 사이에 공유할 수 있는 코드를 포함할 수 있습니다. 예를 들어, 가장 많이 사용되는 Delphi 컴포넌트는 vcl이라는 패키지에 있습니다. 애플리케이션을 생성할 때마다 애플리케이션은 자동으로 vcl을 사용합니다. 이러한 방법으로 생성한 애플리케이션을 컴파일하면 애플리케이션의 실행 이미지는 코드와 코드에 고유한 데이터만 포함합니다. 런타임 패키지의 공통 코드는 vcl60.bpl이라는 런타임 패키지에 있습니다. 몇 개의 패키지 사용 가능 애플리케이션을 설치한 컴퓨터는 vcl60.bpl 복사본 하나만 필요합니다. 이는 모든 애플리케이션과 Delphi IDE가 vcl60.bpl 복사본을 공유하기 때문입니다.

Delphi는 VCL 및 CLX 컴포넌트를 캡슐화하는 몇 개의 미리 컴파일된 런타임 패키지가 함께 포함되어 있습니다. Delphi는 또한 IDE에서 컴포넌트를 처리하는 디자인 타임 패키지를 사용합니다.

애플리케이션을 패키지와 함께 또는 패키지 없이 구축할 수 있습니다. 그러나 IDE에 사용자 지정 컴포넌트를 추가하려면 패키지를 디자인 타임 패키지로 설치해야 합니다.

애플리케이션 사이에 공유하는 런타임 패키지를 사용자가 직접 만들 수 있습니다. Delphi 컴포넌트를 작성하는 경우 컴포넌트를 설치하기 전에 디자인 타임 패키지로 컴파일할 수 있습니다.

패키지의 이점

디자인 타임 패키지는 사용자 지정 컴포넌트를 배포하고 설치하는 작업을 단순화합니다. 옵션으로 사용하는 런타임 패키지는 전통적인 프로그래밍을 능가하는 몇 가지 이점을 제공합니다. 재사용된 코드를 런타임 라이브러리로 컴파일하여 애플리케이션 간에 공유할 수 있습니다. 예를 들어, Delphi를 비롯한 모든 애플리케이션은 패키지를 통해 표준 컴포넌트에 액세스할 수 있습니다. 애플리케이션에는 실행 파일에 바운드된 별도의 컴포넌트 라이브러리의 복사본이 없기 때문에 실행 파일은 더욱 작아져 시스템 자원과 하드 디스크 저장소가 절약됩니다. 더욱이 애플리케이션에 고유한 코드만 빌드 시 컴파일되기 때문에 패키지를 사용하면 컴파일이 빨라집니다.

패키지와 표준 DLL

IDE를 통해 사용할 수 있는 사용자 지정 컴포넌트를 만들려고 하는 경우 패키지를 만듭니다. 애플리케이션 구축에 사용된 개발 툴에 상관 없이 애플리케이션에서 호출할 수 있는 라이브러리를 구축하려는 경우 표준 DLL을 만듭니다.

다음 표는 패키지에 연결된 파일 형식을 나열한 것입니다.

표 11.1 컴파일된 패키지 파일

파일 확장자	내용
dpc	패키지에 포함된 유닛을 나열하는 소스 파일.
dcp	컴파일러가 필요로 하는 모든 기호 정보를 비롯한 패키지 헤더와 패키지의 모든 dcu 파일의 연결을 포함하는 바이너리 이미지. 각 패키지에 대해 dcp 파일이 하나 생성됩니다. dcp에 대한 기본(base) 이름은 dpc 소스 파일의 기본(base) 이름입니다. 패키지와 함께 애플리케이션을 구축하려면 .dcp 파일이 있어야 합니다.
dcu	패키지에 포함된 유닛 파일에 대한 바이너리 이미지. 필요한 경우, 각 유닛 파일에 dcu가 하나씩 생성됩니다.
bpl	런타임 패키지. 이 파일은 특수한 Delphi 특정 기능을 가진 Windows DLL입니다. bpl에 대한 기본(base) 이름은 dpc 소스 파일의 기본(base) 이름입니다.

한 패키지에 VCL이나 CLX 또는 이 두 컴포넌트 유형을 포함할 수 있습니다. 크로스 플랫폼을 나타내는 패키지는 CLX 컴포넌트만 포함해야 합니다.

참고 패키지는 애플리케이션에 있는 다른 모듈과 전역 데이터를 공유합니다.

DLL 및 패키지에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.

런타임 패키지

런타임 패키지는 Delphi 애플리케이션과 함께 배포됩니다. 런타임 패키지는 사용자가 애플리케이션을 실행할 때 기능을 제공합니다.

패키지를 사용하는 애플리케이션을 실행하려면 컴퓨터에 애플리케이션의 실행 파일과 애플리케이션이 사용하는 모든 패키지(.bpl 파일)가 있어야 합니다. .bpl 파일은 애플리

케이션이 사용하는 시스템 경로에 있어야 합니다. 애플리케이션을 배포하는 경우 사용자가 필요한 .bpl 파일의 올바른 버전을 가지고 있는지 확인해야 합니다.

애플리케이션에서 패키지 사용

애플리케이션에서 패키지를 사용하려면 다음과 같이 합니다.

- 1 IDE에서 프로젝트를 로드하거나 만듭니다.
- 2 Project|Options를 선택합니다.
- 3 Packages 탭을 선택합니다.
- 4 "Build with Runtime Packages" 체크 박스를 선택하고 아래에 있는 편집 상자에 패키지 이름을 하나 이상 입력합니다. 설치된 디자인 타임 패키지에 연결된 런타임 패키지를 편집 상자에서 선택해도 됩니다.
- 5 기존 목록에 패키지를 추가하려면 Add 버튼을 클릭하고 Add Runtime Package 대화 상자에 새 패키지 이름을 입력합니다. 사용 가능한 패키지 목록에서 검색하려면 Add 버튼을 클릭한 다음 Add Runtime Package 대화 상자의 Package Name 편집 상자 옆에 있는 Browse 버튼을 클릭합니다.

Add Runtime Package 대화 상자의 Search Path 편집 상자를 편집하면 Delphi의 전역 Library Path가 변경됩니다.

패키지 이름 또는 Delphi 릴리스를 나타내는 숫자와 함께 파일 확장자를 포함하지 않아도 됩니다. 즉, vcl60.bpl은 vcl로 쓰여 있습니다. Runtime Packages 편집 상자에 직접 입력하는 경우 여러 개의 이름은 세미콜론으로 분리해야 합니다. 예를 들면, 다음과 같습니다.

```
rtl;vcl;vclldb;vclado;vclx;Vclbde;
```

Runtime Packages 편집 상자에 나열된 패키지는 컴파일할 때 애플리케이션에 자동으로 연결됩니다. 중복되는 패키지 이름은 무시되며 편집 상자가 비어 있으면 애플리케이션은 패키지 없이 컴파일됩니다.

런타임 패키지는 현재 프로젝트에 대해서만 선택됩니다. 현재 선택 값을 새 프로젝트에 대한 자동 기본값으로 만들려면 대화 상자 아래쪽에 있는 "Defaults" 체크 박스를 선택합니다.

참고 패키지가 포함된 애플리케이션을 생성하려면 원래의 Delphi 유닛 이름을 소스 파일의 **uses** 절에 계속 포함해야 합니다. 예를 들어, 주요 품의 소스 파일은 다음과 같이 시작합니다.

```
unit MainForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
```

이 예제에서 참조한 유닛은 `vcl` 및 `rtl` 패키지에 포함되어 있습니다. 하지만 애플리케이션에 `vcl` 및 `rtl`을 사용하거나 컴파일러 오류가 발생하더라도 **uses** 절에 이러한 참조가 포함되어야 합니다. 생성된 소스 파일에서 Delphi는 이 유닛을 자동으로 **uses** 절에 추가합니다.

패키지 동적 로딩

런타임에 패키지를 로드하려면 `LoadPackage` 함수를 호출합니다. `LoadPackage`는 패키지를 로드하고 중복 유닛을 확인하고 패키지에 포함된 모든 유닛의 초기화 블록을 호출합니다. 예를 들어, 다음 코드는 파일이 파일 선택 대화 상자에서 선택될 때 실행될 수 있습니다.

```
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

패키지를 동적으로 언로드하려면 `UnloadPackage`를 호출합니다. 패키지에 정의된 클래스 인스턴스를 소멸하고 패키지로 등록된 클래스의 등록을 해제할 때 주의하십시오.

사용할 런타임 패키지 결정

Delphi에는 `rtl` 및 `vcl`을 포함하여 기본 랭귀지와 컴포넌트 지원을 제공하는 몇 개의 미리 컴파일된 런타임 패키지가 함께 포함되어 있습니다.

`vcl` 패키지에는 가장 공통적으로 사용되는 컴포넌트가 들어 있으며 `rtl` 패키지에는 컴포넌트 이외의 모든 시스템 함수와 Windows 인터페이스 요소가 들어 있습니다. 그러나 별도의 패키지에서 사용할 수 있는 데이터베이스 컴포넌트나 다른 특수 컴포넌트는 포함하지 않습니다.

패키지를 사용하는 클라이언트/서버 데이터베이스 애플리케이션을 생성하려면 최소한 세 개의 런타임 패키지, 즉 `vcl`과 `vcldb`가 필요하고 애플리케이션에 Outline 컴포넌트를 사용하려는 경우, `vclx`가 필요합니다. 이 패키지를 사용하려면 Project|Options를 선택하고 Packages 탭을 선택하여 Runtime Packages 편집 상자에 다음 목록을 입력합니다.

```
rtl;vcl;Vcldb;vclx;
```

`vcl` 및 `rtl`이 `vcldb`의 Requires 절에 참조되어 있으므로 실제로는 `vcl` 및 `rtl`을 포함하지 않아도 됩니다. 11-9 페이지의 "Requires 절"을 참조하십시오. 애플리케이션은 `vcl` 및 `rtl`이 Runtime Packages 편집 상자에 포함 여부에 관계 없이 동일하게 컴파일합니다.

사용자 지정 패키지

사용자 지정 패키지는 사용자가 직접 코딩하고 컴파일한 `bpl` 또는 벤더가 제공한 미리 컴파일된 패키지입니다. 애플리케이션과 함께 사용자 지정 런타임 패키지를 사용하려면 Project|Options를 선택하고 Packages 페이지에 있는 Runtime Packages 편집 상자에 패키지 이름을 추가합니다. 예를 들어 `stats.bpl`이라고 하는 통계 패키지가 있다

고 가정합니다. 애플리케이션에서 이 통계 패키지를 사용하려면 Runtime Packages 편집 상자에 입력한 행은 다음과 같이 나타납니다.

```
rtl\vcl\vcldb\stats
```

사용자가 패키지를 직접 만드는 경우 필요에 따라 목록에 추가할 수 있습니다.

디자인 타임 패키지

디자인 타임 패키지는 IDE의 컴포넌트 팔레트에 컴포넌트를 설치하고 사용자 지정 컴포넌트에 대한 특수 속성 편집기를 생성하는 데 사용됩니다.

Delphi에는 IDE에 미리 설치되어 있는 여러 디자인 타임 컴포넌트 패키지가 함께 포함되어 있습니다. 설치된 패키지는 사용하는 Delphi 버전 및 사용자 지정 여부에 따라 결정됩니다. Component|Install Packages를 선택하면 시스템에 설치되어 있는 패키지 목록을 볼 수 있습니다.

디자인 타임 패키지는 Requires 절에 참조하는 런타임 패키지를 호출함으로써 실행됩니다. 11-9 페이지의 "Requires 절"을 참조하십시오. 예를 들어, dclstd는 vcl을 참조합니다. dclstd 자체에는 많은 표준 컴포넌트를 컴포넌트 팔레트에서 사용할 수 있는 추가 기능이 있습니다.

이미 설치된 패키지 외에 IDE에 사용자의 컴포넌트 패키지나 타사 개발자가 제공한 컴포넌트 패키지를 설치할 수 있습니다. dclusr 디자인 타임 패키지는 새 컴포넌트에 대한 기본 컨테이너로 제공됩니다.

컴포넌트 패키지 설치

모든 컴포넌트는 패키지로 IDE에 설치됩니다. 직접 컴포넌트를 만든 경우 컴포넌트를 포함하는 패키지를 만들고 컴파일합니다(11-6 페이지의 "패키지 생성 및 편집" 참조). 사용자의 컴포넌트 소스 코드는 5부 "사용자 지정 컴포넌트 생성"에서 설명한 방법을 따라야 합니다.

사용자의 컴포넌트 또는 타사 공급자가 제공하는 컴포넌트를 설치하거나 설치를 제거하려면 다음 단계를 따릅니다.

- 1 새 패키지를 설치하는 경우 패키지 파일을 로컬 디렉토리로 복사하거나 이동합니다. 패키지가 .bpl, .dcp 및 .dcu 파일과 함께 제공된 경우 모든 파일을 복사하십시오. 추가된 파일에 대한 자세한 내용은 "컴파일이 성공했을 때 생성되는 패키지 파일"을 참조하십시오.

.dcp 파일과 .dcu 파일이 배포본에 포함된 경우 이 파일을 저장하는 디렉토리는 Delphi Library Path에 있어야 합니다.

패키지가 .dpc(패키지 모음) 파일과 함께 제공된 경우 이 한 파일만 복사해야 합니다. .dpc 파일은 다른 파일을 포함합니다. (패키지 모음 파일에 대한 자세한 내용은 11-13 페이지의 "패키지 모음 파일"을 참조하십시오.)

- 2 Component|Install Packages를 IDE 메뉴에서 선택하거나 Project|Options를 선택하고 Packages 탭을 클릭합니다.

3 사용 가능한 패키지 목록이 "Design packages" 아래에 나타납니다.

- IDE에 패키지를 설치하려면 옆에 있는 체크 박스를 선택합니다.
- 패키지를 설치 제거하려면 체크 박스의 선택을 해제합니다.
- 설치된 패키지에 포함된 컴포넌트 목록을 보려면 패키지를 선택하고 Components를 클릭합니다.
- 패키지를 목록에 추가하려면 Add를 클릭하고 Open Package 대화 상자에서 .bpl 또는 .dpc 파일이 있는 디렉토리를 검색합니다(1 단계 참조). .bpl이나 .dpc 파일을 선택한 다음 Open을 클릭합니다. .dpc 파일을 선택한 경우 패키지 모음에서 가져온 .bpl 및 기타 파일의 추출을 처리할 수 있도록 새 대화 상자가 나타납니다.
- 목록에서 패키지를 제거하려면 패키지를 선택하고 Remove를 클릭합니다.

4 OK를 클릭합니다.

컴포넌트의 *RegisterComponents* 프로시저에서 지정한 컴포넌트 팔레트에 패키지의 컴포넌트를 동일한 프로시저에서 할당된 이름으로 설치합니다.

기본 설정 값을 변경하지 않는 경우 모든 사용 가능한 패키지를 설치한 새 프로젝트가 생성됩니다. 현재 설치 선택 사항을 새 프로젝트에 대한 자동 기본값으로 만들려면 Project Options 대화 상자의 Packages 탭 아래 쪽에 있는 Default 체크 박스를 선택합니다.

패키지를 설치 제거하지 않고 컴포넌트 팔레트에서 컴포넌트를 제거하려면 Component | Configure Palette를 선택하거나 Tools | Environment Options를 선택하고 Palette 탭을 클릭합니다. Palette 옵션 탭은 컴포넌트가 나타나는 컴포넌트 팔레트 페이지의 이름과 함께 설치된 각 컴포넌트를 나열합니다. 컴포넌트를 선택하고 Hide를 클릭하면 팔레트에서 컴포넌트가 제거됩니다.

패키지 생성 및 편집

패키지를 만들려면 다음을 지정해야 합니다.

- 패키지의 *이름*
- 새 패키지에 필요하거나 새 패키지에 연결되는 다른 패키지의 목록
- 컴파일될 때 패키지가 포함되거나 패키지에 바운드되는 유닛 파일의 목록. 패키지는 본질적으로 컴파일된 .bpl의 기능을 포함하는 소스 코드 유닛에 대한 래퍼(wrapper)입니다. Contains 절에 패키지에 컴파일하려는 사용자 지정 컴포넌트의 소스 코드 유닛을 지정합니다.

.dpc 확장자로 끝나는 패키지 소스 파일은 Package 에디터에 의해 생성됩니다.

패키지 생성

패키지를 만들려면 다음 절차를 따릅니다. 여기에서 대략적으로 설명한 단계에 대한 자세한 내용은 11-8 페이지의 "패키지 구조 이해"를 참조하십시오.

참고 크로스 플랫폼 애플리케이션을 개발하는 경우라면 IFDEF를 패키지 파일(.dpc)에 사용하지 마십시오. 그러나 소스 코드에는 사용할 수 있습니다.

- 1 File|New를 선택하고 Package 아이콘을 선택한 다음 OK를 클릭합니다.
- 2 생성된 패키지는 Package 에디터에 나타납니다.
- 3 Package 에디터는 새 패키지의 *Requires* 노드와 *Contains* 노드를 보여 줍니다.
- 4 Add to package 스피드 버튼을 클릭하여 유닛을 **contains** 절에 추가합니다. Add unit 페이지에서 Unit file name 편집 상자에 .pas 파일명을 입력하거나 Browse를 클릭하여 파일을 검색한 다음 OK를 클릭합니다. 선택한 유닛은 Package 에디터의 Contains 노드 아래에 나타납니다. 추가 유닛도 이 단계를 반복하여 추가할 수 있습니다.
- 5 Add to package 스피드 버튼을 클릭하여 패키지를 **requires** 절에 추가합니다. Requires 페이지에서 Package name 편집 상자에 .dcp 파일명을 입력하거나 Browse를 클릭하여 파일을 검색한 다음 OK를 클릭합니다. 선택한 패키지는 Package 에디터의 Requires 노드 아래에 나타납니다. 추가 패키지도 이 단계를 반복하여 추가할 수 있습니다.
- 6 Options 스피드 버튼을 클릭하고 빌드하려는 패키지 종류를 결정합니다.
 - 디자인 타임 전용 패키지(런타임에 사용할 수 없는 패키지)를 만들려면 Designtime only 라디오 버튼을 선택합니다. 또는 {\$DESIGNONLY} 컴파일러 지시어를 dpc 파일에 추가합니다.
 - 런타임 전용 패키지(설치될 수 없는 패키지)를 만들려면 Runtime only 라디오 버튼을 선택합니다. 또는 {\$RUNONLY} 컴파일러 지시어를 dpc 파일에 추가합니다.
 - 디자인 타임과 런타임 모두에 사용할 수 있는 패키지를 만들려면 Designtime and runtime 라디오 버튼을 선택합니다.
- 7 Package 에디터에서 Compile package 스피드 버튼을 클릭하여 패키지를 컴파일합니다.

기존 패키지 편집

다음과 같은 몇 가지 방법으로 편집을 위해 기존 패키지를 열 수 있습니다.

- File|Open(또는 File|Reopen)을 선택하고 dpc 파일을 선택합니다.
- Component|Install Packages를 선택하고 Design Packages 목록에서 패키지를 선택한 다음 Edit 버튼을 클릭합니다.
- Package 에디터가 열리면 Requires 노드의 패키지 중 하나를 선택하여 마우스 오른쪽 버튼을 클릭한 다음 Open을 선택합니다.

패키지의 설명을 편집하거나 사용 옵션을 설정하려면 Package 에디터에서 Options 스피드 버튼을 클릭하고 Description 탭을 선택합니다.

Project Options 대화 상자는 하단 왼쪽 모서리에 Default 체크 박스가 있습니다. 이 체크 박스를 선택하고 OK를 클릭하면 선택한 옵션은 새 프로젝트에 대한 기본 설정 값으로 저장됩니다. 원래 기본값을 복원하려면 defproj.dof 파일을 삭제하거나 이름을 재지정하십시오.

수동으로 패키지 소스 파일 편집

프로젝트 파일처럼 패키지 소스 파일은 사용자가 제공하는 정보로부터 Delphi에 의해 생성됩니다. 프로젝트 파일처럼 패키지 소스 파일도 수동으로 편집할 수 있습니다. 패키지 소스 파일은 오브젝트 파스칼 소스 코드를 포함하는 다른 파일과의 혼동을 피하기 위해 .dpk (Delphi 패키지) 확장자로 저장해야 합니다.

코드 에디터에서 패키지 소스 파일을 열려면 다음과 같이 합니다.

- 1 Package Editor에서 패키지를 엽니다.
- 2 Package Editor를 마우스 오른쪽 버튼으로 클릭하고 View Source를 선택합니다.
 - **Package** 헤더는 패키지 이름을 지정합니다.
 - **requires** 절은 현재 패키지가 사용하는 다른 외부 패키지를 나열합니다. 다른 패키지에서 유닛을 사용하는 어떤 유닛도 패키지에 포함되지 않으면 **requires** 절은 필요하지 않습니다.
 - **contains** 절은 컴파일되고 패키지에 바운드된 유닛 파일을 식별합니다. 필요한 패키지에 없는 포함된 유닛이 사용하는 모든 유닛은 contains 절에 나열되지 않습니다 (컴파일러가 경고를 보냄) 패키지에 바운드됩니다.

예를 들어, 다음 코드는 소스 파일 vcldb60.bpl에 vcldb 패키지를 선언합니다.

```
package vcldb;
  requires vcldb;
  contains rtl, vcl, Db, DBActns, DB0leCtl, Dbcgrids, dbCommon, dbConsts, Dbctrls,
  Dbgrids, Dblogdlg, SQLTimSt, FmtBcd;
end.
```

패키지 구조 이해

패키지는 다음을 포함합니다.

- 패키지 이름
- Requires 절
- Contains 절

패키지 이름 지정

패키지 이름은 프로젝트 내에서 고유해야 합니다. 패키지 이름을 STATS라고 지정하면 Package 에디터는 STATS.dpk 라는 소스 파일을 생성합니다. 컴파일러는 각각 STATS.bpl과 STATS.dcp 라는 실행 파일과 바이너리 이미지를 생성합니다. 다른 패키지의 **requires** 절에서 패키지를 참조하거나 애플리케이션의 패키지를 사용할 때 STATS를 사용하십시오.

Requires 절

requires 절은 현재 패키지가 사용하는 다른 외부 패키지를 지정합니다. **requires** 절에 포함된 외부 패키지는 외부 패키지에 포함된 유닛 중 하나와 현재 패키지를 모두 사용하는 애플리케이션에 컴파일 시 자동으로 연결됩니다.

패키지에 포함된 유닛 파일이 다른 패키지화된 유닛을 참조하면 다른 패키지는 사용자 패키지의 **requires** 절에 나타나야 하거나 사용자가 다른 패키지를 추가해야 합니다. 다른 패키지가 **requires** 절에서 생략되면 컴파일러는 다른 패키지를 "암시적으로 포함된 유닛"인 사용자의 패키지에 import합니다.

참고 사용자가 만든 대부분의 패키지는 rtl이 필요합니다. VCL 컴포넌트를 사용하는 경우에도 vcl 패키지를 포함해야 할 것입니다. 크로스 플랫폼 프로그래밍에 CLX 컴포넌트를 사용하는 경우에는 VisualCLX를 포함해야 합니다.

순환 패키지 참조 피하기

패키지는 **requires** 절에 순환 참조를 포함할 수 없습니다. 이는 다음을 의미합니다.

- 패키지는 자신의 **requires** 절에서 자신을 참조할 수 없습니다.
- 참조 체인은 체인에서 패키지를 참조하지 않고 종료해야 합니다. 패키지 A가 패키지 B를 요구하면 패키지 B는 패키지 A를 요구할 수 없습니다. 패키지 A가 패키지 B를 요구하고 패키지 B가 패키지 C를 요구하면 패키지 C는 패키지 A를 요구할 수 없습니다.

중복 패키지 참조 처리

패키지의 **requires** 절이나 Runtime Packages 편집 상자의 중복 참조는 컴파일러에 의해 무시됩니다. 그러나 프로그래밍의 명확성과 가독성을 위해 중복 패키지 참조를 찾아서 제거해야 합니다.

Contains 절

contains 절은 패키지로 바운드되는 유닛 파일을 식별합니다. 직접 패키지를 작성하는 경우 소스 코드를 pas 파일로 저장하고 **contains** 절에 이 파일을 포함합니다.

소스 코드 중복 사용 피하기

패키지는 다른 패키지의 **contains** 절에 나타날 수 없습니다.

패키지의 **contains** 절에 직접적으로 포함되거나 유닛에 간접적으로 포함된 모든 유닛은 컴파일 시 패키지에 바운드됩니다.

*Delphi IDE*를 비롯한 동일한 애플리케이션이 사용하는 둘 이상의 패키지에 유닛을 직접적 또는 간접적으로 포함할 수 없습니다. 이는 vcl에 있는 유닛 중 하나를 포함하는 패키지를 만들면 IDE에 패키지를 설치할 수 없다는 것을 의미합니다. 다른 패키지에 이미 패키지로 만든 유닛 파일을 사용하면 첫 번째 패키지를 두 번째 패키지의 **requires** 절에 둡니다.

패키지 컴파일

IDE 나 명령줄에서 패키지를 컴파일할 수 있습니다. IDE 에서 패키지를 다시 컴파일하려면 다음과 같이 합니다.

- 1 File|Open을 선택합니다.
- 2 Files of Type 드롭다운 목록에서 Delphi 패키지(*.dpk)를 선택합니다.
- 3 대화 상자에서 .dpk 파일을 선택합니다.
- 4 Package 에디터를 열 때 Compile 스피드 버튼을 클릭합니다.

컴파일러 지시어를 패키지 소스 코드에 삽입할 수 있습니다. 자세한 내용은 아래의 "패키지 특정 컴파일러 지시어"를 참조하십시오.

명령줄에서 컴파일하면 몇 가지 패키지 특정 스위치를 사용할 수 있습니다. 자세한 내용은 11-12 페이지의 "명령줄 컴파일러와 링커 사용"을 참조하십시오.

패키지 특정 컴파일러 지시어

다음 표는 소스 코드에 삽입할 수 있는 패키지 특정 컴파일러 지시어를 나열한 것입니다.

표 11.2 패키지 특정 컴파일러 지시어

지시어	용도
{\$IMPLICITBUILD OFF}	나중에 패키지가 암시적으로 다시 컴파일되지 않게 합니다. 저수준 기능을 제공하고, 빌드 사이에서 드물게 변경되거나 소스 코드가 배포되지 않을 패키지를 컴파일할 때 .dpk 파일에서 사용합니다.
{\$G-} 또는 {\$IMPORTEDDATA OFF}	import한 데이터 참조를 생성할 수 없습니다. 이 지시어는 메모리 액세스 효율을 높이지만 지시어가 있는 유닛은 다른 패키지에서 변수를 참조할 수 없습니다.
{\$WEAKPACKAGEUNIT ON}	유닛을 "약하게" 패키지화합니다. 아래의 11-11 페이지의 "약한 패키징"을 참조하십시오.
{\$DENYPACKAGEUNIT ON}	유닛을 패키지에 넣을 수 없습니다.
{\$DESIGNONLY ON}	IDE에서 설치를 위해 패키지를 컴파일합니다. .dpk 파일에 놓습니다.
{\$RUNONLY ON}	패키지를 런타임 전용으로 컴파일합니다. .dpk 파일에 놓습니다.

참고 {\$DENYPACKAGEUNIT ON} 을 소스 코드에 포함시키면 유닛 파일은 패키지화될 수 없습니다. {\$G-} 또는 {\$IMPORTEDDATA OFF} 를 포함시키면 패키지는 동일한 애플리케이션에서 다른 패키지와 함께 사용할 수 없습니다. {\$DESIGNONLY ON} 지시어와 함께 컴파일된 패키지는 IDE가 요구한 여분의 코드를 포함하기 때문에 보통 애플리케이션에서 사용하지 않아야 합니다. 적절한 경우 다른 컴파일러 지시어를 패키지 소스 코드에 포함시켜야 합니다. 여기에서 다루지 않은 컴파일러 지시어에 대한 내용은 온라인 도움말의 컴파일러 지시어를 참조하십시오.

모든 라이브러리에 사용할 수 있는 추가 지시어는 5-9 페이지의 "패키지와 DLL 생성"을 참조하십시오.

약한 패키징

{\$WEAKPACKAGEUNIT} 지시어는 패키지의 .dcp 및 .bpl 파일에 .dcu 파일이 저장되는 방법에 영향을 줍니다. 컴파일러가 생성하는 파일에 대한 내용은 11-12 페이지의 "컴파일러가 성공했을 때 생성되는 패키지 파일"을 참조하십시오. **{\$WEAKPACKAGEUNIT ON}** 이 유닛 파일에 나타나면 컴파일러는 가능한 경우 유닛을 bpl에서 생략하고 다른 애플리케이션이나 패키지가 요구할 때 유닛의 패키지화되지 않은 로컬 복사본을 생성합니다. 이 지시어와 함께 컴파일된 유닛을 "약하게 패키지화"되었다고 합니다.

예를 들어 UNIT1이라는 하나의 유닛만 포함하는 PACK이라고 하는 패키지를 만들었다고 가정합니다. UNIT1은 다른 유닛을 사용하지 않지만 RARE.dll을 호출했다고 가정합니다. 패키지를 컴파일할 때 **{\$WEAKPACKAGEUNIT ON}**을 UNIT1.pas에 놓으면 UNIT1은 PACK.bpl에 포함되지 않습니다. RARE.dll의 복사본을 PACK과 함께 배포할 필요는 없습니다. 그러나 UNIT1은 여전히 PACK.dcp에 포함됩니다. PACK을 사용하는 다른 패키지나 애플리케이션이 UNIT1을 참조하면 PACK.dcp에서 UNIT1을 복사하고 프로젝트에 직접 컴파일합니다.

이제 두 번째 유닛인 UNIT2를 PACK에 추가한다고 가정합니다. UNIT2는 UNIT1을 사용한다고 가정합니다. 이번에는 PACK을 **{\$WEAKPACKAGEUNIT ON}**과 함께 UNIT1.pas에서 컴파일해도 컴파일러는 UNIT1을 PACK.bpl에 포함합니다. 그러나 UNIT1을 참조하는 다른 패키지나 애플리케이션은 PACK.dcp에서 가져온 패키지화되지 않은 복사본을 사용합니다.

참고 **{\$WEAKPACKAGEUNIT ON}** 지시어를 포함하는 유닛 파일은 전역 변수, 초기화 섹션 또는 완료 섹션이 없어야 합니다.

{\$WEAKPACKAGEUNIT} 지시어는 다른 Delphi 프로그래머에게 패키지를 배포하는 개발자를 위한 고급 기능입니다. 이 지시어를 사용하면 드물게 사용되는 공유 객체 파일의 배포를 막을 수 있고 동일한 외부 라이브러리에 의존하는 패키지 간의 충돌을 없앨 수 있습니다.

예를 들어, Delphi의 PenWin 유닛은 PenWin.dll을 참조합니다. 대부분의 프로젝트는 PenWin을 사용하지 않고 대부분의 컴퓨터에는 PenWin.dll이 설치되어 있지 않습니다. 이러한 이유에서 PenWin 유닛은 vcl에 약하게 패키지되어 있습니다. PenWin 및 vcl 패키지를 사용하는 프로젝트를 컴파일할 때 PenWin은 VCL60.dcp에서 복사되고 프로젝트에 직접 연결됩니다. 실행 결과는 PenWin.dll에 정적으로 연결됩니다.

PenWin이 약하게 패키지화되지 않은 경우 두 가지 문제가 발생할 것입니다. 우선 vcl 자체가 PenWin.dll에 정적으로 연결되어 PenWin.dll이 설치되어 있지 않은 컴퓨터에서 vcl을 로드할 수 없게 됩니다. 또 PenWin을 포함한 패키지를 만들려는 경우 PenWin 유닛이 vcl과 패키지 모두에 포함되기 때문에 컴파일러 오류가 발생하게 됩니다. 따라서 약하게 패키지를 만들지 않으면 PenWin은 vcl의 표준 배포판에 포함될 수 없습니다.

명령줄 컴파일러와 링커 사용

명령줄에서 컴파일하는 경우 다음 테이블에서 나열하는 패키지 특정 스위치를 사용할 수 있습니다.

표 11.3 패키지 특정 명령줄 컴파일러 스위치

스위치	용도
-\$G-	import한 데이터 참조를 생성할 수 없습니다. 이 스위치를 사용하면 메모리 액세스 효율을 증가시키지만 이 스위치를 사용하여 컴파일된 패키지는 다른 패키지의 변수를 참조할 수 없습니다.
-LEpath	패키지 bpl 파일을 저장할 디렉토리를 지정합니다.
-LNpath	패키지 dcp 파일을 둘 디렉토리를 지정합니다.
-	패키지를 사용합니다.
LUpackage	
-Z	나중에 패키지가 암시적으로 다시 컴파일되지 않게 합니다. 저수준의 기능을 제공하고 드물게 빌드 사이에서 변경되거나 소스 코드가 배포되지 않을 패키지를 컴파일할 때 사용합니다.

참고 -\$G- 스위치를 사용하면 패키지를 동일한 애플리케이션에서 다른 패키지와 함께 사용할 수 없습니다. 적절한 경우 다른 명령줄 옵션을 패키지를 컴파일할 때 사용할 수도 있습니다. 여기에서 설명하지 않은 명령줄 옵션에 대한 내용은 온라인 도움말의 "명령줄 (Command-line) 컴파일러" 를 참조하십시오.

컴파일이 성공했을 때 생성되는 패키지 파일

패키지를 생성하려면 확장자가 .dpc 인 소스 파일을 컴파일합니다. .dpc 파일의 기본 (base) 이름은 컴파일러가 생성한 파일의 기본 이름이 됩니다. 예를 들어, traypak.dpc 라는 패키지 소스 파일을 컴파일하면 컴파일러는 traypak. bpl이라는 패키지를 생성합니다.

다음 테이블은 성공적인 패키지 컴파일이 생성하는 파일을 나열합니다.

표 11.4 컴파일된 패키지 파일

파일 확장자	내용
dcp	패키지 헤더를 포함하는 바이너리 이미지와 패키지의 모든 dcu 파일의 연결. 각 패키지에 대해 dcp 파일이 하나 생성됩니다. dcp에 대한 기본 (base) 이름은 dpc 소스 파일의 기본 (base) 이름입니다.
dcu	패키지에 포함된 유닛 파일에 대한 바이너리 이미지. 필요한 경우, 각 유닛 파일에 dcu가 하나씩 생성됩니다.
bpl	런타임 패키지. 이 파일은 특수한 Delphi 특정 기능을 가진 Windows DLL입니다. bpl에 대한 기본 (base) 이름은 dpc 소스 파일의 기본 (base) 이름입니다.

컴파일 시 bpi, bpl 및 lib 파일은 Tools|Environment Options 대화 상자의 Library 페이지에 지정된 디렉토리의 기본값에 의해 생성됩니다. Package 에디터의 Options 스크드 버튼을 클릭하여 Project Options 대화 상자가 표시되면 Directories/Conditionals 페이지를 변경하여 기본 설정을 오버라이드할 수 있습니다.

패키지 배포

다른 애플리케이션을 배포하는 것처럼 패키지를 배포할 수 있습니다. 일반적인 배포 정보에 대한 내용은 13장 "애플리케이션 배포"를 참조하십시오.

패키지를 사용하는 애플리케이션 배포

런타임 패키지를 사용하는 애플리케이션을 배포하는 경우 사용자가 애플리케이션이 호출하는 모든 라이브러리(.bpl 또는 .dll) 파일과 더불어 애플리케이션의 .exe 파일도 가지고 있는지 확인합니다. 라이브러리 파일이 .exe 파일과 다른 디렉토리에 있으면 사용자의 경로를 통해 액세스할 수 있어야 합니다. Windows\System 디렉토리에 라이브러리 파일을 입력하는 규칙을 따르려는 경우도 있을 것입니다. InstallShield Express를 사용하는 경우 설치 스크립트는 임의로 패키지를 재설치하기 전에 필요한 모든 패키지에 대해 사용자의 시스템을 확인할 수 있습니다.

다른 개발자에 패키지 배포

런타임이나 디자인 타임 패키지를 다른 Delphi 개발자에게 배포하는 경우 .dcp 와 .bpl 파일을 제공해야 합니다. .dcu 파일도 포함시킬 수 있습니다.

패키지 모음 파일

패키지 모음(.dpc 파일)은 다른 개발자에게 패키지를 편리하게 배포할 수 있는 방법을 제공합니다. 각 패키지 모음은 함께 배포하려는 bpl 및 추가 파일 등 하나 이상의 패키지를 포함합니다. IDE 설치를 위해 패키지 모음을 선택하면 구성 파일은 .pce 컨테이너에서 자동으로 추출되고 Installation 대화 상자에서 모음의 모든 패키지를 설치하거나 선택적으로 패키지를 설치하도록 선택할 수 있습니다.

패키지 모음을 만들려면 다음을 수행하십시오.

- 1 Tools|Package Collection Editor를 선택하여 Package Collection 에디터를 엽니다.
- 2 Add a Package 스피드 버튼을 클릭한 다음 Select Package 대화 상자에서 bpl을 선택하고 Open을 클릭합니다. 모음에 bpl을 더 추가하려면 Add a Package 스피드 버튼을 다시 클릭합니다. Package 에디터의 왼쪽에 있는 트리 다이어그램이 추가하는 bpl을 표시합니다. 패키지를 제거하려면 패키지를 선택하고 Remove Package 스피드 버튼을 클릭합니다.
- 3 트리 다이어그램의 상단에 있는 Collection 노드를 선택합니다. Package Collection 에디터의 오른쪽에 두 개의 필드가 나타납니다.
 - 사용자가 패키지를 설치할 때 Installation 대화 상자에 나타날 패키지 모음에 대한 선택 정보를 Author/Vendor Name 편집 상자에 입력할 수 있습니다.
 - Directory List 아래에 패키지 모음을 설치하려는 기본 디렉토리를 나열합니다. Add, Edit 및 Delete 버튼을 사용하여 이 목록을 편집합니다. 예를 들어 모든 소스

코드 파일을 동일한 디렉토리에 복사한다고 가정합니다. 이 경우 Suggested Path가 C:\MyPackage\Source인 Directory Name으로 Source를 입력할 수 있습니다. Installation 대화 상자는 디렉토리에 대한 권장 경로로 C:\MyPackage\Source를 표시합니다.

- 4 패키지 모음은 bpl뿐만 아니라 배포에 포함하려는 .dcp, .dcu 및 .pas(유닛) 파일, 문서, 모든 기타 파일을 포함할 수 있습니다. 보조(ancillary) 파일은 특정 패키지(bpl)에 연관된 파일 그룹에 있으며 그룹 내의 파일은 연관된 bpl이 설치된 경우에만 설치됩니다. 패키지 모음에 보조(ancillary) 파일을 두려는 경우 트리 다이어그램에서 bpl을 선택하고 Add File Group 스피드 버튼을 클릭하고 파일 그룹에 대한 이름을 입력합니다. 필요에 따라 동일한 방법으로 파일 그룹을 추가합니다. 파일 그룹을 선택할 때 새 필드는 Package Collection 에디터의 오른쪽에 나타납니다.
 - Install Directory 리스트 박스에서 이 그룹 내 파일을 설치할 디렉토리를 선택합니다. (드롭다운 목록에는 위의 3 단계에서 Directory List에 입력한 디렉토리가 포함되어 있습니다.)
 - 이 그룹 내 파일의 설치를 선택 사항으로 만들려면 Optional Group 체크 박스를 선택합니다.
 - Include Files 아래에 이 그룹에 포함할 파일을 나열합니다. Add, Delete 및 Auto 버튼을 사용하여 이 목록을 편집합니다. Auto 버튼을 사용하여 패키지의 **contains** 절에 나열된 확장자가 지정되어 있는 모든 파일을 선택할 수 있습니다. Package Collection 에디터는 Delphi의 전역 Library Path를 사용하여 이 파일들을 검색합니다.
- 5 모음에 있는 모든 패키지의 **requires** 절에 나열된 패키지에 대한 설치 디렉토리를 선택할 수 있습니다. 트리 다이어그램에 bpl을 선택하면 네 개의 새 필드가 Package Collection 에디터의 오른쪽에 나타납니다.
 - Required Executables 리스트 박스에서 **requires** 절에 나열된 패키지에 대해 .bpl 파일을 설치할 디렉토리를 선택합니다. (드롭다운 목록에는 위의 3 단계에서 Directory List에 입력한 디렉토리가 포함되어 있습니다.) Package Collection 에디터는 Delphi의 전역 Library Path를 사용하여 이러한 파일을 검색하고 Required Executable Files 아래에 나열합니다.
 - Required Libraries 리스트 박스에서 **requires** 절에 나열된 패키지에 대해 .dcp 파일을 설치할 디렉토리를 선택합니다. (드롭다운 목록에는 위의 3 단계에서 Directory List에 입력한 디렉토리가 포함되어 있습니다.) Package Collection 에디터는 Delphi의 전역 Library Path를 사용하여 이러한 파일을 검색하고 Required Library Files 아래에 나열합니다.
- 6 패키지 모음 소스 파일을 저장하려면 File|Save를 선택합니다. 패키지 모음 소스 파일은 .pce 확장자로 저장해야 합니다.
- 7 패키지 모음을 만들려면 Compile 스피드 버튼을 클릭합니다. Package Collection 에디터는 소스(.pce) 파일과 동일한 이름이 있는 .dcp 파일을 생성합니다. 아직 소스 파일을 저장하지 않은 경우 편집기는 컴파일하기 전에 파일 이름을 쿼리합니다. 기존 .pce 파일을 편집하거나 재컴파일하려면 Package Collection 에디터에서 File|Open을 선택하고 작업하려는 파일을 찾습니다.

12

국제적인 애플리케이션 만들기

이 장에서는 해외 시장에 배포하려는 애플리케이션 개발에 대해 설명합니다. 미리 계획을 잘 세우면 국내 시장뿐 아니라 해외 시장에서 애플리케이션을 사용할 수 있도록 하는데 필요한 시간과 코드를 줄일 수 있습니다.

국제화 및 지역화

해외 시장에 배포할 수 있는 애플리케이션을 만들려면 다음과 같은 중요한 두 가지 단계를 수행해야 합니다.

- 국제화
- 지역화

사용하는 Delphi 버전에 Translation Tool이 포함되어 있으면 지역화를 관리하는 데 사용할 수 있습니다. Translation Tool(ETM.hlp)에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

국제화

국제화란 프로그램이 여러 로케일에서 작동할 수 있도록 하는 것입니다. 로케일은 언어뿐 아니라 대상 국가의 문화적 전통을 비롯한 사용자의 환경을 의미합니다. Windows는 광범위한 로케일 집합을 지원하며 각 로케일은 언어와 국가의 쌍으로 설명됩니다.

지역화

지역화란 특정 로케일에서 작동하도록 애플리케이션을 번역하는 것입니다. 지역화는 사용자 인터페이스를 번역하는 것 외에 기능의 사용자 지정을 포함하기도 합니다. 예를 들어, 재무 애플리케이션은 각 나라에 적합한 세법을 인식하도록 수정될 것입니다.

애플리케이션 국제화

국제화된 애플리케이션을 만들려면 다음 단계를 마쳐야 합니다.

- 코드에서 국제적인 문자 집합의 문자열을 처리할 수 있게 해야 합니다.
- 지역화로 인한 변경 내용을 수용할 수 있도록 사용자 인터페이스를 디자인해야 합니다.
- 지역화가 필요한 모든 리소스를 분리해야 합니다.

애플리케이션 코드 활성화

애플리케이션의 코드는 다양한 대상 로케일의 문자열을 처리할 수 있어야 합니다.

문자 집합

Windows 미국 에디션은 ANSI Latin-1 (1252) 문자 집합을 사용합니다. 하지만 다른 Windows 에디션은 다른 문자 집합을 사용합니다. 예를 들어, Windows의 일본어 버전은 Shift-JIS 문자 집합(코드 페이지 932)을 사용하며 멀티바이트 문자 코드로 일본어 문자를 나타냅니다.

일반적으로 다음과 같은 세 가지 유형의 문자 집합이 있습니다.

- 싱글바이트
- 멀티바이트
- 고정 너비 멀티바이트

Windows와 Linux는 모두 유니코드뿐만 아니라 싱글바이트 및 멀티바이트 문자 집합을 지원합니다. 싱글바이트 문자 집합에서 문자열의 각 바이트는 문자 하나를 나타냅니다. 영어권 운영 체제에서 사용하는 ANSI 문자 집합은 싱글바이트 문자 집합입니다.

멀티바이트 문자 집합에서 일부 문자는 1바이트로 표시하고 그 외 문자들은 2바이트 이상으로 표시합니다. 멀티바이트 문자의 첫 바이트는 선행 바이트라고 합니다. 일반적으로 멀티바이트 문자 집합의 하위 128 문자는 7비트의 ASCII 문자에 매핑됩니다. 그리고 순서 값이 127 이상인 모든 바이트는 멀티바이트 문자의 선행 바이트입니다. 싱글바이트 문자만 Null 값(#0)을 가질 수 있습니다. 멀티바이트 문자 집합, 특히 더블바이트 문자 집합(DBCS)은 아시아 언어권에서 널리 사용되며 Linux에서 사용되는 UTF-8 문자 집합은 유니코드의 멀티바이트 인코딩 방식입니다.

OEM 및 ANSI 문자 집합

간혹 Windows 문자 집합(ANSI)과 사용자 컴퓨터의 코드 페이지에서 지정한 OEM 문자 집합이라고 하는 문자 집합 사이에서 변환해야 하는 경우도 있습니다.

멀티바이트 문자 집합

아시아에서 사용되는 표의 문자 집합은 해당 언어의 문자와 1바이트(8비트) *char* 타입의 문자 사이에 단순한 1:1 매핑을 사용할 수 없습니다. 이러한 언어에는 1바이트 *char* 타입을 사용하여 표현할 수 없는 문자들이 많이 있습니다. 대신 멀티바이트 문자열은 한

문자당 하나 이상의 바이트를 포함할 수 있습니다. AnsiString은 싱글바이트와 멀티바이트 문자를 함께 혼합하여 포함할 수 있습니다.

모든 멀티바이트 문자 코드의 첫 바이트는 특정 문자 집합에 따라 예약된 범위를 사용합니다. 경우에 따라 두 번째 및 다음 바이트들은 각각의 1바이트 문자의 문자 코드와 동일하거나 멀티바이트 문자의 첫 번째 바이트에 대한 예약된 범위를 사용합니다. 따라서 문자열의 특정 바이트가 단일 문자를 나타내는지 또는 멀티바이트 문자인지 구별할 수 있는 유일한 방법은 문자열을 처음부터 읽어서 예약된 범위에서 선행 바이트를 만났을 때 2바이트 이상의 문자로 분석해 보는 것입니다.

아시아 로케일에 대한 코드를 작성할 때는 멀티바이트 문자로 문자열을 구문 분석할 수 있는 함수를 사용하여 모든 문자열을 처리해야 합니다. Delphi는 이러한 작업을 수행할 수 있도록 다음과 같은 여러 가지 런타임 라이브러리 함수를 제공합니다.

AdjustLineBreaks	AnsiStrLower	ExtractFileDir
AnsiCompareFileName	AnsiStrPos	ExtractFileExt
AnsiExtractQuotedStr	AnsiStrRScan	ExtractFileName
AnsiLastChar	AnsiStrScan	ExtractFilePath
AnsiLowerCase	AnsiStrUpper	ExtractRelativePath
AnsiLowerCaseFileName	AnsiUpperCase	FileSearch
AnsiPos	AnsiUpperCaseFileName	IsDelimiter
AnsiQuotedStr	ByteToCharIndex	IsPathDelimiter
AnsiStrComp	ByteToCharLen	LastDelimiter
AnsiStrIComp	ByteType	StrByteType
AnsiStrLastChar	ChangeFileExt	StringReplace
AnsiStrLComp	CharToByteIndex	WrapText
AnsiStrLIComp	CharToByteLen	

바이트 단위의 문자열 길이가 문자열의 문자 수와 같을 필요는 없습니다. 멀티바이트 문자를 반으로 나누어 문자열을 잘라내지 않도록 주의하십시오. 문자의 크기를 알 수 없으므로 함수 또는 프로시저의 매개변수로 문자를 전달하지 마십시오. 그 대신 문자나 문자열에 대한 포인터를 항상 전달하십시오.

와이드 문자

실제로 표의 문자 집합을 사용하는 한 가지 방법은 모든 문자를 유니코드와 같은 와이드 문자 인코딩 구성으로 변환하는 것입니다. 유니코드 문자와 문자열은 또한 와이드 문자 및 와이드 문자열이라고 합니다. 유니코드 문자 집합에서 각 문자는 2바이트로 표시합니다. 따라서 유니코드 문자열은 개별적인 바이트의 연속이 아니라 2바이트 워드의 연속입니다.

첫 256 유니코드 문자는 ANSI 문자 집합에 매핑됩니다. Windows 운영 체제는 유니코드(UCS-2)를 지원합니다. Linux 운영 체제는 UCS-2의 상위 집합인 UCS-4를 지원합니다. Delphi와 Kylix는 두 플랫폼 모두에서 UCS-2를 지원합니다. 와이드 문자는 하나 가 아닌 2바이트이므로 문자 집합은 다른 많은 문자를 나타낼 수 있습니다.

와이드 문자 인코딩 구성을 사용하면 MBCS(멀티바이트 문자 집합) 시스템에서는 작동하지 않는 문자열에 대한 일반적인 가정을 만들 수 있는 이점이 있습니다. 문자열의 바이트 수와 문자 수 사이에는 직접적인 관계가 있습니다. 따라서 문자가 반으로 잘리거나 문자의 뒷부분을 다른 문자의 앞부분으로 오인하는 실수에 대해서 걱정할 필요가 없습니다.

와이드 문자를 사용할 때 최대 단점은 Windows 9x 버전에서만 일부 와이드 문자 API 함수 호출을 지원한다는 점입니다. 이로 인해 VCL 컴포넌트는 모든 문자열 값을 싱글 바이트 또는 MBCS 문자열로 나타냅니다. 문자열 속성을 설정하거나 문자열 값을 읽을 때마다 와이드 문자 시스템과 MBCS 시스템 간의 변환에는 추가 코드가 필요하며 애플리케이션 속도가 느려집니다. 하지만 문자와 *WideChars* 간 1:1 매핑을 이용해야 하는 알고리즘을 처리하는 일부 특수한 문자열에 대한 와이드 문자열로 번역할 수 있습니다.

애플리케이션에 양방향 기능 포함

일부 언어는 서구권 언어에서 흔히 발견되는 왼쪽에서 오른쪽으로 읽는 순서를 따르지 않고 단어는 오른쪽에서 왼쪽으로 숫자는 왼쪽에서 오른쪽으로 읽는 경우가 있습니다. 이 언어들은 그러한 차이 때문에 양방향(BiDi)이라고 합니다. 다른 중동지역의 언어도 양방향이지만 가장 일반적인 양방향 언어는 아랍어와 히브리어입니다.

*TApplication*에는 *BiDiKeyboard*와 *NonBiDiKeyboard* 두 가지 속성이 있는데 이 두 속성을 사용하여 키보드 레이아웃을 지정할 수 있습니다. 또한 VCL은 *BiDiMode*와 *ParentBiDiMode* 속성을 통해 양방향 지역화를 지원합니다. 다음 표는 이러한 속성을 갖는 VCL 객체를 나열한 것입니다.

표 12.1 BiDi 를 지원하는 VCL 객체

컴포넌트 팔레트 페이지	VCL 객체
표준	TButton
	TCheckBox
	TComboBox
	TEdit
	TGroupBox
	TLabel
	TListBox
	TMainMenu
	TMemo
	TPanel
	TPopupMenu
	TRadioButton
	TRadioGroup
	TScrollBar

표 12.1 BiDi 를 지원하는 VCL 객체 (계속)

컴포넌트 팔레트 페이지	VCL 객체
추가	TActionMainMenuBar TActionToolBar TBitBtn TCheckListBox TColorBox TDrawGrid TLabeledEdit TMaskEdit TScrollBar TSpeedButton TStaticLabel TStaticText TStringGrid TValueListEditor
Win32	TComboBoxEx TDateTimePicker THeaderControl THotKey TListView TMonthCalendar TPageControl TRichEdit TStatusBar TTreeView
Data Controls	TDBCheckBox TDBComboBox TDBEdit TDBGrid TDBListBox TDBLookupComboBox TDBLookupListBox TDBMemo TDBRadioGroup TDBRichEdit TDBText
QReport	TQRDBText TQRExpr TQRLabel TQRMemo

표 12.1 BiDi 를 지원하는 VCL 객체 (계속)

컴포넌트 팔레트 페이지	VCL 객체
	TQRPreview
	TQRSysData
Other classes	TApplication (<i>ParentBiDiMode</i> 없음)
	TBoundLabel
	TControl (<i>ParentBiDiMode</i> 없음)
	TCustomHeaderControl (<i>ParentBiDiMode</i> 없음)
	TForm
	TFrame
	THeaderSection
	THintWindow (<i>ParentBiDiMode</i> 없음)
	TMenu
	TStatusPanel
	TTabControl
	TValueListEditor

참고 *THintWindow*는 힌트를 활성화한 컨트롤의 *BiDiMode*를 선택합니다.

양방향 속성

12-4 페이지의 표 12.1 "BiDi를 지원하는 VCL 객체"에 나열된 객체는 *BiDiMode* 및 *ParentBiDiMode* 속성을 갖습니다. 이러한 속성은 *TApplication*의 *BiDiKeyboard* 및 *NonBiDiKeyboard* 속성과 함께 양방향 지역화를 지원합니다.

참고 크로스 플랫폼 프로그래밍의 CLX에서는 양방향 속성을 사용할 수 없습니다.

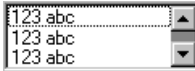
BiDiMode 속성

BiDiMode 속성은 다음 네 가지 상태를 가진 *TBiDiMode*라는 새로운 열거 타입입니다. *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign* 및 *bdRightToLeftReadingOnly*.

bdLeftToRight

*bdLeftToRight*는 왼쪽에서 오른쪽으로 읽는 순서를 사용하여 텍스트를 그리며 정렬과 스크롤 막대는 변경되지 않습니다. 예를 들어, 아랍어나 히브리어와 같이 오른쪽에서 왼쪽으로 텍스트를 입력할 때 커서는 푸시 모드가 되고 텍스트는 오른쪽에서 왼쪽으로 입력됩니다. 영어나 프랑스어와 같은 라틴어 텍스트는 왼쪽에서 오른쪽으로 입력됩니다. *bdLeftToRight*는 기본값입니다.

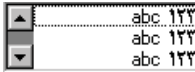
그림 12.1 bdLeftToRight로 설정된 TListBox



bdRightToLeft

*bdRightToLeft*는 오른쪽에서 왼쪽으로 읽는 순서를 사용하여 텍스트를 그리며 정렬이 변경되고 스크롤 막대가 이동합니다. 텍스트는 아랍어나 히브리어와 같은 오른쪽에서 왼쪽으로 읽는 언어에 대해 정상적으로 입력됩니다. 키보드가 라틴어로 변경되면 커서는 푸시 모드가 되고 텍스트는 왼쪽에서 오른쪽으로 입력됩니다.

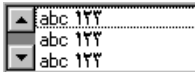
그림 12.2 bdRightToLeft로 설정된 TListBox



bdRightToLeftNoAlign

*bdRightToLeftNoAlign*는 오른쪽에서 왼쪽으로 읽는 순서를 사용하여 텍스트를 그리며 정렬은 변경되지 않고 스크롤 막대는 이동합니다.

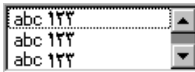
그림 12.3 bdRightToLeftNoAlign로 설정된 TListBox



bdRightToLeftReadingOnly

*bdRightToLeftReadingOnly*는 오른쪽에서 왼쪽으로 읽는 순서를 사용하여 텍스트를 그리며 정렬과 스크롤 막대는 변경되지 않습니다.

그림 12.4 bdRightToLeftReadingOnly로 설정된 TListBox



ParentBiDiMode 속성

*ParentBiDiMode*는 부울 속성입니다. *True*(기본값)일 경우 컨트롤은 컨트롤의 부모를 보고 사용할 *BiDiMode*를 결정합니다. 컨트롤이 *TForm* 객체인 경우 폼은 *Application*의 *BiDiMode* 설정을 사용합니다. 모든 *ParentBiDiMode* 속성이 *True*인 경우 *Application*의 *BiDiMode* 속성을 변경하면 프로젝트의 모든 폼과 컨트롤은 새로운 설정으로 업데이트됩니다.

FlipChildren 메소드

FlipChildren 메소드를 통해 컨테이너 컨트롤의 자식 위치를 바꿀 수 있습니다. 컨테이너 컨트롤은 *TForm*, *TPanel* 및 *TGroupBox*와 같은 다른 컨트롤을 승인할 수 있는 컨트롤입니다. *FlipChildren*은 싱글 부울 매개변수인 *AllLevels*를 가집니다. *False*인 경우 컨테이너 컨트롤의 직계 자식만 바꿉니다. *True*인 경우 컨테이너 컨트롤의 모든 자식이 바뀝니다.

Delphi는 컨트롤의 *Left* 속성과 정렬을 변경하여 컨트롤을 바꿉니다. 컨트롤의 왼쪽이 부모 컨트롤 왼쪽 끝에서 5개 픽셀 떨어져 있다면 바뀌고 난 후 편집 컨트롤의 오른쪽은 부모 컨트롤 오른쪽 끝에서 5개 픽셀 떨어져 있습니다. 편집 컨트롤이 왼쪽 정렬인 경우 *FlipChildren*을 호출하면 컨트롤이 오른쪽으로 정렬됩니다.

디자인 타임 시 컨트롤을 바꾸려면 *Edit|Flip Children*을 선택한 다음 모든 컨트롤을 이동하는지 또는 선택한 컨트롤의 자식만 이동하는지의 여부에 따라 *All* 또는 *Selected*를 선택합니다. 폼에서 컨트롤을 선택하고 마우스 오른쪽 버튼을 클릭한 다음 컨텍스트 메뉴에서 *Flip Children*을 선택하여 컨트롤을 바꿀 수도 있습니다.

참고 편집 컨트롤을 선택하고 *Flip Children|Selected* 명령을 선택해도 아무 것도 실행되지 않습니다. 편집 컨트롤은 컨테이너가 아니기 때문입니다.

추가 메소드

양방향 사용자를 위한 애플리케이션 개발에 유용한 다른 여러 메소드가 있습니다.

메소드	설명
<i>OkToChangeFieldAlignment</i>	데이터베이스 컨트롤과 함께 사용됩니다. 컨트롤의 정렬이 변경될 수 있는지 알려면 선택합니다.
<i>DBUseRightToLeftAlignment</i>	정렬 확인을 위한 데이터베이스 컨트롤에 대한 랩퍼입니다.
<i>ChangeBiDiModeAlignment</i>	전달된 정렬 매개변수를 변경합니다. <i>BiDiMode</i> 설정에 대해 아무 것도 선택되지 않았습니다. 가운데 정렬 컨트롤은 그대로 두고 단지 왼쪽 정렬은 오른쪽 정렬로 오른쪽 정렬은 왼쪽 정렬로 변환합니다.
<i>IsRightToLeft</i>	오른쪽에서 왼쪽 옵션이 선택된 경우 <i>True</i> 를 반환합니다. <i>False</i> 를 반환한 경우 컨트롤은 왼쪽에서 오른쪽 모드입니다.
<i>UseRightToLeftReading</i>	컨트롤이 오른쪽에서 왼쪽 읽기를 사용하는 경우 <i>True</i> 를 반환합니다.
<i>UseRightToLeftAlignment</i>	컨트롤이 오른쪽에서 왼쪽 정렬을 사용하는 경우 <i>True</i> 를 반환합니다. 사용자 지정을 위해 오버라이드할 수 있습니다.
<i>UseRightToLeftScrollBar</i>	컨트롤이 왼쪽 스크롤 막대를 사용하는 경우 <i>True</i> 를 반환합니다.
<i>DrawTextBiDiModeFlags</i>	컨트롤의 <i>BiDiMode</i> 에 대한 올바른 텍스트 그리기 플래그를 반환합니다.

메소드	설명
DrawTextBiDiModeFlagsReadingOnly	플래그를 컨트롤의 읽기 순서로 제한하면서 컨트롤의 <i>BiDiMode</i> 에 대한 올바른 텍스트 그리기 플래그를 반환합니다.
AddBiDiModeExStyle	만드는 중인 컨트롤에 적합한 <i>ExStyle</i> 플래그를 추가합니다.

로케일 특정 기능

특정 로케일의 애플리케이션에 별도의 기능을 추가할 수 있습니다. 특히 아시아 언어 환경에서 애플리케이션이 사용자가 입력한 키스트로크를 문자열로 변환하는 데 사용하는 IME를 제어하도록 할 수 있습니다.

VCL 컴포넌트는 IME 프로그래밍을 지원합니다. 직접적으로 텍스트 입력 기능을 사용하는 대부분의 윈도우 컨트롤에는 컨트롤이 입력 포커스를 가질 때 사용해야 하는 특정 IME를 지정할 수 있는 *ImeName* 속성이 있습니다. 또한 IME가 키보드 입력을 변환하는 방법을 지정하는 *ImeMode* 속성을 제공합니다. *TWinControl*은 사용자가 정의한 클래스에서 IME를 제어하기 위해 사용할 수 있는 여러 protected 메소드를 불러옵니다. 또한 전역 *Screen* 변수는 사용자 시스템에서 사용할 수 있는 IME에 대한 정보를 제공합니다.

또한 VCL 및 CLX에서 사용 가능한 전역 *Screen* 변수는 사용자 시스템에 설치된 키보드 매핑에 대한 정보를 제공합니다. 이 변수를 사용하여 애플리케이션이 실행 중인 환경에 대한 로케일 특정 정보를 얻을 수 있습니다.

사용자 인터페이스 디자인

여러 해외 시장을 위한 애플리케이션을 만들 때는 번역 중 발생하는 변경 내용을 수용할 수 있도록 사용자 인터페이스를 디자인하는 것이 중요합니다.

텍스트

사용자 인터페이스에 나타나는 모든 텍스트가 번역되어야 합니다. 영문 텍스트는 대부분의 경우 번역문보다 짧습니다. 늘어날 텍스트 문자열에 대한 여유 공간을 고려하여 텍스트를 표시하는 사용자 인터페이스의 요소를 디자인하십시오. 쉽게 긴 문자열을 표시할 수 있도록 텍스트를 표시하는 대화 상자, 메뉴, 상태 표시줄 및 기타 사용자 인터페이스 요소를 만드십시오. 표의 문자를 사용하는 언어에 없는 약자는 사용하지 마십시오.

짧은 문자열은 번역할 때 긴 문장보다 더 많이 늘어나는 경향이 있습니다. 표 12.2는 영문 문자열이 번역되었을 때 어느 정도 늘어나는지 대략적으로 보여 줍니다.

표 12.2 문자열 길이 추정

영문 문자열의 길이(문자 수)	예상 증가량
1-5	100%
6-12	80%
13-20	60%

표 12.2 문자열 길이 추정 (계속)

영문 문자열의 길이(문자 수)	예상 증가량
21-30	40%
31-50	20%
50 이상	10%

그래픽 이미지

번역이 필요 없는 이미지를 사용하려면 그래픽 이미지에 번역해야 하는 텍스트가 없어야 합니다. 이미지에 텍스트를 넣어야 하는 경우 이미지의 일부로 텍스트를 포함하는 것 보다는 이미지 위에 투명한 배경과 함께 레이블 객체를 사용하는 것이 좋습니다.

그래픽 이미지를 만들 때 고려해야 할 다른 사항이 있습니다. 특정 문화에만 적합한 이미지는 사용하지 마십시오. 예를 들어, 우체통의 모양은 나라마다 다를 수 있습니다. 우세한 종교는 나라마다 다르므로 이들 국가를 대상으로 하는 애플리케이션에 종교적인 기호는 적절하지 않습니다. 색상조차도 문화마다 다른 상징적인 의미를 가질 수 있습니다.

형식 및 정렬 순서

애플리케이션에서 사용되는 날짜, 시간, 숫자 및 통화 형식은 대상 로케일에 적합하게 지역화되어야 합니다. Windows 형식만 사용하는 경우에는 사용자의 Windows 레지스트리에서 형식을 가져오므로 번역할 필요가 없습니다. 그러나 사용자 고유 형식 문자열을 지정하는 경우, 리소스화된 상수로 선언해서 지역화될 수 있도록 하십시오.

문자열이 정렬되는 순서도 나라마다 다릅니다. 많은 유럽 언어에서는 로케일에 따라 다르게 정렬되는 구분 표시들을 가지고 있습니다. 또한 일부 국가에서는 정렬할 때 2문자 조합을 단일 문자로 취급합니다. 예를 들어, 스페인어에서는 조합 문자 *ch*는 *c*와 *d* 사이의 고유한 단일 문자처럼 정렬됩니다. 간혹 독일어의 *eszett* 같은 단일 문자는 두 개의 개별 문자인 것처럼 정렬됩니다.

키보드 매핑

키 조합으로 단축키를 지정할 때 주의하십시오. US 키보드에 있는 일부 문자는 다른 국제적인 키보드로 쉽게 매핑할 수 없는 경우도 있습니다. 가능한 한 키보드의 모든 키를 사용하여 숫자 키와 단축키를 위한 기능 키를 사용하십시오.

리소스 분리

애플리케이션을 지역화할 때 가장 두드러진 작업은 사용자 인터페이스에 나타나는 문자열을 번역하는 것입니다. 어디서나 코드를 수정하지 않고 번역할 수 있는 애플리케이션을 만들려면 사용자 인터페이스의 문자열을 단일 모듈로 분리해야 합니다. Delphi는 메뉴, 대화 상자 및 비트맵의 리소스를 포함하는 .dfm(CLX 애플리케이션에서는 .xfm) 파일을 자동으로 만듭니다.

이러한 명백한 사용자 인터페이스 요소뿐만 아니라 사용자에게 나타나는 오류 메시지 같은 문자열도 분리해야 합니다. 문자열 리소스는 폼 파일에 포함되지 않습니다. 문자열

리소스에 대해서는 **resourcestring** 키워드를 사용하여 상수로 선언하여 분리해야 합니다. 리소스 문자열 상수에 대한 자세한 내용은 오브젝트 파스칼 랭귀지 안내서를 참조하십시오. 모든 리소스 문자열을 분리된 유닛 하나에 포함하는 것이 가장 좋습니다.

리소스 DLL 작성

리소스를 분리하면 번역 처리가 단순화됩니다. 리소스 분리의 다음 단계는 리소스 DLL을 만드는 것입니다. 리소스 DLL은 모든 리소스 및 특정 프로그램에 대한 리소스만 포함합니다. 리소스 DLL을 사용하여 단순히 리소스 DLL을 교체함으로써 많은 번역을 지원하는 프로그램을 만들 수 있습니다.

리소스 DLL 마법사를 사용하여 프로그램에 대한 리소스 DLL을 만들 수 있습니다. 리소스 DLL 마법사는 열려 있고, 저장되어 있으며, 컴파일된 프로젝트를 필요로 합니다. 그러면 이 마법사는 RC 파일에서 사용되는 문자열 테이블과 프로젝트의 **resourcestring** 문자열을 포함하는 RC 파일을 만들고 관련 폼 및 만들어진 RES 파일을 포함하는 리소스 DLL에 대한 프로젝트만 생성합니다. RES 파일은 새 RC 파일에서 컴파일됩니다.

지원하려는 각 번역에 대한 리소스 DLL을 만들어야 합니다. 각 리소스 DLL에는 대상 로케일에 해당하는 파일 이름 확장자가 있어야 합니다. 처음 두 문자는 대상 언어를 나타내고 세 번째 문자는 로케일의 국가를 나타냅니다. 리소스 DLL 마법사를 사용하는 경우에는 자동으로 처리됩니다. 그렇지 않으면 다음 코드를 사용하여 대상 번역에 대한 로케일 코드를 얻습니다.

```
unit locales;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1:TButton;
    LocaleList:TListBox;
    procedure Button1Click(Sender:TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1:TForm1;

implementation

{$R *.DFM}

function GetLocaleData(ID: LCID; Flag: DWORD):string;
var
  BufSize:Integer;
begin
  BufSize := GetLocaleInfo(ID, Flag, nil, 0);
  SetLength(Result, BufSize);
```

```

    GetLocaleInfo(ID, Flag, PChar(Result), BufSize);
    SetLength(Result, BufSize - 1);
end;

{ Called for each supported locale. }
function LocalesCallback(Name:PChar): Bool; stdcall;
var
    LCID:Integer;
begin
    LCID := StrToInt('$' + Copy(Name, 5, 4));
    Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
    Result := Bool(1);
end;

procedure TForm1.Button1Click(Sender:TObject);
var
    I:Integer;
begin
    with Languages do
        begin
            for I := 0 to Count - 1 do
                begin
                    ListBox1.Items.Add(Name[I]);
                end;
            end;
        end;
    end;
end;

```

리소스 DLL 사용

애플리케이션을 구성하는 실행 파일, DLL 및 패키지는 모든 필요한 리소스를 포함합니다. 하지만 해당 리소스를 지역화된 버전으로 교체하려면 EXE, DLL 또는 BPL 파일과 동일한 이름이 있는 지역화된 리소스 DLL을 사용하여 애플리케이션을 제작해야 합니다.

애플리케이션은 시작할 때 로컬 시스템의 로케일을 확인합니다. 사용 중인 EXE, DLL 또는 BPL 파일과 동일한 이름을 가진 리소스 DLL을 찾을 경우 애플리케이션은 해당 DLL에 대한 확장자를 확인합니다. 리소스 모듈의 확장자가 시스템 로케일의 언어 및 국가와 일치하는 경우 애플리케이션은 실행 파일, DLL 또는 패키지의 리소스 대신 해당 리소스 모듈의 리소스를 사용합니다. 언어 및 국가 모두와 일치하는 리소스 모듈이 없을 경우 애플리케이션은 언어만 일치하는 리소스 모듈을 찾을 것입니다. 언어가 일치하는 리소스 모듈이 없을 경우 애플리케이션은 실행 파일, DLL 또는 패키지로 컴파일한 리소스를 사용할 것입니다.

애플리케이션이 로컬 시스템의 로케일에 일치하는 리소스 모듈이 아닌 다른 리소스 모듈을 사용하게 하려면 Windows 레지스트리에서 로케일 오버라이드 엔트리를 설정할 수 있습니다. HKEY_CURRENT_USER\Software\Borland\Locales 키 아래에 문자열 값으로 애플리케이션의 경로 및 파일 이름을 추가하고 리소스 DLL의 확장자에 데이터 값을 설정합니다. 시작 시 애플리케이션은 시스템 로케일을 시도하기 전에 이 확장자로 리소스 DLL을 검색할 것입니다. 이 레지스트리 엔트리 설정을 사용하여 시스템의 로케일을 변경하지 않고 애플리케이션의 지역화된 버전을 테스트할 수 있습니다.

예를 들어, 다음 프로시저는 Delphi 애플리케이션을 로드할 때 사용할 로케일을 나타내는 레지스트리 키 값을 설정하는 설치 또는 설정 프로그램에서 사용할 수 있습니다.

```

procedure SetLocalOverrides(FileName: string, LocaleOverride:string);
var
    Reg: TRegistry;
begin
    Reg := TRegistry.Create;
    try
        if Reg.OpenKey('Software\Borland\Locales', True) then
            Reg.WriteString(LocaleOverride, FileName);
        finally
            Reg.Free;
    end;

```

애플리케이션 내에서 전역 *FindResourceHInstance* 함수를 사용하여 현재 리소스 모듈의 핸들을 얻습니다. 예를 들면, 다음과 같습니다.

```

    LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery,
        SizeOf(szQuery));

```

적절한 리소스 DLL만 제공하면 애플리케이션이 실행 중인 시스템의 로케일에 맞게 자동으로 적용하는 단일 애플리케이션을 만들 수 있습니다.

리소스 DLL의 동적 전환

애플리케이션 시작 시 리소스 DLL 검색뿐만 아니라 런타임 시 리소스 DLL을 동적으로 전환할 수도 있습니다. 자신의 애플리케이션에 이 기능을 추가하려면 **uses** 문에 *ReInit* 유닛을 포함해야 합니다. (*ReInit*은 Demos 디렉토리에 있는 *Richedit* 예제에 있습니다.) 언어를 전환하려면 새 언어의 LCID를 전달하는 *LoadResourceModule*을 호출한 다음 *ReinitializeForms*를 호출합니다.

예를 들어, 다음 코드는 인터페이스 언어를 프랑스로 전환합니다.

```

const
    FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
    ReinitializeForms;

```

이 방법의 장점은 애플리케이션의 현재 인스턴스와 모든 폼이 사용된다는 점입니다. 레지스트리 설정을 업데이트하고 애플리케이션을 재시작하거나 데이터베이스 서버에 로그인하는 것과 같이 애플리케이션에서 필요한 리소스를 다시 얻을 필요가 없습니다.

리소스 DLL을 전환할 때 새 DLL에 지정된 속성은 해당 폼의 실행 인스턴스에 있는 속성을 덮어 씩니다.

참고 런타임 시 폼 속성에 대한 모든 변경 사항은 잃게 됩니다. 새 DLL이 로드되면 기본값은 재설정되지 않습니다. 지역화로 인한 차이와는 별도로 폼 객체가 시작 상태로 재초기화 된다고 가정하는 코드를 피하십시오.

애플리케이션 지역화

애플리케이션이 일단 국제화되면 배포하려는 다른 해외 시장에 대한 지역화된 버전을 만들 수 있습니다.

리소스 지역화

이상적으로 리소스는 폼 파일 (.dfm 또는 .xfm) 및 리소스 파일을 포함하는 리소스 DLL로 분리되어 있습니다. IDE에서 폼을 열고 관련 속성을 번역할 수 있습니다.

참고 리소스 DLL 프로젝트에서 컴포넌트를 추가하거나 삭제할 수 없습니다. 하지만 속성을 변경하면 런타임 오류가 발생할 수 있으므로 번역이 필요한 해당 속성만 수정하도록 주의하십시오. 이러한 실수를 피하려면 지역화할 수 있는 속성만 표시하도록 Object Inspector를 구성할 수 있습니다. 그렇게 하려면 Object Inspector를 마우스 오른쪽 버튼으로 클릭하고 View 메뉴를 사용하여 필요하지 않은 속성 범주를 필터링합니다.

RC 파일을 열고 관련 문자열을 번역할 수 있습니다. Project Manager에서 RC 파일을 열고 StringTable 편집기를 사용하십시오.

13

애플리케이션 배포

Delphi 애플리케이션 테스트 결과 이상 없이 실행된다면 배포할 수 있습니다. 즉 다른 사람이 애플리케이션을 설치해서 실행하게 할 수 있습니다. 애플리케이션을 다른 컴퓨터에 배포하여 애플리케이션이 완전하게 기능하려면 많은 단계가 필요합니다. 주어진 애플리케이션이 요구하는 단계는 애플리케이션의 종류에 따라 다양합니다. 다음 단원은 다른 종류의 애플리케이션을 배포할 때 고려해야 할 사항에 대해 설명합니다.

- 일반 애플리케이션 배포
- CLX 애플리케이션 배포
- 데이터베이스 애플리케이션 배포
- 웹 애플리케이션 배포
- 다양한 호스트 환경을 위한 프로그래밍
- 소프트웨어 사용권 요구 사항

참고 이 단원에는 Windows에 애플리케이션 배포에 대한 정보가 들어 있습니다. Linux에 배포하기 위한 크로스 플랫폼 애플리케이션을 작성하는 경우, Kylix 설명서에 있는 배포 정보를 참조해야 합니다.

일반 애플리케이션 배포

애플리케이션에는 실행 파일 외에 DLL, 패키지 파일 및 helper 애플리케이션과 같은 많은 지원 파일이 필요합니다. 또한 Windows 레지스트리는 지원 파일 위치 지정부터 단순한 프로그램 설정까지 애플리케이션에 대한 엔트리를 포함해야 합니다. 시스템에 애플리케이션 파일을 복사하고 필요한 레지스트리 설정을 만드는 과정은 InstallShield Express와 같은 설치 프로그램으로 자동화할 수 있습니다. 다음은 거의 모든 종류의 애플리케이션에 공통적인 주요 배포 관련 사항입니다.

- 설치 프로그램 사용
- 애플리케이션 파일 식별

데이터베이스에 액세스하는 Delphi 애플리케이션과 웹에서 실행되는 애플리케이션은 일반 애플리케이션에 적용되는 단계 외에 추가 설치 단계가 필요합니다. 데이터베이스 애플리케이션 설치에 대한 자세한 내용은 13-6 페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오. 웹 애플리케이션 설치에 대한 자세한 내용은 13-10 페이지의 "웹 애플리케이션 배포"를 참조하십시오. ActiveX 컨트롤 설치에 대한 자세한 내용은 38-15 페이지의 "웹에 ActiveX 컨트롤 배포"를 참조하십시오.

설치 프로그램 사용

실행 파일로만 구성된 간단한 Delphi 애플리케이션은 대상 컴퓨터에 설치하기 쉽습니다. 실행 파일을 컴퓨터에 복사하기만 하면 됩니다. 그러나 여러 파일을 포함하는 복잡한 애플리케이션은 확장된 설치 과정이 필요합니다. 이런 애플리케이션은 설치 전용 프로그램이 필요합니다.

Setup 툴킷은 설치 프로그램의 작성 과정을 자동화하며 코드를 작성하지 않아도 되는 경우도 종종 있습니다. Setup 툴킷으로 만든 설치 프로그램은 Delphi 애플리케이션 설치에만 해당하는 다양한 작업, 즉 호스트 시스템에 실행 및 지원 파일 복사, Windows 레지스트리 엔트리 만들기 및 BDE 데이터베이스 애플리케이션에 대한 Borland Database Engine 설치 등을 수행합니다.

InstallShield Express는 Delphi와 함께 제공되는 설치 툴킷입니다. InstallShield Express는 Delphi 및 Borland Database Engine과 함께 사용하도록 인증되어 있습니다. InstallShield Express는 Windows Installer(MSI) 기술을 기반으로 한 것입니다.

InstallShield Express는 Delphi가 설치될 때 자동으로 설치되지 않습니다. 따라서 설치 프로그램을 작성하는 데 사용하려면 직접 설치해야 합니다. InstallShield Express를 설치하려면 Delphi CD에서 설치 프로그램을 실행하십시오. InstallShield Express를 사용하여 설치 프로그램을 작성하는 데 대한 자세한 내용은 InstallShield Express 온라인 도움말을 참조하십시오.

다른 설치 툴킷도 사용할 수 있습니다. 하지만 BDE 데이터베이스 애플리케이션을 배포하는 경우에는 MSI 기술을 기반으로 하고 Borland Database Engine 배포에 인증 받은 툴킷만 사용해야 합니다.

애플리케이션 파일 식별

실행 파일 외에도 많은 파일이 애플리케이션과 함께 배포되어야 합니다.

- 애플리케이션 파일
- 패키지 파일
- Merge 모듈
- ActiveX 컨트롤

애플리케이션 파일

다음과 같은 파일이 애플리케이션과 함께 배포되어야 합니다.

표 13.1 애플리케이션 파일

종류	파일 이름 확장자
프로그램 파일	.exe 및 .dll
패키지 파일	.bpl 및 .dcp
도움말 파일	애플리케이션이 지원하는 .hlp, .cnt 및 .toc(사용된 경우) 또는 기타 도움말 파일
ActiveX 파일	.ocx(때때로 DLL에서 지원)
로컬 테이블 파일	.dbf, .mdx, .dbt, .ndx, .db, .px, .y*, .x*, .mb, .val, .qbe, .gd*

패키지 파일

애플리케이션이 런타임 패키지를 사용하는 경우 패키지 파일은 애플리케이션과 함께 배포되어야 합니다. InstallShield Express는 DLL과 동일한 패키지 파일의 설치, 파일 복사, Windows 레지스트리에서 필요한 엔트리 만들기 등을 처리합니다. 또한 InstallShield Express를 포함하는 MSI 기반 설치 툴과 런타임 패키지 배포용 모듈을 병합할 수 있습니다. 자세한 내용은 다음 단원을 참조하십시오.

Borland가 제공한 Windows\System 디렉토리의 런타임 패키지 파일을 설치할 것을 권장합니다. 이 디렉토리는 공용 디렉토리로 사용되기 때문에 여러 애플리케이션이 파일의 단일 인스턴스에 액세스할 수 있습니다. 사용자가 만든 패키지의 경우 애플리케이션과 같은 디렉토리에 설치하는 것이 좋습니다. .BPL 파일만 배포하면 됩니다.

참고 CLX 애플리케이션에 패키지를 배포하는 경우 vc160.bpl보다는 clx60.bpl을 포함해야 합니다.

다른 개발자에게 패키지를 배포하는 경우 .BPL 및 .DCP 파일을 모두 제공해야 합니다.

Merge 모듈

InstallShield Express 3.0은 Windows Installer(MSI) 기술을 기반으로 합니다. 그런 이유에서 Delphi에 Merge 모듈이 포함되어 있습니다. Merge 모듈은 공유 코드, 파일, 리소스, 레지스트리 엔트리 및 설치 로직을 하나의 복합 파일 형태로 애플리케이션에 전달하는 데 사용할 수 있는 표준 방법을 제공합니다. Merge 모듈은 InstallShield Express와 같은 MSI 기반 설치 툴을 가진 런타임 패키지를 배포하는 데 사용할 수 있습니다.

런타임 라이브러리는 그룹화된 방식 때문에 상호 의존적인 경향이 있습니다. 이 결과로 한 패키지가 설치 프로젝트에 추가될 때 설치 툴은 하나 이상의 다른 패키지에 대한 종속성을 자동으로 추가하거나 보고하게 됩니다. 예를 들어 설치 프로젝트에 VCLInternet Merge 모듈을 추가하는 경우 설치 툴은 VCLDatabase와 StandardVCL 모듈에 대한 종속성도 자동으로 추가하거나 보고합니다.

각 Merge 모듈의 종속성은 아래 표에 나열되어 있습니다. 다양한 설치 틀은 이러한 종속성마다 다르게 반응할 수 있습니다. Windows Installer용 InstallShield는 필요한 모듈을 찾으면 자동으로 추가합니다. 다른 틀은 필요한 모든 모듈이 프로젝트에 포함되어 있지 않으면 단순히 종속성만 보고하거나 빌드 실패를 가져올 수 있습니다.

표 13.2 Merge 모듈과 종속성

Merge 모듈	포함된 BPL 파일	종속성
StandardVCL	vcl60.bpl, rtl60.bpl, vclx60.bpl	종속성 없음
VCLDatabase	vcldb60.bpl, vclmid60.bpl	StandardVCL
VCLLocalDataset	vclld60.bpl	VCLSQL, VCLDatabase, StandardVCL
VCLBDE	vclbde60.bpl, vclldb60.bpl	VCLLocalDataset, VCLSQL, VCLDatabase, StandardVCL
VCLdbExpress	vcldbx60.bpl, vcllds60.bpl	VCLLocalDataset, VCLBDE, VCLSQL, VCLDatabase, StandardVCL
VCLADO	vclado60.bpl, vcllda60.bpl	VCLLocalDataset, VCLSQL, VCLDatabase, StandardVCL
Office2000Components	dcloffice2k.bpl	VCLDatabase, StandardVCL
VCLDataSnap	vclmcn60.bpl	VCLDatabase, StandardVCL
VCLIE	vclie60.bpl	StandardVCL
VCLInternet	inet60.bpl, inetdb60.bpl	VCLDatabase, StandardVCL
InternetBDE	inetdbbde60.bpl	VCLInternet, VCLBDE, VCLLocalDataset, VCLSQL, VCLDatabase, StandardVCL
InternetDBX	inetdbxpress60.bpl	VCLInternet, VCLSQL, VCLDatabase, StandardVCL
VCLXML	vclxml60.bpl	VCLInternet, VCLDatabase, StandardVCL
VCLWebDataSnap	webmid60.bpl	VCLDataSnap, VCLXML, VCLDatabase, VCLInternet, StandardVCL
SiteExpress	site60.bpl, vcljpg60.bpl	VCLWebDataSnap, VCLDataSnap, VCLXML, VCLInternet, VCLDatabase, StandardVCL
VCLInterBase	vclib60.bpl	VCLDatabase, StandardVCL
VCLSOAP	vclsoap60.bpl	VCLXML, VCLInternet, VCLDatabase, StandardVCL
CLXStandard	clx60.bpl	StandardVCL
VCLSample	vclsmp60.bpl	StandardVCL
FastNet	nmfast60.bpl	StandardVCL
DecisionCube	dss60.bpl	TeeChart, VCLDatabase, StandardVCL

표 13.2 Merge 모듈과 종속성 (계속)

Merge 모듈	포함된 BPL 파일	종속성
TeeChart	tee60.bpl, teedb60.bpl, teeqr60.bpl, teeui60.bpl	VCLDatabase, StandardVCL
QuickReport	qrpt60.bpl	VCLDatabase, StandardVCL
InternetDirect	indy60.bpl	StandardVCL
VCLSQL	vclsql60.bpl	VCLDatabase, StandardVCL

ActiveX 컨트롤

Delphi에 함께 제공되는 일부 컴포넌트는 ActiveX 컨트롤입니다. 컴포넌트 랩퍼는 애플리케이션 실행 파일이나 런타임 패키지에 연결되어 있지만 컴포넌트용 .OCX 파일도 애플리케이션에 함께 배포되어야 합니다. 다음과 같은 컴포넌트가 들어 있습니다.

- Chart FX, copyright SoftwareFX Inc.
- VisualSpeller Control, copyright Visual Components, Inc.
- Formula One (spreadsheet), copyright Visual Components, Inc.
- First Impression (VtChart), copyright Visual Components, Inc.
- Graph Custom Control, copyright Bits Per Second Ltd.

직접 작성한 ActiveX 컨트롤은 사용하기 전에 배포할 컴퓨터에 등록해야 합니다. InstallShield Express와 같은 설치 프로그램은 이 등록 프로세스를 자동화합니다. ActiveX 컨트롤을 직접 등록하려면 TRegSvr 데모 애플리케이션 또는 Microsoft 유틸리티 REGSRV32.EXE(일부 Windows 버전에는 포함되어 있지 않음)를 사용하십시오.

또한 ActiveX 컨트롤을 지원하는 DLL은 애플리케이션과 함께 배포되어야 합니다.

Helper 애플리케이션

Helper 애플리케이션은 별도의 프로그램으로 이 프로그램이 없으면 Delphi 애플리케이션의 일부 기능 또는 기능 전체를 사용할 수 없습니다. Helper 애플리케이션은 Boland가 운영 체제와 함께 제공한 것일 수도 있고 협력 업체 제품일 수도 있습니다. helper 애플리케이션의 예로는 InterBase 데이터베이스, 사용자 및 보안을 관리하는 InterBase 유틸리티 프로그램인 Server Manager가 있습니다.

helper 프로그램을 필요로 하는 애플리케이션이라면 가능하면 이 프로그램을 애플리케이션과 함께 배포해야 합니다. helper 프로그램의 배포는 재배포 사용권 계약에 의합니다. 구체적인 내용은 helper에 대한 설명서를 참조하십시오.

DLL 위치

애플리케이션과 동일한 디렉토리에 단일 애플리케이션에서만 사용하는 .dll 파일을 설치할 수 있습니다. 여러 애플리케이션에서 사용할 DLL은 모든 해당 애플리케이션에서 액세스할 수 있는 위치에 설치해야 합니다. 이러한 커뮤니티 DLL 위치에 대한 일반적인 규칙은 DLL을 Windows 또는 Windows\System 디렉토리에 두는 것입니다. 더 좋은 방법은 Borland Database Engine을 설치한 것과 유사한 방법으로 공통적인 .dll 파일에 대한 전용 디렉토리를 만드는 것입니다.

CLX 애플리케이션 배포

Windows와 Linux 모두에 배포할 크로스 플랫폼 애플리케이션을 작성하는 경우 두 플랫폼에서 애플리케이션을 컴파일하고 배포해야 합니다. CLX 애플리케이션을 배포하는 단계는 일반적인 애플리케이션과 동일합니다. 일반적인 애플리케이션 배포에 대한 내용은 13-1 페이지의 "일반 애플리케이션 배포"를 참조하십시오. 데이터베이스 CLX 애플리케이션 설치에 대한 내용은 13-6 페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오.

참고 Windows에서 CLX 애플리케이션을 배포할 때에는 애플리케이션에 qtintf.dll을 포함하여 CLX 런타임을 포함해야 합니다. CLX 애플리케이션에 패키지를 배포하는 경우 vcl60.bpl보다는 clx60.bpl을 포함해야 합니다.

CLX 애플리케이션 작성에 대한 내용은 10장 "크로스 플랫폼 개발을 위한 CLX 사용"을 참조하십시오.

데이터베이스 애플리케이션 배포

데이터베이스에 액세스하는 애플리케이션은 애플리케이션의 실행 파일을 호스트 컴퓨터에 복사하는 것 외에 특별한 설치 고려 사항이 있습니다. 별도의 데이터베이스 엔진, 애플리케이션의 실행 파일에 연결될 수 없는 파일들에 의해 데이터베이스 액세스가 자주 처리됩니다. 미리 생성되지 않은 경우 데이터 파일은 애플리케이션에서 사용할 수 있게 해야 합니다. 애플리케이션을 구성하는 파일은 보통 여러 컴퓨터에 설치되기 때문에 다계층(multi-tier) 데이터베이스 애플리케이션은 설치에 보다 전문적인 처리가 필요합니다.

ADO, BDE, dbExpress 및 InterBase Express와 같은 여러 다른 데이터베이스 기술이 지원되므로 배포 요구 사항은 기술에 따라 달라집니다. 사용하는 기술에 관계 없이 데이터베이스 애플리케이션을 실행할 시스템에 클라이언트측 소프트웨어를 설치해야 합니다. 또한 BDE, ADO 및 dbExpress에서는 드라이버가 데이터베이스의 클라이언트측 소프트웨어와 상호 작용해야 합니다. 그러나 InterBase는 IBX 컴포넌트가 데이터베이스와 직접 통신하므로 드라이버가 필요하지 않습니다.

dbExpress, BDE 및 다계층 애플리케이션 배포 방법에 대한 구체적인 내용은 다음 단원에서 설명합니다.

- dbExpress 데이터베이스 애플리케이션 배포
- BDE 애플리케이션 배포
- 다계층(multi-tier) 데이터베이스 애플리케이션(DataSnap) 배포

TClientDataSet 또는 *TSQLClientDataSet* 또는 데이터셋 프로바이더와 같은 클라이언트 데이터셋을 사용하는 데이터베이스 애플리케이션은 독립 실행 파일 제공 시 정적 연결을 위해 libmidas.dcu 및 crt1.dcu를 포함해야 합니다. 실행 파일과 필요한 DLL을 사용하여 애플리케이션을 패키지화하는 경우 Midas.dll을 포함해야 합니다.

ADO를 사용하는 데이터베이스 애플리케이션을 배포하는 경우 애플리케이션을 실행할 시스템에 MDAC 버전 2.1 이상을 설치해야 합니다. MDAC는 Windows 2000 및 Internet Explorer 버전 5 이상 등의 소프트웨어와 함께 자동으로 설치됩니다. 또한 연결하려는 데이터베이스 서버에 대한 드라이버는 클라이언트에 설치되어야 합니다. 다른 배포 단계는 필요하지 않습니다.

InterBase Express를 사용하는 데이터베이스 애플리케이션을 배포하는 경우 애플리케이션을 실행하려는 시스템에 InterBase 클라이언트를 설치해야 합니다. InterBase는 액세스할 수 있는 디렉토리에 gd32.dll과 interbase.msg를 두어야 합니다. 다른 배포 단계는 필요하지 않습니다. InterBase Express 컴포넌트는 데이터베이스와 직접 통신하므로 드라이버를 추가할 필요가 없습니다. 자세한 내용은 Borland 웹사이트의 Embedded Installation Guide를 참조하십시오.

여기서 설명한 기술 외에도 협력 업체의 데이터베이스 엔진을 사용하여 Delphi 애플리케이션에 대한 데이터베이스 액세스 기능을 제공할 수도 있습니다. 재배포 권한, 설치 및 구성에 대한 데이터베이스 엔진은 설명서를 참조하거나 공급 업체에 문의하십시오.

dbExpress 데이터베이스 애플리케이션 배포

dbExpress는 데이터베이스 정보에 빠른 액세스를 제공하는 드라이버 집합입니다. dbExpress 컴포넌트는 Linux에서도 사용할 수 있으므로 크로스 플랫폼 개발을 지원합니다. dbExpress 컴포넌트 사용에 대한 자세한 내용은 22장 "단방향 데이터셋 사용"을 참조하십시오.

dbExpress 애플리케이션을 독립형 실행 파일로 배포하거나 관련 dbExpress 드라이버 DLL을 포함하는 실행 파일로 배포할 수 있습니다.

dbExpress 애플리케이션을 독립형 실행 파일로 배포하려면 dbExpress 객체 파일은 실행 파일에 정적으로 연결되어야 합니다. lib 디렉토리에 있는 다음과 같은 DCU를 포함하면 정적으로 연결할 수 있습니다.

표 13.3 독립형 실행 파일로 dbExpress 배포

데이터베이스 유닛	포함할 시기
dbExpInt	InterBase 데이터베이스에 연결하는 애플리케이션
dbExpOra	Oracle 데이터베이스에 연결하는 애플리케이션
dbExpDb2	DB2 데이터베이스에 연결하는 애플리케이션
dbExpMy	MySQL 데이터베이스에 연결하는 애플리케이션
Crtl, MidasLib	<i>TSQLClientDataSet</i> 과 같은 클라이언트 데이터셋을 사용하는 dbExpress 실행 파일에서 필요함

독립형 실행 파일을 배포하지 않는 경우 실행 파일과 함께 관련 dbExpress 드라이버와 DataSnap DLL을 배포할 수 있습니다. 다음 표에는 적합한 DLL 및 사용 시기가 나열되어 있습니다.

표 13.4 드라이버 DLL 이 있는 dbExpress 배포

데이터베이스 DLL	배포 시기
dbexpint.dll	InterBase 데이터베이스에 연결하는 애플리케이션
dbexpora.dll	Oracle 데이터베이스에 연결하는 애플리케이션
dbexpdb2.dll.	DB2 데이터베이스에 연결하는 애플리케이션
dbexpmy.dll	MySQL 데이터베이스에 연결하는 애플리케이션
Midas.dll	클라이언트 데이터셋을 사용하는 데이터베이스 애플리케이션에서 필요함

BDE 애플리케이션 배포

Borland Database Engine (BDE)은 데이터베이스와 상호 작용하기 위한 대형 API를 정의합니다. BDE는 모든 데이터 액세스 메커니즘 중에서 가장 광범위한 기능들을 지원하며 최고의 유틸리티를 제공합니다. Paradox나 dBASE 테이블의 데이터에 사용하는 것이 가장 좋습니다.

다양한 데이터베이스 엔진이 애플리케이션에 대한 데이터베이스 액세스를 제공합니다. 애플리케이션은 BDE 또는 협력 업체 데이터베이스 엔진을 사용할 수 있습니다. SQL Link가 제공되어 SQL 데이터베이스 시스템에 대한 원시 액세스가 가능합니다. 일부 버전에는 해당되지 않습니다. 다음 단원은 애플리케이션의 데이터베이스 액세스 요소 설치에 대해 설명합니다.

- Borland Database Engine
- SQL 연결

BDE(Borland Database Engine)

표준 Delphi 데이터 컴포넌트가 데이터 액세스를 하기 위해서는 Borland Database Engine (BDE)이 있어야 하고 액세스할 수 있어야 합니다. BDE 재배포에 대한 특정 권한 및 제한에 대한 내용은 BDEDEPLOY 설명서를 참조하십시오.

Borland는 BDE 설치를 위해 InstallShield Express 또는 다른 인증된 설치 프로그램을 사용할 것을 권장합니다. InstallShield Express는 필수 레지스트리 엔트리를 만들고 해당 애플리케이션에 필요할 수 있는 알리아스를 정의합니다. 인증된 설치 프로그램을 사용한 BDE 파일 및 서브셋 배포는 다음과 같은 이유에서 중요합니다.

- BDE 또는 BDE 서브셋을 잘못 설치하면 BDE를 사용하는 다른 애플리케이션이 실패하게 됩니다. 그러한 애플리케이션에는 Borland 제품뿐 아니라 BDE를 사용하는 많은 협력 업체 프로그램도 포함됩니다.
- Windows 9x 및 Windows NT의 경우, BDE 구성 정보가 16비트 이하의 Windows에서 사용하던 .INI 파일 대신 Windows 레지스트리에 저장되어 있습니다. 설치 및 설치 제거를 위해 올바른 엔트리를 만들고 삭제하는 것은 복잡한 작업입니다.

애플리케이션에서 실제로 필요한 만큼만 BDE를 설치할 수 있습니다. 예를 들어 애플리케이션이 Paradox 테이블만 사용하는 경우 Paradox 테이블에 액세스하는 데 필요한 BDE 부분만 설치하면 됩니다. 이렇게 하면 애플리케이션에 필요한 디스크 공간이 줄어 듭니다. InstallShield Express와 같이 인증된 설치 프로그램은 BDE의 부분 설치를 수행할 수 있습니다. 배포된 애플리케이션에서는 사용하지 않지만 다른 프로그램에서 필요로 하는 BDE 시스템 파일은 반드시 남겨 두어야 합니다.

SQL 연결

SQL Link가 제공하는 드라이버는 BDE를 통해 애플리케이션을 SQL 데이터베이스에 대한 클라이언트 소프트웨어에 연결합니다. SQL Link 재배포에 대한 특정 권한 및 제한에 대한 내용은 DEPLOY 설명서를 참조하십시오. BDE를 사용하는 경우 SQL Link는 InstallShield Express 또는 다른 인증된 설치 프로그램을 사용하여 배포되어야 합니다.

참고 SQL Link는 SQL 데이터베이스 자체가 아니라 클라이언트 소프트웨어에만 BDE를 연결합니다. 이 경우에도 사용하는 SQL 데이터베이스 시스템용 클라이언트 소프트웨어는 설치해야 합니다. 클라이언트 소프트웨어 설치 및 구성에 대한 자세한 내용은 SQL 데이터베이스 시스템에 대한 설명서를 참조하거나 공급 업체에 문의하십시오.

표 13.5는 SQL Link가 다른 SQL 데이터베이스 시스템에 연결하는 데 사용하는 드라이버 및 구성 파일의 이름을 보여 줍니다. 드라이버 및 구성 파일은 SQL Link와 함께 제공되며 Delphi 사용권 계약에 따라 재배포할 수 있습니다.

표 13.5 SQL 데이터베이스 클라이언트 소프트웨어 파일

공급 업체	재배포할 수 있는 파일
Oracle 7	SQLORA32.DLL 및 SQL_ORA.CNF
Oracle8	SQLORA8.DLL 및 SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL 및 SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL 및 SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL 및 SQL_MSS.CNF
Informix 7	SQLINF32.DLL 및 SQL_INF.CNF
Informix 9	SQLINF9.DLL 및 SQL_INF9.CNF
DB/2	SQLDB232.DLL 및 SQL_DB2.CNF
InterBase	SQLINT32.DLL 및 SQL_INT.CNF

InstallShield Express나 다른 인증된 설치 프로그램을 사용하여 SQL Link를 설치합니다. SQL Link의 설치 및 구성에 대한 구체적인 내용은 BDE 주 디렉토리에 기본적으로 설치되는 도움말 파일인 SQLLNK32.HLP를 참조하십시오.

다계층(multi-tier) 데이터베이스 애플리케이션(DataSnap) 배포

DataSnap은 클라이언트 애플리케이션을 애플리케이션 서버의 프로바이더에 연결하여 Delphi 애플리케이션에 다계층 데이터베이스 기능을 제공합니다.

InstallShield Express나 다른 Borland 인증 설치 스크립트 유틸리티를 사용하여 다계층 애플리케이션과 함께 DataSnap을 설치하십시오. 애플리케이션과 함께 재배포하는 데 필요한 파일에 대한 자세한 내용은 Delphi 주 디렉토리에 있는 DEPLOY 설명서를 참조하십시오. 또한 재배포할 수 있는 DataSnap 파일과 재배포 방법에 대한 내용은 REMOTE 설명서를 참조하십시오.

웹 애플리케이션 배포

일부 Delphi 애플리케이션은 서버측 확장 DLL(ISAPI 및 Apache), CGI 애플리케이션 및 ActiveForm의 폼처럼 월드 와이드 웹에 나타낼 수 있도록 설계되었습니다.

웹 애플리케이션 배포 단계는 애플리케이션의 파일을 웹 서버에 배포하는 것만 다를 뿐 일반적인 애플리케이션의 단계와 동일합니다. 일반적인 애플리케이션 설치에 대한 자세한 내용은 13-1 페이지의 "일반 애플리케이션 배포"를 참조합니다. 데이터베이스 웹 애플리케이션 배포에 대한 내용은 13-6 페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오.

웹 애플리케이션 배포에 대한 특별한 주의 사항은 다음과 같습니다.

- BDE 데이터베이스 애플리케이션의 경우 BDE 또는 대체 데이터베이스 엔진은 웹 서버에 애플리케이션 파일과 함께 설치됩니다.
- dbExpress 애플리케이션의 경우 dbExpress DLL은 해당 경로에 포함되어야 합니다. 해당 경로에 포함된 경우 dbExpress 드라이버는 웹 서버에 애플리케이션 파일과 함께 설치됩니다.
- 디렉토리에 대한 보안은 애플리케이션이 필요한 모든 데이터베이스 파일에 액세스할 수 있도록 설정되어야 합니다.
- 애플리케이션을 포함한 디렉토리는 읽기와 실행 속성을 가져야 합니다.
- 애플리케이션은 데이터베이스나 기타 파일에 액세스하기 위해 하드 코딩된 경로를 사용해서는 안 됩니다.
- ActiveX 컨트롤의 위치는 <OBJECT> HTML 태그의 CODEBASE 매개변수로 나타냅니다.

Apache 배포는 다음 단원에서 설명합니다.

Apache 배포

WebBroker는 DLL 및 CGI 애플리케이션에 대해 Apache 버전 1.3.9 및 이후 버전을 지원합니다. Apache는 conf 디렉토리의 파일로 구성됩니다.

Apache DLL을 만드는 경우 httpd.conf라고 하는 Apache 서버 구성 파일에 적합한 지시어를 설정해야 합니다. DLL은 Apache 소프트웨어의 Modules 하위 디렉토리에 물리적으로 위치해야 합니다.

CGI 애플리케이션을 만드는 경우 httpd.conf 파일의 Directory 지시어에 지정된 물리적 디렉토리는 프로그램을 실행할 수 있도록 ExecCGI 옵션을 설정하여 CGI 스크립트를 실행할 수 있어야 합니다. 해당 권한이 바르게 설정되었는지 확인하려면 ScriptAlias 지시어를 사용하거나 Options ExecCGI를 설정해야 합니다.

ScriptAlias 지시어는 서버에 가상 디렉토리를 만들고 CGI 스크립트를 포함하는 것으로 대상 디렉토리를 표시합니다. 예를 들어 httpd.conf 파일에 다음 행을 추가할 수 있습니다.

```
ScriptAlias /cgi-bin "c:\inetpub\cgi-bin"
```

이로 인해 c:\inetpub\cgi-bin\mycgi 스크립트를 실행하여 만족시킬 수 있는 /cgi-bin/mycgi 같은 요청이 발생합니다.

또한 httpd.conf의 Directory 지시어를 사용하여 Options를 All 또는 ExecCGI로 설정할 수 있습니다. Options 지시어는 특정 디렉토리에서 사용할 수 있는 서버의 기능을 제어합니다. Directory 지시어는 지정된 디렉토리와 하위 디렉토리에 적용되는 지시어 집합을 묶는 데 사용됩니다. Directory 지시어의 예는 다음과 같습니다.

```
<Directory <apache-root-dir>\cgi-bin>
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

이 예에서 Options는 cgi-bin 디렉토리에서 CGI 스크립트의 실행을 허용하는 ExecCGI로 설정되어 있습니다.

참고 Apache는 httpd.conf 파일의 User 지시어에 지정된 숫자 내에서 서버에서 지역적으로 실행합니다. 사용자가 애플리케이션에서 필요한 리소스에 액세스할 수 있는 적절한 권한을 가질 수 있게 하십시오.

Apache 소프트웨어 배포에 대한 내용은 Apache 배포에 포함되어 있는 Apache LICENSE 파일에 있습니다. 또한 Apache 웹사이트(www.apache.org)에도 구성 정보가 있습니다.

다양한 호스트 환경을 위한 프로그래밍

다양한 운영 체제 환경의 특성 때문에 사용자 기본 설정이나 구성을 변경하는 요인이 많이 있습니다. 다음 요인은 다른 컴퓨터에 배포되는 애플리케이션에 영향을 줍니다.

- 화면 해상도와 색상 수
- 글꼴
- 운영 체제 버전
- Helper 애플리케이션
- DLL 위치

화면 해상도와 색상 수

컴퓨터의 바탕 화면 크기와 이용 가능한 색상 수를 구성할 수 있는데 설치된 하드웨어에 따라 달라집니다. 이런 속성은 개발 컴퓨터와 배포 컴퓨터가 다를 수도 있습니다.

다른 화면 해상도에 대해 구성된 컴퓨터에서 애플리케이션 외관(창, 객체 및 글꼴 크기)은 다양한 방법으로 처리할 수 있습니다.

- 사용자가 가지는 가장 낮은 해상도(보통, 640x480)로 애플리케이션을 디자인합니다. 호스트 컴퓨터의 화면 표시에 비례하여 동적으로 객체의 크기를 조절하는 어떠한 특별한 작업을 하지 않습니다. 시각적으로 해상도가 더 높게 설정될수록 객체는 더 작게 보입니다.
- 개발 컴퓨터에서 임의의 화면 해상도를 사용하여 디자인하고 런타임에 개발 컴퓨터와 배포 컴퓨터 사이의 화면 해상도 차이에 비례하여 모든 폼과 객체의 크기를 동적으로 조정합니다(화면 해상도 차이 비율).
- 개발 컴퓨터에서 임의의 화면 해상도를 사용하여 디자인하고 런타임에 애플리케이션의 폼 크기만 동적으로 조정합니다. 폼의 비주얼 컨트롤 위치에 따라 사용자가 폼의 모든 컨트롤에 액세스할 수 있도록 폼을 스크롤할 수 있어야 합니다.

동적으로 크기가 조정되지 않는 경우의 고려 사항

애플리케이션을 구성하는 폼과 비주얼 컨트롤이 런타임에 동적으로 크기가 조정되지 않으면 애플리케이션의 요소를 가장 낮은 해상도로 디자인합니다. 그렇지 않으면 개발 컴퓨터보다 낮은 화면 해상도로 구성된 컴퓨터에서 실행하는 애플리케이션의 폼은 화면 경계를 오버랩할 수 있습니다.

예를 들어, 개발 컴퓨터가 1024x768의 화면 해상도로 설정되고 폼이 700픽셀 너비로 디자인된 경우 640x480 화면 해상도로 구성된 컴퓨터의 데스크탑에서는 해당 폼의 일부만 보입니다.

폼과 컨트롤 크기를 동적으로 조정하는 경우의 고려 사항

애플리케이션의 폼과 비주얼 컨트롤을 동적으로 크기 조정하는 경우 가능한 모든 화면 해상도에서 최적의 애플리케이션 모습이 되도록 크기 조정 처리의 모든 측면을 고려합니다. 애플리케이션의 비주얼 요소를 동적으로 크기 조정할 때 고려하는 요인은 다음과 같습니다.

- 폼과 비주얼 컨트롤의 크기를 개발 컴퓨터의 화면 해상도와 애플리케이션이 설치된 컴퓨터의 화면 해상도를 비교하여 계산된 비율로 조정합니다. 개발 컴퓨터의 화면 해상도의 높이 또는 너비를 픽셀 단위의 치수로 표현하는 상수를 사용합니다. *TScreen.Height* 또는 *TScreen.Width* 속성을 사용하여 런타임에 사용자의 컴퓨터에 대해서 동일한 치수를 가져옵니다. 두 컴퓨터의 화면 해상도 사이의 차이 비율을 계산하기 위해 개발 컴퓨터의 값을 사용자 컴퓨터의 값으로 나눕니다.
- 폼에 있는 요소의 크기와 위치를 줄이거나 늘려 애플리케이션(폼과 컨트롤)의 비주얼 요소를 크기 조정합니다. 이 크기 조정은 개발 컴퓨터와 사용자 컴퓨터 사이의 화면 해상도 차이에 비례합니다. *CustomForm.Scaled* 속성을 *True*로 설정하고 *TWinControl.ScaleBy* 메소드(크로스 플랫폼 애플리케이션에 대한 *TWidgetControl.ScaleBy*)를 호출하여 자동으로 폼에 대한 비주얼 컨트롤의 크기를 조정하고 위치를 조정하십시오. *ScaleBy* 메소드가 폼의 높이와 너비는 변경하지는 않습니다. *Height*와 *Width* 속성의 현재 값을 화면 해상도 차이 비율로 곱하여 수동으로 폼의 높이와 너비를 변경합니다.
- 폼의 컨트롤은 *TWinControl.ScaleBy* 메소드(크로스 플랫폼용 애플리케이션의 경우 *TWidgetControl.ScaleBy*)로 자동으로 크기를 조정하는 대신에 순환문에서 각 비주얼 컨트롤을 참조하고 치수와 위치를 설정하여 수동으로 크기 조정할 수 있습니다. 비주얼 컨트롤의 *Height* 및 *Width* 속성 값을 화면 해상도 차이 비율로 곱합니다. *Top* 및 *Left* 속성 값을 같은 비율로 곱하여 화면 해상도 차이에 비례하여 비주얼 컨트롤을 재배치합니다.
- 사용자 컴퓨터의 해상도보다 더 높은 화면 해상도로 구성된 컴퓨터에서 애플리케이션이 디자인된 경우 글꼴 크기는 비주얼 컨트롤의 비례적인 크기 조정 과정에서 줄어들 수 있습니다. 디자인 타임의 글꼴 크기가 너무 작으면 런타임에 크기 조정된 글꼴은 읽을 수가 없습니다. 예를 들어 폼의 기본 글꼴 크기는 8입니다. 개발 컴퓨터의 화면 해상도가 1024x768이고 사용자의 컴퓨터가 640x480인 경우 비주얼 컨트롤 치수는 0.625 ($640 / 1024 = 0.625$)로 감소됩니다. 원래의 글꼴 크기는 8에서 5 ($8 * 0.625 = 5$)로 줄어들 수 없습니다. 애플리케이션의 텍스트는 줄어드는 글꼴 크기로 표시되어 읽을 수 없게 됩니다.
- *TLabel*과 *TEdit*와 같은 일부 비주얼 컨트롤은 컨트롤의 글꼴 크기가 변경될 때 동적으로 크기 조정됩니다. 이것은 폼과 컨트롤이 동적으로 크기가 조정되면 배포된 애플리케이션에 영향을 줍니다. 글꼴 크기 변경으로 인한 컨트롤의 크기 조정이 화면 해상도의 비례적 크기 조정으로 인한 크기의 변경에 추가됩니다. 이 영향은 이들 컨트롤의 *AutoSize* 속성을 *False*로 설정하여 보상할 수 있습니다.
- 캔버스에 직접 그리는 것처럼 명시적인 픽셀 좌표 사용은 피합니다. 대신 비율 비례에 의한 조정으로 개발 컴퓨터와 사용자 컴퓨터 사이의 화면 해상도 차이를 수정합니다. 예를 들어 애플리케이션이 10픽셀 높이와 20픽셀 너비로 캔버스에 직사각형을

그리는 경우 화면 해상도 차이 비율에 10과 20을 곱합니다. 이렇게 하면 다른 화면 해상도에서도 같은 크기로 나타나게 합니다.

다양한 색상 수 지원

색상 표현 능력이 서로 다르게 설정된 모든 배포 컴퓨터에 대한 가장 안전한 방법은 가능한 최소한의 색상 수로 그래픽을 사용하는 것입니다. 이 방법은 전형적인 16 색상 그래픽을 사용해야 하는 컨트롤 glyphs에 대해 적합합니다. 그림을 표시하기 위해서는 다른 해상도와 다른 색상 수를 이용하는 이미지 복사본을 여러 개 제공하거나 애플리케이션에 대해 요구되는 조건으로 최소 해상도와 색상 수를 나타냅니다.

글꼴

Windows와 Linux 운영 체제는 표준 글꼴 집합이 함께 제공됩니다. 다른 컴퓨터에 배포할 애플리케이션을 디자인할 때 일부 컴퓨터는 표준 집합의 글꼴만 가지고 있다는 점을 염두에 두어야 합니다.

애플리케이션에 사용되는 텍스트 컴포넌트는 모든 배포 컴퓨터에서 사용 가능한 글꼴을 모두 사용해야 합니다.

애플리케이션에서 비표준 글꼴을 꼭 사용해야 하는 경우 해당 글꼴을 애플리케이션과 함께 배포해야 합니다. 설치 프로그램이나 애플리케이션은 배포 컴퓨터에 글꼴을 설치해야 합니다. 타사(third-party) 글꼴의 배포는 글꼴 생성자가 부과하는 제약 사항을 지켜야 합니다.

Windows에는 시스템에 없는 글꼴 사용을 대비한 안전한 방법이 있습니다. 가장 비슷하다고 생각되는 기존 글꼴로 대체하는 것입니다. 이 방법은 누락된 글꼴에 대한 오류를 피할 수 있는 반면, 결과적으로 애플리케이션 외관을 손상시킬 수도 있습니다. 따라서 디자인 타임에 이러한 만일의 경우에 대비하는 것이 좋습니다.

Windows 애플리케이션에서 사용할 수 있는 비표준 글꼴을 만들려면 Windows API 함수 *AddFontResource*와 *DeleteFontResource*를 사용하십시오. 비표준 글꼴에 대한 .fot 파일을 애플리케이션에 함께 배포하십시오.

운영 체제 버전

애플리케이션에서 운영 체제 API 또는 운영 체제의 액세스 영역을 사용하는 경우, 다른 운영 체제 버전을 사용하는 시스템에서는 해당 함수, 연산, 영역을 사용하지 못하게 될 수 있습니다.

이러한 가능성을 고려하여 다음과 같은 몇 가지 옵션을 사용할 수 있습니다.

- 애플리케이션이 실행할 수 있는 운영 체제 버전을 애플리케이션의 시스템 요구 사항에 지정합니다. 호환 가능한 운영 체제 버전에서만 애플리케이션을 설치하고 사용하는 것은 사용자의 책임입니다.
- 애플리케이션이 설치된 운영 체제 버전을 확인합니다. 호환되지 않는 운영 체제 버전이 설치되어 있는 경우 설치 프로세스를 중단하거나 설치자에게 문제를 알리십시오.

- 일부 버전에만 해당되는 연산을 실행하기 전에 런타임 시 운영 체제 버전을 확인합니다. 호환되지 않는 운영 체제 버전이 설치되어 있을 경우 프로세스를 중단하고 사용자에게 알리십시오. 다른 방법은 운영 체제 버전에 따라 실행할 수 있는 다른 코드를 제공하는 것입니다. 예를 들어 일부 연산은 Windows 95/98에서 Windows NT/2000과 다르게 수행됩니다. Windows API 함수 *GetVersionEx*를 사용하여 Windows 버전을 결정하십시오.

소프트웨어 사용권 요구 사항

Delphi 애플리케이션에 관련된 일부 파일의 배포는 제한 사항이 있거나 전혀 재배포할 수 없습니다. 다음 문서는 이런 파일의 배포에 관한 법적 규정을 설명합니다.

- DEPLOY
- README
- 사용권 계약서
- 타사(third-party) 제품 설명서

DEPLOY

DEPLOY는 다양한 컴포넌트와 유틸리티, Delphi 애플리케이션의 일부 또는 관련된 다른 제품의 배포에 관련된 몇 가지 법률적인 내용이 들어 있습니다. DEPLOY는 Delphi 주 디렉토리에 설치되어 있는 설명서입니다. 다음과 같은 주제를 다룹니다.

- .exe, .dll 및 .bpl 파일
- 컴포넌트 및 디자인 타임 패키지
- Borland Database Engine (BDE)
- ActiveX 컨트롤
- 샘플 이미지
- SQL 연결

README

README 파일은 컴포넌트, 유틸리티 또는 기타 제품의 재배포 권한에 관련된 정보를 비롯한 Delphi에 대한 최종 정보를 포함합니다. README는 Delphi 주 디렉토리에 설치되어 있는 설명서입니다.

사용권 계약서

인쇄된 문서인 Delphi 사용권 계약서는 Delphi에 관련된 기타 법적 권리와 의무를 규정합니다.

타사(third-party) 제품 설명서

타사 컴포넌트, 유틸리티, helper 애플리케이션, 데이터베이스 엔진 및 기타 제품에 대한 재배포 권리는 제품을 공급하는 협력 업체가 가지고 있습니다. Delphi 애플리케이션에 포함된 제품의 재배포에 관한 정보는 배포하기 전에 제품 설명서를 참조하거나 협력 업체에 문의하십시오.

III

데이터베이스 애플리케이션 개발

"데이터베이스 애플리케이션 개발"에 포함된 장에서는 Delphi 데이터베이스 애플리케이션을 만드는 데 필요한 개념 및 기술을 설명합니다.

참고 데이터베이스 애플리케이션을 개발하려면 Delphi 전문가용 또는 기업용 에디션이 필요합니다. 보다 수준 높은 클라이언트/서버 데이터베이스를 구현하려면 Delphi 기업용 에디션에 있는 기능들이 필요합니다.

14

데이터베이스 애플리케이션 디자인

데이터베이스 애플리케이션은 사용자로 하여금 데이터베이스에 저장되어 있는 정보를 다룰 수 있게 해줍니다. 데이터베이스는 그러한 정보에 대한 구조를 제공하여 각기 다른 애플리케이션 간에 정보를 공유할 수 있게 합니다.

Delphi는 관계형 데이터베이스 애플리케이션을 지원합니다. 관계형 데이터베이스는 행(레코드)과 열(필드)을 포함하는 테이블로 정보를 구성합니다. 이러한 테이블은 관계형 연산 같은 간단한 작업으로 조작될 수 있습니다.

데이터베이스 애플리케이션 디자인 시 데이터 구조를 반드시 이해해야 합니다. 그 구조를 기반으로 사용자 인터페이스를 디자인하여 사용자에게 데이터를 표시할 수 있고 사용자가 새로운 정보를 입력하거나 기존 데이터를 수정하게 할 수 있습니다.

이 장에서는 데이터베이스 애플리케이션 디자인을 위한 일반적인 고려 사항과 사용자 인터페이스 디자인과 관련된 결정 사항을 설명합니다.

데이터베이스 사용

Delphi에는 데이터베이스에 액세스하여 들어 있는 정보를 나타내는 데 사용하는 컴포넌트가 많이 있습니다. 이러한 컴포넌트는 데이터 액세스 메커니즘에 따라 그룹화됩니다.

- 컴포넌트 팔레트의 BDE 페이지에는 Borland Database Engine(BDE)을 사용하는 컴포넌트가 들어 있습니다. BDE는 데이터베이스와 상호 작용하기 위한 대규모의 API를 정의합니다. BDE는 모든 데이터 액세스 메커니즘 중에서 가장 광범위한 기능들을 지원하며 최고의 유틸리티를 제공합니다. Paradox나 dBASE 테이블의 데이터에 사용하는 것이 가장 좋습니다. 그러나 배포하기에 가장 복잡한 메커니즘이기도 합니다. BDE 컴포넌트 사용에 대한 자세한 내용은 20장 "Borland Database Engine 사용"을 참조하십시오.
- 컴포넌트 팔레트의 ADO 페이지에는 OLEDB를 통해 데이터베이스 정보에 액세스하기 위해 ActiveX Data Objects(ADO)를 사용하는 컴포넌트가 들어 있습니다. ADO는 Microsoft 표준입니다. 다른 데이터베이스 서버에 연결하는 데 사용할 수 있는

ADO 드라이버가 많이 있습니다. ADO 기반 컴포넌트를 사용하면 애플리케이션을 ADO 기반 환경(예를 들면, ADO 기반 애플리케이션 서버 사용)으로 통합할 수 있습니다. ADO 컴포넌트 사용에 대한 자세한 내용은 21장 "ADO 컴포넌트 사용"을 참조하십시오.

- 컴포넌트 팔레트의 dbExpress 페이지에 들어 있는 컴포넌트는 dbExpress를 사용하여 데이터베이스 정보에 액세스합니다. dbExpress는 데이터베이스 정보에 대한 가장 빠른 액세스를 제공하는 경량급(lightweight) 드라이버의 집합입니다. 또한 dbExpress 컴포넌트는 Linux에서도 사용 가능하기 때문에 크로스 플랫폼 개발을 지원합니다. 하지만 dbExpress 데이터베이스 컴포넌트는 가장 제한된 범위의 데이터 처리 기능도 지원합니다. dbExpress 컴포넌트 사용에 대한 자세한 내용은 22장 "단방향 데이터셋 사용"을 참조하십시오.
- 컴포넌트 팔레트의 InterBase 페이지에는 별도의 엔진 레이어를 통과하지 않고 InterBase 데이터베이스에 직접 액세스하는 컴포넌트가 들어 있습니다.
- 컴포넌트 팔레트의 Data Access 페이지에는 모든 데이터 액세스 메커니즘을 사용할 수 있는 컴포넌트가 들어 있습니다. 이 페이지에는 디스크에 저장된 데이터를 사용할 수 있거나 이 페이지에 있는 *TDataSetProvider* 컴포넌트를 사용하여 다른 그룹 중 하나의 컴포넌트를 사용할 수 있는 *TClientDataset*이 있습니다. 클라이언트 데이터셋에 대한 자세한 내용은 23장 "클라이언트 데이터셋 사용"을 참조하십시오. *TDataSetProvider*에 대한 자세한 내용은 24장 "프로바이더 컴포넌트 사용"을 참조하십시오.

참고 다른 버전의 Delphi에는 BDE, ADO 또는 dbExpress를 사용하여 데이터베이스 서버에 액세스하기 위한 다른 드라이버가 있습니다.

데이터베이스 애플리케이션을 디자인할 때는 사용할 컴포넌트 집합을 결정해야 합니다. 각 데이터 액세스 메커니즘은 기능 지원, 배포 용이성 및 다른 데이터베이스 서버 지원을 위한 드라이버 가용성에 대해 각기 다른 범위를 가집니다.

데이터 액세스 메커니즘 외에 데이터베이스 서버를 선택해야 합니다. 데이터베이스의 타입이 다르기 때문에 특정 데이터베이스 서버를 정하기 전에 각 타입의 장점과 단점을 고려해야 합니다.

모든 타입의 데이터베이스에는 정보를 저장하는 테이블이 있습니다. 또한 서버 모두는 아니지만 대부분의 서버는 다음과 같은 추가 기능을 지원합니다.

- 데이터베이스 보안
- 트랜잭션
- 참조 무결성, 내장 프로시저 및 트리거

데이터베이스의 타입

관계형 데이터베이스 서버는 정보를 저장하는 방법 및 다중 사용자가 동시에 해당 정보를 액세스할 수 있도록 하는 다양한 방법을 가집니다. Delphi는 다음과 같은 두 가지 타입의 관계형 데이터베이스 서버를 지원합니다.

- **원격 데이터베이스 서버**는 개별적인 시스템에 상주합니다. 경우에 따라 원격 데이터베이스 서버의 데이터는 단일 시스템에 상주하지 않고 여러 서버에 분산됩니다. 원격 데이터베이스 서버는 정보를 저장하는 방법은 다양하지만 클라이언트에 공통적인 인터페이스를 제공합니다. 이 공통적인 인터페이스는 SQL(Structured Query Language)입니다. SQL을 사용하여 액세스하기 때문에 때로 SQL 서버라고도 합니다.(또 다른 이름은 원격 데이터베이스 관리 시스템 또는 RDBMS입니다.) SQL을 구성하는 일반적인 명령에 추가하여 대부분의 원격 데이터베이스 서버는 "조금씩 변경된" 고유한 SQL을 지원합니다. SQL 서버에는 InterBase, Oracle, Sybase, Informix, Microsoft SQL server 및 DB2가 있습니다.
- **로컬 데이터베이스**는 로컬 드라이브 또는 로컬 영역 네트워크에 상주합니다. 로컬 데이터베이스에는 종종 데이터를 액세스하기 위한 전용 API가 있습니다. 여러 사용자가 로컬 데이터베이스를 공유할 때 파일 기반 잠금 메커니즘을 사용합니다. 이 메커니즘 때문에 로컬 데이터베이스는 때로 파일 기반 데이터베이스라고도 합니다. 로컬 데이터베이스에는 Paradox, dBASE, FoxPro 및 Access가 있습니다.

로컬 데이터베이스를 사용하는 애플리케이션은 애플리케이션과 데이터베이스가 하나의 파일 시스템을 공유하기 때문에 **단일 계층 애플리케이션**이라고 합니다. 원격 데이터베이스 서버를 사용하는 애플리케이션은 애플리케이션과 데이터베이스가 독립 시스템(또는 계층)에서 작동하므로 **2계층 애플리케이션** 또는 **다계층 애플리케이션**이라고 합니다.

사용할 데이터베이스 타입 선택은 여러 요인에 따라 다릅니다. 예를 들어, 사용자의 데이터가 기존 데이터베이스에 이미 저장되어 있을 수도 있습니다. 애플리케이션이 사용하는 데이터베이스 테이블을 만드는 경우, 다음 사항을 고려해야 할 것입니다.

- 얼마나 많은 사용자가 이 테이블을 공유할 것인가? 원격 데이터베이스 서버는 동시에 여러 사용자가 액세스할 수 있도록 고안되었습니다. 원격 데이터베이스 서버는 트랜잭션이라는 메커니즘을 통해 다중 사용자를 지원합니다. 또한 Local InterBase 같은 일부 로컬 데이터베이스는 트랜잭션을 지원하지만 대부분의 로컬 데이터베이스는 파일 기반 잠금 메커니즘만 제공하고 클라이언트 데이터셋 파일 같은 일부 로컬 데이터베이스는 다중 사용자를 지원하지 않습니다.
- 테이블은 얼마나 많은 데이터를 보유할 것인가? 원격 데이터베이스 서버는 로컬 데이터베이스보다 많은 데이터를 보유할 수 있습니다. 일부 데이터베이스 서버가 다른 조건(예: 빠른 업데이트)에 대해 최적화되어 있는 반면에 다른 원격 데이터베이스 서버는 대량의 데이터를 보관하도록 고안되어 있습니다.
- 데이터베이스에서 어떤 성능(속도)이 필요한가? 로컬 데이터베이스는 데이터베이스 애플리케이션과 동일한 시스템에 상주하기 때문에 일반적으로 원격 데이터베이스 서버보다 빠릅니다. 다른 원격 데이터베이스 서버는 다른 타입의 연산을 지원하도록 최적화되어 있으므로 원격 데이터베이스 서버를 선택할 때 성능을 고려할 수 있습니다.
- 데이터베이스 관리에 어떤 지원이 가능한가? 로컬 데이터베이스는 원격 데이터베이스 서버보다 더 많은 지원이 필요하지 않습니다. 일반적으로 로컬 데이터베이스는 서버를 따로 설치하거나 비싼 사이트 사용권이 필요하지 않기 때문에 사용하는 데 비용이 덜 듭니다.

데이터베이스 보안

데이터베이스는 종종 기밀을 다루는 정보를 포함합니다. 다른 데이터베이스에서는 그러한 정보를 보호하기 위한 보안 스키마를 제공합니다. Paradox나 dBASE와 같은 일부 데이터베이스에서는 테이블이나 필드 레벨에서의 보안만을 제공합니다. 보호되는 테이블에 사용자가 액세스하려면 암호를 제공해야 합니다. 일단 사용자 인증이 확인되면 승인을 얻은 필드(열)만 볼 수 있습니다.

대부분의 SQL 서버는 데이터베이스 서버를 사용하기 위한 암호와 사용자 이름을 필요로 합니다. 일단 사용자가 데이터베이스에 로그인하면 사용자 이름과 암호는 사용될 수 있는 테이블을 결정합니다. SQL 서버에 암호를 제공하는 것에 대한 내용은 17-4 페이지의 "서버 로그인 제어"를 참조하십시오.

데이터베이스 애플리케이션을 디자인할 때 어떤 종류의 인증이 사용자의 데이터베이스 서버에 필요한지 고려해야 합니다. 종종 애플리케이션은 애플리케이션 자체에 대한 로그인만 필요한 사용자에게 암시적인 데이터베이스 로그인을 표시하지 않도록 디자인됩니다. 사용자가 데이터베이스 암호를 입력하게 만들고 싶지 않다면 암호를 필요로 하지 않는 데이터베이스를 사용하거나 프로그램적으로 서버에 암호와 사용자 이름을 제공해야 합니다. 암호를 프로그램에서 제공할 때 애플리케이션에서 암호를 읽는 것으로부터 보안에 위험이 생기지 않도록 주의해야 합니다.

사용자의 암호를 필요로 하는 경우라면 암호가 필요한 경우를 생각해야 합니다. 로컬 데이터베이스를 사용하고 있지만 나중에 더 큰 SQL 서버로 확대하려면 각각의 테이블을 열 때보다는 SQL 데이터베이스에 로그인할 때 암호를 물을 수 있습니다.

여러 개의 보호되는 시스템 또는 데이터베이스에 로그인하기 때문에 사용자의 애플리케이션이 여러 개의 암호를 필요로 하는 경우라면 보호된 시스템이 필요로 하는 암호들의 테이블을 액세스하는 데 사용되는 하나의 마스터 암호를 제공할 수 있습니다. 그러면 애플리케이션은 사용자가 여러 암호를 제공할 필요 없이 프로그램적으로 암호를 제공할 수 있습니다.

다계층 애플리케이션에서 다른 보안 모델을 함께 사용할 수 있습니다. HTTP, CORBA, 또는 COM+를 사용하여 중간 계층에 대한 액세스를 제어할 수 있고 중간 계층에서 데이터베이스 서버로의 로그인에 대한 모든 상세한 부분을 처리하게 할 수 있습니다.

트랜잭션

트랜잭션은 데이터베이스에 있는 하나 이상의 테이블들이 커밋되기 전에 성공적으로 모두 수행되어야 하는 작업의 그룹입니다. 그룹 내 작업이 하나라도 실패할 경우 모든 작업이 롤백(취소)됩니다.

트랜잭션은 다음 사항을 보증합니다.

- 단일 트랜잭션의 모든 업데이트가 커밋되거나 중지되고 이전 상태로 롤백됩니다. 이것을 **원자성 (atomicity)**이라고 합니다.
- 트랜잭션은 상태 불변 값을 보존하는 시스템 상태의 합당한 변환입니다. 이것을 **일관성 (consistency)**이라고 합니다.

- 병행 트랜잭션은 서로의 부분적인 결과 또는 커밋되지 않은 결과를 알지 못하므로 애플리케이션 상태에서 비일관성을 만들 수 있습니다. 이것을 **분리 (isolation)** 라고 합니다.
- 통신 실패, 프로세스 실패 및 서버 시스템 실패 등의 실패에도 레코드에 커밋된 업데이트는 남아 있습니다. 이것을 **지속성 (durability)** 이라고 합니다.

따라서 트랜잭션은 데이터베이스 명령이나 일련의 명령 중에 발생하는 하드웨어 오류에 대해 보호합니다. 디스크 매체 실패 후 트랜잭션 로그를 사용하여 지속 상태를 복구합니다. 트랜잭션은 또한 SQL 서버에서 다중 사용자 병행 제어의 기반을 형성합니다. 각 사용자가 트랜잭션을 통해서만 데이터베이스와 상호 작용할 때는 한 사용자의 명령이 다른 사용자의 트랜잭션을 방해할 수 없습니다. 그 대신 SQL 서버는 전체로서 성공하거나 실패하는 수신 트랜잭션을 계획합니다.

트랜잭션 지원은 로컬 InterBase에 의해 제공되지만 대부분의 로컬 데이터베이스의 일부는 아닙니다. 또한 BDE 드라이버는 일부 로컬 데이터베이스에 대한 제한된 트랜잭션 지원을 제공합니다. 데이터베이스 트랜잭션 지원은 데이터베이스 연결을 나타내는 컴포넌트에서 제공합니다. 데이터베이스 연결 컴포넌트를 사용하는 트랜잭션 관리에 대한 자세한 내용은 17-6 페이지의 "트랜잭션 관리"를 참조하십시오.

다계층 애플리케이션에서는 데이터베이스 작업 이외의 동작이 포함된 트랜잭션이나 여러 데이터베이스에 걸친 트랜잭션을 만들 수 있습니다. 다계층 애플리케이션에서의 트랜잭션 사용에 대한 자세한 내용은 25-19 페이지의 "다계층 애플리케이션의 트랜잭션 관리"를 참조하십시오.

참조 무결성, 내장 프로시저 및 트리거

모든 관계형 데이터베이스는 애플리케이션이 데이터를 저장하고 조작하도록 하는 특정 기능을 공통적으로 가지고 있습니다. 또한 데이터베이스는 데이터베이스의 테이블 간에 일관적인 관계를 보증하는 데 유용한 데이터베이스 특정 기능을 경우에 따라 제공합니다. 다음과 같은 방법이 있습니다.

- **참조 무결성 (Referential integrity)**. 참조 무결성은 테이블 간의 마스터/디테일 관계가 손상되지 않도록 하는 메커니즘을 제공합니다. 고아 디테일 레코드를 야기할 수 있는 마스터 테이블의 필드를 사용자가 삭제하려는 경우, 참조 무결성 규칙은 삭제를 막거나 자동으로 고아 디테일 레코드를 삭제합니다.
- **내장 프로시저 (Stored procedures)**. 내장 프로시저는 SQL 서버에 이름이 지정되고 저장된 SQL 문의 집합입니다. 내장 프로시저는 보통 서버에서 공통 데이터베이스 관련 작업을 수행하고, 때로 레코드 집합(데이터셋)을 반환합니다.
- **트리거 (Triggers)**. 트리거는 특정 명령에 응답하여 자동으로 실행되는 SQL 문의 집합입니다.

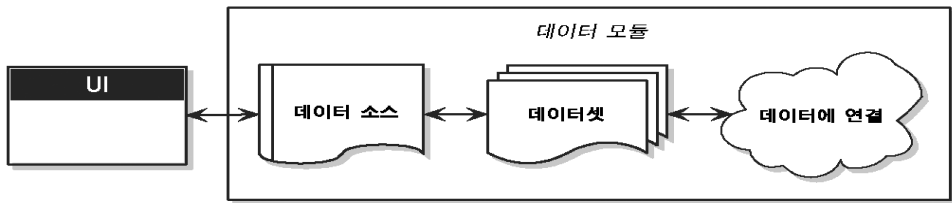
데이터베이스 아키텍처

데이터베이스 애플리케이션은 사용자 인터페이스 요소, 데이터 정보를 나타내는 컴포넌트(데이터셋) 및 이들을 서로 연결하고 데이터베이스 정보 소스에 연결하는 컴포넌트로 구축됩니다. 이러한 요소들을 구성하는 방법이 데이터베이스 애플리케이션의 아키텍처입니다.

일반 구조

데이터베이스 애플리케이션의 컴포넌트를 구성하는 많은 다양한 방법들이 있지만 대부분은 다음 그림 14.1에 설명되어 있는 일반적인 스키마를 따릅니다.

그림 14.1 일반 데이터베이스 아키텍처



사용자 인터페이스 폼

애플리케이션의 나머지 부분과 완전히 분리된 폼에 사용자 인터페이스를 분리하는 것이 좋습니다. 다음과 같은 여러 가지 장점이 있습니다. 데이터베이스 정보 자체를 나타내는 컴포넌트에서 사용자 인터페이스를 분리하여 디자인에 보다 많은 유연성을 부여합니다. 데이터베이스 정보를 관리하는 방법에 대한 변경으로 사용자 인터페이스를 다시 작성할 필요가 없고, 사용자 인터페이스에 대한 변경으로 데이터베이스를 사용하는 애플리케이션의 일부를 변경할 필요는 없습니다. 또한 이러한 타입의 분리를 사용하여 다중 애플리케이션 간에 공유할 수 있는 공통 폼을 개발할 수 있어서 일관성 있는 사용자 인터페이스를 제공합니다. Object Repository에 잘 디자인된 폼과의 연결을 저장하여 개발자들은 새 프로젝트마다 처음부터 시작하지 않고 기존의 기반에서 구축할 수 있습니다. 또한 폼을 공유하여 애플리케이션 인터페이스에 대한 기업 표준을 개발할 수 있습니다. 데이터베이스 애플리케이션용 사용자 인터페이스 생성에 대한 자세한 내용은 14-15 페이지의 "사용자 인터페이스 디자인"을 참조하십시오.

데이터 모듈

사용자 인터페이스를 자체 폼으로 분리한 경우, 데이터 모듈을 사용하여 데이터 정보를 나타내는 컴포넌트(데이터셋)와 이러한 데이터셋을 애플리케이션의 다른 부분에 연결하는 컴포넌트를 수용할 수 있습니다. 사용자 인터페이스 폼처럼 애플리케이션 간에 데이터 모듈을 재사용하고 공유할 수 있도록 Object Repository의 데이터 모듈을 공유할 수 있습니다.

데이터 소스

데이터 모듈의 첫 번째 항목은 데이터 소스입니다. 데이터 소스는 사용자 인터페이스와 데이터베이스의 정보를 나타내는 데이터셋 간에 연결체로서 작용합니다. 폼의 여러 data-aware 컨트롤은 하나의 데이터 소스를 공유할 수 있고 각 컨트롤의 표시는 동기화되어 사용자가 레코드 전체를 스크롤하고 현재 레코드에 대한 필드의 해당 값이 각 컨트롤에 표시됩니다.

데이터셋

데이터베이스 애플리케이션의 핵심은 데이터셋입니다. 이 컴포넌트는 원본으로 사용한 데이터베이스의 레코드 집합을 나타냅니다. 이러한 레코드는 단일 데이터베이스 테이블의 데이터, 테이블의 필드나 레코드의 서브셋 또는 싱글 뷰로 조인된 하나 이상의 테이블 정보일 수 있습니다. 데이터셋을 사용함으로써 애플리케이션 로직은 데이터베이스의 물리적 테이블을 재구성해서 버퍼링됩니다. 원본으로 사용한 데이터베이스가 변경되면 데이터셋 컴포넌트가 포함하고 있는 데이터를 지정하는 방법을 변경해야 할 수도 있지만 애플리케이션의 나머지 부분은 변경하지 않아도 계속 작동됩니다. 데이터셋의 속성 및 메소드에 대한 자세한 내용은 18장 "데이터셋 이해"를 참조하십시오.

데이터 연결

다양한 타입의 데이터셋은 원본으로 사용한 데이터베이스 정보에 연결하기 위한 다양한 메커니즘을 사용합니다. 이러한 다양한 메커니즘은 구축할 수 있는 데이터베이스 애플리케이션의 아키텍처에서 주요 차이점을 차례로 구성합니다. 데이터를 연결하기 위한 네 개의 기본 메커니즘이 있습니다.

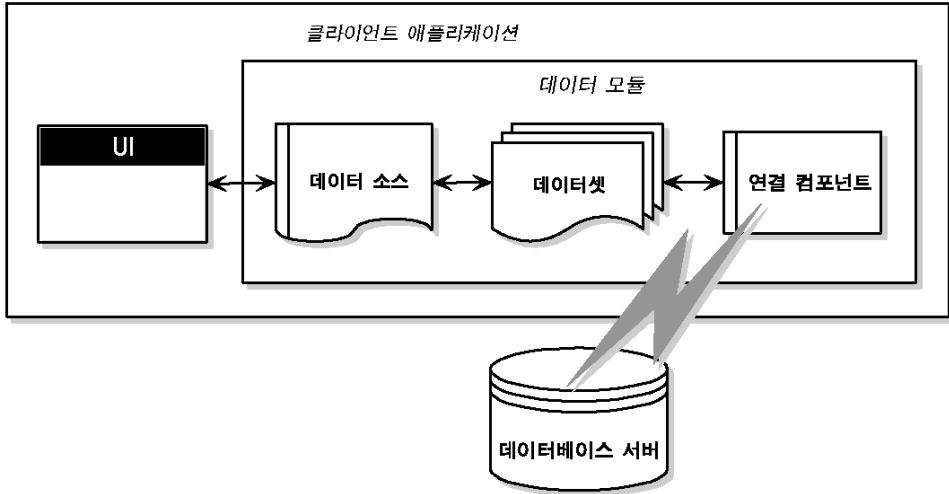
- 데이터베이스 서버에 직접 연결. 대부분의 데이터셋은 *TCustomConnection*의 자손을 사용하여 데이터베이스 서버와의 연결을 나타냅니다.
- 디스크의 전용 파일 사용. 클라이언트 데이터셋은 디스크의 전용 파일을 사용할 수 있는 기능을 지원합니다. 클라이언트 데이터셋 자체가 파일을 읽고 파일에 쓰는 방법을 알기 때문에 전용 파일 사용 시 별도의 연결 컴포넌트가 필요하지 않습니다.
- 다른 데이터셋에 연결. 클라이언트 데이터셋은 다른 데이터셋에 의해 제공되는 데이터를 사용할 수 있습니다. *TDataSetProvider* 컴포넌트는 클라이언트 데이터셋과 소스 데이터셋 간의 매개체 역할을 수행합니다. 이러한 데이터셋 프로바이더는 클라이언트 데이터셋과 동일한 데이터 모듈에 상주하거나 다른 시스템에서 실행하는 애플리케이션 서버의 일부가 될 수 있습니다. 프로바이더가 애플리케이션 서버의 일부인 경우, 애플리케이션 서버에 연결을 나타내려면 *TCustomConnection*의 특정 자손이 필요합니다.
- RDS DataSpace 객체에서 데이터 얻기. ADO 데이터셋은 ADO 기반 애플리케이션 서버를 사용하여 빌드한 다계층 데이터베이스 애플리케이션의 데이터를 마샬링하는 데 *TRDSCONNECTION* 컴포넌트를 사용할 수 있습니다.

경우에 따라 이 메커니즘은 단일 애플리케이션으로 조합될 수 있습니다.

데이터베이스 서버에 직접 연결

대부분의 공통적인 데이터베이스 구조에서 데이터셋은 데이터베이스 서버에 연결하기 위해 연결 컴포넌트를 사용합니다. 그런 다음 데이터셋은 서버에서 데이터를 직접 페치하고 서버에 편집한 내용을 직접 포스트합니다. 그림 14.2에서 그 내용을 보여 줍니다.

그림 14.2 데이터베이스 서버에 직접 연결



각 데이터셋 타입은 단일 데이터 액세스 메커니즘을 나타내는 자신의 연결 컴포넌트 타입을 사용합니다.

- 데이터셋이 *TTable*, *TQuery* 또는 *TStoredProc*와 같은 BDE 데이터셋이면 연결 컴포넌트는 *TDataBase* 객체입니다. *Database* 속성을 설정하여 데이터베이스 컴포넌트에 데이터셋을 연결합니다. BDE 데이터셋을 사용할 때 데이터베이스 컴포넌트를 명시적으로 추가할 필요는 없습니다. 데이터셋의 *DatabaseName* 속성을 설정하면 런타임 시 데이터베이스 컴포넌트가 자동으로 만들어집니다.
- 데이터셋이 *TADODataSet*, *TADOTable*, *TADOQuery* 또는 *TADOStoredProc*와 같은 ADO 데이터셋이면 연결 컴포넌트는 *TADOConnection* 객체입니다. *ADOConnection* 속성을 설정하여 ADO 연결 컴포넌트에 데이터셋을 연결합니다. BDE 데이터셋을 사용할 때와 마찬가지로 연결 컴포넌트를 명시적으로 추가할 필요는 없습니다. 그 대신 데이터셋의 *ConnectionString* 속성을 설정할 수 있습니다.
- 데이터셋이 *TSQLDataSet*, *TSQLTable*, *TSQLQuery* 또는 *TSQLStoredProc*와 같은 dbExpress 데이터셋이면 연결 컴포넌트는 *TSQLConnection* 객체입니다. *SQLConnection* 속성을 설정하여 SQL 연결 컴포넌트에 데이터셋을 연결합니다. dbExpress 데이터셋을 사용할 때는 연결 컴포넌트를 명시적으로 추가해야 합니다. dbExpress 데이터셋과 다른 데이터셋 사이의 또 다른 차이점은 dbExpress 데이터셋은 항상 읽기 전용이며 단방향이라는 것입니다. 이것은 레코드를 순서대로 반복하여 탐색할 수는 있으나 편집을 지원하는 데이터셋 메소드는 사용할 수 없다는 것을 의미합니다.

- 데이터셋이 *TIBDataSet*, *TIBTable*, *TIBQuery* 또는 *TIBStoredProc*와 같은 InterBase Express 데이터셋이면 연결 컴포넌트는 *TIBDatabase* 객체입니다. *Database* 속성을 설정하여 IB 데이터베이스 컴포넌트에 데이터셋을 연결합니다. dbExpress 데이터셋을 사용할 때와 마찬가지로 연결 컴포넌트를 명시적으로 추가해야 합니다.

위에 나열된 컴포넌트 외에도 데이터베이스 연결 컴포넌트와 함께 *TBDEClientDataSet*, *TSQLClientDataSet* 또는 *TIBClientDataSet*과 같은 특수화된 클라이언트 데이터셋을 사용할 수 있습니다. 이러한 클라이언트 데이터셋 중 하나를 사용할 때 *DBConnection* 속성 값으로 알맞은 연결 컴포넌트 타입을 지정하십시오.

각 데이터셋 타입은 다른 연결 컴포넌트를 사용하지만 모두 동일한 많은 작업을 수행하고 동일한 많은 속성, 메소드, 이벤트 등을 표면화합니다. 다양한 데이터베이스 연결 컴포넌트 간의 공통성에 대한 자세한 내용은 17장 "데이터베이스에 연결"을 참조하십시오.

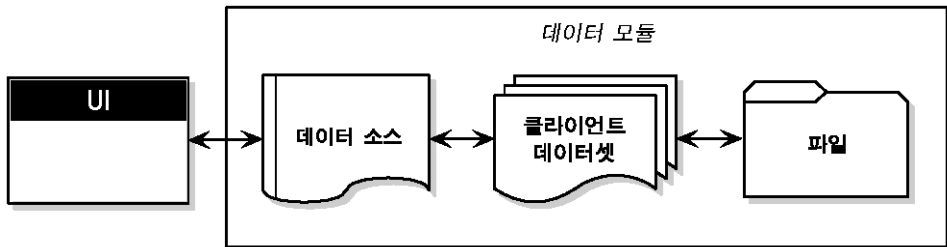
데이터베이스 서버가 로컬 데이터베이스인지 원격 데이터베이스 서버인지에 따라 단일 계층 또는 2계층 애플리케이션을 나타냅니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.

참고 2계층 애플리케이션을 만드는 데 필요한 연결 컴포넌트나 드라이버는 Delphi의 일부 버전에서는 사용할 수 없습니다.

디스크의 전용 파일 사용

작성할 수 있는 데이터베이스 애플리케이션의 가장 단순한 형태에서는 데이터베이스 서버를 사용하지 않습니다. 그 대신 파일에 저장하고 파일의 데이터를 로드하는 클라이언트 데이터셋의 기능을 가진 MyBase를 사용합니다. 다음 그림 14.3에서 이 구조를 그림으로 설명합니다.

그림 14.3 파일 기반 데이터베이스 애플리케이션



이러한 파일에 기반한 방법을 사용할 때 애플리케이션은 클라이언트 데이터셋의 *SaveToFile* 메소드를 사용하여 디스크에 변경 내용을 씁니다. *SaveToFile*은 하나의 매개변수를 취하는데 이 매개변수는 테이블을 포함하는 작성되거나 겹쳐 쓴 파일의 이름입니다. *SaveToFile* 메소드를 사용하여 이전에 작성한 테이블을 읽으려면 *LoadFromFile* 메소드를 사용하십시오. *LoadFromFile* 역시 하나의 매개변수를 취하여 이 매개변수는 테이블을 포함하는 파일의 이름입니다.

항상 동일한 파일을 로드하고 저장하는 경우 *SaveToFile*과 *LoadFromFile* 메소드 대신 *FileName* 속성을 사용할 수 있습니다. *FileName*이 유효한 파일 이름으로 설정된 경우 클라이언트 데이터셋이 열려 있으면 데이터는 자동적으로 파일에서 로드되고 클라이언트 데이터셋이 닫혀 있으면 파일에 저장됩니다.

단일 계층 애플리케이션은 이러한 단순한 파일 기반 구조입니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.

파일 기반 접근 방법의 이점은 간단하다는 것입니다. 설치, 구성, 배포를 할 데이터베이스 서버가 없습니다 (midaslib.dcu에 정적으로 연결하지 않은 경우, 클라이언트 데이터셋은 midas.dll을 필요로 합니다.). 사이트 사용권이나 데이터베이스 관리도 필요하지 않습니다.

또한 Delphi의 일부 버전에서는 임의의 XML 문서와 클라이언트 데이터셋에 의해 사용된 데이터 패킷 간의 변환이 가능합니다. 그러므로 파일 기반 방법은 전용 데이터셋뿐만 아니라 XML 문서를 작업하는 데에도 사용될 수 있습니다. XML 문서와 클라이언트 데이터셋 데이터 패킷 간의 변환에 대한 자세한 내용은 26장 "데이터베이스 애플리케이션에서 XML 사용"을 참조하십시오.

파일 기반 방법은 다중 사용자 지원을 제공하지 않습니다. 데이터셋은 애플리케이션 전용이어야 합니다. 데이터는 디스크의 파일에 저장되고 나중에 로드되기는 하지만 다중 사용자가 데이터 파일을 겹쳐 쓰는 것을 방지하는 기본 제공된 보호 기능은 없습니다.

디스크에 저장된 데이터를 사용하는 클라이언트 데이터셋에 대한 자세한 내용은 23-33 페이지의 "파일 기반 데이터로 클라이언트 데이터셋 사용"을 참조하십시오.

다른 데이터셋에 연결

BDE나 *dbExpress*를 사용하여 데이터베이스 서버에 연결하는 특수화된 클라이언트 데이터셋이 있습니다. 사실 이러한 특수화된 클라이언트 데이터셋은 소스 데이터셋의 데이터를 패키지화하고 데이터베이스 서버에 업데이트를 적용하도록 데이터와 내부 프로바이더 컴포넌트에 액세스하기 위해 내부적으로 다른 데이터셋을 포함하고 있는 합성 컴포넌트입니다. 이러한 합성 컴포넌트는 일부 추가 오버헤드를 필요로 하지만 다음과 같은 이점이 있습니다.

- 클라이언트 데이터셋은 캐시된 업데이트를 사용하는 가장 강력한 방법을 제공합니다. 기본적으로 다른 타입의 데이터셋은 데이터베이스 서버에 편집한 내용을 직접 포스트합니다. 업데이트를 지역적으로 캐시하고 나중에 단일 트랜잭션에서 모든 업데이트를 적용하는 데이터셋을 사용하면 네트워크 트래픽을 줄일 수 있습니다. 업데이트를 캐시하는 클라이언트 데이터셋 사용의 장점에 대한 자세한 내용은 23-16 페이지의 "클라이언트 데이터셋을 사용하여 업데이트 캐시"를 참조하십시오.
- 데이터셋이 읽기 전용인 경우, 클라이언트 데이터셋은 데이터베이스 서버에 편집한 내용을 직접 적용할 수 있습니다. *dbExpress* 사용 시 이는 데이터셋에 있는 데이터를 편집하는 유일한 방법입니다. 이는 또한 *dbExpress*를 사용할 때 데이터를 자유롭게 탐색하는 유일한 방법이기도 합니다. *dbExpress*를 사용하지 않을 때도 일부 쿼리 결과와 모든 내장 프로시저는 읽기 전용입니다. 클라이언트 데이터셋을 사용하면 그러한 데이터를 편집할 수 있습니다.

- 클라이언트 데이터셋은 디스크의 전용 파일을 직접 사용할 수 있기 때문에 클라이언트 데이터셋을 사용하면 파일 기반 모델과 조합하여 유연성 있는 "브리프케이스 (briefcase)" 애플리케이션을 고려할 수 있습니다. 브리프케이스 모델에 대한 자세한 내용은 14-14 페이지의 "방법의 조합"을 참조하십시오.

이러한 특수화된 클라이언트 데이터셋 이외에도 내부 데이터셋 및 데이터셋 프로바이더가 포함되지 않은 일반 클라이언트 데이터셋 (*TClientDataSet*)이 있습니다. *TClientDataSet*에는 기본 제공 데이터베이스 액세스 메커니즘이 없지만 데이터를 페치하고 업데이트를 보내는 외부 데이터셋에 연결할 수 있습니다. 이러한 접근 방법은 다소 복잡하지만 다음과 같은 경우에는 더 많이 사용됩니다.

- 소스 데이터셋과 데이터셋 프로바이더가 외부이기 때문에 데이터를 페치하고 업데이트를 적용하는 방법을 제어합니다. 예를 들면, 프로바이더 컴포넌트는 데이터에 액세스하기 위해 특수화된 클라이언트 데이터셋 사용 시에 사용할 수 없는 많은 이벤트를 표면화합니다.
- 소스 데이터셋이 외부이면 다른 데이터셋을 마스터/디테일 관계에 연결할 수 있습니다. 외부 프로바이더는 이를 중첩 디테일이 있는 단일 데이터셋으로 자동으로 변환합니다. 소스 데이터셋이 내부이면 이런 방식으로 중첩 디테일 집합을 만들 수 없습니다.
- 클라이언트 데이터셋을 외부 데이터셋에 연결하면 여러 계층으로 쉽게 확대할 수 있는 구조가 됩니다. 계층 수가 증가함에 따라 개발 과정이 더 복잡해지고 비용이 많이 들기 때문에 단일 계층이나 2계층 애플리케이션으로 애플리케이션 개발을 시작할 수 있습니다. 데이터의 양, 사용자의 수, 데이터에 액세스하는 다른 애플리케이션의 수 등이 증가함에 따라 나중에 다계층 아키텍처로 확대해야 할 수도 있습니다. 결국 다계층 아키텍처를 사용하게 될 것이라고 생각한다면 외부 소스 데이터셋과 클라이언트 데이터셋을 사용하여 시작하는 것이 더 나을 수 있습니다. 이 방식은 데이터 액세스 및 처리 로직을 중간 계층으로 이동시킬 때 애플리케이션 증가 시 코드를 재사용할 수 있기 때문에 개발 투자비를 줄일 수 있습니다.
- *TClientDataSet*은 모든 소스 데이터셋에 연결할 수 있습니다. 이는 특수화된 해당 클라이언트 데이터셋이 없는 사용자 지정 데이터셋(협력 업체의 컴포넌트)을 사용할 수 있다는 것을 의미합니다. Delphi의 일부 버전에는 다른 데이터셋이 아닌 XML 문서에 클라이언트 데이터셋을 연결하는 특별한 프로바이더 컴포넌트도 들어 있습니다. XML 프로바이더는 데이터셋이 아닌 XML 문서를 사용한다는 것만 제외하면 다른 (소스) 데이터셋에 클라이언트 데이터셋을 연결하는 것과 같은 방법으로 작동합니다. 이러한 XML 프로바이더에 대한 자세한 내용은 26-8 페이지의 "XML 문서를 프로바이더의 소스로 사용"을 참조하십시오.

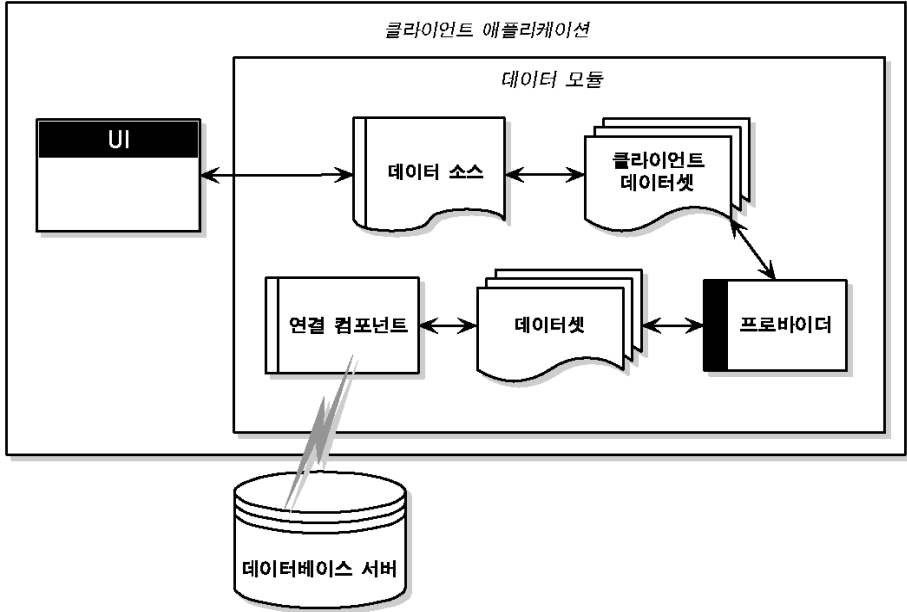
외부 데이터셋에 클라이언트 데이터셋을 연결하는 두 가지 버전의 아키텍처가 있습니다.

- 클라이언트 데이터셋을 동일한 애플리케이션의 다른 데이터셋에 연결
- 다계층 아키텍처 사용

클라이언트 데이터셋을 동일한 애플리케이션의 다른 데이터셋에 연결

프로바이더 컴포넌트를 사용하여 *TClientDataSet*을 다른 (소스) 데이터셋에 연결할 수 있습니다. 프로바이더는 데이터베이스 정보를 클라이언트 데이터셋이 사용할 수 있는 전송 가능한 데이터 패킷으로 패키지화하고 클라이언트 데이터셋이 만든 델타 패킷에 수신된 업데이트를 다시 데이터베이스 서버에 적용합니다. 그림 14.4는 이러한 아키텍처를 보여 줍니다.

그림 14.4 클라이언트 데이터셋과 다른 데이터셋을 조합하는 아키텍처



이 아키텍처는 데이터베이스 서버가 로컬 데이터베이스인지 원격 데이터베이스 서버인지에 따라 단일 계층 또는 2계층 애플리케이션을 나타냅니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.

클라이언트 데이터셋을 프로바이더에 연결하려면 *ProviderName* 속성을 프로바이더 컴포넌트의 이름으로 설정하십시오. 프로바이더는 클라이언트 데이터셋과 동일한 데이터 모듈에 있어야 합니다. 프로바이더를 소스 데이터셋에 연결하려면 *DataSet* 속성을 설정합니다.

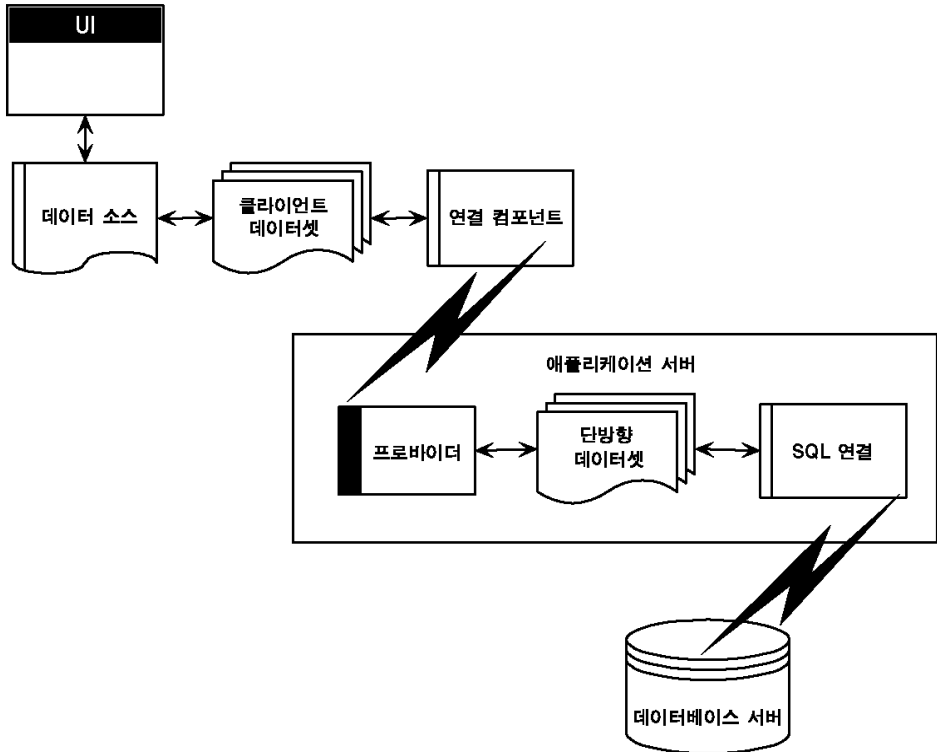
일단 클라이언트 데이터셋이 프로바이더에 연결되고 프로바이더가 소스 데이터셋에 연결되면 이러한 컴포넌트는 소스 데이터셋이 데이터베이스에 연결되었다고 가정하고 데이터베이스 레코드를 페치, 표시, 탐색하는 데 필요한 모든 세부 사항을 자동으로 처리합니다. 데이터베이스에 사용자가 편집한 내용을 적용하기 위해서는 클라이언트 데이터셋의 *ApplyUpdates* 메소드를 호출하기만 하면 됩니다.

프로바이더와 함께 클라이언트 데이터베이스 사용하는 것에 대한 자세한 내용은 23-25 페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

다계층 아키텍처 사용

데이터베이스 정보에 여러 테이블 간의 복잡한 관계가 포함되거나 클라이언트 수가 증가할 때 다계층 애플리케이션을 사용하려고 할 수 있습니다. 다계층 애플리케이션에는 클라이언트 애플리케이션과 데이터베이스 사이에 중간 계층이 있습니다. 그림 14.5에서 이러한 아키텍처를 설명합니다.

그림 14.5 다계층 데이터베이스 아키텍처



앞 그림은 3계층 애플리케이션을 나타냅니다. 데이터베이스 정보를 처리하는 로직은 각 시스템 또는 각 계층에 있습니다. 이 중간 계층은 데이터베이스 상호 작용을 관리하는 로직을 중심에 모으므로 데이터 관계에 대한 중앙화된 제어를 제공합니다. 이를 통해 데이터 로직을 일관되게 하면서 다른 클라이언트 애플리케이션에서 동일한 데이터를 사용할 수 있습니다. 또한 많은 프로세스가 중간 계층으로 분담되기 때문에 더 작은 클라이언트 애플리케이션에서도 사용할 수 있습니다. 이러한 작은 클라이언트 애플리케이션은 설치, 구성 및 유지 관리가 더 용이합니다. 또한 다계층 애플리케이션은 여러 시스템에 데이터 처리 작업을 분산시키므로 성능을 향상시킬 수 있습니다.

다계층 아키텍처는 이전 모델과 매우 유사합니다. 다른 점은 소스 데이터셋과 클라이언트 데이터셋 간에 매개체로서 작동하는 프로바이더와 데이터베이스 서버에 연결하는 소스 데이터셋이 모두 분리 애플리케이션으로 이동했다는 점입니다. 이러한 분리 애플리케이션을 애플리케이션 서버 또는 "원격 데이터 브로커"라고 합니다.

프로바이더가 분리 애플리케이션으로 이동했기 때문에 클라이언트 데이터셋은 더 이상 단순히 *ProviderName* 속성을 설정함으로써 소스 데이터셋에 연결할 수 없습니다. 또한 애플리케이션 서버를 검색하고 연결하는 데 일부 타입의 연결 컴포넌트를 사용해야 합니다.

클라이언트 데이터셋을 애플리케이션 서버에 연결할 수 있는 여러 타입의 연결 컴포넌트가 있습니다. 이 컴포넌트들은 모두 *TCustomRemoteServer*의 자손으로 기본적으로 사용하는 통신 프로토콜(TCP/IP, HTTP, DCOM, SOAP 또는 CORBA)이 다릅니다. *RemoteServer* 속성을 설정하여 연결 컴포넌트에 클라이언트 데이터셋을 연결합니다.

연결 컴포넌트는 애플리케이션 서버에 연결을 설정하고 클라이언트 데이터셋이 사용하는 인터페이스를 반환하여 *ProviderName* 속성으로 지정된 프로바이더를 호출합니다. 클라이언트 데이터셋은 애플리케이션 서버를 호출할 때마다 *ProviderName*의 값을 전달하고 애플리케이션 서버는 프로바이더에 호출을 전달합니다.

애플리케이션 서버와 클라이언트 데이터셋 연결에 대한 자세한 내용은 25 장 "다계층(multi-tiered) 애플리케이션 생성"을 참조하십시오.

방법의 조합

이전 단원에서는 데이터베이스 애플리케이션을 작성할 때 사용할 수 있는 여러 가지 아키텍처에 대해서 설명했습니다. 단일 애플리케이션에서 둘 이상의 사용 가능한 아키텍처를 조합할 수 있습니다. 사실 일부 조합은 매우 강력합니다.

예를 들어, 14-9 페이지의 "디스크의 전용 파일 사용"에서 설명한 디스크 기반 아키텍처를 14-12 페이지의 "클라이언트 데이터셋을 동일한 애플리케이션의 다른 데이터셋에 연결" 또는 14-13 페이지의 "다계층 아키텍처 사용"과 같은 다른 방법으로 조합할 수 있습니다. 모든 모델은 클라이언트 데이터셋을 사용하여 사용자 인터페이스에 있는 데이터를 나타내기 때문에 이러한 조합은 쉽습니다. 이 결과를 브리프케이스 모델(경우에 따라 연결 해제된 모델 또는 모바일 컴퓨팅)이라고 합니다.

이 브리프케이스 모델은 다음과 같은 상황에서 유용합니다. 현장의 회사 데이터베이스는 영업 사원이 해당 현장에서 사용하고 업데이트할 수 있는 고객 연락처 데이터를 포함합니다. 현장에 있을 때 영업 사원은 데이터베이스에서 정보를 다운로드합니다. 나중에 해외 출장 중에도 랩탑으로 데이터를 사용하고 기존 고객 또는 새 고객 위치에 레코드를 업데이트할 수 있습니다. 영업 사원이 현장으로 복귀하면 모든 사람들이 사용할 수 있도록 회사 데이터베이스에 데이터 변경 내용을 업로드합니다.

현장에서 사용할 때 브리프케이스 모델 애플리케이션의 클라이언트 데이터셋은 프로바이더의 데이터를 페치합니다. 그 결과 클라이언트 데이터셋이 데이터에 연결되고 프로바이더를 통해 서버 데이터를 페치하고 다시 서버로 업데이트 내용을 보낼 수 있습니다. 프로바이더에서 연결을 해제하기 전에 클라이언트 데이터셋은 디스크의 파일에 정보의 스냅샷을 저장합니다. 현장이 아닐 때 클라이언트 데이터셋은 파일의 데이터를 로드하고 파일에 변경 내용을 저장합니다. 마침내 현장에 복귀하면 클라이언트 데이터셋은 프로바이더와 다시 연결하여 데이터베이스 서버로 업데이트 내용을 적용하거나 데이터의 스냅샷을 새로 고칠 수 있습니다.

사용자 인터페이스 디자인

컴포넌트 팔레트의 Data Controls 페이지는 데이터베이스 레코드의 필드에서 읽어 온 데이터를 표시하는 data-aware 컨트롤을 제공합니다. 그리고 데이터셋이 허용하는 경우 사용자는 그 데이터를 편집하여 변경 사항을 데이터베이스에 다시 포스트할 수 있습니다. data-aware 컨트롤을 사용하여 데이터베이스 애플리케이션의 사용자 인터페이스(UI)를 빌드할 수 있으므로 정보는 가시성이 있으며 사용자가 액세스할 수 있습니다. data-aware 컨트롤에 대한 자세한 내용은 15장 "데이터 컨트롤 사용"을 참조하십시오.

기본 데이터 컨트롤 이외에도 사용자 인터페이스에 다른 요소를 도입할 수 있습니다.

- 데이터베이스에 포함된 데이터를 분석하는 애플리케이션이 필요할 수도 있습니다. 데이터를 분석하는 애플리케이션은 단지 데이터베이스에 데이터를 표시하는 것 이상을 합니다. 애플리케이션은 사용자가 해당 데이터의 효과를 파악할 수 있도록 유용한 형식으로 정보를 요약하기도 합니다.
- 사용자 인터페이스에 표시된 정보의 하드 카피를 제공하는 리포트를 인쇄할 수 있습니다.
- 웹 브라우저에서 볼 수 있는 사용자 인터페이스를 만들 수 있습니다. 가장 간단한 웹 기반 데이터베이스 애플리케이션은 28-17 페이지의 "응답에서 데이터베이스 정보 사용"에 설명되어 있습니다. 또한 "웹 기반 클라이언트 애플리케이션 작성"에서 설명한 대로 웹 기반 방법을 다계층 아키텍처와 조합할 수 있습니다.

데이터 분석

일부 데이터베이스 애플리케이션은 데이터베이스 정보를 사용자에게 직접 제공하지 않습니다. 그 대신 사용자가 데이터에서 결론을 얻을 수 있도록 데이터베이스의 정보를 분석하고 요약합니다.

컴포넌트 팔레트의 Data Controls 페이지에 있는 *TDBChart* 컴포넌트를 사용하면 그 그래픽 형식으로 데이터베이스 정보를 제공할 수 있으므로 사용자는 데이터베이스 정보의 import를 빠르게 파악할 수 있습니다.

또한 일부 Delphi 버전의 컴포넌트 팔레트에는 Decision Cube 페이지가 있습니다. 여기에는 의사 결정 지원 (Decision Support) 애플리케이션을 빌드할 때 데이터를 분석하고 데이터에 관한 크로스 도표를 작성할 수 있는 여섯 개의 컴포넌트가 있습니다. Decision Cube 컴포넌트 사용에 대한 자세한 내용은 16장 "의사 결정 지원 (Decision Support) 컴포넌트 사용"을 참조하십시오.

다양한 그룹화 조건을 기반으로 데이터 요약을 표시한 컴포넌트를 사용자가 직접 만들려면 클라이언트 데이터셋과 유지 관리되는 집계를 사용할 수 있습니다. 유지 관리되는 집계에 대한 자세한 내용은 23-12 페이지의 "유지 관리되는 집계 (Maintained Aggregates) 속성 사용"을 참조하십시오.

리포트 작성

사용자가 애플리케이션의 데이터셋에서 데이터베이스 정보를 인쇄하도록 할 경우에는 컴포넌트 팔레트의 QReport 페이지에 있는 리포트 컴포넌트를 사용할 수 있습니다. 이러한 컴포넌트를 사용하면 데이터베이스 테이블에 있는 정보를 나타내고 요약해 주는 개요 리포트를 비주얼하게 작성할 수 있습니다. 그룹 머리글 또는 바닥글에 요약을 추가하여 그룹화 기준을 기반으로 데이터를 분석할 수 있습니다.

New Items 대화 상자에서 QuickReport 아이콘을 선택하여 애플리케이션에 대한 리포트를 시작하십시오. 메인 메뉴에서 File|New를 선택한 다음 레이블이 Business인 페이지로 가십시오. QuickReport Wizard 아이콘을 더블 클릭하여 마법사를 실행하십시오.

참고 컴포넌트 사용 방법의 예제를 보려면 Delphi에 들어 있는 QuickReport 데모의 QReport 페이지를 참조하십시오.

15

데이터 컨트롤 사용

컴포넌트 팔레트의 Data Controls 페이지는 데이터베이스 레코드의 필드에서 읽어 온 데이터를 표시하는 data-aware 컨트롤을 제공합니다. 그리고 데이터셋이 허용하는 경우 사용자는 그 데이터를 편집하여 변경 사항을 데이터베이스에 다시 포스트할 수 있습니다. 데이터 컨트롤을 사용자의 데이터베이스 애플리케이션 폼에 두면 데이터베이스 애플리케이션의 사용자 인터페이스(UI)를 구축하여 정보를 볼 수 있고 액세스할 수 있습니다.

사용자 인터페이스에 추가하는 data-aware 컨트롤은 다음과 같은 몇 가지 요소에 따라 달라집니다.

- 표시할 데이터 타입. 일반 텍스트, 서식있는 텍스트와 함께 작동하는 컨트롤, 그래픽용 컨트롤, 멀티미디어 요소 등을 표시하고 편집하도록 고안된 컨트롤들을 선택할 수 있습니다. 다양한 타입의 정보를 표시하는 컨트롤은 15-7 페이지의 "단일 레코드 표시"에서 설명합니다.
- 정보를 구성하는 방법. 화면에 단일 레코드의 정보를 표시하거나 그리드를 사용하여 여러 레코드의 정보를 나열할 수도 있습니다. 데이터를 구성하는 몇 가지 방법은 15-7 페이지의 "데이터 구성 방법 선택"에서 설명합니다.
- 컨트롤에 데이터를 제공하는 데이터셋 타입. 원본으로 사용한 데이터셋의 제한 사항을 반영하는 컨트롤을 사용할 수 있습니다. 예를 들어, 단방향의 데이터셋은 한 번에 하나의 레코드만을 공급할 수 있기 때문에 단방향의 데이터셋에는 그리드를 사용하지 않을 것입니다.
- 사용자들이 데이터셋의 레코드를 탐색하고 데이터를 추가하거나 편집할 수 있도록 하는 방법. 탐색과 편집을 위해 자신의 컨트롤 또는 메커니즘을 추가하거나 데이터 탐색기와 같은 기본 제공 컨트롤을 사용할 수도 있습니다. 데이터 탐색기 사용에 대한 자세한 내용은 15-28 페이지의 "레코드 탐색 및 처리"를 참조하십시오.

참고 보다 복잡한 의사 결정 지원용 data-aware 컨트롤은 16장 "의사 결정 지원 (Decision Support) 컴포넌트 사용"에서 설명합니다.

사용자의 인터페이스에 추가하기 위해 선택하는 data-aware 컨트롤과는 상관 없이 공통적인 기능이 적용됩니다. 이러한 기능은 다음과 같습니다.

공통적인 데이터 컨트롤 기능의 사용

다음은 대부분의 데이터 컨트롤에서 공통적인 작업입니다.

- 데이터셋에 데이터 컨트롤 연결
- 데이터 편집과 업데이트
- 데이터 표시 사용 불가능 및 사용 가능
- 데이터 표시 새로 고침
- 마우스, 키보드 및 타이머 이벤트 사용 가능

일반적으로 데이터 컨트롤을 사용하면 데이터셋의 현재 레코드에 연결된 데이터 필드를 표시하고 편집할 수 있습니다. 표 15.1은 컴포넌트 팔레트의 Data Controls 페이지에 나타나는 데이터 컨트롤을 요약해서 보여 줍니다.

표 15.1 데이터 컨트롤

데이터 컨트롤	설명
<i>TDBGrid</i>	데이터 소스의 정보를 표 서식으로 표시합니다. 그리드의 열은 원본으로 사용한 테이블이나 쿼리의 데이터셋의 열에 해당합니다. 그리드의 행은 레코드에 해당합니다.
<i>TDBNavigator</i>	데이터셋의 데이터 레코드를 탐색합니다. 레코드 업데이트, 레코드 포스트, 레코드 삭제, 레코드 편집 취소 및 데이터 표시 새로 고침.
<i>TDBText</i>	필드의 데이터를 레이블로 표시합니다.
<i>TDBEdit</i>	필드의 데이터를 편집 상자에 표시합니다.
<i>TDBMemo</i>	메모 또는 BLOB 필드의 데이터를 스크롤 가능한 여러 줄 편집 상자에 표시합니다.
<i>TDBImage</i>	데이터 필드의 그래픽을 그래픽 상자에 표시합니다.
<i>TDBListBox</i>	현재 데이터 레코드의 필드를 업데이트할 항목의 목록을 표시합니다.
<i>TDBComboBox</i>	필드를 업데이트할 항목의 목록을 표시하고 표준 data-aware 편집 상자과 같이 직접적인 텍스트 입력을 허용합니다.
<i>TDBCheckBox</i>	부울 필드의 값을 나타내는 체크 박스를 표시합니다.
<i>TDBRadioGroup</i>	필드에 대해 상호 배타적인 옵션을 표시합니다.
<i>TDBLookupListBox</i>	필드 값을 기준으로 다른 데이터셋에서 조회한 항목의 목록을 표시합니다.
<i>TDBLookupComboBox</i>	필드 값을 기준으로 다른 데이터셋에서 조회된 항목의 목록을 표시하고 표준 data-aware 편집 상자과 같은 직접적인 텍스트 입력을 허용합니다.
<i>TDBCtrlGrid</i>	그리드 내에 구성 가능한 반복되는 data-aware 컨트롤 집합을 표시합니다.
<i>TDBRichEdit</i>	필드의 데이터를 편집 상자에 표시합니다.

데이터 컨트롤은 디자인 타임에도 데이터를 인식 (data-aware) 합니다. 애플리케이션을 구축하는 동안 활성 데이터셋과 데이터 컨트롤을 연결하면 컨트롤에서 활성 데이터를 즉시 볼 수 있습니다. Fields Editor를 사용하면 디자인 타임 시 데이터셋을 스크롤하여 애플리케이션을 컴파일하고 실행할 필요 없이 애플리케이션이 데이터를 올바르게 표시하는지 확인할 수 있습니다. Fields Editor에 대한 자세한 내용은 19-4 페이지의 "영구적 필드 생성"을 참조하십시오.

런타임 시 데이터 컨트롤은 데이터를 표시합니다. 그리고 애플리케이션, 컨트롤 및 데이터셋이 모두 허용한다면 컨트롤을 통해 데이터를 편집할 수 있습니다.

데이터셋에 데이터 컨트롤 연결

데이터 컨트롤은 데이터 소스를 사용하여 데이터베이스에 연결됩니다. 데이터 소스 컴포넌트 (*TDataSource*)는 컨트롤과 데이터를 포함하고 있는 데이터셋 사이의 중개자 같은 역할을 합니다. 데이터를 표시하고 처리하려면 각 data-aware 컨트롤을 데이터 소스 컴포넌트에 연결해야 합니다. 이와 비슷하게 데이터를 폼의 data-aware 컨트롤에 표시하고 처리하려면 모든 데이터셋을 데이터 소스 컴포넌트에 연결해야 합니다.

참고 데이터 소스 컴포넌트는 마스터 디테일 관계의 중첩되지 않은 데이터셋 연결을 위해서도 필요합니다.

데이터 컨트롤을 데이터셋에 연결하려면 다음과 같이 합니다.

- 1 데이터셋을 데이터 모듈이나 폼에 두고 속성을 적절히 설정합니다.
- 2 데이터 소스를 동일한 데이터 모듈이나 폼에 둡니다. Object Inspector를 사용하여 *DataSet* 속성을 1 단계에서 둔 데이터셋으로 설정합니다.
- 3 컴포넌트 팔레트의 Data Access 페이지의 데이터 컨트롤을 폼에 둡니다.
- 4 Object Inspector를 사용하여 컨트롤의 *DataSource* 속성을 2 단계에서 둔 데이터 소스 컴포넌트로 설정합니다.
- 5 컨트롤의 *DataField* 속성을 표시할 필드 이름으로 설정하거나 속성의 드롭다운 목록에서 필드 이름을 선택합니다. 이 단계는 *TDBGrid*와 *TDBCtrlGrid* 및 *TDBNavigator*에 적용되지 않습니다. 왜냐하면 이들은 데이터셋의 사용 가능한 모든 필드에 액세스하기 때문입니다.
- 6 데이터셋의 *Active* 속성을 *True*로 설정하여 컨트롤에 데이터를 표시합니다.

런타임 시 연결된 데이터셋 변경

앞의 예제에서는 디자인 타임 시 *DataSet* 속성을 설정하여 데이터 소스를 해당 데이터셋에 연결했습니다. 런타임 시 필요에 따라 데이터 소스 컴포넌트에 대한 데이터셋을 교체할 수 있습니다. 예를 들어, 다음의 코드는 *CustSource* 데이터 소스 컴포넌트에 대한 데이터셋을 *Customers*와 *Orders*라는 데이터셋 컴포넌트로 바꿉니다.

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
  else
    DataSet := Customers;
end;
```

또한 *DataSet* 속성을 다른 폼의 데이터셋으로 설정하여 두 폼에 있는 데이터 컨트롤을 동기화할 수 있습니다. 예를 들면, 다음과 같습니다.

```
procedure TForm2.FormCreate (Sender :TObject);
begin
  DataSource1.DataSet := Form1.Table1;
end;
```

데이터 소스의 사용 가능 및 사용 불가능

데이터 소스에는 데이터셋과의 연결 여부를 결정하는 *Enabled* 속성이 있습니다. *Enabled* 속성이 *True*이면 데이터 소스는 데이터셋에 연결됩니다.

Enabled 속성을 *False*로 설정하여 데이터셋에서 단일 데이터 소스를 일시적으로 연결 해제할 수 있습니다. *Enabled* 속성이 *False*가 되면 데이터 소스 컴포넌트에 첨부된 모든 데이터 컨트롤은 빈 상태가 되고 *Enabled*를 *True*로 설정할 때까지 비활성 상태가 됩니다. 그러나 데이터셋 컴포넌트의 *DisableControls*와 *EnableControls* 메소드는 첨부된 모든 데이터 소스에 영향을 주기 때문에 이들 메소드를 통해 데이터셋에 대한 액세스를 제어하는 것이 좋습니다.

데이터 소스에서 변경 사항에 대한 응답

데이터 소스는 데이터 컨트롤과 데이터셋 사이의 연결을 제공하기 때문에 이 둘 사이에서 발생하는 모든 통신을 조정합니다. 일반적으로 data-aware 컨트롤은 데이터셋의 변경 사항에 자동으로 응답합니다. 그러나 사용자 인터페이스에서 data-aware가 아닌 컨트롤을 사용하는 경우 데이터 소스 컴포넌트의 이벤트를 사용하여 같은 종류의 응답을 수동으로 제공할 수 있습니다.

OnDataChange 이벤트는 필드 편집을 비롯하여 레코드의 데이터가 변경되거나 커서가 새 레코드로 이동할 때 발생합니다. 이 이벤트는 변경이 발생할 때마다 트리거되기 때문에 컨트롤이 데이터셋의 현재 필드 값 반영을 확실히 해야 할 때 유용합니다. 일반적으로 *OnDataChange* 이벤트 핸들러는 필드 데이터를 표시하는 data-aware가 아닌 컨트롤의 값을 새로 고칩니다.

OnUpdateData 이벤트는 현재 레코드의 데이터가 포스트되기 직전에 발생합니다. 예를 들면, *OnUpdateData* 이벤트는 *Post*가 호출되고 난 후, 그러나 데이터가 실제로 원본으로 사용한 데이터베이스 서버나 로컬 캐시에 포스트되기 전에 발생합니다.

StateChange 이벤트는 데이터셋의 상태가 변경될 때 발생합니다. 이 이벤트가 발생할 때 데이터셋의 *State* 속성을 검사하여 현재 상태를 알 수 있습니다.

예를 들어, 다음의 *OnStateChange* 이벤트 핸들러는 현재 상태에 따라 버튼이나 메뉴 항목을 활성화하거나 비활성화합니다.

```
procedure Form1.DataSource1.StateChange(Sender:TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
  ;
end;
```

참고 데이터셋 상태에 대한 자세한 내용은 18-3 페이지의 "데이터셋 상태 알아보기"를 참조하십시오.

데이터 편집과 업데이트

탐색기를 제외한 모든 데이터 컨트롤은 데이터베이스 필드의 데이터를 표시합니다. 또한 원본으로 사용한 데이터셋이 허용하는 한, 이들 컨트롤을 사용하여 데이터를 편집하고 업데이트할 수 있습니다.

참고 단방향 데이터셋에서는 사용자가 데이터를 편집하고 업데이트할 수 없습니다.

컨트롤에서 사용자 입력의 편집 사용 가능

데이터를 편집하기 위해서는 데이터셋이 *dsEdit* 상태여야 합니다. 데이터 소스의 *AutoEdit* 속성이 기본값 *True*인 경우, 데이터 컨트롤은 사용자가 데이터를 편집하는 순간 데이터셋을 *dsEdit* 모드로 변경합니다.

*AutoEdit*가 *False*인 경우에는 데이터셋을 편집 모드로 변경하려면 다른 대안 메커니즘을 제공해야 합니다. 대안 메커니즘 중 하나는 *Edit* 버튼과 함께 *TDBNavigator* 컨트롤을 사용하여 데이터셋을 명시적으로 편집 모드로 변경하는 것입니다. *TDBNavigator*에 대한 자세한 내용은 15-28 페이지의 "레코드 탐색 및 처리"를 참조하십시오. 다른 방법으로는 데이터셋을 편집 모드로 변경하고 싶을 때 데이터셋의 *Edit* 메소드를 호출하는 코드를 생성하는 것입니다.

컨트롤에서의 데이터 편집

데이터셋의 *CanModify* 속성이 *True*인 경우 데이터 컨트롤은 편집 사항을 연결된 데이터셋에 포스트만 할 수 있습니다. *CanModify*는 단방향 데이터셋에 대해 항상 *False*입니다. 일부 데이터셋에는 *CanModify*가 *True*인지 지정할 수 있는 *ReadOnly* 속성이 있습니다.

참고 데이터셋이 데이터를 업데이트할 수 있는지의 여부는 원본으로 사용한 데이터베이스 테이블이 업데이트를 허용하는지의 여부에 달려 있습니다.

데이터셋의 *CanModify* 속성이 *True*인 경우라도 컨트롤이 데이터베이스 테이블로 업데이트 사항을 다시 포스트하기 전에 데이터셋을 컨트롤에 연결하는 데이터 소스의 *Enabled* 속성이 *True*여야 합니다. 데이터 소스의 *Enabled* 속성은 컨트롤이 데이터셋의 필드 값을 표시할 수 있는지를 결정하며 그에 따라 사용자가 값을 편집하고 포스트할 수 있는지의 여부도 결정합니다. *Enabled*가 기본값 *True*인 경우, 컨트롤은 필드 값을 표시할 수 있습니다.

마지막으로 사용자가 컨트롤에 표시된 데이터에 편집 사항을 입력할 수 있는지의 여부를 제어할 수 있습니다. 데이터 컨트롤의 *ReadOnly* 속성은 사용자가 컨트롤로 표시된 데이터를 편집할 수 있는지 결정합니다. 기본값 *False*이면 데이터를 사용자가 편집할 수 있습니다. 데이터셋의 *CanModify* 속성이 *False*인 경우, 사용자는 컨트롤의 *ReadOnly* 속성이 *True*인지 확인하려고 할 것입니다. 그렇지 않으면 잘못된 인상을 주어 사용자가 원본으로 사용한 데이터베이스 테이블의 데이터에 영향을 끼칠 수 있습니다.

필드를 수정할 때 *TDBGrid*를 제외한 모든 데이터 컨트롤에서 수정 사항은 컨트롤에서 *Tab*을 누르면 원본으로 사용한 데이터셋으로 복사됩니다. 필드에서 *Tab*을 누르기 전에 *Esc*를 누르면 데이터 컨트롤은 수정 사항을 버리고 필드 값은 수정하기 전의 값으로 되 돌아갑니다.

*TDBGrid*에서 수정 사항은 사용자가 다른 레코드로 이동할 때 포스트됩니다. 다른 레코드로 이동하기 전에 필드의 아무 레코드에서나 *Esc*를 누르면 레코드의 모든 변경 사항을 취소할 수 있습니다.

레코드가 포스트될 때 Delphi는 상태 변경에 대해 데이터셋에 연결된 모든 data-aware 컨트롤을 확인합니다. 수정된 데이터를 포함하는 필드를 업데이트하는 데 문제가 있는 경우, Delphi는 예외 사항을 발생하여 레코드가 수정되지 않습니다.

참고 애플리케이션이 업데이트 사항을 캐시로 저장하는 경우(예를 들어, 클라이언트 데이터셋의 사용), 모든 수정 사항은 내부 캐시에 포스트됩니다. 이러한 수정 사항은 데이터셋의 *ApplyUpdates* 메소드를 호출한 후 원본으로 사용한 데이터베이스 테이블에 적용됩니다.

데이터 표시 사용 불가능 및 사용 가능

애플리케이션이 데이터셋을 반복하거나 검색을 수행할 때 현재 레코드가 변경될 때마다 data-aware 컨트롤에 표시된 값을 일시적으로 새로 고쳐지지 않게 해야 합니다. 새로 고침을 일시적으로 중지하면 반복 또는 검색 속도가 빨라지고 화면이 깜빡이는 현상을 막을 수 있습니다.

*DisableControls*는 데이터셋에 연결된 모든 data-aware 컨트롤의 표시를 사용 가능하게 하는 데이터셋 메소드입니다. 반복이나 검색이 끝나자마자 애플리케이션은 즉시 데이터셋의 *EnableControls* 메소드를 호출하여 컨트롤의 표시를 다시 사용 가능하게 합니다.

일반적으로 반복 프로세스를 시작하기 전에 컨트롤을 사용하지 못하게 합니다. 반복 프로세스 자체는 **try...finally** 문장 내에 있으며 프로세스 동안 예외가 발생하더라도 컨트롤을 다시 사용 가능하게 할 수 있습니다. **finally** 절은 *EnableControls*를 호출합니다. 다음 코드는 *DisableControls*와 *EnableControls*를 사용하는 방법을 보여 줍니다.

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    :
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

데이터 표시 새로 고침

데이터셋의 *Refresh* 메소드는 지역 버퍼를 플러시하고 개방형 데이터셋의 데이터를 다시 페치(fetch)합니다. 이 메소드를 사용하면 다른 애플리케이션이 사용자의 애플리케이션에서 사용되는 데이터에 대한 동시 액세스를 가지기 때문에 원본으로 사용한 데이터가 변경되었다고 생각하는 경우, data-aware 컨트롤의 표시를 업데이트할 수 있습니다. 캐시로 저장된 업데이트 사항을 사용하는 경우, 데이터셋을 새로 고치기 전에 데이터셋이 현재 캐시한 모든 업데이트 사항을 적용해야 합니다.

새로 고침은 가끔 예기치 못한 결과를 초래합니다. 예를 들어, 사용자가 다른 애플리케이션에서 삭제된 레코드를 보고 있는 경우, 사용자의 애플리케이션이 *Refresh*를 호출하는 순간 레코드가 사라집니다. 또한 사용자가 데이터를 페치하고 나서 *Refresh*를 호출하기 전에 다른 사용자가 레코드를 변경하면 데이터의 변경 내용이 나타날 수도 있습니다.

마우스, 키보드 및 타이머 이벤트 사용 가능

데이터 컨트롤의 *Enabled* 속성은 마우스, 키보드 또는 타이머 이벤트에 대해 응답할지의 여부를 결정하고 데이터 소스에 정보를 전달합니다. 이 속성의 기본 설정은 *True*입니다.

마우스, 키보드 또는 타이머 이벤트를 데이터 컨트롤에 접근하지 못하게 하려면 *Enabled* 속성을 *False*로 설정합니다. *Enabled*가 *False*인 경우 컨트롤을 데이터셋에 연결하는 데이터 소스는 데이터 컨트롤에서 정보를 받지 못합니다. 데이터 컨트롤은 데이터를 계속 표시하지만 컨트롤에 표시된 텍스트는 흐리게 표시됩니다.

데이터 구성 방법 선택

데이터베이스 애플리케이션에 대한 사용자 인터페이스를 구축할 경우, 정보 표시 및 해당 정보를 처리하는 컨트롤의 구성 방법을 선택합니다.

먼저 내려야 하는 결정 중 하나는 한 번에 레코드 하나를 표시할지 아니면 여러 레코드를 표시할지 선택하는 것입니다.

레코드를 탐색하고 처리하는 컨트롤을 추가하기를 원할 수도 있습니다. *TDBNavigator* 컨트롤은 수행하고자 하는 많은 기능에 대해 기본 제공 지원을 제공합니다.

단일 레코드 표시

많은 애플리케이션의 경우, 한 번에 데이터의 레코드 하나에 대한 정보를 표시할 수 있습니다. 예를 들어, 주문 입력 애플리케이션은 다른 주문이 현재 기록되었는지 나타내지 않고도 하나의 주문에 대한 정보를 표시할 수도 있습니다. 이 정보는 주문 데이터셋의 단일 레코드로부터 비롯됩니다.

모든 데이터베이스 정보가 동일한 것(앞에서는 동일한 주문)에 관한 것이기 때문에 단일 레코드를 표시하는 애플리케이션이 일반적으로 읽고 이해하기 쉽습니다. 이러한 사용자

인터페이스의 data-aware 컨트롤은 데이터베이스 레코드의 단일 필드를 나타냅니다. 컴포넌트 팔레트의 Data Controls 페이지는 여러 종류의 필드를 나타내는 여러 가지 컨트롤을 제공합니다. 이러한 컨트롤은 일반적으로 컴포넌트 팔레트에서 사용할 수 있는 다른 컨트롤의 data-aware 버전입니다. 예를 들어, *TDBEdit* 컨트롤은 사용자가 텍스트 문자열(string)을 보면서 편집할 수 있는 표준 *TEdit* 컨트롤의 data-aware 버전입니다.

어느 컨트롤을 사용할지는 필드에 포함된 데이터의 타입(텍스트, 서식있는 텍스트, 그래픽, 부울 정보 등)에 따라 달라집니다.

데이터를 레이블로 표시

*TDBText*는 컴포넌트 팔레트의 Standard 페이지에 있는 *TLabel* 컴포넌트와 비슷한 입기 전용 컨트롤입니다. *TDBText* 컨트롤은 사용자가 다른 컨트롤에 입력할 수 있도록 하는 폼으로 표시 전용 데이터를 제공하고자 할 때 유용합니다. 예를 들어, 고객 목록 테이블의 필드에 폼을 생성한다고 가정하십시오. 사용자가 주소, 시, 도 정보를 폼에 입력하면 별도의 테이블에서 우편 번호 필드를 자동으로 결정하기 위해 동적 조회를 사용한다고 간주합니다. 우편 번호 테이블에 연결된 *TDBText* 컴포넌트는 사용자가 입력한 주소와 일치하는 우편 번호 필드를 표시하는 데 사용할 수 있습니다.

*TDBText*는 표시하는 텍스트를 데이터셋의 현재 레코드에 지정된 필드에서 얻습니다. *TDBText*는 데이터셋에서 텍스트를 얻기 때문에 표시하는 텍스트는 동적입니다. 즉, 사용자가 데이터베이스 테이블을 탐색하면 텍스트가 변경됩니다. 따라서 *TLabel*에서와 같이 디자인 타임 시 *TDBText*의 표시 텍스트를 지정할 수는 없습니다.

참고 *TDBText* 컴포넌트를 폼에 둘 때는 *AutoSize* 속성이 기본값 *True*인지를 확인하여 컨트롤이 폭이 변하는 데이터를 표시하기 위해 필요에 따라 자체 크기를 변경하도록 합니다. *AutoSize*가 *False*이고 컨트롤이 너무 작으면 데이터가 잘려서 표시됩니다.

편집 상자의 필드 표시 및 편집

*TDBEdit*는 편집 상자 컴포넌트의 data-aware 버전입니다. *TDBEdit*는 연결된 데이터 필드의 현재 값을 표시하고 표준 편집 상자 기법을 사용하여 편집될 수 있도록 합니다.

예를 들어, *CustomersSource*가 *CustomersTable*이라는 개방형 *TSQLClientDataSet*에 연결되고 활성화된 *TDataSource* 컴포넌트라고 가정하십시오. 그런 다음 폼에 *TDBEdit* 컴포넌트를 두고 속성을 다음과 같이 설정할 수 있습니다.

- *DataSource*: CustomersSource
- *DataField*: CustNo

data-aware 편집 상자 컴포넌트는 디자인 타임과 런타임 시 모두 *CustomersTable* 데이터셋의 *CustNo* 열의 현재 행의 값을 즉시 표시합니다.

메모 컨트롤에 텍스트 표시 및 편집

*TDBMemo*는 긴 텍스트 데이터를 표시할 수 있는 표준 *TMemo* 컴포넌트와 비슷한 *data-aware* 컴포넌트입니다. *TDBMemo*는 여러 줄의 텍스트를 표시할 뿐만 아니라 여러 줄의 텍스트를 입력할 수도 있습니다. *TDBMemo* 컨트롤을 사용하여 큰 텍스트 필드나 BLOB 필드에 포함된 텍스트 데이터를 표시할 수 있습니다.

기본적으로 *TDBMemo*를 사용하면 사용자가 메모 텍스트를 편집할 수 있습니다. 편집을 방지하려면 메모 컨트롤의 *ReadOnly* 속성을 *True*로 설정합니다. Tab을 표시하고 사용자가 메모에 이를 입력하도록 하려면 *WantTabs* 속성을 *True*로 설정합니다. 사용자가 데이터베이스 메모에 입력할 수 있는 문자 수를 제한하려면 *MaxLength* 속성을 사용합니다. *MaxLength*의 기본값은 0입니다. 즉, 운영 체제에서 부과된 것 이외의 문자 제한은 없습니다.

일부 속성은 데이터베이스 메모의 표시 방법과 텍스트 입력 방법에 영향을 줍니다. *ScrollBars* 속성을 가진 메모 스크롤 바를 제공할 수 있습니다. 줄 바꿈을 방지하려면 *WordWrap* 속성을 **False**로 설정합니다. *Alignment* 속성은 컨트롤 내에서 텍스트의 정렬 방법을 결정합니다. 기본값인 *taLeftJustify*, *taCenter* 및 *taRightJustify*에서 선택할 수 있습니다. 텍스트의 글꼴을 변경하려면 *Font* 속성을 사용합니다.

런타임 시 사용자는 데이터베이스 메모 컨트롤에서 텍스트를 잘라내고, 복사하고, 붙여넣을 수 있습니다. *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 사용하여 프로그램으로 동일한 작업을 구현할 수 있습니다.

*TDBMemo*는 많은 양의 데이터를 표시할 수 있기 때문에 런타임 시 표시하는 데 시간이 걸릴 수 있습니다. 데이터 레코드를 스크롤하는 데 걸리는 시간을 줄이기 위해 *TDBMemo*에는 액세스된 데이터를 자동으로 표시할지 여부를 제어하는 *AutoDisplay* 속성이 있습니다. *AutoDisplay*를 **False**로 설정하면 *TDBMemo*는 실제 데이터가 아닌 필드 이름을 표시합니다. 컨트롤 안을 더블 클릭하면 실제 데이터를 볼 수 있습니다.

Rich edit 메모 컨트롤에 텍스트 표시 및 편집

*TDBRichEdit*는 *data-aware* 컴포넌트로서 표준 *TRichEdit* 컴포넌트와 비슷하여 binary large object (BLOB) 필드에 저장된 서식있는 텍스트를 표시할 수 있습니다. *TDBRichEdit*는 여러 줄의 텍스트를 표시할 뿐만 아니라 여러 줄의 텍스트를 입력할 수도 있습니다.

참고 *TDBRichEdit*는 서식있는 텍스트를 입력하고 사용할 수 있는 속성과 메소드를 제공하는 반면, 이러한 서식 옵션을 사용자가 사용할 수 있도록 하는 사용자 인터페이스 컴포넌트를 제공하지 않습니다. 애플리케이션은 서식있는 텍스트 용량을 구체화시키기 위해서 사용자 인터페이스를 구현해야 합니다.

기본적으로 *TDBRichEdit*를 사용하면 사용자가 메모 텍스트를 편집할 수 있습니다. 편집을 방지하려면 메모 컨트롤의 *ReadOnly* 속성을 *True*로 설정합니다. Tab을 표시하고 사용자가 메모에 이를 입력하도록 하려면 *WantTabs* 속성을 *True*로 설정합니다. 사용자가 데이터베이스 메모에 입력할 수 있는 문자 수를 제한하려면 *MaxLength* 속성을 사용합니다. *MaxLength*의 기본값은 0입니다. 즉, 운영 체제에서 부과된 것 이외의 문자 제한은 없습니다.

*TDBRichEdit*는 많은 양의 데이터를 표시할 수 있기 때문에 런타임 시 표시하는 데 시간이 걸릴 수 있습니다. 데이터 레코드를 스크롤하는 데 걸리는 시간을 줄이기 위해 *TDBRichEdit*에는 액세스된 데이터를 자동으로 표시할지 여부를 제어하는 *AutoDisplay* 속성이 있습니다. *AutoDisplay*를 *False*로 설정하면 *TDBRichEdit*는 실제 데이터가 아닌 필드 이름을 표시합니다. 컨트롤 안을 더블 클릭하면 실제 데이터를 볼 수 있습니다.

이미지 컨트롤의 그래픽 필드 표시 및 편집

*TDBImage*는 BLOB 필드에 포함된 그래픽을 표시하는 data-aware 컨트롤입니다.

기본적으로 *TDBImage*를 통해서 사용자는 *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 사용하여 클립보드에서 잘라내기 및 붙여넣기를 함으로써 그래픽 이미지를 편집할 수 있습니다. 대신 이벤트 핸들러에 첨부된 사용자의 편집 메소드를 컨트롤에 제공할 수도 있습니다.

기본적으로 이미지 컨트롤은 너무 큰 이미지를 잘라내어 컨트롤에 들어갈 수 있는 만큼의 그래픽을 표시합니다. *Stretch* 속성을 *True*로 설정하여 이미지 컨트롤의 크기가 변경되면 이미지 컨트롤에 맞도록 그래픽의 크기를 조정할 수 있습니다.

*TDBImage*는 많은 양의 데이터를 표시할 수 있기 때문에 런타임 시 표시하는 데 시간이 걸릴 수 있습니다. 데이터 레코드를 스크롤하는 시간을 줄이기 위해 *TDBImage*에는 액세스된 데이터를 자동으로 표시할지 여부를 제어하는 *AutoDisplay* 속성이 있습니다. *AutoDisplay*를 *False*로 설정하면 *TDBImage*는 실제 데이터가 아닌 필드 이름을 표시합니다. 컨트롤 안을 더블 클릭하면 실제 데이터를 볼 수 있습니다.

리스트 박스와 콤보 박스의 데이터 표시 및 편집

런타임 시 사용자에게 선택할 수 있는 기본 데이터 값 모음을 제공하는 데이터 컨트롤은 4개가 있습니다. 이러한 컨트롤은 표준 리스트 박스와 콤보 박스 컨트롤의 data-aware 버전입니다.

- *TDBListBox*는 사용자가 데이터 필드에 입력하기 위해 선택할 수 있는 항목의 스크롤 가능한 목록을 표시합니다. data-aware 리스트 박스는 현재 레코드의 필드에 대한 기본값을 표시하고 목록의 해당 입력을 강조하여 표시합니다. 현재 행의 필드 값이 목록에 없는 경우 리스트 박스에 어떤 값도 강조하여 표시되지 않습니다. 사용자가 목록 항목을 선택하면 원본으로 사용한 데이터셋에서 해당 필드 값이 바뀝니다.
- *TDBComboBox*는 data-aware 편집 컨트롤 기능과 드롭다운 목록을 조합합니다. 런타임 시 이 컨트롤은 사용자가 이미 정의된 값 모음에서 선택할 수 있는 드롭다운 목록을 표시하고 사용자가 완전히 다른 값을 선택할 수 있도록 합니다.
- *TDBLookupListBox*는 표시 항목의 목록이 다른 데이터셋에서 조회된다는 점을 제외하면 *TDBListBox*처럼 행동합니다.
- *TDBLookupComboBox*는 표시 항목의 목록이 다른 데이터셋에서 조회된다는 점을 제외하면 *TDBComboBox*처럼 행동합니다.

참고 런타임 시 사용자는 증가 검색을 사용하여 리스트 박스 항목을 찾을 수 있습니다. 예를 들어, 컨트롤에 포커스가 있을 때 "ROB"라고 입력하면 리스트 박스에서 "ROB" 글자로 시작하는 첫 번째 항목이 선택됩니다. 추가로 "E"를 입력하면 "Robert Johnson"과 같이 "ROBE"로 시작하는 첫 번째 항목이 선택됩니다. 검색은 대소문자 구분을 구분합니다. *Backspace*와 *Esc*를 누르면 키를 입력하는 사이에 두 번째 멈춤을 두 번 할 때와 같이 현재 검색 문자열(선택은 바뀌지 않음)을 취소합니다.

TDBListBox 및 TDBComboBox 사용

TDBListBox 또는 *TDBComboBox*를 사용할 때는 표시할 항목의 목록을 생성하기 위해 디자인 타임 시 String List 에디터를 사용해야 합니다. String List 에디터를 열려면 Object Inspector의 *Items* 속성에서 생략 버튼을 클릭합니다. 그런 다음 목록에 표시하려는 항목을 입력합니다. 런타임 시 *Items* 속성의 메소드를 사용하면 문자열 목록을 처리할 수 있습니다.

TDBListBox 또는 *TDBComboBox* 컨트롤이 *DataField* 속성을 통해 필드에 연결된 경우, 필드 값은 목록에 선택되어 나타납니다. 현재 값이 목록에 없는 경우 어떤 항목도 선택되어 표시되지 않습니다. 그러나 *TDBComboBox*는 *Items* 목록에 나타나는지의 여부와 상관 없이 편집 상자에 필드에 대한 현재 값을 표시합니다.

*TDBListBox*에서 *Height* 속성은 얼마나 많은 항목들이 리스트 박스에서 동시에 보여질 수 있는지 결정합니다. *IntegralHeight* 속성은 마지막 항목이 표시되는 방법을 제어합니다. *IntegralHeight*가 *False*(기본값)이면 리스트 박스의 하단은 *ItemHeight* 속성에 의해 결정되고 하단 항목은 완전하게 표시되지 않을 수도 있습니다. *IntegralHeight*가 *True*일 경우에는 리스트 박스에서 보이는 하단 항목이 모두 표시됩니다.

*TDBComboBox*의 경우, *Style* 속성은 사용자와 컨트롤과의 상호 작용을 결정합니다. 기본적으로 *Style*은 사용자가 키보드로 값을 입력하거나 드롭다운 목록에서 항목을 선택할 수 있는 것을 의미하는 *csDropDown*입니다. 다음 속성은 *Items* 목록이 런타임 시 표시되는 방법을 결정합니다.

- *Style*은 다음과 같이 컴포넌트의 표시 스타일을 결정합니다.
 - *csDropDown*(기본값): 사용자가 텍스트를 입력할 수 있는 편집 상자와 드롭다운 목록을 표시합니다. 모든 항목은 문자열이고 높이가 동일합니다.
 - *csSimple*: 항상 표시되는 항목의 고정된 크기 목록과 편집 컨트롤을 연결합니다. *Style*을 *csSimple*로 설정할 때 목록이 표시되도록 *Height* 속성을 증가시켜야 합니다.
 - *csDropDownList*: 드롭다운 리스트 박스와 편집 상자를 표시합니다. 그러나 사용자는 런타임에 드롭다운 목록에 없는 값을 입력하거나 변경할 수 없습니다.
 - *csOwnerDrawFixed* 및 *csOwnerDrawVariable*: 항목 목록에서 문자열 이외의 값(예: 비트맵)을 표시할 수 있도록 하거나 목록의 개별 항목에 대해 다른 글꼴을 사용할 수 있도록 합니다.
- *DropDownCount*: 목록에 표시되는 항목의 최대 *Items*의 수가 *DropDownCount* 이 상이면 사용자는 목록을 스크롤할 수 있습니다. *Items*의 수가 *DropDownCount* 이 하이면 목록은 모든 항목들을 표시하기에 충분이 큼니다.

- *ItemHeight*: 스타일이 *csOwnerDrawFixed*인 경우에 각 항목의 높이
- *Sorted: True*이면 *Items* 목록이 알파벳 순서로 표시됩니다.

데이터 표시 및 편집

조회 리스트 박스와 조회 콤보 박스 (*TDBLookupListBox*와 *TDBLookupComboBox*)는 사용자에게 유효한 필드 값을 설정하는 제한된 선택 목록을 제공합니다. 사용자가 목록 항목을 선택하면 원본으로 사용한 데이터셋에서 해당 필드 값이 바뀝니다.

예를 들어, *OrdersTable*에 연결된 필드가 있는 주문 폼을 생각해 보십시오. *OrdersTable*에는 고객 ID에 해당하는 *CustNo* 필드는 있지만 그 외 고객 정보는 *OrdersTable*에 없습니다. 반면, *CustomersTable*은 고객 ID에 해당하는 *CustNo* 필드가 있으며 고객의 회사나 메일 주소와 같은 추가적인 정보도 포함되어 있습니다. 점원이 송장을 생성할 때 주문 폼에서 고객 ID 대신 회사 이름으로 고객을 선택할 수 있다면 편리할 것입니다. *CustomersTable*에 모든 회사 이름을 표시하는 *TDBLookupListBox*를 사용하면 사용자가 목록에서 회사 이름을 선택할 수 있고 주문 폼에서 적절한 *CustNo*를 설정할 수 있습니다.

이런 조회 컨트롤은 다음 두 소스 중 하나로부터 표시 항목 목록을 파생시킵니다.

• 데이터셋에 대해 정의된 조회 필드

조회 필드를 사용하여 리스트 박스 항목을 지정하려면 컨트롤을 연결하는 데이터셋은 이미 조회 필드를 정의해야 합니다. 이 프로세스는 17-10 페이지의 "서버에 명령 전송"에 설명되어 있습니다. 리스트 박스 항목에 대한 조회 필드를 지정하려면 다음과 같이 합니다.

1 리스트 박스의 *DataSource* 속성을 사용할 조회 필드를 포함하는 데이터셋에 대한 데이터 소스로 설정합니다.

2 *DataField* 속성에 대한 드롭다운 목록에서 사용할 조회 필드를 선택합니다.

조회 컨트롤에 연결된 테이블을 활성화할 때 컨트롤은 데이터 필드가 조회 필드라는 것을 인식하고 조회로부터 적절한 값을 표시합니다.

• 보조 데이터 소스, 데이터 필드 및 키

데이터셋에 대한 조회 필드를 정의하지 않은 경우 보조 데이터 소스, 보조 데이터 소스에서 검색할 필드 값 및 목록 항목으로 반환할 필드 값을 사용하여 유사한 관계를 구성할 수 있습니다. 리스트 박스 항목의 보조 데이터 소스를 지정하려면 다음과 같이 합니다.

1 리스트 박스의 *DataSource* 속성을 컨트롤에 대한 데이터 소스로 설정합니다.

2 *DataField* 속성에 대한 드롭다운 목록의 조회 값을 삽입할 필드를 선택합니다. 선택한 필드는 조회 필드가 될 수 없습니다.

3 리스트 박스의 *ListSource* 속성을 조회하려는 값의 필드가 포함된 데이터셋에 대한 데이터 소스로 설정합니다.

- 4 *KeyField* 속성에 대한 드롭다운 목록에서 조회 키로 사용할 필드를 선택합니다. 드롭다운 목록은 3 단계에서 지정한 데이터 소스에 연결된 데이터셋에 대한 필드를 표시합니다. 선택한 필드는 색인의 일부가 될 필요는 없지만 만약 색인의 일부라면 조회 성능이 훨씬 더 빨라집니다.
- 5 *ListField* 속성에 대한 드롭다운 목록에서 값을 반환할 필드를 선택합니다. 드롭다운 목록은 3 단계에서 지정한 데이터 소스에 연결된 데이터셋의 필드를 표시합니다.

조회 컨트롤에 연결된 테이블을 활성화하면 컨트롤은 목록 항목이 보조 소스로부터 파생된 것을 인식하고 해당 소스로부터 적절한 값을 표시합니다.

TDBLookupListBox 컨트롤에 단 한 번 나타나는 항목의 수를 지정하려면 *RowCount* 속성을 사용합니다. 리스트 박스의 높이는 이 행 수와 정확히 일치하도록 조정됩니다.

*TDBLookupComboBox*의 드롭다운 목록에 나타나는 항목의 수를 지정하려면 *DropDownRows* 속성을 대신 사용합니다.

참고 또한 데이터 그리드의 열을 설정하여 조회 콤보 박스처럼 행동하게 할 수 있습니다. 자세한 내용은 15-21 페이지의 "조회 목록 열 정의"를 참조하십시오.

체크 박스로 부울 필드 값 처리

*TDBCheckBox*는 data-aware 체크 박스 컨트롤입니다. 이 컨트롤은 데이터셋의 부울 필드 값을 설정하는 데 사용될 수 있습니다. 예를 들어, 고객 송장 폼에는 컨트롤이 선택된 경우 고객의 세금이 면제되고, 선택되지 않은 경우 고객의 세금이 면제되지 않는다는 것을 나타내는 체크 박스 컨트롤이 포함될 수 있습니다.

data-aware 체크 박스 컨트롤은 *ValueChecked* 및 *ValueUnchecked* 속성에 대한 현재 필드의 값을 비교하여 컨트롤의 선택 여부를 관리합니다. 필드 값이 *ValueChecked* 속성과 일치하면 컨트롤이 선택된 것입니다. 필드가 *ValueUnchecked* 속성과 일치하면 컨트롤이 선택되지 않은 것입니다.

참고 *ValueChecked* 및 *ValueUnchecked*의 값은 동일할 수 없습니다.

사용자가 다른 레코드로 이동할 때 컨트롤이 선택되는 경우, *ValueChecked* 속성을 컨트롤이 데이터베이스에 포스트해야 하는 값으로 설정합니다. 기본적으로 이 값은 "true"로 설정되지만 사용자의 필요에 따라 적절히 영숫자 값으로 설정할 수 있습니다. *ValueChecked*의 값으로서 세미콜론으로 구분된 항목 목록을 입력할 수도 있습니다. 현재 레코드의 필드 내용과 일치하는 항목이 있는 경우에는 체크 박스가 선택됩니다. 예를 들어, *ValueChecked* 문자열을 다음과 같이 지정할 수 있습니다.

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

현재 레코드의 필드에 "true", "Yes" 또는 "On" 값이 있으면 체크 박스가 선택됩니다. *ValueChecked* 문자열의 필드 비교는 대소문자가 구분됩니다. 사용자가 여러 *ValueChecked* 문자열이 있는 상자를 선택하는 경우, 첫 번째 문자열이 데이터베이스에 포스트되는 값입니다.

사용자가 다른 레코드로 이동할 때 컨트롤이 선택되지 않으면 *ValueUnchecked* 속성을 컨트롤이 데이터베이스에 포스트해야 하는 값으로 설정합니다. 기본적으로 이 값은 "false"로 설정되지만 필요에 따라 적절한 영숫자 값으로 설정할 수 있습니다. 또한

ValueUnchecked 값으로서 세미콜론으로 구분된 항목 목록을 입력할 수도 있습니다. 현재 레코드의 필드 내용과 일치하는 항목이 있으면 체크 박스가 선택되지 않습니다.

data-aware 체크 박스는 현재 레코드의 필드에 *ValueChecked* 또는 *ValueUnchecked* 속성에 나열된 값이 하나도 포함되지 않을 때마다 사용 불가능이 됩니다.

체크 박스가 연결되는 필드가 논리 필드라면 필드 내용이 *True*인 경우 항상 선택되고 필드 내용이 *False*인 경우 선택되지 않습니다. 이런 경우 *ValueChecked* 및 *ValueUnchecked* 속성에 입력된 문자열은 논리 필드에 어떤 영향도 주지 않습니다.

라디오 컨트롤로 필드 값 제한

*TDBRadioGroup*은 라디오 그룹 컨트롤의 data-aware 버전입니다. 이 컨트롤을 사용하면 필드에 대한 가능한 값의 수가 제한되어 있는 라디오 버튼 컨트롤로 데이터 필드 값을 설정할 수 있습니다. 라디오 그룹에는 필드가 사용할 수 있는 각 값에 대해 버튼이 하나 있습니다. 사용자는 원하는 라디오 버튼을 선택함으로써 데이터 필드에 대한 값을 설정할 수 있습니다.

Items 속성은 그룹에 나타나는 라디오 버튼을 결정합니다. *Items*는 문자열 목록입니다. *Items*의 문자열 하나에 대해 라디오 버튼이 하나 표시되고 각 문자열은 버튼의 레이블로서 라디오 버튼의 오른쪽에 나타납니다.

라디오 그룹에 연결된 필드의 현재 값이 *Items* 속성의 문자열 중 하나와 일치하면 해당 라디오 버튼이 선택됩니다. 예를 들어, "Red", "Yellow" 및 "Blue"의 세 문자열이 *Items*에 나열되고 현재 레코드의 필드에 "Blue" 값이 포함되는 경우, 그룹의 세 번째 버튼이 선택되어 표시됩니다.

참고 필드가 *Items*의 어떤 문자열과도 일치하지 않으면 필드가 *Values* 속성의 문자열과 일치하는 경우, 라디오 버튼은 여전히 선택될 수 있습니다. 현재 레코드에 대한 필드가 *Items* 또는 *Values*의 어떤 문자열과도 일치하지 않는 경우 라디오 버튼은 선택되지 않습니다.

Values 속성은 사용자가 라디오 버튼을 선택하고 레코드를 포스트할 때 데이터셋에 반환될 수 있는 문자열 목록을 옵션으로 포함할 수 있습니다. 문자열은 버튼에 순서대로 연결됩니다. 첫 번째 문자열은 첫 번째 버튼, 두 번째 문자열은 두 번째 버튼 등의 순으로 연결됩니다. 예를 들어, *Items*에는 "Red", "Yellow" 및 "Blue"가 포함되고 *Values*에는 "Magenta", "Yellow" 및 "Cyan"이 포함된다고 가정하십시오. 사용자가 "Red" 버튼을 선택하면 "Magenta"가 데이터베이스에 포스트됩니다.

*Values*에 대한 문자열이 제공되지 않으면 레코드가 포스트될 때 선택된 라디오 버튼에 대한 *Item* 문자열이 데이터베이스에 반환됩니다.

여러 레코드 표시

경우에 따라 동일한 폼에 많은 레코드를 표시해야 할 수도 있습니다. 예를 들어, 송장 애플리케이션은 동일한 폼에 한 명의 고객이 낸 모든 주문을 보여 주어야 합니다.

여러 레코드를 표시하려면 그리드 컨트롤을 사용합니다. 그리드 컨트롤은 애플리케이션의 사용자 인터페이스를 보다 우수하고 효과적으로 만들 수 있는 데이터의 여러 필드 및 여러 레코드 뷰를 제공합니다. 이것들은 15-15 페이지의 "TDBGrid로 데이터 보기

및 편집" 및 15-27 페이지의 "기타 data-aware 컨트롤을 포함하는 그리드 생성"에서 설명합니다.

참고 단방향 데이터셋을 사용할 때는 여러 레코드를 표시할 수 없습니다.

여러 레코드를 나타내는 그리드와 단일 레코드의 필드 모두를 표시하는 사용자 인터페이스를 디자인할 수도 있습니다. 이들을 조합하는 두 가지 모델이 있습니다.

- **마스터/디테일 폼:** 단일 필드 및 그리드 컨트롤을 표시하는 두 컨트롤을 포함시켜서 마스터 테이블과 디테일 테이블 모두로부터 정보를 나타낼 수 있습니다. 예를 들어, 한 명의 고객에 대한 정보를 해당 고객에 대한 주문을 표시하는 디테일 그리드로 표시할 수 있습니다. 마스터/디테일 폼의 원본으로 사용한 테이블 연결에 대한 내용은 18-35 페이지의 "마스터/디테일 관계 생성" 및 18-47 페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"을 참조하십시오.
- **드릴다운 폼:** 여러 레코드를 표시하는 폼에서 현재 레코드에 대한 상세한 정보를 표시하는 단일 필드 컨트롤을 포함할 수 있습니다. 이러한 접근 방식은 특히 레코드에 긴 메모나 그래픽 정보가 있을 때 유용합니다. 사용자가 그리드의 레코드를 통해 스크롤할 때 메모나 그래픽은 현재 레코드의 값을 나타내도록 업데이트됩니다. 이렇게 설정하는 것은 매우 쉽습니다. 두 개의 화면 표시 사이의 동기화는 그리드와 메모 또는 이미지 컨트롤이 공통적인 데이터 소스를 공유하는 경우 자동으로 처리됩니다.

팁 일반적으로 이러한 두 접근 방식을 하나의 폼에 조합하는 것은 좋은 방법이 아닙니다. 대개의 경우 사용자가 그런 폼의 데이터 관계를 이해하는 데 혼란을 줄 수 있습니다.

TDBGrid로 데이터 보기 및 편집

TDBGrid 컨트롤을 사용하여 표 모양의 그리드 형식으로 데이터셋의 레코드를 보고 편집할 수 있습니다.

그림 15.1 TDBGrid 컨트롤

현재 필드	열 제목	VendorName	Address1	City	State
레코드 지시자		Cacor Corporation	161 Southfield Rd	Southfield	OH
		Underwater	50 N 3rd Street	Indianapolis	IN
		J.W. Luscher Mfg.	65 Addams Street	Berkely	MA
		Scuba Professionals	3105 East Brace	Rancho Dominguez	CA
		Divers' Supply Shop	5208 University Dr	Macon	GA
		Techniques	52 Dolphin Drive	Redwood City	CA
		Perry Scuba	3443 James Ave	Hapeville	GA

그리드 컨트롤에 표시되는 레코드의 외형에 영향을 주는 세 가지 요인은 다음과 같습니다.

- Columns Editor를 사용하는 그리드에 대해 정의된 영구적 열 객체의 존재. 영구적 열 객체는 그리드 및 데이터 모습의 설정에 많은 유용성을 제공합니다. 영구적 열 사용에 대한 내용은 15-17 페이지의 "사용자 지정 그리드 생성"을 참조하십시오.

- 그리드에 표시된 데이터셋에 대한 영구적 필드 컴포넌트의 생성. Fields Editor를 사용한 영구적 필드 컴포넌트 생성에 대한 내용은 19장 "필드 컴포넌트 사용"을 참조하십시오.
- ADT와 배열 필드를 표시하는 그리드에 대한 데이터셋의 *ObjectView* 속성 설정. 15-22 페이지의 "ADT 및 배열 필드 표시"를 참조하십시오.

그리드 컨트롤은 자체적으로 *TDBGridColumn*s 객체의 증첩기인 *Columns* 속성을 가집니다. *TDBGridColumn*s는 그리드 컨트롤에 모든 열을 나타내는 *TColumn* 객체의 모음입니다. Columns Editor를 사용하여 디자인 타임 시 열 속성을 설정하거나 또는 그리드의 *Columns* 속성을 사용하여 런타임 시 *TDBGridColumn*s의 속성, 이벤트 및 메소드에 액세스할 수 있습니다.

기본 상태에서 그리드 컨트롤 사용

그리드의 *Columns* 속성의 *State* 속성은 그리드에 대한 영구적 열 객체의 존재 여부를 나타냅니다. *Columns.State*는 그리드에 대해 자동으로 설정되는 런타임 전용 속성입니다. 기본 상태는 *csDefault*이며 영구적 열 객체는 그리드에 존재하지 않습니다. 이러한 경우 그리드의 데이터 표시는 주로 그리드의 데이터셋의 필드 속성에 의해 결정되거나 영구적 필드 컴포넌트가 없을 경우에는 표시 특성의 기본 설정에 의해 결정됩니다.

그리드의 *Columns.State* 속성이 *csDefault*인 경우, 그리드 열은 데이터셋의 시각적 필드로부터 동적으로 생성되고 그리드의 열 순서는 데이터셋의 필드 순서와 일치합니다. 그리드의 각 열은 필드 컴포넌트에 연결됩니다. 필드 컴포넌트의 속성 변경은 즉시 그리드에 나타납니다.

동적으로 생성된 열이 있는 그리드 컨트롤을 사용하면 런타임 시 선택된 임의의 테이블의 내용을 보거나 편집하는 데 유용합니다. 그리드의 구조는 설정되어 있지 않기 때문에 다양한 데이터셋에 맞도록 동적으로 변경될 수 있습니다. 동적으로 생성된 열을 가지는 단일 그리드는 Paradox 테이블을 표시하고, 그런 다음 그리드의 *DataSource* 속성이 변하거나 데이터 소스의 *DataSet* 속성이 변할 때 MySQL 쿼리 결과를 표시하도록 전환될 수 있습니다.

디자인 타임 또는 런타임에 동적 열의 외형을 변경할 수 있습니다. 그러나 실제로 수정하는 것은 열에 표시된 필드 컴포넌트의 속성입니다. 동적 열의 속성은 열이 단일 데이터셋의 특정 필드에 연결된 경우에만 존재합니다. 예를 들어, 열의 *Width* 속성을 변경하면 해당 열에 연결된 필드의 *DisplayWidth* 속성이 변경됩니다. *Font*와 같이 필드 속성에 기준하지 않는 열 속성에 취해진 변경 사항은 열의 수명 동안만 존재합니다.

그리드의 데이터셋이 동적 필드 컴포넌트로 구성되는 경우, 필드는 데이터셋이 닫힐 때마다 없어집니다. 필드 컴포넌트가 없다면 이와 연결된 모든 동적 열도 함께 없어집니다. 그리드의 데이터셋이 영구적 필드 컴포넌트로 구성되는 경우, 필드는 데이터셋이 닫힌 경우에도 존재합니다. 따라서 이들 필드에 연결된 열도 데이터셋이 닫힌 경우라도 속성을 계속 유지합니다.

참고 그리드의 *Columns.State* 속성을 런타임 시 *csDefault*로 바꾸면 영구적 열을 비롯한 그리드의 모든 열 객체가 삭제되고 그리드의 데이터셋의 시각적 필드에 따라 동적 열이 재구성됩니다.

사용자 지정 그리드 생성

사용자 지정 그리드는 열이 나타나는 방법과 열의 데이터가 표시되는 방법을 설명하는 영구적 열 객체를 정의하기 위한 것입니다. 사용자 지정 그리드를 사용하면 동일한 데이터셋의 다양한 뷰(예를 들어, 다양한 열 순서, 다양한 필드 선택 및 다양한 열 색상과 글꼴)를 나타내는 여러 그리드를 구성할 수 있습니다. 사용자 지정 그리드를 사용하면 그리드에서 사용되는 필드나 데이터셋의 필드 순서에 영향을 주지 않고 런타임 시 그리드의 모습을 사용자가 수정할 수도 있습니다.

사용자 지정 그리드는 디자인 타임에 구조가 알려진 데이터셋과 함께 가장 많이 사용됩니다. 사용자 지정 그리드는 디자인 타임에 선언된 필드 이름이 데이터셋에 존재한다고 예상하기 때문에 런타임 시 선택되는 임의의 테이블을 검색하는 데에는 적합하지 않습니다.

영구적 열(Persistent Columns) 이해

그리드에 대한 영구적 열 객체를 만들 때 이 열들은 그리드의 데이터셋의 원본으로 사용한 필드에 느슨하게 연결됩니다. 영구적 열에 대한 기본 속성 값은 열 속성에 값이 할당될 때까지 기본 소스(연결된 필드 또는 그리드 자체)에 동적으로 페치(fetch)됩니다. 열 속성에 값을 할당하기 전까지는 기본 소스가 변경되면 이 값도 변경됩니다. 열 속성에 값을 할당하면 기본 소스가 변경되어도 이 값은 더 이상 변경되지 않습니다.

예를 들어, 열 제목 캡션에 대한 기본 소스는 연결된 필드의 *DisplayLabel* 속성입니다. *DisplayLabel* 속성을 수정하면 열 제목은 변경 내용을 즉시 반영합니다. 열 제목의 캡션에 문자열(string)을 할당하면 제목 캡션은 연결된 필드의 *DisplayLabel* 속성과는 무관하게 됩니다. 필드의 *DisplayLabel* 속성에 대한 추후 변경 사항은 더 이상 열 제목에 영향을 주지 않습니다.

영구적 열은 연결된 필드 컴포넌트와는 독립적으로 존재합니다. 실제로 영구적 열은 필드 객체와 연결될 필요가 없습니다. 영구적 열의 *FieldName* 속성이 비어 있거나 필드 이름이 그리드의 현재 데이터셋의 어떤 필드 이름과도 일치하지 않으면, 열의 *Field* 속성은 NULL이고 열은 빈 셀로 그려집니다. 셀의 기본 그리기 메소드를 오버라이드하면 빈 셀에 자신의 사용자 정의 정보를 표시할 수 있습니다. 예를 들어, 빈 열을 사용하여 요약을 집계하는 레코드 그룹의 마지막 레코드에 대한 집계 값을 표시할 수 있습니다. 다른 가능성은 레코드 데이터의 일부를 그래픽으로 묘사하는 비트맵이나 막대 도표를 표시하는 것입니다.

두 개 이상의 영구적 열은 데이터셋의 동일한 필드에 연결될 수 있습니다. 예를 들어, 그리드를 스크롤하지 않고도 부분 번호를 찾기 쉽도록 하기 위해 보다 넓은 그리드의 왼쪽과 오른쪽 맨끝에 부분 번호 필드를 표시할 수도 있습니다.

참고 영구적 필드는 데이터셋의 필드에 연결될 필요가 없기 때문에, 그리고 여러 열은 동일한 필드를 참조할 수 있기 때문에, 사용자 지정 그리드의 *FieldCount* 속성은 그리드의 열 수보다 작거나 같습니다. 또한 사용자 지정 그리드의 현재 선택된 열이 필드에 연결되지 않은 경우에는 그리드의 *SelectedField* 속성은 NULL이고 *SelectedIndex* 속성은 -1 임에 주의하십시오.

영구적 열은 그리드 셀을 다른 데이터셋이나 정적 선택 목록의 조회 값의 콤보 박스 드롭다운 목록으로 또는 현재 셀과 관련된 특수 데이터 뷰어나 대화 상자를 실행하기 위해 사용자가 클릭할 수 있는 셀의 생략 버튼(...)으로 표시하도록 구성될 수 있습니다.

영구적 열 생성

디자인 타임에 그리드의 외양을 사용자 정의하려면 Columns Editor를 실행하여 그리드에 대한 영구적 열 객체들을 생성합니다. 런타임 시 영구적 열 객체가 있는 그리드의 *State* 속성은 *csCustomized*으로 자동 설정됩니다.

그리드 컨트롤에 대한 영구적 열을 생성하려면 다음과 같이 합니다.

- 1 폼에서 그리드 컴포넌트를 선택합니다.
- 2 Object Inspector에서 그리드의 *Columns* 속성을 더블 클릭하여 Columns Editor를 실행합니다.

Columns 리스트 박스가 선택된 그리드에 대해 정의된 영구적 열을 표시합니다. Columns Editor를 처음 실행하면 그리드는 동적 열만 포함하는 기본 상태이기 때문에 목록이 비어 있습니다.

데이터셋의 모든 필드에 대한 영구적 열을 한 번에 생성하거나 영구적 열을 개별적으로 생성할 수 있습니다. 모든 필드에 대한 영구적 열을 생성하려면 다음과 같이 합니다.

- 1 그리드를 마우스 오른쪽쪽을 클릭하여 컨텍스트 메뉴를 표시하고 Add All Fields를 선택합니다. 그리드가 아직 데이터 소스에 연결되지 않은 경우에는 Add All Fields를 사용할 수 없다는 점에 주의하십시오. Add All Fields를 선택하기 전에 활성 데이터셋을 갖고 있는 데이터 소스에 그리드를 연결하십시오.
- 2 그리드에 이미 영구적 열이 있는 경우에는 대화 상자가 나타나 기존의 열을 삭제할지 아니면 열에 첨부할지를 묻습니다. Yes를 선택하면 기존의 영구적 열 정보가 제거되고 필드 이름에 의해 현재 데이터셋의 모든 필드가 데이터셋의 순서대로 삽입됩니다. No를 선택하면 모든 기존의 영구적 열 정보가 유지되고 새로운 열 정보가 데이터셋의 추가 필드에 따라 데이터셋에 추가됩니다.
- 3 Close를 클릭하면 그리드에 영구적 열이 적용되고 대화 상자가 닫힙니다.

영구적 열을 개별적으로 생성하려면 다음과 같이 합니다.

- 1 Columns Editor에서 Add 버튼을 선택합니다. 리스트 박스에서 새로운 열이 선택됩니다. 새로운 열에는 순차적인 번호와 기본 이름(예를 들어, 0-TColumn)이 있습니다.
- 2 이 새로운 열에 필드를 연결하려면 Object Inspector의 *FieldName* 속성을 설정합니다.
- 3 새로운 열의 제목을 설정하려면 Object Inspector의 *Title* 속성을 확장하고 *Caption* 속성을 설정합니다.
- 4 Columns Editor를 닫으면 그리드에 영구적 열이 적용되고 대화 상자가 닫힙니다.

런타임 시 *Columns.State* 속성에 *csCustomized*를 할당하여 영구적 열로 전환할 수 있습니다. 그리드의 기존 열이 모두 없어지고 새로운 영구적 열이 그리드의 데이터셋의 각 필드에 대해 생성됩니다. 그런 다음 열 목록에 대해 *Add* 메소드를 호출하여 런타임 시 영구적 열을 추가할 수 있습니다.

```
DBGrid1.Columns.Add;
```

영구적 열 삭제

그리드에서 영구적 열을 삭제하는 것은 표시하고 싶지 않은 필드를 제거하는 데 유용합니다. 그리드에서 영구적 열을 삭제하려면 다음과 같이 합니다.

- 1 그리드를 더블 클릭하여 Columns Editor를 표시합니다.
- 2 Columns 리스트 박스에서 제거할 필드를 선택합니다.
- 3 Delete를 클릭합니다. 컨텍스트 메뉴나 *Del* 키를 사용해도 열을 삭제할 수 있습니다.

참고 그리드에서 모든 열을 삭제하면 *Columns.State* 속성이 *csDefault* 상태로 되돌아가고 데이터셋의 각 필드에 대해 동적인 열을 자동으로 구성합니다.

런타임 시 열 객체를 해제해도 영구적 열을 삭제할 수 있습니다.

```
DBGrid1.Columns[5].Free;
```

영구적 열 순서 정렬

열이 Columns Editor에 나타나는 순서는 열이 그리드에 나타나는 순서와 동일합니다. Columns 리스트 박스 내에 열을 끌어다 놓음으로써 열 순서를 변경할 수 있습니다.

열 순서를 변경하려면 다음과 같이 합니다.

- 1 Columns 리스트 박스에서 열을 선택합니다.
- 2 열을 리스트 박스의 새로운 위치로 끌어 놓습니다.

또한 런타임에서와 같이 실제 그리드의 열 제목을 클릭하고 열을 새로운 위치로 끌어 놓아 열 순서를 변경할 수 있습니다.

참고 Fields Editor에서의 영구적 필드를 재정렬하면 사용자 지정 그리드가 아닌 기본 그리드의 열도 재정렬됩니다.

중요 디자인 타임에 동적 열과 동적 필드를 모두 포함하는 그리드의 열은 변경된 필드나 열 순서를 기록할 영구적인 것이 아무 것도 없기 때문에 재정렬할 수 없습니다.

런타임 시 *DragMode* 속성이 *dmManual*로 설정되어 있으면 사용자는 마우스를 사용하여 열을 그리드의 새로운 위치로 끌어 놓을 수 있습니다. *csDefault* 상태의 *State* 속성으로 그리드의 열을 재정렬하면 그리드의 원본으로 사용한 데이터셋의 필드 컴포넌트도 재정렬됩니다. 실제 테이블의 필드 순서는 영향을 받지 않습니다. 사용자가 런타임 시 열을 재정렬할 수 없도록 하려면 그리드의 *DragMode* 속성을 *dmAutomatic*으로 설정합니다.

런타임 시 그리드의 *OnColumnMoved* 이벤트는 열이 이동된 후에 발생합니다.

디자인 타임 시 열 속성 설정

열 속성은 해당 열의 셀에 데이터가 표시되는 방법을 결정합니다. 대부분의 열 속성은 그리드나 연결된 필드 컴포넌트와 같은 *default source*라는 다른 컴포넌트에 연결된 속성에서 기본값을 얻습니다.

열의 속성을 설정하려면 Columns Editor에서 열을 선택하고 Object Inspector에서 속성을 설정합니다. 다음 표는 사용자가 설정할 수 있는 주요 열 속성을 요약해서 보여 줍니다.

표 15.2 열 속성

속성	용도
Alignment	열에서 필드 데이터를 왼쪽, 오른쪽 또는 가운데 정렬합니다. 기본 소스: <i>TField.Alignment</i> .
ButtonStyle	<i>cbAuto</i> : (기본값) 연결된 필드가 조회 필드이거나 열의 <i>PickList</i> 속성에 데이터가 포함된 경우 드롭다운 목록을 표시합니다. <i>cbEllipsis</i> : 셀의 오른쪽에 생략(...) 버튼을 표시합니다. 이 버튼을 클릭하면 그리드의 <i>OnEditButtonClick</i> 이벤트가 발생합니다. <i>cbNone</i> : 일반적인 편집 컨트롤만 사용하여 열의 데이터를 편집합니다.
Color	열의 셀 배경색을 지정합니다. 기본 소스: <i>TDBGrid.Color</i> . (텍스트 전경 색은 Font 속성 참조)
DropDownRows	드롭다운 목록에서 표시되는 텍스트의 줄 수. 기본값: 7.
Expanded	열을 확장할지 여부를 지정합니다. ADT 또는 배열 필드를 나타내는 열들에만 적용합니다.
FieldName	이 열에 연결된 필드 이름을 지정합니다. 비워 둘 수 있습니다.
ReadOnly	<i>True</i> : 사용자가 열의 데이터를 편집할 수 없습니다. <i>False</i> : (기본값) 열의 데이터를 편집할 수 있습니다.
Width	화면 픽셀로 열의 폭을 지정합니다. 기본 소스: <i>TField.DisplayWidth</i> .
Font	열에서 텍스트를 나타내는 데 사용되는 글꼴, 크기 및 색상을 지정합니다. 기본 소스: <i>TDBGrid.Font</i> .
PickList	열의 드롭다운 목록에 표시할 값의 목록을 포함합니다.
Title	선택된 열의 제목에 대한 속성을 설정합니다.

다음 표는 *Title* 속성에 대해 지정할 수 있는 옵션을 요약해서 보여 줍니다.

표 15.3 확장된 TColumn Title 속성

속성	용도
Alignment	열 제목의 캡션 텍스트를 왼쪽(기본값), 오른쪽 또는 가운데 정렬합니다.
Caption	열 제목에 표시할 텍스트를 지정합니다. 기본 소스: <i>TField.DisplayLabel</i> .
Color	열 제목 셀에 그리는 데 사용되는 배경 색을 지정합니다. 기본 소스: <i>TDBGrid.FixedColor</i> .
Font	열 제목에 텍스트를 나타내는 데 사용되는 글꼴, 크기 및 색상을 지정합니다. 기본 소스: <i>TDBGrid.TitleFont</i> .

조회 목록 열 정의

조회 콤보 박스 컨트롤과 유사하게 값의 드롭다운 목록을 표시하는 열을 생성할 수 있습니다. 열이 콤보 박스처럼 행동하도록 지정하려면 열의 *ButtonStyle* 속성을 *cbsAuto* 로 설정합니다. 목록을 값으로 채우고 나면 해당 열의 셀이 편집 모드로 될 때 그리드가 콤보 박스와 같은 드롭다운 버튼을 자동으로 표시합니다.

사용자가 선택한 값으로 목록을 채우는 방법에는 두 가지가 있습니다.

- 조회 테이블에서 값을 폐치할 수 있습니다. 열이 독립된 조회 테이블로부터 그려진 값의 드롭다운 목록을 표시하게 하려면 데이터셋의 조회 필드를 정의해야 합니다. 조회 필드 생성에 대한 내용은 17-10 페이지의 "서버에 명령 전송"을 참조하십시오. 일단 조회 필드를 정의하면 열의 *FieldName*을 조회 필드 이름으로 설정합니다. 드롭다운 목록은 조회 필드에서 정의된 조회 값으로 자동으로 채워집니다.
- 디자인 타임에 값 목록을 명시적으로 지정할 수 있습니다. 디자인 타임에 목록 값을 입력하려면 Object Inspector에서 열에 대한 *PickList* 속성을 더블 클릭합니다. 그러면 String List Editor가 나타나서 열에 대한 선택 목록을 채울 값을 입력할 수 있습니다.

기본적으로 드롭다운 목록은 7개의 값을 표시합니다. 이 목록의 길이는 *DropDownRows* 속성을 설정하여 변경할 수 있습니다.

참고 명시적인 선택 목록이 있는 열을 정상 행동으로 복구하려면 String List Editor를 사용하여 선택 목록에서 모든 텍스트를 삭제합니다.

열에 버튼 두기

열은 생략(...) 버튼을 보통 셀 편집기의 오른쪽에 표시할 수 있습니다. *Ctrl+Enter* 또는 마우스를 클릭하면 그리드의 *OnEditButtonClick* 이벤트가 시작됩니다. 생략 버튼을 사용하면 열의 데이터에 대한 상세 뷰가 있는 폼이 나타납니다. 예를 들어, 송장 요약을 표시하는 테이블에서 송장 전체 열에 생략 버튼을 설정하여 해당 송장의 항목, 세금 계산 방법 등을 표시하는 폼을 불러올 수 있습니다. 그래픽 필드의 경우에는 이 생략 버튼을 사용하여 이미지를 표시하는 폼을 불러올 수 있습니다.

열에 생략 버튼을 생성하려면 다음과 같이 합니다.

- 1 *Columns* 리스트 박스에서 열을 선택합니다.
- 2 *ButtonStyle*을 *cbsEllipsis*로 설정합니다.
- 3 *OnEditButtonClick* 이벤트 핸들러를 생성합니다.

열에 기본값 복구

런타임 시 열의 *AssignedValues* 속성을 테스트하여 열 속성이 명시적으로 할당되었는지 여부를 결정할 수 있습니다. 명시적으로 정의되지 않은 값은 연결된 필드나 그리드의 기본값에 동적으로 유도된 값으로 복원됩니다.

하나 이상의 열에 대해 변경된 속성을 실행 취소할 수 있습니다. Columns Editor에서 복구할 열을 선택하고 컨텍스트 메뉴에서 Restore Defaults를 선택합니다. Restore defaults는 할당된 속성 설정을 버리고 원본으로 사용한 필드 컴포넌트에서 파생된 값으로 열 속성을 복구합니다.

런타임 시 열의 *RestoreDefaults* 메소드를 호출하면 단일 열에 대한 모든 기본 속성을 재설정할 수 있습니다. 또한 열 목록의 *RestoreDefaults* 메소드를 사용하면 그리드의 모든 열에 대한 기본 속성을 재설정할 수도 있습니다.

```
DBGrid1.Columns.RestoreDefaults;
```

ADT 및 배열 필드 표시

경우에 따라 그리드의 데이터셋의 필드는 텍스트, 그래픽, 숫자 값 등과 같은 간단한 값을 나타내지 않습니다. 일부 데이터베이스 서버는 ADT 필드나 배열 필드와 같이 보다 단순한 데이터 타입의 복잡한 필드를 허용합니다.

그리드가 복합 필드를 표시하는 방법은 두 가지입니다.

- 필드를 구성하는 보다 단순한 타입 각각이 데이터셋에서 독립된 필드로서 나타나도록 필드를 "평평하게(flatten out)" 할 수 있습니다. 복합 필드가 평평하게 되면 필드의 구성 요소는 원본으로 사용한 데이터베이스 테이블의 공통 부모 필드의 이름이 각 필드 이름에 선행하는 공통 소스를 반영하는 독립된 필드로서 나타납니다.

복합 필드가 평평한 것처럼 표시하려면 데이터셋의 *ObjectView* 속성을 *False*로 설정해야 합니다. 데이터셋은 복합 필드를 별개의 필드 모음으로 저장하고, 그리드는 각 구성 요소에 독립된 열을 할당하여 이를 반영합니다.

- 이 속성은 단일 열에 복합 필드를 표시하여 필드가 단일 필드라는 사실을 반영할 수 있습니다. 복합 필드를 단일 열에 표시할 때는 필드의 제목 표시줄의 화살표를 클릭하거나 열의 *Expanded* 속성을 설정하여 열을 확장하거나 축소할 수 있습니다.
 - 열을 확장하는 경우, 각각의 자식 필드는 부모 필드의 제목 표시줄 아래에 열이 분리되어 나타납니다. 즉, 그리드에 대한 제목 표시줄은 높이가 늘어나면서 첫 번째 행에는 복합 필드의 이름이 표시되고 두 번째 행에서는 각각의 열마다 이름이 표시됩니다. 복합이 아닌 필드는 높이가 늘어난 제목 표시줄로 나타납니다. 이러한 확장은 교대로 복합 필드인 구성 요소에 대해서도 계속되며(예를 들어, 디테일 테이블에 중첩된 디테일 테이블) 그에 따라 제목 표시줄의 높이가 증가합니다.
 - 필드가 축소되면 자식 필드를 포함한 쉼표로 구분된 편집 불가능한 문자열과 함께 하나의 열만이 나타납니다.

확장 및 제거 열에 복합 필드를 표시하려면 데이터셋의 *ObjectView* 속성을 *True*로 설정합니다. 데이터셋은 중첩된 하위 필드를 포함하는 단일 필드 컴포넌트로 복합 필드를 저장합니다. 그리드는 확장하거나 축소할 수 있는 열에 이를 반영합니다.

그림 15.2는 ADT 필드와 배열 필드를 가진 그리드를 보여 줍니다. 각각의 자식 필드가 열을 가지도록 데이터셋의 *ObjectView* 속성을 *False*로 설정합니다.

그림 15.2 False로 설정된 ObjectView를 가지는 TDBGrid 컨트롤

ADT 자식 필드			배열 자식 필드			
ID_KEY	NAME_ADT.FIRST	NAME_ADT.MIDDLE	NAME_ADT.LAST	TELNOS_ARRAY[0]	TELNOS_ARRAY[1]	TELNOS_ARR[2]
1	Stephan		Wright	415-908-9875	902-786-1245	
2	Whitney	N	Long			
3	Chris	T	Scanlan	234-232-1343		

그림 15.3과 15.4는 ADT 필드와 배열 필드를 가지는 그리드를 보여 줍니다. 그림 15.3은 필드가 축소된 것을 보여 줍니다. 이 상태에서는 필드를 편집할 수 없습니다. 그림 15.4는 필드가 확장된 것을 보여 줍니다. 필드 제목 표시줄에 있는 화살표를 클릭하면 필드가 확장되거나 축소됩니다.

그림 15.3 Expanded가 False로 설정된 TDBGrid 컨트롤

ID_KEY	NAME_ADT	TELNOS_ARRAY
1	(Stephan, ., Wright)	(415-908-9875, 902-786-1245,
2	(Whitney, N, Long)	(,.., 510-454-7234,
3	(Chris, T, Scanlan)	(234-232-1343,

그림 15.4 Expanded가 True로 설정된 TDBGrid 컨트롤

ADT 자식 필드 열			배열 자식 필드 열				
ID_KEY	NAME_ADT	TELNOS_ARRAY			TELNO		
	FIRST	MIDDLE	LAST	TELNOS_ARRAY[0]	TELNOS_ARRAY[1]	TELNOS_ARRAY[2]	TELNO
1	Stephan		Wright	415-908-9875	902-786-1245		
2	Whitney	N	Long				510-454
3	Chris	T	Scanlan	234-232-1343			

다음 표는 ADT와 배열 필드가 *TDBGrid*에 나타나는 방식에 영향을 주는 속성들을 나열한 것입니다.

표 15.4 복합 필드가 나타나는 방식에 영향을 주는 속성

속성	객체	용도
Expandable	TColumn	독립되고 편집 가능한 열로 자식 필드를 나타내도록 열을 확장할 수 있는지 여부를 나타냅니다. (읽기 전용)
Expanded	TColumn	열을 확장할지 여부를 지정합니다.
MaxTitleRows	TDBGrid	그리드에 나타날 수 있는 최대 제목 행의 수를 지정합니다.
ObjectView	TDataSet	필드가 객체 필드를 확장하고 축소할 수 있는 객체 모드로 또는 평평하게 (flatten out) 표시되는지 여부를 지정합니다.
ParentColumn	TColumn	자식 필드의 열을 소유하는 TColumn 객체를 참조합니다.

참고 ADT 및 배열 필드를 비롯한 일부 데이터셋은 다른 데이터셋(데이터셋 필드)이나 다른 데이터셋(참조) 필드의 레코드를 참조하는 필드를 포함합니다. Data-aware 그리드는 이 필드를 "(DataSet)" 또는 "(Reference)"로 각각 표시합니다. 런타임 시 생략 버튼은 오른쪽에 나타납니다. 생략 버튼을 클릭하면 그리드가 필드 내용을 표시하는 새로운 폼을 불러옵니다. 데이터셋 필드의 경우, 이 그리드는 필드의 값인 데이터셋을 표시합니다. 참조 필드의 경우, 이 그리드는 다른 데이터셋의 레코드를 표시하는 단일 행을 포함합니다.

그리드 옵션 설정

디자인 타임에 그리드 *Options* 속성을 사용하면 기본 그리드의 행동과 런타임 시의 외형을 조정할 수 있습니다. 디자인 타임에 그리드 컴포넌트를 처음으로 폼에 두면 Object Inspector의 *Options* 속성이 + 기호로 표시되어 *Options* 속성을 확장하여 개별적으로 설정할 수 있는 일련의 Boolean 속성을 표시할 수 있다는 것을 나타냅니다. 이러한 속성을 보거나 설정하려면 + 기호를 클릭합니다. Object Inspector의 옵션 목록이 *Options* 속성 아래에 표시됩니다. - 기호를 클릭하면 목록이 축소되면서 + 기호로 변경됩니다.

다음 표는 설정할 수 있는 *Options* 속성을 보여 주며, 이 속성은 런타임 시 그리드에 영향을 줍니다.

표 15.5 Expanded TDBGrid Options 속성

옵션	용도
dgEditing	<i>True</i> : (기본값). 그리드 레코드의 편집, 삽입 및 삭제를 가능하게 합니다. <i>False</i> : 그리드 레코드의 편집, 삽입 및 삭제를 불가능하게 합니다.
dgAlwaysShowEditor	<i>True</i> : 필드를 선택하면 Edit 상태가 됩니다. <i>False</i> : (기본값). 필드가 선택될 때 자동으로 Edit 상태로 되지는 않습니다.
dgTitles	<i>True</i> : (기본값). 그리드 상단에 필드 이름을 표시합니다. <i>False</i> : 필드 이름 표시가 비활성화됩니다.
dgIndicator	<i>True</i> : (기본값). 지시자 열이 그리드 왼쪽에 표시되고 현재 레코드 지시자(그리드 왼쪽의 화살표)가 활성화되어 현재 레코드를 보여 줍니다. 삽입 시 화살표는 별표가 됩니다. 편집 시 화살표는 I 빔이 됩니다. <i>False</i> : 지시자 열이 꺼집니다.
dgColumnResize	<i>True</i> : (기본값). 제목 부분의 열 조정자를 끌어서 열 크기를 조정할 수 있습니다. 열 크기를 조정하면 원본으로 사용한 <i>TField</i> 컴포넌트의 폭을 변경할 수 있습니다. <i>False</i> : 그리드에서 열 크기를 조정할 수 없습니다.
dgColLines	<i>True</i> : (기본값). 열 사이의 수직 구분 선을 표시합니다. <i>False</i> : 열 사이의 구분 선을 표시하지 않습니다.
dgRowLines	<i>True</i> : (기본값). 레코드 사이의 수평 구분 선을 표시합니다. <i>False</i> : 레코드 사이의 구분 선을 표시하지 않습니다.

표 15.5 Expanded TDBGrid Options 속성 (계속)

옵션	용도
dgTabs	<i>True</i> : (기본값). 레코드의 필드 사이에 탭 이동을 가능하게 합니다. <i>False</i> : 탭을 누르면 그리드 컨트롤을 빠져 나갑니다.
dgRowSelect	<i>True</i> : 선택 막대가 그리드의 전체 폭에 미칩니다. <i>False</i> : (기본값). 레코드의 필드를 선택하면 해당 필드만 선택됩니다.
dgAlwaysShowSelection	<i>True</i> : (기본값). 다른 컨트롤에 포커스가 있는 경우에도 그리드의 선택 막대는 항상 보입니다. <i>False</i> : 그리드에 포커스가 있는 경우에만 그리드의 선택 막대가 보입니다.
dgConfirmDelete	<i>True</i> : (기본값). 레코드 삭제 (<i>Ctrl+Del</i>) 시 사용자에게 확인을 요구합니다. <i>False</i> : 확인 없이 레코드를 삭제합니다.
dgCancelOnExit	<i>True</i> : (기본값). 그리드에서 포커스가 사라지면 보류 중인 삽입을 취소합니다. 이 옵션은 부분 또는 빈 레코드의 우연한 포스트를 방지합니다. <i>False</i> : 보류 중인 삽입을 허용합니다.
dgMultiSelect	<i>True</i> : 사용자가 <i>Ctrl+Shift</i> 또는 <i>Shift+화살표</i> 키를 사용하여 그리드의 비연속적인 행을 선택할 수 있도록 합니다. <i>False</i> : (기본값). 사용자가 행을 여러 개 선택할 수 없도록 합니다.

그리드 편집

런타임 시 다음의 기본 조건이 만족되는 경우에는 그리드를 사용하여 기존의 데이터를 수정하거나 새로운 레코드를 입력할 수 있습니다.

- *Dataset*의 *CanModify* 속성은 *True*입니다.
- 그리드의 *ReadOnly* 속성은 *False*입니다.

사용자가 그리드의 레코드를 편집할 때 각 필드의 변경 사항은 내부 레코드 버퍼에 포스트됩니다. 변경 내용은 사용자가 그리드의 다른 레코드로 이동한 다음에 포스트됩니다. 폼의 다른 컨트롤로 포커스가 변경되어도 데이터셋에 대한 커서가 다른 레코드로 이동할 때까지 그리드는 변경 사항을 포스트하지 않습니다. 레코드가 포스트되는 경우 데이터셋은 모든 연결된 data-aware 컴포넌트의 상태 변경을 확인합니다. 수정된 데이터를 포함하는 필드를 업데이트하는 데 문제가 발생하면 예외를 발생시키고 레코드를 수정하지 않습니다.

참고 애플리케이션이 업데이트를 캐시로 저장하면 레코드 변경 사항의 포스트는 단지 내부 캐시에 추가합니다. 애플리케이션이 업데이트 사항을 적용한 후에 원본으로 사용한 데이터셋 테이블에 다시 포스트됩니다.

다른 레코드로 이동하기 전에 필드에서 *Esc*를 누르면 레코드의 모든 편집 사항이 취소됩니다.

그리드 그리기 제어

그리드 컨트롤이 자신을 그리는 방법에 대한 첫 번째 수준의 조정은 열 속성을 설정하는 것입니다. 그리드는 자동으로 열의 글꼴, 색상 및 정렬 속성을 사용하여 해당 열의 셀을 그립니다. 열에 연결된 필드 컴포넌트의 *DisplayFormat* 또는 *EditFormat* 속성을 사용하여 데이터 필드의 텍스트를 그립니다.

기본 그리드 표시 로직을 그리드의 *OnDrawColumnCell* 이벤트 코드의 인수로 전달할 수 있습니다. 그리드의 *DefaultDrawing* 속성이 *True*이면 모든 일반적인 그리기는 *OnDrawColumnCell* 이벤트 핸들러가 호출되기 전에 수행됩니다. 그런 다음 코드가 기본 표시 위에 그려질 수 있습니다. 이는 빈 영구적 열을 정의하고 해당 열의 셀에 특수 값을 그리려고 할 때 주로 유용합니다.

그리드의 그리기 로직을 완전히 바꾸려면 *DefaultDrawing*을 *False*로 설정하고 그리기 코드를 그리드의 *OnDrawColumnCell* 이벤트에 둡니다. 그리기 로직을 특정 열이나 특정 필드 데이터 타입에 대해서만 교체하려면 *OnDrawColumnCell* 이벤트 핸들러 내의 *DefaultDrawColumnCell* 메소드를 호출하여 그리드가 선택된 열에 대해 일반적인 그리기 코드를 이용하도록 할 수 있습니다. 예를 들어, 부울 필드 타입에 대해서만 그리기 방식을 변경하려는 경우에 사용자가 해야 하는 작업의 양이 줄어듭니다.

런타임 시 사용자 동작에 응답

그리드 내의 특정 동작에 응답하도록 이벤트 핸들러를 작성하여 런타임 시 그리드 동작을 수정할 수 있습니다. 그리드는 보통 한 번에 많은 필드와 레코드를 표시하기 때문에 개별적인 열에 대한 변경 사항에 응답해야 할 필요가 있을 수 있습니다. 예를 들어, 사용자가 특정 열을 입력하고 열을 빠져나갈 때마다 사용자는 폼에서 버튼을 활성화하고 비활성화할 수도 있습니다.

다음 표는 Object Inspector에서 사용할 수 있는 그리드 이벤트를 보여 줍니다.

표 15.6 그리드 컨트롤 이벤트

이벤트	용도
OnCellClick	사용자가 그리드의 셀을 클릭할 때 발생합니다.
OnColEnter	사용자가 그리드의 열로 이동할 때 발생합니다.
OnColExit	사용자가 그리드의 열을 떠날 때 발생합니다.
OnColumnMoved	사용자가 새로운 위치로 열을 이동할 때 발생합니다.
OnDbClick	사용자가 그리드를 더블 클릭할 때 발생합니다.
OnDragDrop	사용자가 그리드에서 끌어다 놓을 때 발생합니다.
OnDragOver	사용자가 그리드 위로 끌 때 발생합니다.
OnDrawColumnCell	애플리케이션이 개별적인 셀을 그릴 필요가 있을 때 발생합니다.
OnDrawDataCell	(폐기됨) <i>State</i> 가 <i>csDefault</i> 인 경우 애플리케이션이 개별적인 셀을 그릴 필요가 있을 때 발생합니다.
OnEditButtonClick	사용자가 열에서 생략 버튼을 클릭할 때 발생합니다.
OnEndDrag	사용자가 그리드에서 끌기를 멈추었을 때 발생합니다.
OnEnter	그리드가 포커스를 얻었을 때 발생합니다.

표 15.6 그리드 컨트롤 이벤트 (계속)

이벤트	용도
OnExit	그리드가 포커스를 잃었을 때 발생합니다.
OnKeyDown	사용자가 그리드에서 키보드의 키나 키 조합을 눌렀을 때 발생합니다.
OnKeyPress	사용자가 그리드에서 키보드의 영숫자 키 하나를 눌렀을 때 발생합니다.
OnKeyUp	사용자가 그리드에서 키를 놓았을 때 발생합니다.
OnStartDrag	사용자가 그리드에서 끌어가기를 시작할 때 발생합니다.
OnTitleClick	사용자가 열에 대한 제목을 클릭할 때 발생합니다.

이러한 이벤트는 여러 가지 용도로 사용됩니다. 예를 들어, *OnDbClick* 이벤트에 대한 핸들러를 작성하여 사용자가 열에 값을 입력하려고 할 때 가능한 값이 있는 팝업 목록이 표시되도록 할 수 있습니다. 이러한 핸들러는 *SelectedField* 속성을 사용하여 현재 행과 열을 결정합니다.

기타 data-aware 컨트롤을 포함하는 그리드 생성

TDBCtrlGrid 컨트롤은 표 모양의 그리드 형식으로 여러 레코드의 여러 필드를 표시합니다. 그리드의 각 셀은 단일 행에서 여러 필드를 표시합니다. 다음과 같은 방법으로 데이터베이스 컨트롤 그리드를 사용합니다.

- 1 데이터베이스 컨트롤 그리드를 폼에 둡니다.
- 2 그리드의 *DataSource* 속성을 데이터 소스의 이름으로 설정합니다.
- 3 그리드용 디자인 셀 내에 개별적인 데이터 컨트롤을 둡니다. 그리드용 디자인 셀은 그리드에서 맨위쪽 또는 맨왼쪽 셀이며, 다른 컨트롤들을 둘 수 있는 유일한 셀입니다.
- 4 *DataField* 속성을 필드 이름으로 설정합니다. 이러한 데이터 컨트롤에 대한 데이터 소스는 이미 데이터베이스 컨트롤 그리드의 데이터 소스로 설정되어 있습니다.
- 5 컨트롤을 원하는 셀 안에 배열합니다.

데이터베이스 컨트롤 그리드를 포함하는 애플리케이션을 컴파일하고 실행할 때 런타임 시 디자인 셀에 설정한 데이터 컨트롤의 배열은 그리드의 각 셀에 복사됩니다. 각 셀은 데이터셋의 각기 다른 레코드를 표시합니다.

그림 15.5 디자인 타임 시 TDBCtrlGrid



다음 표는 디자인 타임에 설정할 수 있는 데이터베이스 컨트롤 그리드에 대한 고유 속성 몇 가지를 요약한 것입니다.

표 15.7 선택된 데이터베이스 컨트롤 그리드 속성

속성	용도
AllowDelete	<i>True</i> (기본값): 레코드 삭제를 허용합니다. <i>False</i> : 레코드 삭제를 금지합니다.
AllowInsert	<i>True</i> (기본값): 레코드 삽입을 허용합니다. <i>False</i> : 레코드 삽입을 금지합니다.
ColCount	그리드의 열 수를 설정합니다. 기본값 = 1.
Orientation	<i>goVertical</i> (default): 레코드를 위에서 아래로 표시합니다. <i>goHorizontal</i> : Displays 레코드를 왼쪽에서 오른쪽으로 표시합니다.
PanelHeight	개별적인 패널에 대한 높이를 설정합니다. 기본값 = 72.
PanelWidth	개별적인 패널의 폭을 설정합니다. 기본값 = 200.
RowCount	표시할 패널의 수를 설정합니다. 기본값 = 3.
ShowFocus	<i>True</i> (기본값): 런타임 시 현재 레코드의 패널 주변에 포커스 직사각형을 표시합니다. <i>False</i> : 포커스 직사각형을 표시하지 않습니다.

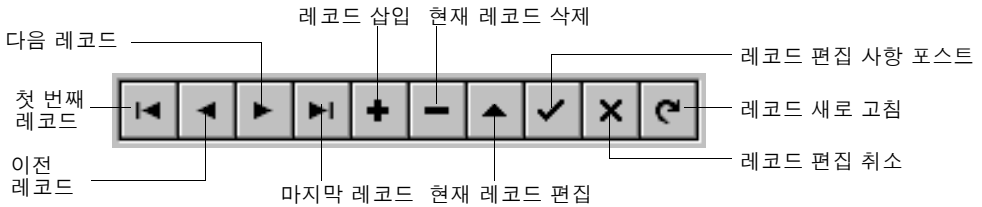
데이터베이스 컨트롤 그리드 속성 및 메소드에 대한 자세한 내용은 온라인 *VCL Reference* 를 참고하십시오.

레코드 탐색 및 처리

*TDBNavigator*는 사용자에게 데이터셋의 레코드를 탐색하고 처리하는 간단한 컨트롤을 제공합니다. 탐색기는 일련의 버튼으로 구성됩니다. 이 버튼을 이용하여 사용자는 한 레코드씩 앞으로 또는 뒤로 스크롤하고, 첫 번째 레코드 또는 마지막 레코드로 이동하며, 새로운 레코드를 삽입하고, 기존 레코드를 업데이트하고, 데이터 변경 사항을 포스트하고, 데이터 변경을 취소하고, 레코드를 삭제하며, 레코드 표시를 새로 고칠 수 있습니다.

그림 15.6은 디자인 타임 시 폼에 추가할 때 기본으로 나타나는 탐색기를 보여 줍니다. 탐색기는 사용자가 데이터셋의 한 레코드에서 다른 레코드로 이동하거나 레코드를 편집, 삭제, 삽입 및 포스트할 수 있게 해주는 일련의 버튼으로 구성됩니다. 탐색기의 *VisibleButtons* 속성을 통해 사용자가 이러한 버튼의 일부를 동적으로 숨기거나 보일 수 있습니다.

그림 15.6 TDBNavigator 컨트롤의 버튼



다음 표는 탐색기의 버튼을 설명합니다.

표 15.8 TDBNavigator 버튼

버튼	용도
첫 번째	데이터셋의 <i>First</i> 메소드를 호출하여 현재 레코드를 첫 번째 레코드로 설정합니다.
이전	데이터셋의 <i>Prior</i> 메소드를 호출하여 현재 레코드를 이전 레코드로 설정합니다.
다음	데이터셋의 <i>Next</i> 메소드를 호출하여 현재 레코드를 다음 레코드로 설정합니다.
마지막	데이터셋의 <i>Last</i> 메소드를 호출하여 현재 레코드를 마지막 레코드로 설정합니다.
삽입	데이터셋의 <i>Insert</i> 메소드를 호출하여 현재 레코드 앞에 새로운 레코드를 삽입하고 데이터셋을 Insert 상태로 설정합니다.
삭제	현재 레코드를 삭제합니다. <i>ConfirmDelete</i> 속성이 <i>True</i> 인 경우, 삭제하기 전에 사용자에게 확인을 요구합니다.
편집	데이터셋을 Edit 상태로 만들어 현재 레코드를 수정할 수 있도록 합니다.
포스트	현재 레코드의 변경 사항을 데이터베이스에 적용합니다.
취소	현재 레코드의 편집을 취소하고 데이터셋을 Browse 상태로 만듭니다.
새로 고침	데이터 컨트롤 표시 버퍼를 지우고 실제 테이블이나 쿼리로부터 버퍼를 새로 고칩니다. 원본으로 사용한 데이터가 다른 애플리케이션에 의해 변경된 경우에 유용합니다.

표시할 탐색기 버튼 선택

디자인 타임에 *TDBNavigator*를 폼에 처음으로 두면 모든 버튼이 보입니다. *VisibleButtons* 속성을 사용하면 폼에서 사용하고 싶지 않은 버튼을 보이지 않게 할 수 있습니다. 예를 들어, 단방향 데이터셋을 동작할 때 *First*, *Next* 및 *Refresh* 버튼만 사용할 수 있습니다. 편집보다는 탐색을 목적으로 하는 폼에서는 *편집*, *삽입*, *삭제*, *포스트* 및 *취소* 버튼을 사용하지 못하게 할 수도 있습니다.

디자인 타임 시 탐색기 버튼 보이기 및 숨기기

Object Inspector의 *VisibleButtons* 속성은 + 기호와 함께 표시되며 속성을 확장하여 탐색기의 각 버튼에 대한 부울 값을 표시할 수 있습니다. 이들 값을 보거나 설정하려면 + 기호를 클릭합니다. 켜거나 끌 수 있는 버튼 목록이 *VisibleButtons* 속성 아래의 Object Inspector에 나타납니다. - 기호를 클릭하면 속성 목록이 축소되면서 + 기호로 바뀝니다.

버튼의 가시성(visibility)은 버튼 값의 *Boolean* 상태에 의해 결정됩니다. 값이 *True*로 설정된 경우, 버튼은 *TDBNavigator*에 나타납니다. 값이 *False*인 경우, 버튼은 디자인 타임 및 런타임에 탐색기에서 제거됩니다.

참고 버튼 값을 *False*로 설정하면 폼의 *TDBNavigator*에서 버튼이 제거되며 남아 있는 버튼은 폭이 넓어져서 컨트롤을 채웁니다. 컨트롤의 핸들을 마우스로 끌어서 버튼의 크기를 조정합니다.

런타임 시 탐색기 버튼 보이기 및 숨기기

런타임 시 사용자의 동작이나 애플리케이션의 상태에 따라 탐색기 버튼을 보이게 하거나 숨길 수 있습니다. 예를 들어, 하나는 사용자가 레코드를 편집하도록 허용하고 다른 하나는 읽기 전용인 두 개의 다른 데이터셋의 검색을 위한 단일 탐색기를 제공한다고 가정하십시오. 데이터셋 사이를 전환할 때 읽기 전용인 데이터셋에 대해서는 탐색기의 *삽입*, *삭제*, *편집*, *포스트*, *취소* 및 *새로 고침* 버튼을 숨기고, 다른 데이터셋에 대해서는 이 버튼들을 보여 주려고 합니다.

예를 들어, 탐색기의 *삽입*, *삭제*, *편집*, *포스트*, *취소* 및 *새로 고침* 버튼을 숨겨서 *OrdersTable*에 대한 편집을 막거나 *CustomersTable*에 대해서는 편집을 허용하는 경우를 가정하십시오. *VisibleButtons* 속성은 어느 버튼을 탐색기에 표시할지를 제어합니다. 다음 코드는 *OnEnter* 이벤트 핸들러를 코딩하는 방법을 보여 줍니다.

```

procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    begin
      DBNavigatorAll.DataSource := CustomerCompany.DataSource;
      DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
    end
  else
    begin
      DBNavigatorAll.DataSource := OrderNum.DataSource;
      DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
        nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
    end;
end;

```

풍선 도움말 표시

런타임 시 각 탐색기 버튼에 대한 풍선 도움말을 표시하려면 탐색기의 *ShowHint* 속성을 *True*로 설정합니다. *ShowHint*가 *True*인 경우 마우스 커서를 탐색기 버튼 위로 가져갈 때마다 풍선 도움말 힌트가 표시됩니다. *ShowHint*는 기본값이 *False*입니다.

Hints 속성은 각 버튼에 대한 풍선 도움말 텍스트를 제어합니다. 기본적으로 *Hints*는 빈 문자열 목록입니다. *Hints*가 비어 있으면 각 탐색기 버튼은 기본 도움말 텍스트를 표시합니다. 탐색기 버튼에 사용자 지정 풍선 도움말을 제공하려면 String List Editor를 사용하여 *Hints* 속성의 각 버튼에 대해 독립된 힌트 텍스트 줄을 입력합니다. 텍스트를 입력하면 탐색기 컨트롤에서 제공한 기본 힌트 대신 사용자가 제공한 문자열이 풍선 도움말에 나타납니다.

여러 데이터셋에 단일 탐색기 사용

다른 data-aware 컨트롤에서와 같이 탐색기의 *DataSource* 속성은 컨트롤을 데이터셋에 연결하는 데이터 소스를 지정합니다. 런타임 시 탐색기의 *DataSource* 속성을 변경하면 단일 탐색기로 여러 데이터셋에 대한 레코드를 탐색하거나 처리할 수 있습니다.

각각의 *CustomersSource* 및 *OrdersSource* 데이터 소스를 통해 *CustomersTable* 및 *OrdersTable* 데이터셋에 연결된 두 개의 편집 컨트롤이 폼에 있다고 가정하십시오. 사용자가 *CustomersSource*에 연결된 편집 컨트롤을 입력하면 탐색기는 또한 *CustomersSource*를 사용할 수 있고, 사용자가 *OrdersSource*에 연결된 편집 컨트롤을 입력하면 탐색기는 또한 *OrdersSource*로 전환할 수 있습니다. 편집 컨트롤 중 하나에 대해 *OnEnter* 이벤트 핸들러를 코딩할 수 있고, 그런 다음 이 이벤트를 다른 편집 컨트롤과 공유할 수 있습니다. 예를 들면, 다음과 같습니다.

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```


16

의사 결정 지원(Decision Support) 컴포넌트 사용

의사 결정 지원 컴포넌트를 사용하여 크로스탭 테이블 및 그래프를 만들 수 있습니다. 그런 다음 이러한 테이블과 그래프를 사용하여 다른 시각에서 데이터를 보고 요약할 수 있습니다. 크로스탭 데이터에 대한 자세한 내용은 16-2 페이지의 "크로스탭"을 참조하십시오.

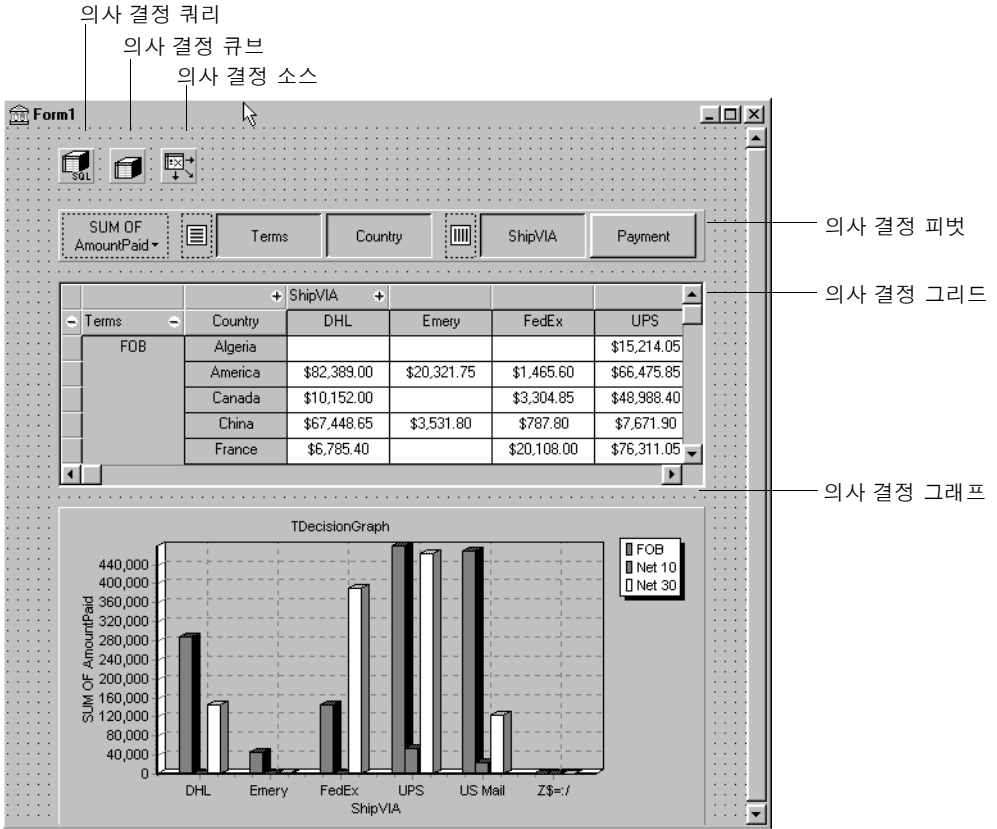
개요

의사 결정 지원 컴포넌트는 컴포넌트 팔레트의 Decision Cube 페이지에 나타납니다.

- 의사 결정 큐브(Decision Cube)인 *TDecisionCube*는 다차원의 데이터 저장소입니다.
- 의사 결정 소스(Decision Source)인 *TDecisionSource*는 의사 결정 그리드 또는 의사 결정 그래프의 현재 피벗 상태를 정의합니다.
- 의사 결정 쿼리(Decision Query)인 *TDecisionQuery*는 의사 결정 큐브에서 데이터를 정의할 때 사용하는 *TQuery*의 특수한 형태입니다.
- 의사 결정 피벗(Decision Pivot)인 *TDecisionPivot*을 사용할 경우 버튼을 눌러 의사 결정 큐브 차원이나 필드를 열거나 닫을 수 있습니다.
- 의사 결정 그리드(Decision Grid)인 *TDecisionGrid*는 1차원과 다차원 데이터를 테이블 형태로 표시합니다.
- 의사 결정 그래프(Decision Graph)인 *TDecisionGraph*는 의사 결정 그리드의 필드를 데이터 차원이 수정될 때 변경되는 동적 그래프로 표시합니다.

그림 16.1은 디자인 타임 시 폼에 놓여 있는 모든 의사 결정 지원 컴포넌트를 보여 줍니다.

그림 16.1 디자인 타임의 의사 결정 지원 컴포넌트



크로스탭

크로스탭은 관계와 흐름이 더 잘 보이도록 데이터의 서브셋을 나타내는 방법입니다. 테이블 필드는 크로스탭의 차원이 되며 필드 값은 차원 내의 범주와 요약을 정의합니다.

의사 결정 지원 컴포넌트를 사용하면 폼에 크로스탭을 설정할 수 있습니다. *TDecisionGrid*는 데이터를 테이블로 표시하는 반면 *TDecisionGraph*는 그래프로 표시합니다. *TDecisionPivot*에는 더 쉽게 차원을 표시하고 숨기고 열과 행 간에 이동할 수 있게 하는 버튼이 있습니다.

크로스탭은 1차원이나 다차원 모두 가능합니다.

1차원 크로스탭

1차원 크로스탭은 1차원의 범주에 대한 요약 행(또는 열)을 보여 줍니다. 예를 들어, 열 차원으로 Payment가 선택되고 Amount Paid가 요약 범주인 경우에 그림 16.2의 크로스탭은 각 지불 방식에 따라 지불되는 금액을 표시합니다.

그림 16.2 1차원 크로스탭

	AmEx	Cash	Check	COD	Credit	MC
Payment	\$134,753.40	\$164,003.65	\$270,492.15	\$33,776.55	\$1,332,430.25	\$250,163.25

다차원 크로스탭

다차원 크로스탭은 행이나 열에 대해 추가적인 차원을 사용합니다. 예를 들면, 2차원 크로스탭은 각 나라의 지불 방식에 따라 지불되는 금액을 표시할 수 있습니다.

3차원 크로스탭은 그림 16.3에서처럼 각 나라의 지불 방식과 조건에 따라 지불되는 금액을 표시할 수 있습니다.

그림 16.3 3차원 크로스탭

Terms	Country	Check	COD	Credit	MC
FOB	Algeria	\$2,577.95		\$1,400.00	\$13,814.05
	America			\$356,816.20	\$20,881.35
	Canada			\$24,485.00	\$3,304.85
	China	\$61,936.90		\$6,641.55	

의사 결정 지원 컴포넌트 사용 지침

16-1 페이지에 나열된 의사 결정 지원 컴포넌트는 다차원 데이터를 테이블과 그래프로 나타내는 데 사용할 수 있습니다. 둘 이상의 그리드나 그래프를 각 데이터셋에 연결할 수 있습니다. 두 개 이상의 *TDecisionPivot* 인스턴스는 런타임 시 다른 시각으로 데이터를 표시하는 데 사용할 수 있습니다.

다차원 데이터의 테이블과 그래프를 갖는 폼을 만들려면 다음 단계를 따릅니다.

- 1 폼을 생성합니다.
- 2 다음과 같은 컴포넌트들을 폼에 추가하고 Object Inspector를 사용하여 바인딩합니다.

- 데이터셋은 일반적으로 *TDecisionQuery*(자세한 내용은 16-6 페이지의 "Decision Query 에디터로 의사 결정 데이터셋 생성"을 참조) 또는 *TQuery*입니다.
 - 의사 결정 큐브인 *TDecisionCube*는 이 컴포넌트의 *DataSet* 속성을 데이터셋의 이름으로 설정하여 데이터셋에 바인딩합니다.
 - 의사 결정 소스인 *TDecisionSource*는 이 컴포넌트의 *DecisionCube* 속성을 의사 결정 큐브의 이름으로 설정하여 의사 결정 큐브에 바인딩합니다.
- 3** 의사 결정 피벗인 *TDecisionPivot*을 추가하고 Object Inspector로 *DecisionSource* 속성을 해당 의사 결정 소스 이름으로 설정하여 의사 결정 소스에 바인딩합니다. 의사 결정 피벗은 옵션이지만 유용합니다. 이 피벗을 통해 폼 개발자와 최종 사용자는 버튼을 눌러 의사 결정 그리드나 의사 결정 그래프에 표시된 차원을 변경할 수 있습니다.
- 기본 방향인 수평에서 의사 결정 피벗의 왼쪽 버튼은 의사 결정 그리드(행)의 왼쪽 필드에 적용됩니다. 오른쪽 버튼은 의사 결정 그리드(열)의 맨 위쪽 필드에 적용됩니다.
- GroupLayout* 속성을 *xtVertical*, *xtLeftTop*, *xtHorizontal*(기본값) 등으로 설정하여 의사 결정 피벗의 버튼이 나타나는 위치를 결정할 수 있습니다. 의사 결정 피벗 속성에 대한 자세한 내용은 16-9 페이지의 "의사 결정 피벗(Decision Pivot) 사용"을 참조하십시오.
- 4** 의사 결정 소스에 바인딩되는 하나 이상의 의사 결정 그리드 및 그래프를 추가합니다. 자세한 내용은 16-10 페이지의 "의사 결정 그리드(Decision Grid) 생성 및 사용" 및 16-13 페이지의 "의사 결정 그래프(Decision Graph) 생성 및 사용"을 참조하십시오.
- 5** Decision Query 에디터 또는 *TDecisionQuery*(또는 *TQuery*)의 *SQL* 속성을 사용하여 그리드나 그래프에 표시할 테이블, 필드 및 요약을 지정합니다. SQL SELECT의 마지막 필드는 요약 필드여야 합니다. SELECT의 다른 필드는 GROUP BY 필드여야 합니다. 자세한 내용은 16-6 페이지의 "Decision Query 에디터로 의사 결정 데이터셋 생성"을 참조하십시오.
- 6** 의사 결정 쿼리 또는 대체 데이터셋 컴포넌트의 *Active* 속성을 *True*로 설정합니다.
- 7** 의사 결정 그리드와 그래프를 사용하여 다른 데이터 차원을 표시하고 차트로 만듭니다. 자세한 내용은 16-11 페이지의 "의사 결정 그리드 사용"과 16-13 페이지의 "의사 결정 그래프 사용"을 참조하십시오.
- 폼에 사용할 수 있는 모든 의사 결정 지원 컴포넌트에 대한 그림은 16-2 페이지의 그림 16.1을 참조하십시오.

의사 결정 지원 컴포넌트에 데이터셋 사용

데이터셋에 직접 바인딩하는 유일한 의사 결정 지원 컴포넌트는 의사 결정 큐브인 *TdecisionCube*입니다. *TDecisionCube*는 적용 가능한 형식의 SQL 문에서 정의한 그룹 데이터와 요약 데이터를 받습니다. GROUP BY 절에는 SELECT 절과 동일하게 요약 필드 이외의 필드가 같은 순서로 있어야 하고 요약 필드는 식별되어야 합니다.

의사 결정 쿼리 컴포넌트인 *TDecisionQuery*는 *TQuery*의 특별한 형태입니다. *TDecisionQuery*를 사용하면 의사 결정 큐브(*TDecisionCube*)에 데이터를 제공하기 위해 사용되는 차원(행과 열)과 요약 값의 설정을 더 간단하게 정의할 수 있습니다. 일반적인 *TQuery*나 다른 BDE 활성 데이터셋을 *TDecisionCube*의 데이터셋으로 사용할 수도 있지만 데이터셋과 *TDecisionCube*를 올바르게 설정하는 것은 디자이너의 책임입니다.

의사 결정 큐브로 올바르게 작업하려면 데이터셋에 반영된 모든 필드가 차원 또는 요약이어야 합니다. 요약은 합이나 카운트 같은 추가적인 값이어야 하고 각 차원 값의 조합에 대한 합계를 나타내야 합니다. 설정을 최대한 쉽게 하기 위해서는 데이터셋에서 합계의 이름을 "Sum..."으로 하고 카운트는 "Count..."로 해야 합니다.

의사 결정 큐브는 셀이 추가적인 요약인 경우에만 정확하게 피벗하고, 부분 합계를 내고, 드릴 인(drill-in)할 수 있습니다. (SUM과 COUNT는 추가할 수 있지만 AVERAGE, MAX, MIN은 추가할 수 없습니다.) 피벗 만들기 크로스탭은 추가적인 집계자를 포함하는 그리드일 때만 표시됩니다. 추가할 수 없는 집계자를 사용할 경우에는 피벗 또는 드릴하거나 부분 합계를 내지 않는 정적인 의사 결정 그리드를 사용하십시오.

평균은 SUM을 COUNT로 나누어 계산될 수 있기 때문에 피벗하는 평균은 필드에 대한 SUM과 COUNT 차원이 데이터셋에 포함될 때 자동으로 추가됩니다. AVERAGE 문을 사용하여 계산되는 평균보다는 이렇게 계산되는 평균을 사용하십시오.

평균은 COUNT(*)를 사용하여 계산할 수도 있습니다. COUNT(*)를 사용하여 평균을 계산하려면 쿼리에 "COUNT(*) COUNTALL" 선택자(selector)를 포함하십시오. COUNT(*)를 사용하여 평균을 계산할 경우 단일 집계자(aggregator)를 모든 필드에 사용할 수 있습니다. 요약 중인 모든 필드에 빈 값이 없거나 COUNT 집계자를 어떤 필드에서도 사용할 수 없을 경우에만 COUNT(*)를 사용하십시오.

TQuery 또는 TTable로 의사 결정 데이터셋 생성

보통의 *TQuery* 컴포넌트를 의사 결정 데이터 셋으로 사용할 경우 SQL 문을 직접 설정해야 하는데 SELECT 절과 같은 필드가 동일한 순서로 포함된 GROUP BY 절을 제공해야 합니다.

SQL은 다음과 같아야 합니다.

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
       ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

SELECT 필드의 순서는 GROUP BY 필드의 순서와 일치해야 합니다.

*TTable*의 경우, 쿼리의 필드 중에서 어떤 필드가 그룹 필드이고 어느 것이 요약 필드인지에 대한 정보를 의사 결정 큐브에 제공해야 합니다. 이렇게 하려면 의사 결정 큐브의 *DimensionMap*에 있는 각 필드의 Dimension Type을 채웁니다. 각 필드가 차원인지 요약인지를 나타내고 요약인 경우 요약 타입을 제공해야 합니다. 피벗하는 평균은 SUM/COUNT 계산에 의존하므로 의사 결정 큐브가 SUM과 COUNT 집계자 쌍에 맞도록 기본 필드 이름도 제공해야 합니다.

Decision Query 에디터로 의사 결정 데이터셋 생성

의사 결정 지원 컴포넌트에서 사용하는 모든 데이터는 의사 결정 큐브를 거치는데 SQL 쿼리로 가장 쉽게 만든 특별한 형식의 데이터셋을 받아들입니다. 자세한 내용은 16-4 페이지의 "의사 결정 지원 컴포넌트에 데이터셋 사용"을 참조하십시오.

*TTable*과 *TQuery* 모두 의사 결정 데이터셋으로 사용될 수 있지만 *TDecisionQuery* 를 사용하는 것이 더 쉽습니다. 함께 들어 있는 Decision Query 에디터는 의사 결정 큐브에 나타낼 테이블, 필드, 요약물 지정하는 데 사용할 수 있으며 SQL의 SELECT와 GROUP BY 부분을 정확하게 설정할 수 있습니다.

다음과 같은 방법으로 Decision Query 에디터를 사용합니다.

- 1 폼에서 의사 결정 쿼리 컴포넌트를 선택한 다음 마우스 오른쪽 버튼을 클릭하여 Decision Query 에디터를 선택합니다. Decision Query 에디터 대화 상자가 나타납니다.
- 2 사용할 데이터베이스를 선택합니다.
- 3 단일 테이블 쿼리의 경우에는 Select Table 버튼을 클릭합니다.
여러 테이블의 조인을 포함하는 복잡한 쿼리의 경우 Query Builder 버튼을 클릭하여 SQL Builder를 표시하거나 SQL 탭 페이지의 편집 상자에 SQL 문을 입력합니다.
- 4 Decision Query 에디터 대화 상자로 돌아옵니다.
- 5 Decision Query 에디터 대화 상자의 Available Fields 리스트 박스에서 필드를 선택하고 해당 오른쪽 화살표 버튼을 클릭하여 Dimensions 또는 Summaries에 할당합니다. Summaries 목록에 필드 추가할 때 사용할 요약 타입에 표시된 메뉴에서 합계, 카운트, 평균 중 하나를 선택하십시오.
- 6 기본적으로 의사 결정 쿼리의 SQL 속성에 정의된 모든 필드와 요약은 Active Dimensions와 Active Summaries 리스트 박스에 나타납니다. 차원이나 요약물 제거하려면 목록에서 선택하고 목록 옆에 있는 왼쪽 화살표를 클릭하거나 제거할 항목을 더블 클릭합니다. 다시 추가하려면 Available Fields 리스트 박스에서 선택하고 해당 오른쪽 화살표를 클릭합니다.

의사 결정 큐브의 내용을 정의했다면 *DimensionMap* 속성과 *TDecisionPivot* 버튼으로 차원 표시를 더 자세하게 처리할 수 있습니다. 자세한 내용은 다음 단원 "의사 결정 큐브 (Decision Cube) 사용", 16-8 페이지의 "의사 결정 소스 (Decision Source) 사용", 16-9 페이지의 "의사 결정 피벗 (Decision Pivot) 사용"을 참조하십시오.

참고 Decision Query 에디터를 사용할 때 쿼리는 초기에 ANSI-92 SQL 구문으로 처리된 다음 필요한 경우 서버가 사용하는 것으로 번역되어야 합니다. Decision Query 에디터는 ANSI 표준 SQL만 읽고 표시합니다. 이러한 번역은 *TDecisionQuery*의 SQL 속성에 자동으로 할당됩니다. 쿼리를 수정하려면 SQL 속성보다는 Decision Query에서 ANSI-92 버전을 편집하십시오.

의사 결정 큐브(Decision Cube) 사용

의사 결정 큐브 컴포넌트인 *TDecisionCube*는 데이터셋에서 데이터를 페치 (fetch) 하는 다차원 데이터 저장소입니다(대개 *TDecisionQuery* 또는 *TQuery*를 통해 특수한 구조의 SQL 문이 입력됩니다.). 데이터는 쿼리를 두 번 실행할 필요 없이 쉽게 피벗(즉, 데이터를 구성하고 요약하는 방식을 변경)할 수 있게 만드는 형태로 저장됩니다.

의사 결정 큐브 속성과 이벤트

*TDecisionCube*의 *DimensionMap* 속성을 사용하면 나타나는 차원과 요약을 제어할 수 있을 뿐만 아니라 날짜 범위를 설정하고 의사 결정 큐브가 지원하는 차원의 최대 수치를 지정할 수 있습니다. 디자인하는 동안 데이터를 표시할 것인지 여부도 지시할 수 있습니다. 이름, (별주) 값, 부분 합계, 데이터를 표시할 수 있습니다. 디자인 타임에 데이터를 표시하는 것은 데이터 소스에 따라 시간이 걸릴 수도 있습니다.

Object Inspector에서 *DimensionMap* 옆에 있는 생략 부호를 클릭하면 Decision Cube 에디터 대화 상자가 나타납니다. 페이지와 컨트롤을 사용하여 *DimensionMap* 속성을 설정할 수 있습니다.

의사 결정 큐브 캐시를 다시 만들 때마다 *OnRefresh* 이벤트가 일어납니다. 개발자는 새 차원 맵에 액세스하고 메모리를 해제할 때 변경할 수 있으며 요약이나 차원의 최대값 등을 변경할 수 있습니다. *OnRefresh*는 사용자가 Decision Cube 에디터에 액세스하는 경우에도 유용합니다. 애플리케이션 코드는 사용자의 변경 사항에 대해 즉각적으로 응답할 수 있습니다.

Decision Cube 에디터 사용

Decision Cube 에디터를 사용하여 의사 결정 큐브의 *DimensionMap* 속성을 설정할 수 있습니다. 앞에서 설명한 대로 Object Inspector를 통해 Decision Cube 에디터를 표시할 수도 있고 디자인 타임 시 의사 결정 큐브를 마우스 오른쪽 버튼으로 클릭하고 Decision Cube 에디터를 선택하여 표시할 수도 있습니다.

Decision Cube 에디터 대화 상자에는 다음 두 개 탭이 있습니다.

- Dimension Settings는 사용할 수 있는 차원을 활성화 또는 비활성화하고, 차원의 이름과 형식을 바꾸고, 차원을 영구적으로 드릴된 상태에 놓고, 표시할 날짜 범위를 설정하는 데 사용됩니다.
- Memory Control은 한 번에 활성화될 수 있는 차원과 요약의 최대값을 설정하고, 메모리 사용에 관한 정보를 표시하고, 디자인 타임 시 나타나는 이름과 데이터를 결정하는 데 사용됩니다.

차원 설정의 보기 및 변경

차원 설정을 보려면 Decision Cube 에디터를 표시하고 Dimension Settings 탭을 클릭합니다. 그런 다음 Available Fields 목록에서 차원이나 요약을 선택합니다. 에디터의 오른쪽에 있는 상자에 해당 정보가 나타납니다.

- 의사 결정 피벗, 의사 결정 그리드, 의사 결정 그래프 등에 나타나는 차원이나 요약 이름을 변경하려면 Display Name 편집 상자에 새 이름을 입력합니다.
- 선택된 필드가 차원인지 요약인지를 파악하려면 Type 편집 상자에 있는 텍스트를 읽습니다. 데이터셋이 *TTable* 컴포넌트이면 선택된 필드가 차원인지 요약인지를 지정하기 위해 Type을 사용할 수 있습니다.
- 선택된 차원이나 요약을 사용할 수 없게 하거나 활성화하려면 Active Type 드롭다운 리스트 박스에서 설정을 Active, As Needed 또는 Inactive 중 하나로 변경합니다. 차원을 사용하지 않거나 As Needed로 설정하면 메모리가 절약됩니다.
- 차원이나 요약의 서식을 바꾸려면 Format 편집 상자에 서식 문자열을 입력하십시오.
- 차원이나 요약을 Year, Quarter, Month 등을 기준으로 표시하려면 Binning 드롭다운 리스트 박스에서 설정을 변경합니다. Binning 리스트 박스에서 Set을 선택하여 선택된 차원이나 요약을 영구적으로 "드릴 다운된" 상태로 놓을 수 있습니다. 이것은 차원이 많은 값을 가질 때 메모리를 절약하는 데 유용하게 사용됩니다. 자세한 내용은 16-19 페이지의 "의사 결정 지원(Decision Support) 컴포넌트와 메모리 제어"를 참조하십시오.
- 범위의 시작 값이나 "Set" 차원의 드릴 다운 값을 결정하려면 먼저 Grouping 드롭다운에서 해당 Grouping 값을 선택한 다음 Initial Value 드롭다운 목록에 시작 범위 값이나 영구 드릴 다운 값을 입력합니다.

사용 가능한 차원과 요약의 최대값 설정

선택된 의사 결정 큐브에 바인딩된 의사 결정 피벗, 의사 결정 그리드, 의사 결정 그래프에 사용할 수 있는 차원과 요약의 최대값을 결정하려면 Decision Cube 에디터를 표시하고 Memory Control 탭을 클릭합니다. 필요에 따라서는 편집 컨트롤을 사용하여 현재 설정을 조정합니다. 이 설정으로 의사 결정 큐브에서 필요한 메모리 양을 제어할 수 있습니다. 자세한 내용은 16-19 페이지의 "의사 결정 지원(Decision Support) 컴포넌트와 메모리 제어"를 참조하십시오.

디자인 옵션의 보기 및 변경

디자인 타임 시 얼마나 많은 정보가 나타나는지 결정하려면 Decision Cube 에디터를 표시하고 Memory Control 탭을 클릭합니다. 그런 다음 표시할 이름과 데이터를 나타내는 설정을 선택합니다. 디자인 타임 시 데이터나 필드 이름을 표시하는 것은 데이터를 폐치하는 데 필요로 하는 시간 때문에 경우에 따라서는 성능을 지연시킬 수 있습니다.

의사 결정 소스(Decision Source) 사용

의사 결정 소스 컴포넌트인 *TDecisionSource*는 의사 결정 그리드나 의사 결정 그래프의 현재 피벗 상태를 정의합니다. 동일한 의사 결정 소스를 사용하는 두 객체는 피벗 상태도 공유합니다.

속성과 이벤트

다음은 의사 결정 소스의 외관과 동작을 제어하는 몇 가지 특수한 속성과 이벤트입니다.

- *TDecisionSource*의 *ControlType* 속성은 의사 결정 피벗 버튼이 체크 박스(동시에 여러 개 선택)나 라디오 버튼(상호 배타적 선택)처럼 동작하는지를 나타냅니다.
- *TDecisionSource*의 *SparseCols*와 *SparseRows* 속성은 값이 없는 열이나 행을 표시할 것인지를 나타냅니다. 값이 *True*라면 희소(sparse) 열이나 행이 표시됩니다.
- *TDecisionSource*에는 다음과 같은 이벤트가 있습니다.
 - *OnLayoutChange*는 사용자가 데이터를 재구성하는 피벗이나 드릴 다운을 수행할 때 발생합니다.
 - *OnNewDimensions*는 요약이나 차원 필드가 변경될 때처럼 데이터가 완전히 변경될 때 발생합니다.
 - *OnSummaryChange*는 현재 요약이 변경될 때 발생합니다.
 - *OnStateChange*는 의사 결정 큐브가 활성화되거나 활성화되지 않을 때 발생합니다.
 - *OnBeforePivot*은 변경 사항이 커밋되었지만 아직 사용자 인터페이스에 반영되지 않았을 때 발생합니다. 개발자는 애플리케이션 사용자가 이전 작업의 결과를 보기 전에 용량이나 피벗 상태 등에 대해 변경할 기회를 갖습니다.
 - *OnAfterPivot*은 피벗 상태가 변경된 다음 발생합니다. 개발자는 그 때 정보를 포착할 수 있습니다.

의사 결정 피벗(Decision Pivot) 사용

의사 결정 피벗 컴포넌트인 *TDecisionPivot*을 사용하면 버튼을 눌러 의사 결정 큐브 차원이나 필드를 열거나 닫을 수 있습니다. *TDecisionPivot* 버튼을 눌러 행이나 열이 열릴 경우 해당 차원이 *TDecisionGrid* 또는 *TDecisionGraph* 컴포넌트에 나타납니다. 차원이 닫혀 있으면 자세한 데이터가 나타나지 않으며 다른 차원의 합계에 묻힙니다. 차원은 차원 필드의 특정 값에 대한 요약만 나타나는 "드릴된" 상태일 수도 있습니다.

의사 결정 그리드와 의사 결정 그래프에 표시된 차원을 재구성하는 데 의사 결정 피벗을 사용할 수도 있습니다. 단지 버튼을 행이나 열 영역으로 끌거나 동일한 영역 내의 버튼을 재정렬합니다.

디자인 타임의 의사 결정 피벗을 그림으로 보려면 그림 16.1, 16.2, 16.3을 참조하십시오.

의사 결정 피벗 속성

다음은 의사 결정 피벗의 외관과 동작을 제어하는 특별한 속성입니다.

- *TDecisionPivot*에 대해 나열된 첫 속성들은 전체적인 동작과 외관을 정의합니다. *TDecisionPivot*에 대한 *ButtonAutoSize*를 *False*로 설정하면 컴포넌트의 크기를 조정하듯이 버튼을 크거나 작게 할 수 있습니다.
- *TDecisionPivot*의 *Groups* 속성은 어떤 차원 버튼이 나타나는지를 정의합니다. 행, 열, 요약 선택 버튼 그룹을 어떠한 조합으로도 표시할 수 있습니다. 이러한 그룹의 배치를 더 유연하게 하기 위해 폼의 한 위치에는 행만 포함된 하나의 *TDecisionPivot*을 배치하고 다른 위치에는 열만 포함된 두 번째 것을 배치할 수 있습니다.
- 일반적으로 *TDecisionPivot*은 *TDecisionGrid* 위에 추가됩니다. 기본 방향인 수평에서 *TDecisionPivot* 왼쪽에 있는 버튼은 *TDecisionGrid* 왼쪽에 있는 필드(행)에 적용됩니다. 오른쪽에 있는 버튼은 *TDecisionGrid*의 위쪽에 있는 필드(열)에 적용됩니다.
- *GroupLayout* 속성을 *xtVertical*, *xtLeftTop*, *xtHorizontal*(기본값) 중 하나로 설정하여 *TDecisionPivot*의 버튼이 나타나는 위치를 결정할 수 있습니다.

의사 결정 그리드(Decision Grid) 생성 및 사용

의사 결정 그리드 컴포넌트인 *TDecisionGrid*는 크로스탭 데이터를 테이블 형태로 나타냅니다. 이러한 테이블을 16-2 페이지에서 설명하듯이 크로스탭이라고도 합니다. 16-2 페이지의 그림 16.1은 디자인 타임 시 폼의 의사 결정 그리드를 보여 주는 것입니다.

의사 결정 그리드 생성

다음과 같은 방법으로 하나 이상의 크로스탭 데이터의 테이블을 갖는 폼을 만듭니다.

- 1 16-3 페이지의 "의사 결정 지원 컴포넌트 사용 지침"에 나열된 1-3 단계를 따릅니다.
- 2 하나 이상의 의사 결정 그리드 컴포넌트(*TDecisionGrid*)를 추가하고 Object Inspector에서 의사 결정 소스인 *TDecisionSource*의 *DecisionSource* 속성을 해당 의사 결정 소스 컴포넌트로 설정해서 의사 결정 소스인 *TDecisionSource*에 바인딩합니다.
- 3 "의사 결정 지원 컴포넌트 사용 지침"에 나열된 5-7 단계를 계속 진행하십시오.

의사 결정 그리드에 무엇이 나타나고 어떻게 사용하는지에 대한 설명은 16-11 페이지의 "의사 결정 그리드 사용"을 참조하십시오.

폼에 그래프를 추가하려면 16-13 페이지의 "의사 결정 그래프 생성"의 지침을 따릅니다.

의사 결정 그리드 사용

의사 결정 그리드 컴포넌트인 *TDecisionGrid*는 의사 결정 소스(*TDecisionSource*)에 바인딩된 의사 결정 큐브(*TDecisionCube*)의 데이터를 표시합니다.

기본적으로 그리드는 데이터셋에 정의된 그룹화 방법에 따라 왼쪽이나 위쪽에 차원 필드와 함께 나타납니다. 각 필드의 아래에 범주가 데이터 값마다 하나씩 나타납니다. 다음 작업을 할 수 있습니다.

- 차원 열기와 닫기
- 행과 열의 재구성 또는 피벗
- 자세한 내용 드릴 다운
- 각각의 축에 차원을 하나만 선택하도록 제한

의사 결정 그리드의 특별한 속성과 이벤트에 대한 자세한 내용은 16-12 페이지의 "의사 결정 그리드 속성"을 참조하십시오.

의사 결정 그리드 필드 열기와 닫기

차원 또는 요약 필드에 더하기 기호(+)가 나타나면 오른쪽에 있는 필드가 하나 이상 닫히거나 숨겨집니다. 기호를 클릭하면 추가적인 필드와 범주를 열 수 있습니다. 빼기 기호(-)는 완전히 열려 있거나 확장된 필드를 나타냅니다. 이 기호를 클릭하면 필드가 닫힙니다. 이러한 기능은 사용하지 못할 수도 있습니다. 자세한 내용은 16-12 페이지의 "의사 결정 그리드 속성"을 참조하십시오.

의사 결정 그리드에서 행과 열 재구성

행과 열의 헤더를 끌어서 동일한 축이나 다른 축의 새 위치에 놓을 수 있습니다. 이러한 방법으로 그리드를 재구성하고 데이터 그룹화가 바뀔 때 새로운 관점에서 데이터를 볼 수 있습니다. 이러한 피벗 기능은 사용하지 못하게 할 수도 있습니다. 자세한 내용은 16-12 페이지의 "의사 결정 그리드 속성"을 참조하십시오.

의사 결정 피벗을 포함시키면 해당 버튼을 누르고 끌어 표시를 재구성할 수 있습니다. 지침은 16-9 페이지의 "의사 결정 피벗(Decision Pivot) 사용"을 참조하십시오.

의사 결정 그리드의 자세한 내용 드릴 다운

드릴 다운하면 차원을 자세히 볼 수 있습니다.

예를 들어, 다른 차원들이 그 밑에 감춰진 차원에서 범주 레이블(행 헤더)을 마우스 오른쪽 버튼으로 클릭하면 드릴 다운하고 해당 범주의 데이터만 보도록 선택할 수 있습니다. 차원이 드릴되면 단일 범주 값에 대한 레코드만 표시되기 때문에 그리드에 표시된 해당 차원의 범주 레이블은 보이지 않습니다. 폼에 의사 결정 피벗이 있으면 범주 값이 표시되고 원할 경우 다른 값으로 변경할 수 있습니다.

다음과 같은 방법으로 차원을 드릴 다운 합니다.

- 범주 레이블을 마우스 오른쪽 버튼으로 클릭하고 Drill In To This Value를 선택하거나
- 피벗 버튼을 마우스 오른쪽 버튼으로 클릭하고 Drilled In을 선택합니다.

다음과 같은 방법으로 전체 차원을 다시 활성화합니다.

- 해당 피벗 버튼을 마우스 오른쪽 버튼으로 클릭하거나 왼쪽 위 모서리에 있는 의사 결정 그리드를 마우스 오른쪽 버튼으로 클릭하고 차원을 선택합니다.

의사 결정 그리드의 차원 선택 제한

그리드의 각 축에 대해 두 개 이상의 차원을 선택할 수 있는가를 결정하기 위해 의사 결정 소스의 *ControlType* 속성을 변경할 수 있습니다. 자세한 내용은 16-8 페이지의 "의사 결정 소스 (Decision Source) 사용"을 참조하십시오.

의사 결정 그리드 속성

의사 결정 그리드 컴포넌트인 *TDecisionGrid*는 *TDecisionSource*에 바인딩된 *TDecisionCube* 컴포넌트의 데이터를 표시합니다. 기본적으로 데이터는 그리드의 왼쪽과 위쪽에 범주 필드가 표시된 그리드에 나타납니다.

다음은 의사 결정 그리드의 외관과 동작을 제어하는 특별한 속성들입니다.

- *TDecisionGrid*는 각 차원에 대해 고유한 속성을 갖습니다. 이를 설정하려면 Object Inspector에서 *Dimensions*를 선택한 다음 차원을 선택합니다. 그러면 Object Inspector에 속성이 나타납니다. *Alignment*는 해당 차원에 대한 범주 레이블의 정렬을 정의하며, *Caption*은 기본 차원 이름을 오버라이드하는 데 사용될 수 있고, *Color*는 범주 레이블의 색상을 정의하고, *FieldName*은 활성화 차원의 이름을 표시하고, *Format*은 해당 데이터 타입의 표준 형식을 가질 수 있으며, *Subtotals*는 해당 차원의 부분 합계를 표시할지 여부를 나타냅니다. 요약 필드가 있으면 이러한 동일한 속성들은 그리드의 요약 영역에 나타나는 데이터의 모습을 변경하기 위해 사용할 수 있습니다. 차원 속성을 설정할 경우에는 폼에서 컴포넌트를 클릭하거나 Object Inspector의 위쪽에 있는 드롭다운 리스트 박스에서 컴포넌트를 선택합니다.
- *TDecisionGrid*의 *Options* 속성을 사용하면 그리드 라인 (*cgGridLines = True*)의 표시를 조절할 수 있고, 윤곽 기능 (+ 와 - 지시자로 차원의 축소와 확장, *cgOutliner = True*)을 사용할 수 있고, 드래그 앤 드롭 피벗 (*cgPivotable = True*)을 사용할 수 있습니다.
- *TDecisionGrid*의 *OnDecisionDrawCell* 이벤트는 각 셀의 외관을 그리는 대로 변경할 수 있게 해줍니다. 이벤트는 현재 셀의 *String*, *Font*, *Color*를 참조 매개변수로 전달합니다. 음수 값에 대해 특별한 색을 주는 것과 같은 효과를 얻기 위해 그러한 매개변수를 변경할 수 있습니다. *TCustomGrid*에 의해 전달되는 *DrawState* 외에도 이벤트는 그려지는 셀이 어느 타입인지를 결정하는 데 사용할 수 있는 *TDecisionDrawState*를 전달합니다. *Cells*, *CellValueArray* 또는 *CellDrawState* 함수를 사용하여 셀에 대한 자세한 내용을 폐지할 수 있습니다.

- *TDecisionGrid*의 *OnDecisionExamineCell* 이벤트는 마우스 오른쪽 버튼 클릭 이벤트를 데이터 셀에 연결하기 위해 사용할 수 있으며 프로그램이 해당 특정 데이터 셀에 관한 정보(예: 상세 레코드)를 표시하도록 할 수 있습니다. 사용자가 데이터 셀을 마우스 오른쪽 버튼으로 클릭하면 이벤트에 현재 활성 요약 값과 요약 값을 만들기 위해 사용한 모든 차원 값의 *ValueArray*를 포함하여 데이터 값을 구성하기 위해 사용된 모든 정보가 제공됩니다.

의사 결정 그래프(Decision Graph) 생성 및 사용

의사 결정 그래프 컴포넌트인 *TDecisionGraph*는 크로스탭 데이터를 그래프 형태로 나타냅니다. 각각의 의사 결정 그래프는 Sum, Count, Avg와 같은 단일 요약 값을 표시하며 하나 이상의 차원을 가진 차트로 작성되었습니다. 크로스탭에 대한 자세한 내용은 16-3 페이지를 참조하십시오. 디자인 타임 시 의사 결정 그래프의 그림은 16-2 페이지의 그림 16.1과 16-14 페이지의 그림 16.4를 참조하십시오.

의사 결정 그래프 생성

다음과 같은 방법으로 하나 이상의 의사 결정 그래프를 갖는 폼을 만듭니다.

- 1 16-3 페이지의 "의사 결정 지원 컴포넌트 사용 지침"에 나오는 1-3 단계를 따릅니다.
- 2 하나 이상의 의사 결정 그래프 컴포넌트(*TDecisionGraph*)를 추가하고 Object Inspector에서 *DecisionSource* 속성을 해당 의사 결정 소스 컴포넌트로 설정해서 의사 결정 소스인 *TDecisionSource*에 바인딩합니다.
- 3 "의사 결정 지원 컴포넌트 사용 지침"에 나오는 5-7 단계를 계속 진행하십시오.
- 4 마지막으로 그래프를 마우스 오른쪽 버튼으로 클릭하고 Edit Chart를 선택하여 그래프 시리즈의 외관을 수정합니다. 각 그래프 차원의 템플릿 속성을 설정한 다음 각 시리즈 속성을 설정하여 이 기본값을 오버라이드합니다. 자세한 내용은 16-15 페이지의 "의사 결정 그래프 사용자 지정"을 참조하십시오.

의사 결정 그래프에 무엇이 나타나고 어떻게 사용하는지에 대한 설명은 다음 단원 "의사 결정 그래프 사용"을 참조하십시오.

폼에 의사 결정 그리드 또는 크로스탭 테이블을 추가하려면 16-10 페이지의 "의사 결정 그리드(Decision Grid) 생성 및 사용"의 지침을 따르십시오.

의사 결정 그래프 사용

의사 결정 그래프 컴포넌트인 *TDecisionGraph*는 의사 결정 소스(*TDecisionSource*)의 필드를 데이터 차원이 열리고, 닫히고, 드래그되고, 드롭되고, 의사 결정 피벗(*TDecisionPivot*)으로 재정렬될 때 변경되는 동적 그래프로 표시합니다.

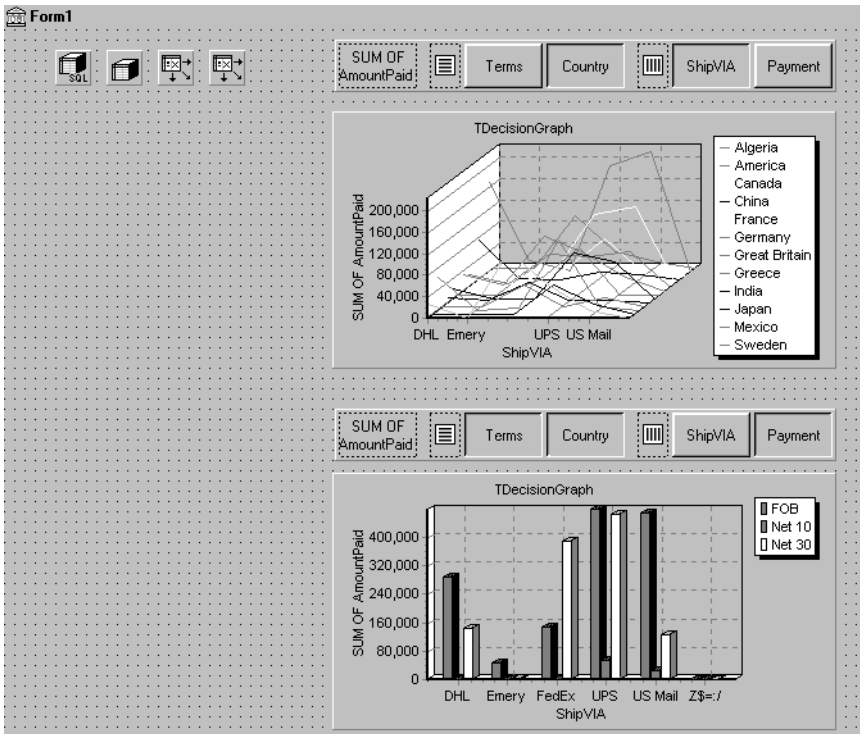
그래프화된 데이터는 *TDecisionQuery*와 같은 특수한 형식의 데이터셋에서 가져옵니다. 의사 결정 지원 컴포넌트가 이 데이터를 처리하고 배열하는 방법의 개요는 16-1 페이지를 참조하십시오.

기본적으로 첫 번째 행 차원은 x축으로 나타나고 첫 번째 열 차원은 y축으로 나타납니다.

크로스탭 데이터를 표 형태로 나타내는 의사 결정 그리드 대신 의사 결정 그래프를 사용할 수 있습니다. 동일한 의사 결정 소스에 바인딩된 의사 결정 그리드와 의사 결정 그래프는 동일한 데이터 차원을 나타냅니다. 동일한 차원에 대한 다른 요약 데이터를 표시하기 위해 둘 이상의 의사 결정 그래프를 동일한 의사 결정 소스에 바인딩할 수 있습니다. 다른 차원을 표시하려면 의사 결정 그래프를 다른 의사 결정 소스에 바인딩하십시오.

예를 들어, 그림 16.4에서 첫 번째 의사 결정 피벗과 그래프는 첫 번째 의사 결정 소스에 바인딩되고 두 번째 의사 결정 피벗과 그래프는 두 번째 의사 결정 소스에 바인딩됩니다. 따라서 각 그래프는 다른 차원을 표시할 수 있습니다.

그림 16.4 다른 의사 결정 소스에 바인딩된 의사 결정 그래프



의사 결정 그래프에 무엇이 나타나는가에 대한 자세한 내용은 다음 단원 "의사 결정 그래프 표시"를 참조하십시오.

의사 결정 그래프를 만들려면 앞 단원 "의사 결정 그래프 생성"을 참조하십시오.

의사 결정 그래프 속성과 의사 결정 그래프의 외관과 동작을 변경하는 방법에 대한 설명은 16-15 페이지의 "의사 결정 그래프 사용자 지정"을 참조하십시오.

의사 결정 그래프 표시

기본적으로 의사 결정 그래프는 x축에는 첫 번째 활성 열 필드의 값을, y축에는 첫 번째 활성 행 필드의 범주에 대한 요약 값을 나타냅니다. 그래프화된 각 범주는 별개 시리즈로 나타납니다.

예를 들면, 하나의 *TDecisionPivot* 버튼만 클릭한 경우처럼 한 차원만 선택한 경우에는 하나의 시리즈만 그래프화됩니다.

의사 결정 피벗을 사용했으면 해당 버튼을 눌러 그래프화할 의사 결정 큐브 필드(차원)를 결정할 수 있습니다. 그래프 축을 바꾸려면 분리 공간의 한 쪽에 있는 의사 결정 피벗 차원 버튼을 다른 쪽으로 끌어다 놓습니다. 분리 공간의 한쪽에 모든 버튼이 있는 1차원 그래프가 있으면 분리 공간의 다른 쪽에 버튼을 추가하고 그래프를 다차원으로 만들기 위한 드롭 대상으로 Row 또는 Column 아이콘을 사용할 수 있습니다.

한 번에 하나의 열과 하나의 행만 활성화하길 원한다면 *TDecisionSource*의 *ControlType* 속성을 *xtRadio*로 설정할 수 있습니다. 그러면 각 의사 결정 큐브 축에 대해 한 번에 하나의 활성 필드만 있을 수 있으며, 의사 결정 피벗의 기능은 그래프의 동작에 대응됩니다. *xtRadioEx*는 *xtRadio*와 동일하게 동작하지만 모든 행이나 모든 열 차원이 닫힌 상태는 허용하지 않습니다.

의사 결정 그리드와 그래프가 모두 동일한 *TDecisionSource*에 연결되어 있으면 *ControlType*을 다시 *xtCheck*로 설정하여 *TDecisionGrid*의 보다 유연한 동작에 대응시킬 수 있습니다.

의사 결정 그래프 사용자 지정

의사 결정 그래프 컴포넌트인 *TDecisionGraph*는 의사 결정 소스(*TDecisionSource*)의 필드를 차원이 열리고, 닫히고, 드래그되고, 드롭되고 의사 결정 피벗(*TDecisionPivot*)으로 재정렬될 때 변경되는 동적 그래프로 표시합니다. 선 그래프의 타입, 색상, 표식 타입 및 의사 결정 그래프의 다른 많은 속성을 변경할 수 있습니다.

다음과 같은 방법으로 그래프를 사용자 지정합니다.

- 1 마우스 오른쪽 버튼으로 클릭하고 Edit Chart를 선택합니다. Chart Editing 대화 상자가 나타납니다.
- 2 Chart Editing 대화 상자의 Chart 페이지를 사용하여 가시적(visible) 시리즈의 목록을 보고, 동일한 시리즈에 대해 두 개 이상의 시리즈 정의를 사용할 수 있을 경우 사용할 시리즈를 선택하고, 템플릿이나 시리즈의 그래프 타입을 변경하고, 전체적인 그래프 속성을 설정합니다.

Chart 페이지의 Series 목록에는 Template: 뒤에 오는 모든 의사 결정 큐브 차원과 현재의 가시적 범주가 표시됩니다. 각 범주 또는 시리즈는 별개의 객체입니다. 다음 작업을 할 수 있습니다.

- 기존 의사 결정 그래프 시리즈에서 파생된 시리즈를 추가하거나 삭제합니다. 파생된 시리즈는 기존 시리즈의 주석을 제공하거나 다른 시리즈에서 계산된 값을 표현할 수 있습니다.

- 기본 그래프 타입을 변경하고 템플릿과 시리즈의 제목을 변경합니다.

다른 Chart 페이지 탭에 대한 설명은 온라인 도움말의 "Chart 페이지 (Chart Editing 대화 상자)" 항목을 검색하십시오.

- 3 Series 페이지를 사용하여 차원 템플릿을 설정한 다음 각 그래프 시리즈의 속성을 사용자 지정합니다.

기본적으로 모든 시리즈는 막대 그래프로 그래프화되고 최대 16가지의 기본 색상이 할당됩니다. 템플릿 타입과 속성을 편집하여 새 기본값을 만들 수 있습니다. 그런 다음 의사 결정 소스를 다른 상태로 피벗할 때 템플릿을 사용하여 각각의 새로운 상태 시리즈를 동적으로 만듭니다. 자세한 내용은 16-16 페이지의 "의사 결정 그래프 템플릿 기본값 설정"을 참조하십시오.

개별 시리즈를 사용자 지정하려면 16-17 페이지의 "의사 결정 그래프 시리즈 사용자 지정"의 지침을 따릅니다.

각각의 Series 페이지 탭에 대한 설명은 온라인 도움말의 "Series 페이지 (Chart Editing 대화 상자)" 항목을 검색하십시오.

의사 결정 그래프 템플릿 기본값 설정

의사 결정 그래프는 의사 결정 큐브의 두 차원으로부터 값을 표시합니다. 한 차원은 그래프의 축으로 표시되고 다른 차원은 일련의 시리즈를 생성하기 위해 사용됩니다. 해당 차원의 템플릿은 그러한 시리즈에 대한 기본 속성(예를 들면, 시리즈가 막대, 선, 영역 등인지 여부)을 제공합니다. 사용자가 한 상태에서 다른 상태로 피벗할 때 템플릿에 지정된 시리즈 타입 및 다른 기본값을 사용하여 차원에 대해 필요한 시리즈가 생성됩니다.

사용자가 한 차원만 활성화되는 상태로 피벗하는 경우에는 별개의 템플릿이 제공됩니다. 1차원 상태는 원형 차트로 표현되는 경우도 있으므로 이 경우에 별개의 템플릿이 제공됩니다.

다음 작업을 할 수 있습니다.

- 기본 그래프 타입을 변경합니다.
- 다른 그래프 템플릿 속성을 변경합니다.
- 전체적인 그래프 속성을 보고 설정합니다.

기본 의사 결정 그래프 타입 변경

다음과 같은 방법으로 기본 그래프 타입을 변경합니다.

- 1 Chart Editing 대화 상자의 Chart 페이지에 있는 Series 목록에서 템플릿을 선택합니다.
- 2 Change 버튼을 클릭합니다.
- 3 새로운 타입을 선택하고 Gallery 대화 상자를 닫습니다.

다른 의사 결정 그래프 템플릿 속성 변경

다음과 같은 방법으로 템플릿의 색상이나 다른 속성을 변경합니다.

- 1 Chart Editing 대화 상자의 맨 위에 있는 Series 페이지를 선택합니다.
- 2 페이지의 위에 있는 드롭다운 목록에서 템플릿을 선택합니다.
- 3 적절한 속성 탭을 선택한 후 설정을 선택합니다.

전체적인 의사 결정 그래프 속성 보기

다음과 같은 방법으로 타입과 시리즈 이외의 의사 결정 그래프 속성을 보고 설정합니다.

- 1 Chart Editing 대화 상자의 위에 있는 Chart 페이지를 선택합니다.
- 2 적절한 속성 탭을 선택한 후 설정을 선택합니다.

의사 결정 그래프 시리즈 사용자 지정

템플릿은 그래프 타입 및 시리즈가 표시되는 방법과 같은 각각의 의사 결정 큐브 차원에 대한 많은 기본값을 제공합니다. 시리즈 색상과 같은 다른 기본값은 *TDecisionGraph*에 의해 정의됩니다. 원한다면 각 시리즈의 기본값을 오버라이드할 수 있습니다.

템플릿은 필요하면 프로그램으로 범주에 대한 시리즈를 생성할 때 사용하고 더 이상 필요 없을 때는 버리도록 고안되어 있습니다. 원한다면 특정 범주 값에 대해 사용자 지정 시리즈를 설정할 수도 있습니다. 이렇게 하려면 사용자 지정하려는 범주에 대한 시리즈가 표시되도록 그래프를 피벗합니다. 시리즈가 그래프에 표시되면 Chart Editor를 사용하여 다음 작업을 할 수 있습니다.

- 그래프 타입을 변경합니다.
- 다른 시리즈 속성을 변경합니다.
- 사용자 지정된 특정 그래프 시리즈를 저장합니다.

시리즈 템플릿을 정의하고 전체적인 그래프 기본값을 설정하려면 16-16 페이지의 "의사 결정 그래프 템플릿 기본값 설정"을 참조하십시오.

시리즈 그래프 타입 변경

기본적으로 각 시리즈에는 해당 차원의 템플릿으로 정의된 동일한 그래프 타입이 있습니다. 모든 시리즈를 동일한 그래프 타입으로 변경하기 위해 템플릿 타입을 변경할 수 있습니다. 지침은 16-16 페이지의 "기본 의사 결정 그래프 타입 변경"을 참조하십시오.

단일 시리즈의 그래프 타입을 변경하려면 다음과 같이 합니다.

- 1 Chart Editor의 Chart 페이지에 있는 Series 목록에서 시리즈를 선택합니다.
- 2 Change 버튼을 클릭합니다.
- 3 새로운 타입을 선택하고 Gallery 대화 상자를 닫습니다.
- 4 Save Series 체크 박스를 선택합니다.

다른 의사 결정 그래프 시리즈 속성 변경

의사 결정 그래프 시리즈의 색상이나 다른 속성을 변경하려면 다음과 같이 합니다.

- 1 Chart Editing 대화 상자의 맨 위에 있는 Series 페이지를 선택합니다.
- 2 페이지의 위에 있는 드롭다운 목록에서 시리즈를 선택합니다.
- 3 적절한 속성 탭을 선택한 후 설정을 선택합니다.
- 4 Save Series 체크 박스를 선택합니다.

의사 결정 그래프 시리즈 설정 저장

기본적으로 디자인 타임에는 템플릿의 설정만 저장됩니다. 특정 시리즈의 변경 사항은 Chart Editing 대화 상자에서 해당 시리즈에 대해 Save 상자가 선택되었을 때만 저장됩니다.

시리즈를 저장할 때는 메모리가 많이 필요하므로 저장할 필요가 없을 때는 Save 상자를 선택하지 않습니다.

런타임의 의사 결정 지원 컴포넌트

런타임 시 사용자는 가시적인 의사 결정 지원 컴포넌트를 마우스 왼쪽 클릭, 마우스 오른쪽 클릭 또는 드래그하여 많은 작업을 수행할 수 있습니다. 이러한 작업들은 이 장의 앞 부분에서 다루었으며 아래에 요약되어 있습니다.

런타임의 의사 결정 피벗

사용자는 다음 작업을 할 수 있습니다.

- 의사 결정 피벗의 왼쪽 끝에 있는 요약 버튼을 마우스 왼쪽 버튼으로 클릭하여 사용할 수 있는 요약 목록을 표시합니다. 이 목록을 사용하여 의사 결정 그리드와 의사 결정 그래프에 표시된 요약 데이터를 변경할 수 있습니다.
- 차원 버튼을 마우스 오른쪽 버튼으로 클릭하고 다음 중에서 선택합니다.
 - 행 영역을 열 영역으로 또는 그 반대로 이동합니다.
 - 상세 데이터를 표시하기 위해 드릴 인(Drill In)합니다.
- Drill In 명령 다음에 차원 버튼을 마우스 왼쪽 버튼으로 클릭하고 다음 중에서 선택합니다.
 - 해당 차원의 상위 레벨로 이동하려면 Open Dimension을 선택합니다.
 - 의사 결정 그리드에서 단지 요약을 표시하는 것과 요약에 덧붙여 다른 모든 값을 표시하는 것 사이를 전환하려면 All Values를 선택합니다.
 - 해당 차원의 사용 가능한 범주 목록에서 자세한 값을 얻기 위해 드릴할 범주를 선택합니다.

- 차원 버튼을 마우스 왼쪽 버튼으로 클릭하여 해당 차원을 열거나 닫습니다.
- 차원 버튼을 행 영역에서 열 영역으로 또는 그 반대로 드래그하여 끌어 놓습니다. 그런 다음 그 영역에 놓거나 행이나 열 아이콘 위에 있는 기존 버튼 옆에 놓습니다.

런타임 시의 의사 결정 그리드

사용자는 다음 작업을 할 수 있습니다.

- 의사 결정 그리드 내에서 마우스 오른쪽 버튼을 클릭하고 다음 중에서 선택합니다.
 - 부분 합계를 개별 데이터 그룹, 차원의 모든 값, 전체 그리드 등에 대해 on/off로 토글합니다.
 - 16-7 페이지에서 설명되는 Decision Cube 에디터를 표시합니다.
 - 차원과 요약물 열고 닫기를 전환합니다.
- 행과 열 헤더에 있는 +와 -를 클릭하여 차원을 열고 닫습니다.
- 행에서 열로 또는 그 반대로 차원을 드래그 앤 드롭합니다.

런타임 시의 의사 결정 그래프

사용자는 오프스크린(off-screen)의 범주와 값을 스크롤하기 위해 그래프 그리드 영역의 옆에서 옆으로 또는 위에서 아래로 끌어다 놓을 수 있습니다.

의사 결정 지원(Decision Support) 컴포넌트와 메모리 제어

차원이나 요약이 의사 결정 큐브로 로드되면 메모리를 차지하게 됩니다. 새로운 요약물 추가하면 메모리 소모가 점차 늘어납니다. 즉, 두 개의 요약물 갖는 의사 결정 큐브는 요약이 하나인 큐브의 두 배에 해당하는 메모리를 사용하게 되며, 세 개의 요약물 갖는 의사 결정 큐브는 요약이 하나인 큐브의 세 배에 해당하는 메모리를 사용하게 된다는 뜻입니다. 차원에 대한 메모리 소모는 더 빠르게 증가합니다. 10개의 값을 갖는 차원을 추가하면 10배의 메모리를 소모하고, 100개의 값을 갖는 차원을 추가하면 100배의 메모리를 소모합니다. 이와 같이 의사 결정 큐브에 차원을 추가하면 메모리 사용에 엄청난 영향을 줄 수 있으며 더 나아가 성능 문제를 일으킬 수 있습니다. 이 영향은 특히 많은 값을 갖는 차원을 추가할 때 뚜렷해집니다.

의사 결정 지원 컴포넌트에는 메모리가 사용되는 방법과 시기를 제어할 수 있게 하는 많은 설정이 있습니다. 여기서 언급되는 속성 및 기술에 대한 자세한 내용은 온라인 도움말에서 *TDecisionCube*를 검색하십시오.

차원, 요약, 셀의 최대값 설정

의사 결정 큐브의 *MaxDimensions*와 *MaxSummaries* 속성은 *CubeDim.ActiveFlag* 속성과 함께 사용되어 한 번에 로드될 수 있는 차원과 요약 수를 조절할 수 있습니다.

Decision Cube 에디터의 Cube Capacity 페이지에 최대값을 설정하여 동시에 메모리에 가져올 수 있는 차원이나 요약 수를 제어할 수 있습니다.

차원이나 요약 수를 제한하면 의사 결정 큐브에서 사용하는 메모리 양을 제한할 수 있습니다. 하지만 많은 값을 갖는 차원과 적게 갖는 차원이 구분되지 않습니다. 의사 결정 큐브의 절대적인 메모리 수요를 더 많이 제어하기 위해 큐브의 셀 수를 제한할 수도 있습니다. Decision Cube 에디터의 Cube Capacity 페이지에 셀의 최대 수치를 설정합니다.

차원 상태 설정

ActiveFlag 속성은 로드되는 차원을 제어합니다. Activity Type 컨트롤을 사용하여 Decision Cube 에디터의 Dimension Settings 탭에서 이 속성을 설정할 수 있습니다. 이 컨트롤을 *Active*로 설정하면 차원이 바로 로드되고 항상 공간을 차지하게 됩니다. 이 상태의 차원 수는 항상 *MaxDimensions*보다 적어야 하며 *Active*로 설정된 요약 수는 *MaxSummaries*보다 적어야 합니다. 항상 사용 가능한 경우에만 차원이나 요약을 *Active*로 설정해야 합니다. *Active* 설정은 사용 가능한 메모리를 관리하는 큐브의 능력을 감소시킵니다.

*ActiveFlag*가 *AsNeeded*로 설정되었으면 차원이나 요약은 *MaxDimensions*, *MaxSummaries* 또는 *MaxCells* 등의 한계를 초과하지 않으면서 로드될 수 있는 경우에만 로드됩니다. 의사 결정 큐브는 메모리 내부와 외부에서 *AsNeeded* 표시된 차원과 요약을 바꾸어 *MaxCells*, *MaxDimensions*, *MaxSummaries*가 나타내는 한계 내에서 유지합니다. 차원이나 요약이 사용되지 않으면 메모리에 로드되지 않을 수도 있습니다. 자주 사용되지 않는 차원을 *AsNeeded*로 설정하면 현재 로드되지 않은 차원을 액세스하는 데 시간이 걸리기는 하지만 로드와 피벗의 성능은 더 좋아집니다.

페이지화된 차원 사용

Decision Cube 에디터의 Dimension Settings 탭에서 Binning이 Set으로 설정되고 Start Value가 NULL이 아니면 차원이 "페이지화"되었거나 "영구적으로 드릴 다운"되었다고 할 수 있습니다. 프로그램에서 일련의 값을 연속해서 액세스할 수 있어도 그 차원의 값은 한 번에 하나만 액세스할 수 있습니다. 그러한 차원은 피벗되거나 열리지 않을 수도 있습니다.

아주 많은 값을 갖는 차원에 대해 차원 데이터를 포함하기 위해서는 아주 많은 메모리가 필요합니다. 그러한 차원을 페이지 표시하면 한 번에 하나의 값에 대한 요약 정보를 표시할 수 있습니다. 이렇게 표시될 때 정보는 일반적으로 더 읽기 쉬우며 메모리 소모를 관리하는 것도 훨씬 더 쉬워집니다.

17

데이터베이스에 연결

대부분의 데이터셋 컴포넌트는 데이터베이스 서버에 직접 연결할 수 있습니다. 일단 연결되면 데이터셋은 서버와 자동으로 통신합니다. 데이터셋을 열면 서버의 데이터로 데이터셋이 채워지고 레코드를 포스트할 때 서버로 되돌아 가서 적용됩니다. 단일 연결 컴포넌트는 여러 데이터셋에 의해 공유되거나 각 데이터셋은 고유한 연결을 사용할 수 있습니다.

각 데이터셋 타입은 단일 데이터 액세스 메커니즘에 의해 작동하도록 디자인된 고유한 타입의 연결 컴포넌트를 사용하여 데이터베이스 서버에 연결합니다. 다음 표는 이러한 데이터 액세스 메커니즘 및 관련된 연결 컴포넌트를 나열한 것입니다.

표 17.1 데이터베이스 연결 컴포넌트

데이터 액세스 메커니즘	연결 컴포넌트
Borland Database Engine (BDE).	TDatabase
ActiveX Data Objects (ADO).	TADOConnection
dbExpress.	TSQLConnection
InterBase Express.	TIBDatabase

참고 각 메커니즘의 장단점에 대한 설명은 14-1 페이지의 "데이터베이스 사용"을 참조하십시오.

연결 컴포넌트는 데이터베이스 연결에 필요한 모든 정보를 제공합니다. 이 정보는 다음과 같이 연결 컴포넌트의 타입에 따라 다릅니다.

- BDE 기반 연결에 대한 설명은 20-14 페이지의 "데이터베이스 식별"을 참조하십시오.
- ADO 기반 연결에 대한 설명은 21-3 페이지의 "TADOConnection을 사용하여 데이터 저장소에 연결"을 참조하십시오.
- dbExpress 연결에 대한 설명은 22-3 페이지의 "TSQLConnection 설정"을 참조하십시오.

- InterBase Express를 설명하는 내용은 온라인 도움말의 *TIBDatabase*를 참조하십시오.

각 타입의 데이터셋은 각기 다른 연결 컴포넌트를 사용하지만 모두 *TCustomConnection*의 자손입니다. 이들은 모두 동일한 여러 작업을 수행하고 동일한 여러 속성, 메소드 및 이벤트를 표면화합니다. 이 장에서는 이러한 공통적인 여러 작업에 대해 설명합니다.

암시적인 연결(Implicit connections) 사용

사용하는 데이터 액세스 메커니즘의 종류에 상관 없이 언제든지 연결 컴포넌트를 명시적으로 생성하여 이를 통해 데이터베이스 서버와의 연결을 관리하고 통신할 수 있습니다. BDE 활성 데이터셋 및 ADO 기반 데이터셋에는 데이터셋의 속성들을 통해 데이터베이스 연결을 설명하고 데이터셋이 암시적인 연결을 생성하도록 하는 옵션이 있습니다. BDE 활성 데이터셋에서는 *DatabaseName* 속성을 사용하여 암시적인 연결을 지정합니다. ADO 기반 데이터셋에서는 *ConnectionString* 속성을 사용합니다.

암시적인 연결을 사용할 때 연결 컴포넌트를 명시적으로 생성할 필요는 없습니다. 이로 인해서 애플리케이션 개발을 단순화할 수 있고, 지정하는 기본 연결에서 매우 다양한 상황을 처리할 수 있습니다. 하지만 많은 사용자와 다른 데이터베이스 연결 요건을 가지고 있는 복잡하고 중요한 임무를 수행하는 클라이언트/서버 애플리케이션에서는 애플리케이션의 필요에 따라 각 데이터베이스 연결을 조정하기 위해서 사용자 고유의 연결 컴포넌트를 생성해야 합니다. 명시적인 연결 컴포넌트는 보다 많은 제어를 합니다. 예를 들면, 다음과 같은 작업을 수행하기 위해 연결 컴포넌트에 액세스해야 합니다.

- 데이터베이스 서버 로그인 지원을 사용자 지정합니다.(암시적인 연결은 사용자 이름과 암호를 묻는 기본 로그인 대화 상자를 표시합니다.)
- 트랜잭션을 제어하고 트랜잭션 분리 레벨을 지정합니다.
- 데이터셋을 사용하지 않고 서버에서 SQL 명령을 실행합니다.
- 동일한 데이터베이스에 연결된 모든 개방형 데이터셋에서 동작을 수행합니다.

또한 동일한 서버를 사용하는 여러 데이터셋이 있는 경우, 연결 컴포넌트를 사용하면 한 곳에서 사용할 서버만 지정하면 되므로 더 용이합니다. 나중에 서버를 변경하는 경우, 여러 데이터셋 컴포넌트를 업데이트할 필요 없이 연결 컴포넌트만 업데이트하면 됩니다.

연결 제어

데이터베이스 서버에 대한 연결을 설정하려면 먼저 원하는 서버 연결을 설명하는 어떤 중요 정보를 애플리케이션이 제공해야 합니다. 각 타입의 연결 컴포넌트는 서버를 식별하도록 하는 다른 속성 집합을 나타냅니다. 하지만 일반적으로 모든 연결 컴포넌트는 원하는 서버를 명명하는 방법과 연결이 이루어지는 방식을 제어하는 연결 매개변수 집합을 제공합니다. 연결 매개변수는 서버에 따라 다양합니다. 사용자 이름과 암호, BLOB 필드의 최대 크기, SQL 역할 등과 같은 정보를 포함할 수 있습니다.

원하는 서버와 연결 매개변수를 식별하고 나면 연결 컴포넌트를 사용하여 명시적으로 연결을 열거나 닫습니다. 연결 컴포넌트는 연결을 열거나 닫을 때 이벤트를 생성하여 데이터베이스 연결의 변경 사항에 대한 애플리케이션의 응답을 사용자 지정하는 데 사용할 수 있습니다.

데이터베이스 서버에 연결

연결 컴포넌트를 사용하여 데이터베이스 서버에 연결하는 방법에는 다음과 같이 두 가지가 있습니다.

- *Open* 메소드를 호출합니다.
- *Connected* 속성을 *True*로 설정합니다.

Open 메소드를 호출하면 *Connected*가 *True*로 설정됩니다.

참고 연결 컴포넌트가 서버에 연결되지 않고 애플리케이션이 연결된 데이터셋 중 하나를 열려고 하는 경우, 데이터셋은 자동으로 연결 컴포넌트의 *Open* 메소드를 호출합니다.

*Connected*를 *True*로 설정한 경우, 연결 컴포넌트는 초기화를 수행할 수 있는 *BeforeConnect* 이벤트를 먼저 생성합니다. 예를 들면, 연결 매개변수를 변경하기 위해 이 이벤트를 사용할 수 있습니다.

BeforeConnect 이벤트 후, 연결 컴포넌트는 서버 로그인 제어를 선택하는 방법에 따라 기본 로그인 대화 상자를 표시합니다. 그런 다음 사용자 이름과 암호를 드라이버에 전달하고 연결을 엽니다.

일단 연결이 열리면 연결 컴포넌트는 *AfterConnect* 이벤트를 생성하여 개방형 연결을 필요로 하는 작업들을 수행할 수 있습니다.

참고 일부 연결 컴포넌트는 연결이 만들어질 때 추가적인 이벤트를 생성합니다.

일단 연결이 만들어지면 연결을 사용하는 활성 데이터셋이 하나 이상 되는 한 계속 유지됩니다. 더 이상의 활성 데이터셋이 없을 때 연결 컴포넌트는 연결을 해제합니다. 일부 연결 컴포넌트는 모든 데이터셋이 닫혀 있더라도 연결이 여전히 개방형 이벤트로 남아 있도록 하는 *KeepConnection* 속성을 나타냅니다. *KeepConnection*이 *True*인 경우 연결이 유지됩니다. 원격 데이터베이스 서버에 대한 연결의 경우나 데이터셋을 자주 열고 닫는 애플리케이션의 경우에 *KeepConnection*을 *True*로 설정하면 네트워크 트래픽이 감소되고 애플리케이션의 속도가 빨라집니다. *KeepConnection*이 *False*인 경우에는 데이터베이스를 사용하는 활성 데이터셋이 없으면 연결이 해제됩니다. 데이터베이스를 사용하는 데이터셋이 나중에 열릴 때 연결이 재구성되고 초기화되어야 합니다.

데이터베이스 서버로부터 연결 해제

연결 컴포넌트를 사용하여 서버를 연결 해제하는 방법에는 다음과 같이 두 가지가 있습니다.

- *Connected* 속성을 *False*로 설정합니다.
- *Close* 메소드를 호출합니다.

*Close*를 호출하면 *Connected*가 *False*로 설정됩니다.

*Connected*가 *False*로 설정되는 경우, 연결 컴포넌트는 연결을 닫기 전에 클린업을 실행할 수 있는 *BeforeDisconnect* 이벤트를 생성합니다. 예를 들어, 이 이벤트를 사용하여 데이터셋이 닫히기 전에 모든 열린 데이터셋에 대한 정보를 캐시로 저장할 수 있습니다.

BeforeConnect 이벤트 후, 연결 컴포넌트는 모든 열린 데이터셋을 닫고 서버로부터 연결을 해제합니다.

마지막으로 연결 컴포넌트는 *AfterDisconnect* 이벤트를 생성하여 사용자 인터페이스의 Connect 버튼을 활성화하는 것과 같은 연결 상태의 변경에 응답할 수 있습니다.

참고 *Close*를 호출하거나 *Connected*를 *False*로 설정하면 연결 컴포넌트의 *KeepConnection* 속성이 *True*일지라도 데이터베이스 서버로부터 연결을 해제합니다.

서버 로그인 제어

대부분의 원격 데이터베이스 서버는 승인되지 않은 액세스를 방지하는 보안 기능을 포함합니다. 일반적으로 서버는 데이터베이스 액세스를 허용하기 전에 사용자 이름과 암호 로그인을 요구합니다.

디자인 타임 시 서버에 로그인이 필요하면 데이터베이스에 대한 연결을 처음으로 시도할 때 표준 로그인 대화 상자가 나타나 사용자 이름과 암호 입력을 기다립니다.

런타임 시 서버의 로그인 요청을 처리할 수 있는 방법에는 세 가지가 있습니다.

- 기본 로그인 대화 상자와 프로세스가 로그인을 처리하도록 합니다. 이것은 기본적인 방법입니다. 연결 컴포넌트의 *LoginPrompt* 속성을 *True*(기본값)로 설정하고 연결 컴포넌트를 선언하는 유닛의 *uses* 절에 *DBLogDlg*를 추가합니다. 애플리케이션은 서버가 사용자 이름과 암호를 요청할 때 표준 로그인 대화 상자를 표시합니다.
- 로그인이 시도되기 전에 로그인 정보를 제공합니다. 각 타입의 연결 컴포넌트는 사용자 이름과 암호를 지정하기 위해 다음과 같이 각기 다른 메커니즘을 사용합니다.
 - BDE, dbExpress 및 InterBase express 데이터셋에서는 사용자 이름과 암호 연결 매개변수가 *Params* 속성을 통해 액세스될 수 있습니다. (BDE에서는 매개변수 값을 BDE 알리아스와 관련시킬 수 있는 반면, dbExpress 데이터셋에서는 매개변수 값을 연결 이름과 관련시킬 수 있습니다.)
 - ADO 데이터셋에서는 사용자 이름과 암호가 *ConnectionString* 속성에 포함될 수 있습니다(또는 *Open* 메소드에 대한 매개변수로서 제공될 수 있습니다.).

서버가 요청하기 전에 사용자 이름과 암호를 지정하려면 기본 로그인 대화 상자가 나타나지 않도록 *LoginPrompt*를 *False*로 설정해야 합니다. 예를 들어, 다음 코드는 *BeforeConnect* 이벤트 핸들러에서 현재 연결 이름과 연관된 암호화된 암호를 해독하여 SQL 연결 컴포넌트에 대한 사용자 이름과 암호를 설정합니다.


```

procedure TForm1.SQLConnectionBeforeConnect(Sender:TObject);
begin
  with Sender as TSQLConnection do
  begin
    if LoginPrompt = False then
      begin
        Params.Values['User_Name'] := 'SYSDBA';
        Params.Values['Password'] := Decrypt(Params.Values['Password']);
      end;
    end;
  end;
end;

```

디자인 타임 시 사용자 이름과 암호를 설정하거나 코드에서 하드 코딩된 문자열을 사용하면 애플리케이션의 실행 파일에 값이 포함된다는 점에 유의하십시오. 그렇게 해도 값을 쉽게 찾을 수 있으므로 서버 보안성이 떨어집니다.

- 로그인 이벤트에 대한 고유한 사용자 지정 처리를 제공합니다. 연결 컴포넌트는 사용자 이름과 암호가 필요할 때 이벤트를 생성합니다.
 - TDatabase*, *TSQLConnection* 및 *TIBDatabase*의 경우 *OnLogin* 이벤트입니다. 이벤트 핸들러는 두 개의 매개변수, 연결 컴포넌트 및 문자열 목록의 사용자 이름과 암호 매개변수의 로컬 복사본을 갖습니다. *TSQLConnection*은 데이터베이스 매개변수도 포함합니다. 이 이벤트가 발생하도록 하기 위해서는 *LoginPrompt* 속성을 *True*로 설정해야 합니다. *False*의 *LoginPrompt* 값을 가지고 *OnLogin* 이벤트에 대한 핸들러를 할당하면 기본 대화 상자가 나타나지 않고 *OnLogin* 이벤트 핸들러가 실행되지 않기 때문에 데이터베이스에 로그인할 수 없는 상황이 발생합니다.
 - TADOConnection*의 경우에는 *OnWillConnect* 이벤트입니다. 이벤트 핸들러는 다섯 개의 매개변수, 즉 연결 컴포넌트 및 연결에 영향을 주는 값을 반환하는 네 개의 매개변수(사용자 이름과 암호의 두 매개변수 포함)를 가집니다. 이 이벤트는 *LoginPrompt* 값과 상관 없이 항상 발생합니다.

로그인 매개변수를 설정하는 이벤트에 대한 이벤트 핸들러를 작성합니다. 다음 예제에는 전역 변수(*UserName*)로부터 제공되는 USER NAME과 PASSWORD 매개변수에 대한 값 및 사용자 이름이 주어진 암호를 반환하는 메소드(*PasswordSearch*)가 있습니다.

```

procedure TForm1.Database1Login(Database:TDatabase; LoginParams:TStrings);
begin
  LoginParams.Values['USER NAME'] := UserName;
  LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
end;

```

로그인 매개변수를 제공하는 다른 메소드에서와 같이 *OnLogin* 또는 *OnWillConnect* 이벤트 핸들러를 작성할 때는 애플리케이션 코드에 암호를 하드 코딩하지 마십시오. 암호는 암호화된 값, 즉 애플리케이션이 값을 조회하기 위해 사용하는 보안 데이터베이스의 항목으로만 나타나거나 사용자로부터 동적으로 얻어질 수 있어야 합니다.

트랜잭션 관리

트랜잭션은 커밋(영구적으로)되기 전에 데이터베이스의 하나 이상의 테이블에서 모두 성공적으로 수행되어야 하는 작업 그룹입니다. 그룹의 작업 중 하나라도 실패하면 모든 작업이 롤백(실행 취소)됩니다. 트랜잭션을 사용함으로써 트랜잭션을 구성하는 작업 중 하나를 완료하면서 문제가 발생할 때 데이터베이스가 불일치 상태가 되지 않도록 합니다.

예를 들면, 은행 업무용 애플리케이션에서 한 계좌에서 다른 계좌로의 자금 전송은 트랜잭션으로 보호해야 하는 작업입니다. 한 계좌의 잔액을 줄이고 나서 다른 계좌의 잔액을 증가시킬 때 오류가 발생하면 트랜잭션을 롤백하여 데이터베이스가 정확한 총 잔액을 반영하도록 합니다.

SQL 명령을 데이터베이스에 직접 전송하여 트랜잭션을 관리하는 것이 언제나 가능합니다. 일부 데이터베이스에는 트랜잭션 지원이 전혀 없지만 대부분의 데이터베이스는 고유한 트랜잭션 관리 모델을 제공합니다. 트랜잭션을 지원하는 서버의 경우, 스키마 캐시와 같은 특정 데이터베이스 서버의 고급 트랜잭션 관리 기능을 이용하여 사용자 고유의 트랜잭션 관리를 코드로 직접 작성할 수 있습니다.

고급 트랜잭션 관리 기능을 사용할 필요가 없는 경우, 연결 컴포넌트는 SQL 명령을 명시적으로 전송하지 않고 트랜잭션을 관리하기 위해 사용할 수 있는 일련의 메소드와 속성을 제공합니다. 이들 속성과 메소드를 사용하면 서버가 트랜잭션을 지원하는 한 사용하는 데이터베이스 서버의 각 타입에 따라 애플리케이션을 사용자 지정할 필요가 없다는 이점이 있습니다. (BDE 또한 서버 트랜잭션 지원 없이 로컬 테이블에 대해 제한된 트랜잭션 지원을 제공합니다. BDE를 사용하지 않는 경우, 트랜잭션을 지원하지 않는 데이터베이스에서 트랜잭션을 시작하려고 하면 연결 컴포넌트가 예외를 발생시킵니다.)

경고 데이터셋 프로바이더 컴포넌트가 업데이트를 적용할 때 업데이트 사항에 대한 트랜잭션을 암시적으로 생성합니다. 명시적으로 시작하는 트랜잭션이 프로바이더에 의해 생성된 트랜잭션과 충돌하지 않도록 유의하십시오.

트랜잭션 시작

트랜잭션을 시작할 경우, 트랜잭션이 명시적으로 종료될 때까지 또는 오버랩된 트랜잭션의 경우는 다른 트랜잭션이 시작될 때까지 데이터베이스에서 읽거나 쓰기를 하는 이어지는 모든 문장은 해당 트랜잭션의 컨텍스트에 있습니다. 각 문장은 그룹의 일부로 간주됩니다. 변경 사항이 데이터베이스에 성공적으로 커밋되어야 하며, 그렇지 않으면 그룹에 취해진 모든 변경 사항이 실행 취소됩니다.

트랜잭션이 진행되는 동안 데이터베이스 테이블의 데이터 뷰는 트랜잭션 분리 레벨에 의해 결정됩니다. 트랜잭션 분리 레벨에 대한 내용은 17-9 페이지의 "트랜잭션 분리 레벨 지정"을 참조하십시오.

TADOConnection에서는 다음과 같이 *BeginTrans* 메소드를 호출하여 트랜잭션을 시작합니다.

```
Level := ADOConnection1.BeginTrans;
```

*BeginTrans*는 시작된 트랜잭션에 대한 중첩 레벨을 반환합니다. 중첩 트랜잭션은 다른 부모 트랜잭션 내에 중첩된 트랜잭션입니다. 서버가 트랜잭션을 시작한 후, ADO 연결은 *OnBeginTransComplete* 이벤트를 받습니다.

*TDatabase*에서는 대신 *StartTransaction* 메소드를 사용합니다. *TDatabase*는 다음과 같이 중첩되거나 오버랩된 트랜잭션을 지원하지 않습니다. 다른 트랜잭션이 진행되는 동안 *TDatabase* 컴포넌트의 *StartTransaction* 메소드를 호출하는 경우 예외를 발생시킵니다. *StartTransaction*을 호출하는 것을 방지하기 위해서 다음과 같이 *InTransaction* 속성을 확인할 수 있습니다.

```
if not Database1.InTransaction then
    Database1.StartTransaction;
```

*TSQLConnection*은 또한 *StartTransaction* 메소드를 사용하지만 더 많은 제어를 제공하는 버전을 사용합니다. 구체적으로 말하면 *StartTransaction*이 트랜잭션 디스크립터를 취해서 사용자가 동시적인 여러 트랜잭션을 관리하고 트랜잭션 단위로 트랜잭션 분리 레벨을 지정하게 합니다. (트랜잭션 레벨에 대한 자세한 내용은 17-9 페이지의 "트랜잭션 분리 레벨 지정"을 참조하십시오.) 동시적인 여러 트랜잭션을 관리하려면 트랜잭션 디스크립터의 *TransactionID* 필드를 고유한 값으로 설정합니다. *TransactionID*가 고유한(현재 사용 중인 다른 트랜잭션과 충돌하지 않는) 값인 경우, 어떤 값이라도 선택할 수 있습니다. 서버에 따라서는 *TSQLConnection*에 의해 시작된 트랜잭션이 ADO를 사용할 때처럼 중첩되거나 오버랩될 수 있습니다.

```
var
    TD:TTransactionDesc;
begin
    TD.TransactionID := 1;
    TD.IsolationLevel := xilREADCOMMITTED;
    SQLConnection1.StartTransaction(TD);
```

기본적으로 오버랩된 트랜잭션에서는 첫 번째 트랜잭션의 커밋 또는 롤백을 나중까지 연기할 수 있다고 하더라도 두 번째 트랜잭션이 시작될 때 첫 번째 트랜잭션이 비활성화됩니다. InterBase 데이터베이스를 가지고 *TSQLConnection*을 사용하는 경우, *TransactionLevel* 속성을 설정하여 특별한 활성 트랜잭션을 가지는 애플리케이션에서 각 데이터셋을 식별할 수 있습니다. 즉, 두 번째 트랜잭션을 시작한 후에 단지 데이터셋을 원하는 트랜잭션에 연결함으로써 두 트랜잭션 모두 동시에 작업할 수 있습니다.

참고 *TADOConnection*과 달리, *TSQLConnection*과 *TDatabase*는 트랜잭션을 시작할 때 이벤트를 받지 않습니다.

InterBase express는 연결 컴포넌트를 사용하여 트랜잭션을 시작하는 대신 개별 트랜잭션 컴포넌트를 사용함으로써 *TSQLConnection*보다 훨씬 더 많은 제어를 제공합니다. 하지만 다음과 같이 *TIBDatabase*를 사용하여 기본 트랜잭션을 시작할 수 있습니다.

```
if not IBDatabase1.DefaultTransaction.InTransaction then
    IBDatabase1.DefaultTransaction.StartTransaction;
```

두 개의 개별 트랜잭션 컴포넌트를 사용하여 오버랩된 트랜잭션을 가질 수 있습니다. 각 트랜잭션 컴포넌트에는 사용자가 트랜잭션을 구성하도록 하는 매개변수 집합이 있습니다. 이로 인해서 사용자는 트랜잭션의 다른 속성뿐만 아니라 트랜잭션 분리 레벨을 지정할 수 있습니다.

트랜잭션 종료

이상적으로 보면 트랜잭션은 필요한 만큼 지속되어야 합니다. 트랜잭션이 활성화되는 시간이 길면 길수록 데이터베이스에 액세스하는 동시 사용자가 더 많아지고, 트랜잭션 수명 동안 시작하고 종료하는 동시에 발생하는 트랜잭션이 더 많아지며, 어떤 변경 사항을 커밋하려고 시도할 때 트랜잭션이 다른 트랜잭션과 충돌할 가능성이 더 커집니다.

성공적인 트랜잭션 종료

트랜잭션을 구성하는 작업이 모두 성공하면 트랜잭션을 커밋하여 데이터베이스의 변경 사항을 영구적으로 만들 수 있습니다. *TDatabase*에서는 다음과 같이 *Commit* 메소드를 사용하여 트랜잭션을 커밋합니다.

```
MyOracleConnection.Commit;
```

*TSQLConnection*에서는 다음과 같이 *Commit* 메소드를 사용하지만 *StartTransaction* 메소드에 부여했던 트랜잭션 디스크립터를 제공하여 커밋할 트랜잭션을 지정해야 합니다.

```
MyOracleConnection.Commit(TD);
```

*TIBDatabase*에서는 다음과 같이 *Commit* 메소드를 사용하여 트랜잭션 객체를 커밋합니다.

```
IBDatabase1.DefaultTransaction.Commit;
```

*TADOConnection*에서는 다음과 같이 *CommitTrans* 메소드를 사용하여 트랜잭션을 커밋합니다.

```
ADOConnection1.CommitTrans;
```

참고 부모 트랜잭션이 롤백되면 변경 사항이 나중에 롤백되도록 하기 위해서만 중첩 트랜잭션이 커밋될 수 있습니다.

트랜잭션이 성공적으로 커밋된 후, ADO 연결 컴포넌트는 *OnCommitTransComplete* 이벤트를 받습니다. 다른 연결 컴포넌트는 비슷한 이벤트를 받지 않습니다.

현재 트랜잭션을 커밋하기 위한 호출은 보통 **try...except** 문에서 시도됩니다. 이러한 방식으로 트랜잭션을 성공적으로 커밋할 수 없으면 **except** 블록을 사용하여 오류를 처리하고 작업을 재시도하거나 트랜잭션을 롤백할 수 있습니다.

성공적이지 않은 트랜잭션 종료

트랜잭션의 일부인 변경 사항을 작성하거나 트랜잭션 커밋을 시도할 때 오류가 발생하면 트랜잭션을 구성하는 모든 변경 사항을 취소할 수 있습니다. 모든 변경 사항을 취소하는 것을 트랜잭션을 롤백한다고 표현합니다.

*TDatabase*에서는 다음과 같이 *Rollback* 메소드를 호출하여 트랜잭션을 롤백합니다.

```
MyOracleConnection.Rollback;
```

*TSQLConnection*에서는 *Rollback* 메소드를 사용하지만 *StartTransaction* 메소드에 부여했던 트랜잭션 디스크립터를 제공하여 다음과 같이 롤백할 트랜잭션을 지정해야 합니다.

```
MyOracleConnection.Rollback(TD);
```

*TIBDatabase*에서는 다음과 같이 *Rollback* 메소드를 호출하여 트랜잭션 객체를 롤백합니다.

```
IBDatabase1.DefaultTransaction.Rollback;
```

*TADOConnection*에서는 다음과 같이 *RollbackTrans* 메소드를 호출하여 트랜잭션을 롤백합니다.

```
ADOConnection1.RollbackTrans;
```

트랜잭션이 성공적으로 롤백된 후, ADO 연결 컴포넌트는 *OnRollbackTransComplete* 이벤트를 받습니다. 다른 연결 컴포넌트는 비슷한 이벤트를 받지 않습니다.

현재 트랜잭션을 롤백하는 호출은 보통 다음에서 발생합니다.

- 데이터베이스 오류를 복구할 수 없는 경우의 예외 처리 코드
- 사용자가 Cancel 버튼을 클릭할 때와 같은 버튼이나 메뉴 이벤트 코드

트랜잭션 분리 레벨 지정

트랜잭션 분리 레벨은 동일한 테이블을 사용할 때 트랜잭션이 다른 동시 트랜잭션과 상호 작용하는 방법을 결정합니다. 특히 이 필드는 얼마나 많은 트랜잭션이 테이블에 대한 다른 트랜잭션의 변경 사항을 "보는"지에 영향을 줍니다.

각 서버 타입은 가능한 트랜잭션 분리 레벨의 다른 집합을 지원합니다. 다음과 같이 트랜잭션 분리 레벨에는 세 가지가 있습니다.

- *DirtyRead*: 분리 레벨이 *DirtyRead*인 경우 트랜잭션은 커밋되지 않은 경우에도 다른 트랜잭션에 의해 만들어진 모든 변경 사항을 봅니다. 커밋되지 않은 변경 사항은 영구적이지 않으며 언제든지 롤백될 수도 있습니다. 이 값은 최소한의 분리를 제공하며 많은 데이터베이스 서버(예: Oracle, Sybase, MS-SQL 및 InterBase)에서 사용할 수 없습니다.
- *ReadCommitted*: 분리 레벨이 *ReadCommitted*인 경우 다른 트랜잭션에서 취해진 커밋된 변경 사항만 볼 수 있습니다. 이 설정은 롤백될 수도 있는 커밋되지 않은 변경 사항을 트랜잭션이 보지 못하도록 보호하지만, 사용자가 읽는 중일 때 다른 트랜잭션이 커밋될 경우 데이터베이스 상태가 일치되어 보이지 않을 수도 있습니다. 이 레벨은 BDE가 관리하는 로컬 트랜잭션을 제외한 모든 트랜잭션에서 사용할 수 있습니다.
- *RepeatableRead*: 분리 레벨이 *RepeatableRead*인 경우 트랜잭션은 일관된 상태의 데이터베이스 데이터를 볼 수 있습니다. 트랜잭션은 데이터의 단일 스냅샷을 봅니다. 트랜잭션은 다른 동시 트랜잭션이 커밋된 경우에도 다른 트랜잭션에 의한 이후의 데이터 변경 사항을 볼 수 없습니다. 이 분리 레벨은 분명 사용자의 트랜잭션이 레코드를 읽고 난 후에는 레코드의 뷰가 변하지 않게 합니다. 이 레벨에서 트랜잭션은 다른 트랜잭션에 의한 변경 사항으로부터 가장 많이 분리됩니다. 이 레벨은 Sybase와 MS-SQL 같은 일부 서버에서 사용할 수 없고 BDE가 관리하는 로컬 트랜잭션에서도 사용할 수 없습니다.

또한 *TSQLConnection*은 사용자가 데이터베이스 특정 사용자 지정 분리 레벨을 지정할 수 있게 합니다. 사용자 지정 분리 레벨은 *dbExpress* 드라이버에 의해 정의됩니다. 자세한 내용은 드라이버 설명서를 참조하십시오.

참고 각 분리 레벨의 구현 방법에 대한 자세한 내용은 서버 설명서를 참조하십시오.

*TDatabase*와 *TADOConnection*은 *TransIsolation* 속성을 설정함으로써 사용자가 트랜잭션 분리 레벨을 지정하게 합니다. *TransIsolation*을 데이터베이스 서버가 지원하지 않는 값으로 설정할 때 더 높은 레벨이 가능한 경우, 가장 높은 다음 분리 레벨을 연습니다. 더 높은 레벨이 가능하지 않은 경우, 연결 컴포넌트는 트랜잭션을 시작하려고 할 때 예외를 발생시킵니다.

*TSQLConnection*을 사용하는 경우, 트랜잭션 분리 레벨은 트랜잭션 디스크립터의 *IsolationLevel* 필드에 의해 제어됩니다.

InterBase express를 사용하는 경우, 트랜잭션 분리 레벨은 트랜잭션 매개변수에 의해 제어됩니다.

서버에 명령 전송

*TIBDatabase*를 제외한 모든 데이터베이스 연결 컴포넌트는 *Execute* 메소드를 호출하여 연결된 서버에 SQL 문을 실행하도록 합니다. *Execute*는 문장이 SELECT 문인 경우 커서를 반환할 수 있는데 이러한 사용은 권장되지 않습니다. 데이터를 반환하는 문을 실행하는 더 선호되는 방법은 데이터셋을 사용하는 것입니다.

Execute 메소드는 레코드를 반환하지 않는 단순한 SQL 문을 실행하는 데 매우 편리합니다. 그러한 문장들은 Data Definition Language (DDL) 문을 포함하는데 이는 CREATE INDEX, ALTER TABLE 및 DROP DOMAIN과 같은 데이터베이스의 메타데이터에서 작동하거나 생성합니다. 일부 Data Manipulation Language (DML) SQL 문은 또한 결과 집합을 반환하지 않습니다. 데이터에 대한 작업을 수행하지만 결과 집합을 반환하지 않는 DML 문은 INSERT, DELETE 및 UPDATE 문입니다.

Execute 메소드에 대한 구문은 다음과 같이 연결 타입에 따라 다양합니다.

- *TDatabase*의 경우, *Execute*는 다음과 같은 네 개의 매개변수를 갖습니다. 실행하고자 하는 단일 SQL 문을 지정하는 문자열, 그 문장에 대한 매개변수 값을 제공하는 *TParams* 객체, 재호출로 인해 문장을 캐시로 저장할지 여부를 나타내는 부울, 반환될 수 있는 BDE 커서에 대한 포인터(*nil*을 전달하도록 권장됨)가 그것입니다.
- *TADOConnection*에는 두 가지 버전의 *Execute*가 있습니다. 첫 번째 구문은 SQL 문을 지정하는 *WideString*과 그 문장이 비동기적으로 수행되는지 여부 및 레코드 반환 여부를 제어하는 옵션 집합을 지정하는 매개변수를 갖습니다. 이 첫 번째 구문은 반환된 레코드에 대한 인터페이스를 반환합니다. 두 번째 구문은 SQL 문을 지정하는 *WideString*, 문장이 실행될 때 영향을 받는 레코드의 수를 반환하는 두 번째 매개변수, 문장을 비동기적으로 실행하는지의 여부와 같은 옵션을 지정하는 세 번째 매개변수를 갖습니다. 구문이 매개변수를 전달하지 않는다는 점에 유의하십시오.

- *TSQLConnection*의 *Execute*에서는 다음과 같은 세 개의 매개변수를 갖습니다. 실행하고자 하는 단일 SQL 문을 지정하는 문자열, 문장에 대한 매개변수 값을 제공하는 *TParams* 객체, 레코드를 반환하기 위해 생성된 *TCustomSQLDataSet*을 받을 수 있는 포인터가 그것입니다.

참고 *Execute*는 한 번에 하나의 SQL 문만 실행할 수 있습니다. SQL 스크립트 유틸리티에서와 같이 *Execute*의 단일 호출로 여러 SQL 문을 실행하는 것은 불가능합니다. 하나 이상의 문장을 실행하려면 *Execute*를 반복해서 호출합니다.

어떤 매개변수도 포함하지 않는 문을 실행하는 것은 비교적 쉽습니다. 예를 들어, 다음 코드는 *TSQLConnection* 컴포넌트의 매개변수 없이 CREATE TABLE 문(DDL 문)을 실행합니다.

```

procedure TForm1.CreateTableButtonClick(Sender:TObject);
var
    SQLstmt: String;
begin
    SQLConnection1.Connected := True;
    SQLstmt := 'CREATE TABLE NewCusts ' +
        '( ' +
        ' CustNo INTEGER, ' +
        ' Company CHAR(40), ' +
        ' State CHAR(2), ' +
        ' PRIMARY KEY (CustNo) ' +
        ')';
    SQLConnection1.Execute(SQLstmt, nil, nil);
end;

```

매개변수를 사용하려면 *TParams* 객체를 생성해야 합니다. 각각의 매개변수 값에 대해 *TParams.CreateParam* 메소드를 사용하여 *TParam* 객체를 추가합니다. 그런 다음 *TParam* 속성을 사용하여 매개변수를 설명하고 값을 설정합니다.

이 과정은 *TDatabase*를 사용하여 INSERT 문을 실행하는 다음 예제에서 설명됩니다. INSERT 문은 *StateParam*이라는 단일 매개변수를 갖습니다. *stmtParams*라는 *TParams* 객체가 생성되어 해당 매개변수에 "CA" 값을 제공합니다.

```

procedure TForm1.INSERT_WithParamsButtonClick(Sender:TObject);
var
    SQLstmt: String;
    stmtParams: TParams;
begin
    stmtParams := TParams.Create;
    try
        Database1.Connected := True;
        stmtParams.CreateParam(ftString, 'StateParam', ptInput);
        stmtParams.ParamByName('StateParam').AsString := 'CA';
        SQLstmt := 'INSERT INTO "Custom.db" ' +
            '(CustNo, Company, State) ' +
            'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
        Database1.Execute(SQLstmt, stmtParams, False, nil);
    finally
        stmtParams.Free;
    end;
end;

```

SQL 문이 매개변수를 포함하는 경우, 이에 대한 값을 제공하기 위해 *TParam* 객체를 제공하지 않으면 SQL 문이 실행될 때 오류가 발생할 수 있습니다 (이것은 백엔드에서 사용되는 특정 데이터베이스에 따라 다릅니다.). *TParam* 객체가 제공되지만 SQL 문에 일치하는 매개변수가 없는 경우에는 애플리케이션이 *TParam*을 사용하려고 시도할 때 예외가 발생합니다.

연결된 데이터셋 작업

모든 데이터베이스 연결 컴포넌트는 데이터베이스에 연결하는 데 사용하는 모든 데이터셋의 목록을 유지합니다. 예를 들면, 연결 컴포넌트는 데이터베이스 연결을 닫을 때 이 목록을 사용하여 모든 데이터셋을 닫습니다.

또한 이 목록을 사용하여 특정 데이터베이스에 연결하기 위해 특정 연결 컴포넌트를 사용하는 모든 데이터셋에 대한 작업을 수행할 수 있습니다.

서버로부터 연결을 해제하지 않고 데이터셋 닫기

연결 컴포넌트는 연결을 닫을 때 모든 데이터셋을 자동으로 닫습니다. 하지만 데이터베이스 서버로부터 연결을 해제하지 않고 모든 데이터셋을 닫고자 하는 경우가 있습니다.

서버로부터 연결 해제하지 않고 모든 열린 데이터셋을 닫기 위해 *CloseDataSets* 메소드를 사용할 수 있습니다.

*TADOConnection*과 *TIBDatabase*에서 *CloseDataSets*를 호출하면 연결이 항상 열린 상태로 둡니다. *TDatabase*와 *TSQLConnection*에서는 또한 *KeepConnection* 속성을 *True*로 설정해야 합니다.

연결된 데이터셋을 통한 반복

연결 컴포넌트를 사용하는 모든 데이터셋에 대한 작업(데이터셋을 모두 닫는 작업은 제외)을 수행하려면 *DataSets* 및 *DataSetCount* 속성을 사용합니다. *DataSets*는 연결 컴포넌트에 연결된 모든 데이터셋의 인덱스화된 배열입니다. *TADOConnection*을 제외한 모든 연결 컴포넌트에서 이 목록은 활성 데이터셋만을 포함합니다. *TADOConnection*은 비활성 데이터셋을 나열합니다. *DataSetCount*는 이 배열에 있는 데이터셋의 수입니다.

참고 특수화된 클라이언트 데이터셋을 사용하여 업데이트 내용을 캐시로 저장하는 경우(일반적인 클라이언트 데이터셋인 *TClientDataSet*과 반대), *DataSets* 속성은 클라이언트 데이터셋 자체가 아닌 클라이언트 데이터셋이 소유한 내부 데이터셋을 나열합니다.

*DataSetCount*로 *DataSets*을 사용하여 코드의 현재 활성 상태인 모든 데이터셋을 순환할 수 있습니다. 예를 들면, 다음 코드에서는 모든 활성 데이터셋을 순환하고 제공하는 데이터를 사용하는 모든 컨트롤을 사용 불가능하게 합니다.

```
var  
  I:Integer;  
begin  
  with MyDBConnection do
```



```

begin
  for I := 0 to DataSetCount - 1 do
    DataSets[I].DisableControls;
  end;
end;

```

참고 *TADOConnection*은 데이터셋뿐만 아니라 명령 객체를 지원합니다. *Commands*와 *CommandCount* 속성을 사용하여 데이터셋을 통한 반복과 매우 유사하게 이를 통해 반복할 수 있습니다.

메타데이터 얻기

모든 데이터베이스 연결 컴포넌트는 검색하는 메타데이터의 타입이 다양함에도 불구하고 데이터베이스 서버에 대한 메타데이터의 목록을 검색할 수 있습니다. 메타데이터를 검색하는 메소드는 문자열 목록을 서버에서 사용 가능한 다양한 엔티티의 이름으로 채웁니다. 예를 들어, 이 정보를 사용하여 사용자는 런타임 시 테이블을 동적으로 선택할 수 있습니다.

TADOConnection 컴포넌트를 사용하여 ADO 데이터 저장에서 사용 가능한 테이블과 내장 프로시저(stored procedures)에 대한 메타데이터를 검색할 수 있습니다. 예를 들어, 이 정보를 사용하여 사용자는 런타임 시 테이블이나 내장 프로시저를 동적으로 선택할 수 있습니다.

사용 가능한 테이블 열거

GetTableNames 메소드는 기존의 문자열 목록 객체에 테이블 이름 목록을 복사합니다. 예를 들면, 사용자가 테이블을 열기 위해 선택할 때 사용할 수 있는 테이블 이름으로 리스트 박스를 채우는 데 사용될 수 있습니다. 다음 줄은 데이터베이스의 모든 테이블 이름으로 리스트 박스를 채웁니다.

```
MyDBConnection.GetTableNames(ListBox1.Items, False);
```

*GetTableNames*는 다음과 같은 두 개의 매개변수, 즉 테이블 이름으로 채울 문자열 목록과 그 목록에 시스템 테이블과 순서 테이블 중 어느 것을 포함해야 할지를 지정하는 부울(boolean), 두 가지가 있습니다. 모든 서버가 메타데이터를 저장하기 위해 시스템 테이블을 사용하는 것은 아닙니다. 따라서 시스템 테이블을 요구했을 때 빈 목록이 반환될 수도 있습니다.

참고 대부분의 데이터베이스 연결 컴포넌트에서 *GetTableNames*는 두 번째 매개변수가 *False*일 때 모든 사용 가능한 시스템 이외의 테이블 목록을 반환합니다. 하지만 *TSQLConnection*에서는 시스템 테이블의 이름만을 페치(fetch)하지 않을 때 사용자는 목록에 추가되는 타입을 더 많이 제어합니다. *TSQLConnection*을 사용하는 경우, 목록에 추가되는 이름의 타입은 *TableScope* 속성에 의해 제어됩니다. *TableScope*는 목록이 보통 테이블, 시스템 테이블, 동의어, 뷰 중에서 일부나 전체를 포함해야 할지 여부를 지정합니다.

테이블의 필드 열거

GetFieldNames 메소드는 기존 문자열 목록을 특정 테이블에 있는 모든 필드(열)의 이름으로 채웁니다. *GetFieldNames*는 필드를 열거하려는 테이블의 이름 및 필드 이름으로 채워야 할 기존 문자열 목록이라는 두 개의 매개변수를 갖습니다.

```
MyDBConnection.GetFieldNames('Employee', ListBox1.Items);
```

사용 가능한 내장 프로시저(Stored procedures) 열거

데이터베이스에 포함된 모든 내장 프로시저 목록을 얻으려면 *GetProcedureNames* 메소드를 사용합니다. 이 메소드는 다음과 같은 단일 매개변수, 즉 채워야 하는 기존 문자열 목록을 갖습니다.

```
MyDBConnection.GetProcedureNames(ListBox1.Items);
```

참고 *GetProcedureNames*는 *TADOConnection*과 *TSQLConnection*에 대해서만 사용 가능합니다.

사용 가능한 인덱스 열거

특정 테이블에 대해 정의된 모든 인덱스 목록을 얻으려면 *GetIndexNames* 메소드를 사용합니다. 이 메소드는 다음과 같은 두 개의 매개변수, 즉 인덱스를 필요로 하는 테이블과 채워야 하는 기존 문자열 목록을 갖습니다.

```
SQLConnection1.GetIndexNames('Employee', ListBox1.Items);
```

참고 대부분의 테이블 타입 데이터셋은 동일한 메소드를 갖지만 *GetIndexNames*는 *TSQLConnection*에 대해서만 사용 가능합니다.

내장 프로시저 매개변수 열거

특정 내장 프로시저에 대해 정의된 모든 매개변수의 목록을 얻으려면 *GetProcedureParams* 메소드를 사용합니다. *GetProcedureParams*는 각각의 레코드가 이름, 인덱스, 매개변수 타입, 필드 타입 등을 포함한 특정 내장 프로시저의 매개변수를 설명하는 매개변수 설명 레코드에 대한 포인터로 *TList* 객체를 채웁니다.

*GetProcedureParams*는 다음과 같은 두 개의 매개변수, 즉 내장 프로시저의 이름과 채워야 하는 기존 *TList* 객체를 갖습니다.

```
SQLConnection1.GetProcedureParams('GetInterestRate', List1);
```

전역 *LoadParamListItems* 프로시저를 호출함으로써 목록에 추가된 매개변수 설명을 더 익숙한 *TParams* 객체로 변환할 수 있습니다. *GetProcedureParams*는 개별 레코드를 동적으로 할당하기 때문에 애플리케이션은 정보를 끝마쳤을 때 이를 해제해야 합니다. 전역 *FreeProcParams* 루틴이 이를 수행합니다.

참고 *GetProcedureParams*는 *TSQLConnection*에서만 사용 가능합니다.

18

데이터셋 이해

데이터 액세스의 기본 유닛은 객체의 데이터셋 패밀리에 있습니다. 애플리케이션은 모든 데이터베이스 액세스를 위해 데이터셋을 사용합니다. 데이터셋 객체는 논리 테이블로 구성된 데이터베이스의 레코드 집합을 나타냅니다. 이러한 레코드는 단일 데이터베이스 테이블의 레코드이거나 쿼리나 내장 프로시저를 실행한 결과를 표시할 수 있습니다.

데이터베이스 애플리케이션에서 사용하는 모든 데이터셋 객체는 *TDataSet*의 자손이며 이 클래스에서 데이터 필드, 속성, 이벤트 및 메소드를 상속합니다. 이 장에서는 데이터베이스 애플리케이션에서 사용할 데이터셋 객체가 상속하는 *TDataSet*의 기능에 대해 설명합니다. 데이터셋 객체를 사용하려면 이러한 공유 기능을 이해해야 합니다.

*TDataSet*은 가상화된 데이터셋인데 이는 속성과 메소드의 많은 부분이 **virtual** 또는 **abstract**임을 의미합니다. *가상 메소드*는 자손 객체에 메소드의 구현을 오버라이드할 수 있는(대개는 오버라이드되는) 함수 또는 프로시저 선언입니다. 실제 구현이 없는 함수 또는 프로시저 선언을 *추상 메소드*라고 합니다. 선언은 모든 자손 데이터셋 객체에 구현되어야 하지만 객체마다 다르게 구현될 수 있는 메소드(매개변수와 반환 타입)의 프로토타입입니다.

*TDataSet*은 **abstract** 메소드를 포함하기 때문에 런타임 오류를 발생하지 않고 애플리케이션에서 직접 사용할 수 없습니다. 그 대신 기본 제공 *TDataSet* 자손의 인스턴스를 만들어 애플리케이션에서 사용하거나 *TDataSet* 또는 그 자손으로부터 사용자 고유의 데이터셋 객체를 파생하여 모든 **abstract** 메소드 구현을 작성합니다.

*TDataSet*은 모든 데이터셋 객체에 공통적인 많은 부분을 정의합니다. 예를 들어, *TDataSet*은 둘 이상의 데이터베이스 테이블의 실제 열에 해당하는 *TField* 컴포넌트의 배열, 애플리케이션에서 제공하는 조회(lookup) 필드 또는 애플리케이션에서 제공하는 계산된 필드 등 모든 데이터셋의 기본 구조를 정의합니다. *TField* 컴포넌트에 대한 자세한 내용은 19장 "필드 컴포넌트 사용"을 참조하십시오.

이 장에서는 *TDataSet*에 들어 있는 공통적인 데이터베이스 기능의 사용 방법을 설명합니다. 하지만 *TDataSet*에 이 기능에 대한 메소드가 있어도 모든 *TDataSet* 자손이 이 메소드를 구현하지는 않는다는 점에 유의하십시오. 특히 단방향 데이터셋은 제한된 서브셋만 구현합니다.

TDataSet 자손 사용

*TDataSet*에는 여러 직계 자손이 있으며 각 자손은 서로 다른 데이터 액세스 메커니즘에 해당합니다. 이 자손을 직접 사용하지는 않습니다. 오히려 각 자손은 특정 데이터 액세스 메커니즘을 사용하는 메소드 및 속성을 갖습니다. 그러면 이러한 속성 및 메소드는 다른 타입의 서버 데이터에 적용된 자손 클래스에 의해 노출됩니다. *TDataSet*에는 다음과 같은 직계 자손이 있습니다.

- *TBDEDataSet*은 BDE(Borland Database Engine)를 사용하여 데이터베이스 서버와 통신합니다. 사용자가 사용하는 *TBDEDataSet* 자손에는 *TTable*, *TQuery*, *TStoredProc* 및 *TNestedTable*이 있습니다. BDE 활성 데이터셋의 고유한 기능에 대해서는 20장 "Borland Database Engine 사용"에서 설명합니다.
- *TCustomADODataset*은 ADO(ActiveX Data Objects)를 사용하여 OLEDB 데이터 저장소와 통신합니다. 사용자가 사용하는 *TCustomADODataset* 자손에는 *TADODataset*, *TADOTable*, *TADOQuery* 및 *TADOStoredProc*가 있습니다. ADO 기반 데이터셋의 고유한 기능에 대해서는 21장 "ADO 컴포넌트 사용"에서 설명합니다.
- *TCustomSQLDataSet*은 dbExpress를 사용하여 데이터베이스 서버와 통신합니다. 사용자가 사용하는 *TCustomSQLDataSet* 자손에는 *TSQLDataSet*, *TSQLTable*, *TSQLQuery* 및 *TSQLStoredProc*가 있습니다. dbExpress 데이터셋의 고유한 기능에 대해서는 22장 "단방향 데이터셋 사용"에서 설명합니다.
- *TIBCustomDataSet*은 InterBase 데이터베이스 서버와 직접 통신합니다. 사용자가 사용하는 *TIBCustomDataSet* 자손에는 *TIBDataSet*, *TIBTable*, *TIBQuery* 및 *TIBStoredProc*가 있습니다.
- *TCustomClientDataSet*은 다른 데이터셋 컴포넌트의 데이터 또는 디스크에 있는 전용 파일의 데이터를 나타냅니다. 사용자가 사용하는 *TCustomClientDataSet* 자손에는 외부(소스) 데이터셋에 연결할 수 있는 *TClientDataSet*과 내부 소스 데이터셋을 사용하는 특정 데이터 액세스 메커니즘으로 특수화되는 클라이언트 데이터셋(*TBDEClientDataSet*, *TSQLClientDataSet* 및 *TIBClientDataSet*)이 있습니다. 클라이언트 데이터셋의 고유한 기능에 대해서는 23장 "클라이언트 데이터셋 사용"에서 설명합니다.

이러한 *TDataSet* 자손에서 사용하는 다양한 데이터 액세스 메커니즘의 장단점에 대해서는 14-1 페이지의 "데이터베이스 사용"에서 설명합니다.

기본 제공 데이터셋뿐만 아니라 고유한 사용자 정의 *TDataSet* 자손을 직접 만들 수 있습니다. 예를 들면, 스프레드시트와 같은 데이터베이스 서버 이외의 프로세스로부터 데이터를 제공하도록 할 수 있습니다. 사용자 지정 데이터셋을 작성하면 VCL 데이터 컨트롤을 사용하여 사용자 인터페이스를 구축할 수 있으면서도 선택한 모든 메소드를 사

용하여 유연성 있게 데이터를 관리할 수 있습니다. 사용자 지정 컴포넌트 만들기에 대한 자세한 내용은 40장 "컴포넌트 생성 개요"를 참조하십시오.

각 *TDataSet* 자손에는 고유한 속성 및 메소드가 있지만 자손 클래스에 있는 일부 속성 및 메소드는 다른 데이터 액세스 메커니즘을 사용하는 다른 자손 클래스에 있는 속성 및 메소드와 동일합니다. 예를 들어, "테이블" 컴포넌트 (*TTable*, *TADOTable*, *TSQLTable* 및 *TIBTable*) 사이에는 유사점이 있습니다. *TDataSet* 자손에 있는 공통점에 대한 자세한 내용은 18-23 페이지의 "데이터셋 형식"을 참조하십시오.

데이터셋 상태 알아보기

데이터셋의 *상태*(또는 *모드*)는 데이터에 수행될 작업을 결정합니다. 예를 들어, 데이터셋이 닫혀 있을 때 데이터셋의 상태는 *dsInactive*이며, 이는 데이터에 어떤 작업도 수행될 수 없다는 것을 의미합니다. 런타임 시 데이터셋의 읽기 전용 *State* 속성을 검사하여 현재 상태를 알 수 있습니다. 다음 표는 *State* 속성에 대한 가능한 값과 의미를 요약해서 보여 줍니다.

표 18.1 데이터셋 State 속성의 값

값	상태	의미
<i>dsInactive</i>	Inactive	DataSet이 닫혀 있습니다. 데이터를 사용할 수 없습니다.
<i>dsBrowse</i>	Browse	DataSet이 열려 있습니다. 데이터를 볼 수는 있지만 변경할 수 없습니다. 개방형 데이터셋의 기본 상태입니다.
<i>dsEdit</i>	Edit	DataSet이 열려 있습니다. 현재 행을 수정할 수 있습니다. 단방향 데이터셋에서는 지원되지 않습니다.
<i>dsInsert</i>	Insert	DataSet이 열려 있습니다. 새 행이 삽입되거나 추가됩니다. 단방향 데이터셋에서는 지원되지 않습니다.
<i>dsSetKey</i>	SetKey	DataSet이 열려 있습니다. 범위 설정 및 범위와 <i>GotoKey</i> 작동에 사용되는 키 값의 설정을 가능하게 합니다. 모든 데이터셋에서 지원되는 것은 아닙니다.
<i>dsCalcFields</i>	CalcFields	DataSet이 열려 있습니다. <i>OnCalcFields</i> 이벤트가 진행 중임을 나타냅니다. 계산되지 않은 필드에 대한 변경을 방지합니다.
<i>dsCurValue</i>	CurValue	DataSet이 열려 있습니다. 필드의 <i>CurValue</i> 속성이 캐시된 업데이트 적용 시 발생하는 오류에 응답하는 이벤트 핸들러에 페치(fetch)되고 있음을 나타냅니다.
<i>dsNewValue</i>	NewValue	DataSet이 열려 있습니다. 필드의 <i>NewValue</i> 속성이 캐시된 업데이트 적용 시 발생하는 오류에 응답하는 이벤트 핸들러에 페치(fetch)되고 있음을 나타냅니다.
<i>dsOldValue</i>	OldValue	DataSet이 열려 있습니다. 필드의 <i>OldValue</i> 속성이 캐시된 업데이트 적용 시 발생하는 오류에 응답하는 이벤트 핸들러에 페치(fetch)되고 있음을 나타냅니다.
<i>dsFilter</i>	Filter	DataSet이 열려 있습니다. 필터 작동이 진행 중임을 나타냅니다. 제한된 데이터 집합을 볼 수 있지만, 어떤 데이터도 변경될 수 없습니다. 단방향 데이터셋에서는 지원되지 않습니다.

표 18.1 데이터셋 State 속성의 값 (계속)

값	상태	의미
<i>dsBlockRead</i>	Block Read	DataSet이 열려 있습니다. data-aware 컨트롤이 업데이트 되지 않고 현재 레코드가 변경될 때 이벤트가 트리거되지 않습니다.
<i>dsInternalCalc</i>	Internal Calc	DataSet이 열려 있습니다. <i>OnCalcFields</i> 이벤트가 레코드로 저장된 계산된 값에 대해 진행 중에 있습니다. (클라이언트 데이터셋만 해당)
<i>dsOpening</i>	Opening	데이터셋이 열기 작업 중에 있지만 완료되지 않았습니다. 이 상태는 데이터셋이 비동기 폐치로 열릴 때 발생합니다.

일반적으로 애플리케이션은 데이터셋 상태를 검사하여 특정 작업을 수행할 시기를 결정합니다. 예를 들어, *dsEdit*나 *dsInsert* 상태를 검사하여 업데이트를 포스트해야 하는지 확인할 수 있습니다.

참고 데이터셋의 상태가 변경될 때마다 *OnStateChange* 이벤트는 데이터셋에 연결된 모든 데이터 소스 컴포넌트에 대해 호출됩니다. 데이터 소스 컴포넌트와 *OnStateChange*에 대한 자세한 내용은 15-4 페이지의 "데이터 소스에서 변경 사항에 대한 응답"을 참조하십시오.

데이터셋 열기와 닫기

데이터셋의 데이터를 읽거나 쓰려면 애플리케이션은 먼저 데이터셋을 열어야 합니다. 데이터셋은 두 가지 방법으로 열 수 있습니다.

- 디자인 타임에 Object Inspector에서 또는 런타임 시 코드에서 데이터셋의 *Active* 속성을 *True*로 설정합니다.

```
CustTable.Active := True;
```

- 런타임 시 데이터셋에 대한 *Open* 메소드를 호출합니다.

```
CustQuery.Open;
```

데이터셋을 열면 데이터셋은 먼저 *BeforeOpen* 이벤트를 받은 다음 커서를 열어 데이터로 채우고 마지막으로 *AfterOpen* 이벤트를 받습니다.

새로 열린 데이터셋은 찾아보기 모드에 있으므로 애플리케이션이 데이터를 읽고 탐색할 수 있습니다.

데이터셋은 두 가지 방법으로 닫을 수 있습니다.

- 디자인 타임에 Object Inspector에서 또는 런타임 시 코드에서 데이터셋의 *Active* 속성을 *False*로 설정합니다.

```
CustQuery.Active := False;
```

- 런타임 시 데이터셋에 대한 *Close* 메소드를 호출합니다.

```
CustTable.Close;
```

데이터셋을 열 때 *BeforeOpen* 및 *AfterOpen* 이벤트를 받는 것처럼 닫을 때는 *BeforeClose* 및 *AfterClose* 이벤트를 받습니다. 핸들러는 데이터셋의 *Close* 메소드에 응답합니다.

예를 들어, 이 이벤트를 사용하면 보류 중인 변경 내용을 게시하거나 데이터셋을 닫기 전에 변경 내용을 취소할지를 묻는 메시지를 표시할 수 있습니다. 다음 코드는 이러한 핸들러를 보여 줍니다.

```

procedure TForm1.CustTableVerifyBeforeClose(DataSet:TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Post changes before closing?', mtConfirmation, mbYesNoCancel, 0)
    of
      mrYes:CustTable.Post; { save the changes }
      mrNo:CustTable.Cancel; { abandon the changes}
      mrCancel:Abort;      { abort closing the dataset }
    end;
  end;
end;

```

참고 *TTable* 컴포넌트의 *TableName*과 같은 특정 속성을 변경하려고 할 때는 데이터셋을 닫아야 할 수도 있습니다. 데이터셋을 다시 열 때 새로운 속성 값이 적용됩니다.

데이터셋 탐색

각각의 활성 데이터셋은 데이터셋의 현재 행에 커서나 포인터가 있습니다. 데이터셋의 현재 행은 필드 값이 *TDBEdit*, *TDBLabel* 및 *TDBMemo*와 같이 폼의 단일 필드의 data-aware 컨트롤에 현재 나타나는 행입니다. 데이터셋이 편집을 지원하면 현재 레코드는 편집, 삽입 및 삭제 메소드로 처리될 수 있는 값을 포함합니다.

커서를 이동시켜 다른 행을 가리키면 현재 행을 변경할 수 있습니다. 다음 표는 다른 레코드로 이동하기 위해 애플리케이션 코드에서 사용할 수 있는 메소드를 나열한 것입니다.

표 18.2 데이터셋의 탐색 메소드

메소드	커서 이동
<i>First</i>	데이터셋의 첫 번째 행
<i>Last</i>	데이터셋의 마지막 행(단방향 데이터셋에서는 사용할 수 없음)
<i>Next</i>	데이터셋의 다음 행
<i>Prior</i>	데이터셋의 이전 행(단방향 데이터셋에서는 사용할 수 없음)
<i>MoveBy</i>	데이터셋의 앞 또는 뒤쪽 행의 지정된 수

data-ware 비주얼 컴포넌트 *TDBNavigator*는 사용자가 런타임 시 레코드 간에 이동할 때 클릭하는 버튼으로 이러한 메소드를 캡슐화합니다. 탐색기 컴포넌트에 대한 자세한 내용은 15-28 페이지의 "레코드 탐색 및 처리"를 참조하십시오.

이러한 메소드 중 하나를 사용하거나 검색 조건을 기반으로 탐색하는 다른 메소드로 현재 레코드를 변경할 때마다 데이터셋은 두 가지 이벤트, *BeforeScroll*(현재 레코드를 빠져 나가기 전) 및 *AfterScroll*(새 레코드에 도달한 후)을 받습니다. 이 이벤트를 사용하면 사용자 인터페이스를 업데이트할 수 있습니다. 예를 들어, 현재 레코드에 대한 정보를 나타내는 상태 표시줄을 업데이트할 수 있습니다.

또한 *TDataSet*은 데이터셋의 레코드를 통해 반복할 때 유용한 정보를 제공하는 두 개의 부울 속성을 정의합니다.

표 18.3 데이터셋의 탐색 속성

속성	설명
<i>Bof</i> (Beginning-of-file)	<i>True</i> : 커서가 데이터셋의 첫 번째 행에 있습니다. <i>False</i> : 커서가 데이터셋의 첫 번째 행에 없습니다.
<i>Eof</i> (End-of-file)	<i>True</i> : 커서가 데이터셋의 마지막 행에 있습니다. <i>False</i> : 커서가 데이터셋의 마지막 행에 없습니다.

First 및 Last 메소드 사용

First 메소드는 커서를 데이터셋의 첫 번째 행으로 이동하고 *Bof* 속성을 *True*로 설정합니다. 커서가 이미 데이터셋의 첫 번째 행에 있으면 *First* 메소드는 아무 것도 하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 첫 번째 레코드로 이동합니다.

```
CustTable.First;
```

Last 메소드는 커서를 데이터셋의 마지막 행으로 이동하고 *Eof* 속성을 *True*로 설정합니다. 커서가 이미 데이터셋의 마지막 행에 있으면 *Last* 메소드는 아무 것도 하지 않습니다.

다음 코드는 *CustTable*의 마지막 레코드로 이동합니다.

```
CustTable.Last;
```

참고 *Last* 메소드는 단방향 데이터셋에서 예외를 발생시킵니다.

팁 사용자의 조작 없이 데이터셋의 첫 번째 또는 마지막 행으로 이동하도록 프로그램으로 만들 수도 있지만 사용자가 *TDBNavigator* 컴포넌트를 사용하여 레코드에서 레코드로 탐색하도록 할 수도 있습니다. 활성이고 보이는 (visible) 탐색기 컴포넌트는 사용자가 활성 데이터셋의 첫 번째 및 마지막 행으로 이동할 수 있는 버튼을 포함합니다. 이러한 버튼에 대한 *OnClick* 이벤트는 데이터셋의 *First* 및 *Last* 메소드를 호출합니다. 탐색기 컴포넌트의 효과적인 사용에 대한 자세한 내용은 15-28 페이지의 "레코드 탐색 및 처리"를 참조하십시오.

Next 및 Prior 메소드 사용

Next 메소드는 데이터셋에서 한 행 앞으로 커서를 이동하고 데이터셋이 비어 있지 않으면 *Bof* 속성을 *False*로 설정합니다. *Next*를 호출할 때 이미 커서가 데이터셋의 마지막 행에 있으면 아무 것도 발생하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 다음 레코드로 이동합니다.

```
CustTable.Next;
```


Prior 메소드는 데이터셋의 한 행 뒤로 커서를 이동하고 데이터셋이 비어 있지 않으면 *Eof*를 *False*로 설정합니다. *Prior*를 호출할 때 커서가 이미 데이터셋의 첫 번째 행에 있으면 *Prior*는 아무 것도 하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 이전 레코드로 이동합니다.

```
CustTable.Prior;
```

참고 *Prior* 메소드는 단방향 데이터셋에서 예외를 발생시킵니다.

MoveBy 메소드 사용

*MoveBy*를 사용하면 데이터셋에서 커서를 앞이나 뒤의 행으로 얼마나 많이 이동할지 지정할 수 있습니다. *MoveBy*가 호출되면 현재 레코드에 상대적으로 이동합니다. *MoveBy*는 또한 필요에 따라 적절히 데이터셋의 *Bof* 및 *Eof* 속성을 설정합니다.

이 함수는 이동할 레코드의 수인 정수 매개변수를 사용합니다. 양의 정수는 앞으로 이동을 나타내고, 음의 정수는 뒤로 이동을 나타냅니다.

참고 *MoveBy*는 음의 인수를 사용할 때 단방향 데이터셋에서 예외를 발생시킵니다.

*MoveBy*는 이동하는 행의 수를 반환합니다. 데이터셋의 시작 또는 끝 부분을 지나서 이동하려는 경우, *MoveBy*에서 반환한 행의 수는 이동을 요청한 행의 수와 다릅니다. *MoveBy*가 데이터셋의 첫 번째 레코드나 마지막 레코드에 도달하면 멈추기 때문입니다.

다음 코드는 *CustTable* 두 레코드 뒤로 커서를 이동합니다.

```
CustTable.MoveBy(-2);
```

참고 애플리케이션이 다중 사용자 데이터베이스 환경에서 *MoveBy*를 사용하면 데이터셋이 유동적임을 기억하십시오. 여러 사용자가 동시에 데이터베이스에 액세스하여 데이터를 변경하면 조금 전에 다섯 레코드 뒤에 있었던 레코드가 지금은 넷, 여섯 또는 알 수 없는 숫자만큼의 레코드 뒤에 있을 수 있습니다.

Eof 및 Bof 속성 사용

두 개의 읽기 전용 런타임 속성인 *Eof*(End-of-file) 및 *Bof*(Beginning-of-file)는 데이터셋의 모든 레코드를 순환하려는 경우에 유용합니다.

Eof

*Eof*가 *True*인 경우 커서가 데이터셋의 마지막 행에 있다는 것을 나타냅니다. 애플리케이션이 다음의 작업을 수행할 때 *Eof*는 *True*로 설정됩니다.

- 빈 데이터셋을 엽니다.
- 데이터셋의 *Last* 메소드를 호출합니다.
- 데이터셋의 *Next* 메소드를 호출했지만 커서가 현재 데이터셋의 마지막 행에 있기 때문에 메소드가 실패합니다.

- 빈 범위 또는 데이터셋에서 *SetRange*를 호출합니다.

다른 모든 경우에는 *Eof*가 *False*로 설정됩니다. 위의 조건 중 하나라도 만족하지 않고, 속성을 직접 테스트하지 않은 경우에는 *Eof*를 *False*로 생각해야 합니다.

*Eof*는 일반적으로 순환문의 조건에서 테스트되어 데이터셋의 모든 레코드의 반복 처리를 제어합니다. 레코드를 포함하는 데이터셋을 열거나 *First*를 호출하면 *Eof*는 *False*입니다. 데이터셋을 통해 한 번에 한 레코드씩 반복하려면 *Next*를 호출하여 각 레코드를 하나씩 통과하고 *Eof*가 *True*일 때 종료하는 순환문을 만듭니다. *Eof*는 커서가 이미 마지막 레코드에 있을 때 사용자가 *Next*를 호출하기 전까지는 *False* 상태로 남습니다.

다음 코드는 *CustTable*이라는 데이터셋에 대한 레코드 처리 순환문을 코딩할 수 있는 방법 중 한 가지를 보여 줍니다.

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets Eof False }
  while not CustTable.Eof do { Cycle until Eof is True }
  begin
    { Process each record here }
    ?
    CustTable.Next; { Eof False on success; Eof True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

팁 이 예제는 또한 데이터셋에 연결된 data-aware 비주얼 컨트롤을 사용 가능 및 사용 불가능하게 하는 방법을 보여 줍니다. 데이터셋 반복 동안 비주얼 컨트롤을 사용 불가능하게 하면 현재 레코드가 변경될 때 애플리케이션이 컨트롤의 내용을 업데이트할 필요가 없기 때문에 처리 속도가 빨라집니다. 반복이 완료된 후, 컨트롤을 다시 사용 가능하게 해야 새로운 현재 행의 값으로 업데이트할 수 있습니다. 비주얼 컨트롤을 사용 가능하게 하는 것은 **try...finally** 문의 **finally** 절에서 이루어집니다. 이는 예외로 인해 순환문 처리가 너무 빨리 종료된 경우라도 컨트롤이 사용 불가능한 상태로 남지 않도록 보장합니다.

Bof

*Bof*가 *True*인 경우는 커서가 명백히 데이터셋의 첫 번째 행에 있다는 것을 나타냅니다. *Bof*는 애플리케이션이 다음과 같은 작업을 수행할 때에는 *True*로 설정됩니다.

- 데이터셋을 엽니다.
- 데이터셋의 *First* 메소드를 호출합니다.
- 데이터셋의 *Prior* 메소드를 호출하면 커서가 현재 데이터셋의 첫 번째 행에 있기 때문에 메소드가 실패합니다.
- 빈 범위 또는 데이터셋에서 *SetRange*를 호출합니다.

*Bof*는 다른 모든 경우에 *False*로 설정됩니다. 위의 조건 중 하나라도 만족하지 않는 경우, 또한 속성을 직접 테스트하지 않는 경우에 *Bof*는 *False*라고 생각해야 합니다.

*Eof*와 같이 *Bof*는 데이터셋의 반복적인 레코드 처리를 제어하기 위해 순환문 조건에 있을 수 있습니다. 다음 코드는 *CustTable*이라는 데이터셋에 대한 레코드 처리 순환문을 코딩할 수 있는 방법 중 한 가지를 보여 줍니다.

```
CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.Bof do { Cycle until Bof is True }
  begin
    { Process each record here }
    :
    CustTable.Prior; { Bof False on success; Bof True when Prior fails on first record }
  end;
finally
  CustTable.EnableControls; { Display new current row in controls }
end;
```

레코드 표시 및 반환

데이터셋의 레코드에서 레코드로 이동하는 것(또는 특정 레코드 수만큼 한 레코드에서 다른 레코드로 이동) 외에도 데이터셋의 특정 위치를 표시하여 원할 때마다 표시된 위치로 빨리 돌아올 수 있습니다. *TDataSet*에는 *Bookmark* 속성 및 다섯 개의 북마크 메소드로 구성된 북마크 기능이 있습니다.

*TDataSet*은 **virtual** 북마크 메소드를 구현합니다. 이 메소드는 북마크 메소드가 호출되는 경우, *TDataSet*에서 파생된 데이터셋 객체가 값을 반환하도록 보장하지만 반환 값은 현재 위치를 추적하지 않는 단순한 기본값입니다. *TDataSet* 자손은 북마크에 제공하는 지원 레벨이 다양합니다. dbExpress 데이터셋은 북마크 지원을 추가하지 않습니다. ADO 데이터셋은 원본으로 사용한 데이터베이스 테이블에 따라 북마크를 지원할 수 있습니다. BDE 데이터셋, InterBase express 데이터셋 및 클라이언트 데이터셋은 항상 북마크를 지원합니다.

Bookmark 속성

Bookmark 속성은 애플리케이션의 여러 북마크 중에서 어떤 북마크가 현재 북마크인지 나타냅니다. *Bookmark*는 현재 북마크를 식별하는 문자열입니다. 다른 북마크를 추가할 때마다 그 북마크가 현재 북마크가 됩니다.

GetBookmark 메소드

북마크를 만들려면 애플리케이션에서 *TBookmark* 타입의 변수를 선언한 다음 *GetBookmark*를 호출하여 변수에 대한 저장소를 할당하고 데이터셋의 특정 위치로 그 값을 설정합니다. *TBookmark* 타입은 포인터입니다.

GotoBookmark 및 BookmarkValid 메소드

*GotoBookmark*는 북마크를 전달할 때 데이터셋의 커서를 북마크에서 지정된 위치로 이동합니다. *GotoBookmark*를 호출하기 전에 *BookmarkValid*를 호출하면 북마크가 레코드를

가리키는지 확인할 수 있습니다. 지정된 북마크가 레코드를 가리키면 *BookmarkValid*는 *True*를 반환합니다.

CompareBookmarks 메소드

또한 *CompareBookmarks*를 호출하여 이동하려는 북마크가 다른(또는 현재) 북마크와 다른지 확인할 수 있습니다. 두 북마크가 동일한 레코드를 참조하거나 두 값이 모두 *nil*이면 *CompareBookmarks*는 0을 반환합니다.

FreeBookmark 메소드

*FreeBookmark*는 지정된 북마크가 더 이상 필요하지 않을 때 지정된 북마크에 할당된 메모리를 해제합니다. 또한 기존 북마크를 재사용하기 전에 *FreeBookmark*를 호출해야 합니다.

북마크 예제

다음 코드는 북마크 사용 예제를 보여 줍니다.

```

procedure DoSomething (const Tbl:Ttable)
var
  Bookmark:TBookmark;
begin
  Bookmark := Tbl.GetBookmark; { Allocate memory and assign a value }
  Tbl.DisableControls; { Turn off display of records in data controls }
  try
    Tbl.First; { Go to first record in table }
    while not Tbl.Eof do {Iterate through each record in table }
    begin
      { Do your processing here }
      :
      Tbl.Next;
    end;
  finally
    Tbl.GotoBookmark(Bookmark);
    Tbl.EnableControls; { Turn on display of records in data controls, if necessary }
    Tbl.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
  end;
end;

```

레코드를 반복하기 전에 컨트롤은 사용 불가능합니다. 레코드 반복 도중에 오류가 발생하는 경우, **finally** 절을 사용하여 컨트롤이 항상 사용 가능한지와 순환문이 너무 빨리 종료되더라도 북마크가 항상 해제되어 있는지 보장할 수 있습니다.

데이터셋 검색

데이터셋이 단방향성이 아닌 경우에는 *Locate* 및 *Lookup* 메소드를 사용하여 데이터셋을 검색할 수 있습니다. 이러한 메소드를 사용하면 모든 데이터셋의 모든 열 타입을 검색할 수 있습니다.

참고 일부 *TDataSet* 자손에는 인덱스를 기반으로 검색하는 추가 메소드 패밀리가 있습니다. 이러한 추가 메소드에 대한 자세한 내용은 18-28 페이지의 "인덱스를 사용하여 레코드 검색"을 참조하십시오.

Locate 사용

*Locate*는 지정된 검색 조건과 일치하는 첫 번째 행으로 커서를 이동합니다. 가장 간단한 폼에서 검색할 열 이름, 일치하는 필드 값 및 검색이 대소문자 구별인지 또는 부분 키 일치룰 사용할 수 있는지를 지정하는 옵션 플래그를 *Locate*에 전달합니다. (부분 키 일치는 기준 문자열이 필드 값의 접두사일 필요가 있을 경우입니다.) 예를 들어, 다음 코드는 *Company* 열의 값이 "Professional Divers, Ltd."인 *CustTable*의 첫 번째 행으로 커서를 이동시킵니다.

```
var
    LocateSuccess:Boolean;
    SearchOptions:TLocateOptions;
begin
    SearchOptions := [loPartialKey];
    LocateSuccess := CustTable.Locate('Company', 'Professional Divers,
    Ltd.', SearchOptions);
end;
```

*Locate*가 일치하는 값을 찾으면 이 일치하는 값을 포함하는 첫 번째 레코드가 현재 레코드가 됩니다. *Locate*는 일치하는 레코드를 찾으면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다. 검색이 실패하면 현재 레코드는 변경되지 않습니다.

*Locate*의 장점은 여러 열을 검색하고 검색할 여러 값을 지정하려는 경우에 나타납니다. 검색 값은 가변 타입인데 이는 검색 조건에 각각 다른 데이터 타입을 지정할 수 있다는 것을 의미합니다. 검색 문자열에서 여러 열을 지정하려면 문자열의 개별 항목을 세미콜론으로 구분합니다.

검색 값이 가변 타입이기 때문에 값을 여러 개 전달하는 경우, 인수로 가변 배열 타입을 전달하거나(예: *Lookup* 메소드로부터의 반환값), 아니면 *ArrayOf* 함수를 사용하여 가변 배열을 구성해야 합니다. 다음 코드는 여러 검색 값과 부분 키 일치를 사용하는 여러 열에서의 검색을 보여 줍니다.

```
with CustTable do
    Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver', 'P']), loPartialKey);
```

*Locate*는 가능한 가장 빠른 메소드를 사용하여 일치하는 레코드를 찾습니다. 검색하려는 열이 인덱싱되어 있고 이 인덱스가 사용자가 지정하는 검색 옵션과 호환되는 경우, *Locate*는 이 인덱스를 사용합니다.

Lookup 사용

*Lookup*은 지정된 검색 조건과 일치하는 첫 번째 행을 검색합니다. 일치하는 행을 찾으면 모든 계산된 필드와 데이터셋에 연결된 조희 필드를 다시 계산한 다음 일치하는 행에서 하나 이상의 필드를 반환합니다. *Lookup*은 커서를 일치하는 행으로 이동하지 않고 행의 값만 반환합니다.

가장 간단한 폼에서 검색하는 필드 이름, 일치하는 필드 값 및 반환하는 필드를 *Lookup*에 전달합니다. 예를 들어, 다음 코드는 *Company* 필드의 값이 "Professional Divers, Ltd."인 *CustTable*의 첫 번째 레코드를 찾는 다음, 회사 이름, 담당자 및 전화 번호를 반환합니다.

```
var
    LookupResults:Variant;
begin
    LookupResults := CustTable.Lookup('Company', 'Professional Divers, Ltd.',
        'Company;Contact; Phone');
end;
```

*Lookup*은 첫 번째 찾아낸 레코드에서 지정된 필드에 대한 값을 반환합니다. 값은 가변 타입으로 반환됩니다. 둘 이상의 반환 값이 요청되면 *Lookup*은 가변 배열을 반환합니다. 일치하는 레코드가 없으면 *Lookup*은 Null 가변을 반환합니다. 가변 배열에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

*Lookup*의 장점은 여러 열을 검색하고 검색할 여러 값을 지정하려는 경우에 나타납니다. 여러 열이나 결과 필드를 포함하는 문자열을 지정하려면 문자열의 개별 필드를 세미콜론으로 구분합니다.

검색 값이 가변 타입이기 때문에 값을 여러 개 전달하는 경우, 인수로 가변 배열 타입을 전달하거나(예: *Lookup* 메소드로부터의 반환 값), 아니면 *VarArrayOf* 함수를 사용하여 가변 배열을 구성해야 합니다. 다음 코드는 여러 열에서의 조회 검색을 설명합니다.

```
var
    LookupResults:Variant;
begin
    with CustTable do
        LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver',
            'Christiansted']),
            'Company; Addr1; Addr2; State; Zip');
    end;
```

*Locate*처럼 *Lookup*은 일치하는 레코드를 찾는 가장 빠른 방법을 사용합니다. 검색할 열이 인덱싱되어 있는 경우, *Lookup*은 이 인덱스를 사용합니다.

필터를 사용한 데이터 서브셋 표시 및 편집

애플리케이션은 데이터셋 내의 레코드셋에만 주로 관심이 있습니다. 예를 들어, 고객 데이터베이스에서 캘리포니아에 소재한 회사에 대한 레코드만 검색, 보기 또는 특정 필드 값의 집합을 포함하는 레코드를 찾을 수도 있습니다. 각각의 경우, 필터를 사용하면 데이터셋에 있는 모든 레코드의 서브셋에 대한 애플리케이션의 액세스를 제한할 수 있습니다.

단방향 데이터셋의 경우 데이터셋의 레코드를 제한하는 쿼리를 사용하여 데이터셋의 레코드만 제한할 수 있습니다. 그러나 다른 *TDataSet* 자손의 경우 이미 페치된 데이터의 서브셋을 정의할 수 있습니다. 데이터셋에 있는 모든 레코드의 서브셋에 대한 애플리케이션의 액세스를 제한하기 위해 필터를 사용할 수 있습니다.

필터는 레코드를 표시하기 위해 만족해야 하는 조건을 지정합니다. 필터 조건은 데이터셋의 *Filter* 속성으로 규정되거나 *OnFilterRecord* 이벤트 핸들러로 코딩될 수 있습니다. 필터 조건은 필드가 인덱싱되어 있는지 여부에 관계 없이 데이터셋에 있는 특정 필드 값을 기준으로 합니다. 예를 들어, 캘리포니아에 소재한 회사의 레코드를 보기 위해 간단한 필터는 레코드에 State 필드의 값이 "CA"인 필드를 필터링할 것입니다.

참고 필터는 데이터셋에서 검색된 모든 레코드에 적용됩니다. 많은 양의 데이터를 필터링하려는 경우, 레코드 검색을 제한하는 쿼리를 사용하거나 필터를 사용하기 보다는 인덱싱된 테이블에 범위를 설정하는 것이 더 효과적일 수 있습니다.

필터링 사용 가능 및 사용 불가능

데이터셋의 필터를 사용 가능하도록 하려면 다음 세 단계 과정을 거쳐야 합니다.

- 1 필터를 만듭니다.
- 2 필요한 경우 문자열 기반의 필터 테스트를 위한 필터 옵션을 설정합니다.
- 3 *Filtered* 속성을 *True*로 설정합니다.

필터링이 사용 가능한 경우, 필터 조건을 만족하는 레코드만 애플리케이션에서 사용할 수 있습니다. 필터링은 항상 임시 조건입니다. *Filtered* 속성을 *False*로 설정하여 필터링을 해제할 수 있습니다.

필터 생성

데이터셋에 대한 필터를 만드는 방법은 다음 두 가지입니다.

- *Filter* 속성에서 간단한 필터 조건을 지정합니다. *Filter*는 특히 런타임 시 필터를 만들고 사용하는 데 유용합니다.
- 단순하거나 복잡한 필터 조건을 위해 *OnFilterRecord* 이벤트 핸들러를 작성합니다. *OnFilterRecord*를 사용하여 디자인 타임에 필터 조건을 지정합니다. 필터 로직을 포함하는 단일 문자열로 제한된 *Filter* 속성과는 달리 *OnFilterRecord* 이벤트는 분기 및 순환문 로직을 사용하여 복잡하고 다양한 수준의 필터 조건을 만들 수 있습니다.

Filter 속성을 사용하여 필터를 만들면 애플리케이션에서 필터를 동적으로 생성, 변경 및 적용할 수 있다는 장점이 있습니다(예를 들어, 사용자 입력에 대한 응답으로). 그러나 필터 조건이 단일 텍스트 문자열에 표현될 수 있어야 하고, 분기 및 순환문 구조를 사용할 수 없으며, 데이터셋에 아직 없는 값에 대해 그 값을 비교하거나 테스트할 수 없다는 단점이 있습니다.

OnFilterRecord 이벤트는 필터가 복잡하고 다양할 수 있으며, 분기 및 순환문 구조를 사용하는 여러 줄의 코드에 적용할 수 있으며, 편집 상자의 텍스트와 같은 데이터셋 외부의 값에 대해 데이터셋 값을 테스트할 수 있다는 장점이 있습니다. *OnFilterRecord*를 사용하면 필터를 디자인 타임에 설정하고, 사용자 입력의 응답으로 수정할 수 없다는 단점이 있습니다. 그러나 몇몇 필터 핸들러를 작성하여 일반적인 애플리케이션 조건에 대한 응답 시 핸들러를 전환할 수 있습니다.

다음 단원은 *Filter* 속성과 *OnFilterRecord* 이벤트 핸들러를 사용하여 필터를 만드는 방법을 설명합니다.

Filter 속성 설정

Filter 속성을 사용하여 필터를 작성하려면 필터의 테스트 조건을 포함하는 문자열로 속성 값을 설정합니다. 예를 들어, 다음 문장은 캘리포니아 주에 대한 값이 포함되는지 알기 위해서 데이터셋의 *State* 필드를 테스트하는 필터를 작성합니다.

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

사용자가 제공한 텍스트를 기반으로 *Filter*의 값을 제공할 수도 있습니다. 예를 들어, 다음 문장은 편집 상자의 텍스트를 *Filter*에 할당합니다.

```
Dataset1.Filter := Edit1.Text;
```

물론 하드 코딩된 텍스트 및 사용자 제공 데이터를 기반으로 문자열을 작성할 수도 있습니다.

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

빈 필드 값은 명시적으로 필터에 포함되지 않으면 나타나지 않습니다.

```
Dataset1.Filter := 'State <> ''CA'' or State = BLANK';
```

참고 데이터셋에 필터를 적용하기 위해 *Filter*에 대한 값을 지정한 다음에는 *Filtered* 속성을 *True*로 설정합니다.

필터는 필드 값을 리터럴과 비교하고 다음과 같은 비교 및 논리 연산자를 사용하는 상수와 비교할 수 있습니다.

표 18.4 필터에 나타날 수 있는 비교 및 논리 연산자

연산자	의미
<	작다
>	크다
>=	같거나 크다
<=	같거나 작다
=	같다
<>	같지 않다
AND	두 문장이 모두 <i>True</i> 인지 테스트
NOT	뒤에 있는 문장이 <i>True</i> 가 아닌지 테스트
OR	두 문장 중 적어도 하나가 <i>True</i> 인지 테스트
+	숫자 더하기, 문자열 연결, 날짜/시간 값에 숫자 추가(일부 드라이버에만 사용 가능)
-	숫자 빼기, 날짜 빼기 또는 날짜에서 숫자 빼기(일부 드라이버에만 사용 가능)
*	두 수를 곱하기(일부 드라이버에만 사용 가능)
/	두 수를 나누기(일부 드라이버에만 사용 가능)
*	부분 비교에 사용하는 와일드카드(<i>FilterOptions</i> 에 <i>foPartialCompare</i> 가 포함되어야 함)

이들 연산자를 조합하여 매우 복잡한 필터를 만들 수 있습니다. 예를 들어, 다음 문장은 레코드를 표시하기 전에 두 테스트 조건을 만족하는지 확인합니다.

```
(Custno > 1400) AND (Custno < 1500);
```


참고 필터링을 사용하는 경우, 사용자가 레코드를 편집하면 더 이상 레코드가 필터의 테스트 조건에 맞지 않을 수도 있습니다. 따라서 데이터셋에서 레코드를 검색한 다음에는 레코드가 "사라질" 수도 있습니다. 레코드가 사라지면 필터 조건을 만족하는 다음 레코드가 현재 레코드가 됩니다.

OnFilterRecord 이벤트 핸들러 작성

검색한 각 레코드에 대해 데이터셋에 의해 생성되는 *OnFilterRecord* 이벤트를 사용하여 레코드를 필터링하는 코드를 작성할 수 있습니다. 이 이벤트 핸들러는 레코드를 테스트하여 애플리케이션에 레코드를 표시할지 여부를 결정합니다.

레코드가 필터 조건을 만족하는지 나타내려면 *OnFilterRecord* 핸들러에서 *Accept* 매개변수를 *True*로 설정하여 레코드를 포함하거나 *False*로 설정하여 제외시켜야 합니다. 예를 들어, 다음 필터는 State 필드가 "CA"로 설정된 레코드만 표시합니다.

```
procedure TForm1.Table1FilterRecord(DataSet:TDataSet; var Accept:Boolean);
begin
    Accept := DataSet['State'].AsString = 'CA';
end;
```

필터링이 사용 가능하면 *OnFilterRecord* 이벤트가 검색된 각 레코드에 대해 생성됩니다. 이벤트 핸들러는 각각의 레코드를 테스트하고 필터 조건을 만족하는 레코드만 표시합니다. *OnFilterRecord* 이벤트는 데이터셋의 모든 레코드에 대해 생성되기 때문에 애플리케이션의 성능에 나쁜 영향을 미치지 않도록 가능한 한 이벤트 핸들러를 빈틈없이 코딩해야 합니다.

런타임 시 필터 이벤트 핸들러 전환

여러 개의 *OnFilterRecord* 이벤트 핸들러를 코딩하여 런타임 시 이벤트 핸들러 간에 전환할 수 있습니다. 예를 들어, 다음 문장은 *NewYorkFilter*라는 *OnFilterRecord* 이벤트 핸들러로 전환합니다.

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

필터 옵션 설정

FilterOptions 속성을 사용하면 문자열 기반의 필드를 비교하는 필터가 레코드를 부분 비교하는지, 대소문자를 구별하는 비교인지 지정할 수 있습니다. *FilterOptions*는 빈 설정(기본값)이거나 다음 값을 포함할 수 있는 설정 속성입니다.

표 18.5 FilterOptions 값

값	의미
<i>foCaseInsensitive</i>	문자열 비교 시 대소문자를 무시합니다.
<i>foNoPartialCompare</i>	부분 문자열 일치를 사용할 수 없게 합니다. 즉, 별표(*)로 끝나는 문자열을 일치시키지 않게 합니다.

예를 들어, 다음 문장은 *State* 필드의 값을 비교할 때 대소문자를 무시하는 필터를 설정합니다.

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```

필터링된 데이터셋의 레코드 탐색

필터링된 데이터셋의 레코드를 탐색하는 데이터셋 메소드에는 네 가지가 있습니다. 다음 표는 이러한 메소드를 나열하고 메소드의 기능을 설명합니다.

표 18.6 필터링된 데이터셋의 탐색 메소드

메소드	용도
<i>FindFirst</i>	현재 필터 조건과 일치하는 첫 번째 레코드로 이동합니다. 첫 번째로 일치하는 레코드의 검색은 항상 필터링되지 않은 데이터셋의 첫 번째 레코드에서 시작합니다.
<i>FindLast</i>	현재 필터 조건과 일치하는 마지막 레코드로 이동합니다.
<i>FindNext</i>	필터링된 데이터셋의 현재 레코드에서 다음 레코드로 이동합니다.
<i>FindPrior</i>	필터링된 데이터셋의 현재 레코드에서 이전 레코드로 이동합니다.

예를 들어, 다음 문장은 데이터셋의 첫 번째 필터링된 레코드를 찾습니다.

```
DataSet1.FindFirst;
```

애플리케이션에 대해 *Filter* 속성을 설정하거나 *OnFilterRecord* 이벤트 핸들러를 생성한다고 가정할 때 이들 메소드는 현재 필터링이 사용 가능한지에 상관 없이 지정된 레코드에 커서를 둡니다. 필터링이 사용 가능하지 않을 때 이들 메소드를 호출하면 다음과 같이 됩니다.

- 일시적으로 필터링을 사용 가능하게 합니다.
- 일치하는 레코드를 찾으면 이 레코드에 커서를 둡니다.
- 필터링을 사용 불가능하게 합니다.

참고 필터링이 사용 불가능하고 *Filter* 속성을 설정하지 않거나 *OnFilterRecord* 이벤트 핸들러를 생성하지 않는 경우, 이 메소드는 *First*, *Last*, *Next* 및 *Prior*과 동일한 작업을 수행합니다.

모든 탐색 필터 메소드는 커서를 일치하는 레코드(있을 경우)에 두고 이 레코드를 현재 레코드로 만든 다음 *True*를 반환합니다. 일치하는 레코드를 찾지 못하면 커서 위치는 변하지 않고 이 메소드는 *False*를 반환합니다. 이러한 호출을 랩(wrap)하기 위해 *Found* 속성의 상태를 확인하여 *Found*가 *True*일 때만 동작을 수행합니다. 예를 들어, 커서가 이미 데이터셋의 일치하는 마지막 레코드에 있고 *FindNext*를 호출하는 경우, 메소드는 *False*를 반환하고 현재 레코드는 변경되지 않습니다.

데이터 수정

읽기 전용 *CanModify* 속성이 *True*이면 아래 나열된 데이터셋 메소드를 사용하여 데이터를 삽입, 업데이트 및 삭제할 수 있습니다. 데이터셋이 단방향이 아니거나, 데이터셋을 원본으로 하는 데이터베이스가 읽기 및 쓰기 권한을 허용하지 않거나, 일부 다른 요소가 없다면 *CanModify*는 *True*입니다. 개입하는 요소에는 일부 데이터셋의 *ReadOnly* 속성이나 *TQuery* 컴포넌트의 *RequestLive* 속성이 있습니다.

표 18.7 데이터 삽입, 업데이트 및 삭제를 위한 데이터셋 메소드

메소드	설명
<i>Edit</i>	데이터셋이 아직 <i>dsEdit</i> 또는 <i>dsInsert</i> 상태가 아닌 경우, 데이터셋을 <i>dsEdit</i> 상태로 둡니다.
<i>Append</i>	보류 중인 데이터를 포스트하고, 현재 레코드를 데이터셋의 끝 부분으로 이동하며, 데이터셋을 <i>dsInsert</i> 상태로 둡니다.
<i>Insert</i>	모든 보류 중인 데이터를 포스트하고 데이터셋을 <i>dsInsert</i> 상태로 둡니다.
<i>Post</i>	새로운 레코드나 변경된 레코드를 데이터베이스에 포스트하려고 시도합니다. 성공하면 데이터셋은 <i>dsBrowse</i> 상태가 되고, 실패하면 현재 상태로 남습니다.
<i>Cancel</i>	현재 작업을 취소하고 데이터셋을 <i>dsBrowse</i> 상태로 둡니다.
<i>Delete</i>	현재 레코드를 삭제하고 데이터셋을 <i>dsBrowse</i> 상태로 둡니다.

레코드 편집

데이터셋은 애플리케이션이 레코드를 수정하기 전에 *dsEdit* 모드여야 합니다. 데이터셋의 읽기 전용 *CanModify* 속성이 *True*인 경우, 코드에서 *Edit* 메소드를 사용하여 데이터셋을 *dsEdit* 모드로 둘 수 있습니다.

데이터셋이 *dsEdit* 모드로 전환되면 먼저 *BeforeEdit* 이벤트를 받습니다. 편집 모드로의 전환이 성공적으로 완료되면 데이터셋은 *AfterEdit* 이벤트를 받습니다. 일반적으로 이러한 이벤트는 데이터셋의 현재 상태를 나타내도록 사용자 인터페이스를 업데이트하는 데 사용됩니다. 몇 가지 이유로 데이터셋이 편집 모드가 될 수 없으면 *OnEditError* 이벤트가 발생하는데 여기서 사용자에게 문제를 알려 주거나 데이터셋이 편집 모드로 들어가지 못하는 상황을 수정할 수 있습니다.

애플리케이션의 폼에서 다음의 경우에 일부 data-aware 컨트롤은 데이터셋을 자동으로 *dsEdit* 상태로 둘 수 있습니다.

- 컨트롤의 *ReadOnly* 속성은 기본값인 *False*입니다.
- 컨트롤에 대한 데이터 소스의 *AutoEdit* 속성은 *True*입니다.
- *CanModify*는 데이터셋에 대해 *True*입니다.

참고 데이터셋이 *dsEdit* 상태인 경우라도 애플리케이션 사용자가 적절한 SQL 액세스 권한이 없으면 SQL 기반 데이터베이스에 대한 레코드 편집을 하지 못할 수도 있습니다.

일단 데이터셋이 *dsEdit* 모드이면 사용자는 폼의 data-aware 컨트롤에 나타나는 현재 레코드에 대한 필드 값을 수정할 수 있습니다. 사용자가 현재 레코드를 변경하는 작업(예를 들어, 그리드의 다른 레코드로 이동)을 수행할 때 편집이 사용 가능한 data-aware 컨트롤은 자동으로 *Post*를 호출합니다.

폼에 탐색기 컴포넌트가 있으면 사용자는 탐색기의 Cancel 버튼을 클릭하여 편집을 취소할 수 있습니다. 편집을 취소하면 데이터셋을 *dsBrowse* 상태로 돌려 놓습니다.

코드에서 적절한 메소드를 호출하여 편집 내용을 작성하거나 취소해야 합니다. 변경 내용은 *Post*를 호출하여 작성합니다. *Cancel*을 호출하여 변경 내용을 취소합니다. 코드에서 *Edit*와 *Post*는 주로 함께 사용됩니다. 예를 들면, 다음과 같습니다.

```
with CustTable do
begin
    Edit;
    FieldValues['CustNo'] := 1234;
    Post;
end;
```

앞의 예제에서 코드의 첫 번째 줄은 데이터셋을 *dsEdit* 모드로 둡니다. 코드의 다음 줄은 숫자 1234를 현재 레코드의 *CustNo* 필드에 할당합니다. 끝으로 마지막 줄은 수정된 레코드를 쓰거나 포스트합니다. 업데이트를 캐시하지 못하면 포스트는 데이터베이스에 변경 내용을 다시 씁니다. 업데이트를 캐시하면 변경 내용이 임시 버퍼에 쓰여지고 데이터셋의 *ApplyUpdates* 메소드가 호출될 때까지 임시 버퍼에 보관됩니다.

새 레코드 추가

애플리케이션이 새 레코드를 추가할 수 있으려면 데이터셋은 *dsInsert* 모드여야 합니다. 코드에서 데이터셋의 읽기 전용 *CanModify* 속성이 *True*인 경우, *Insert* 또는 *Append* 메소드를 사용하여 데이터셋을 *dsInsert* 모드로 둘 수 있습니다.

데이터셋이 *dsInsert* 모드로 전환되면 우선 *BeforeInsert* 이벤트를 받습니다. 삽입 모드로의 전환이 성공적으로 완료되면 데이터셋은 우선 *OnNewRecord* 이벤트를 받은 다음 *AfterInsert* 이벤트를 받습니다. 예를 들면, 이 이벤트를 사용하여 새로 삽입된 레코드에 초기 값을 제공합니다.

```
procedure TForm1.OrdersTableNewRecord(DataSet:TDataSet);
begin
    DataSet.FieldByName('OrderDate').AsDateTime := Date;
end;
```

애플리케이션의 폼에서 data-aware 그리드와 탐색기 컨트롤은 데이터셋을 *dsInsert* 상태로 둘 수 있습니다.

- 컨트롤의 *ReadOnly* 속성은 기본값인 *False*입니다.
- *CanModify*는 데이터셋에 대해 *True*입니다.

참고 데이터셋이 *dsInsert* 상태일지라도 애플리케이션의 사용자에게 적절한 SQL 액세스 권한이 없으면 SQL 기반 데이터베이스에 데이터를 추가하지 못할 수 있습니다.

일단 데이터셋이 *dsInsert* 모드이면 사용자나 애플리케이션은 새 레코드에 연결된 필드에 값을 입력할 수 있습니다. 그리드와 탐색 컨트롤을 제외하면 사용자에게 *Insert*와 *Append* 사이에 가시적인 차이는 없습니다. *Insert*를 호출하면 현재 레코드의 위 그리드에 빈 행이 새로 나타납니다. *Append*를 호출하면 그리드는 데이터셋의 마지막 레코드로 스크롤되고, 그리드의 아래에 빈 행이 나타나고, 데이터셋에 연결된 모든 탐색기 컴포넌트에서 *Next* 및 *Last* 버튼이 흐리게 표시됩니다.

삽입할 수 있는 data-aware 컨트롤은 사용자가 현재 레코드를 변경하는 작업(예를 들어, 그리드의 다른 레코드로의 이동)을 수행할 때 자동으로 *Post*를 호출합니다. 그렇지 않으면 코드에서 *Post*를 호출해야 합니다.

*Post*는 새 레코드를 데이터베이스에 쓰거나, 업데이트를 캐시하는 경우에는 인메모리 캐시에 레코드를 씁니다. 캐시된 삽입 및 추가를 데이터베이스에 쓰려면 데이터셋의 *ApplyUpdates* 메소드를 호출합니다.

레코드 삽입

*Insert*는 현재 레코드 앞에 새로운 빈 레코드를 열고 빈 레코드를 현재 레코드로 만들어서 사용자나 애플리케이션 코드가 레코드에 대한 필드 값을 입력할 수 있도록 합니다.

애플리케이션이 *Post*를 호출하거나 캐시된 업데이트를 사용하는 경우, *ApplyUpdates*를 호출하면 다음 세 가지 방법 중 하나를 이용해서 데이터베이스에 새로 삽입된 레코드를 씁니다.

- 인덱스된 Paradox 및 dBASE 테이블의 경우, 레코드는 해당 인덱스에 따라 데이터셋의 특정 위치에 삽입됩니다.
- 인덱싱되지 않은 Paradox 및 dBASE 테이블의 경우, 레코드는 데이터셋의 현재 위치에 삽입됩니다.
- SQL 데이터베이스에서는 실제 삽입 위치가 구현에 따라 다릅니다. 테이블이 인덱싱된 경우, 인덱스는 새 레코드 정보를 가지고 업데이트됩니다.

레코드 추가

*Append*는 데이터셋 끝에 새로운 빈 레코드를 열고 빈 레코드를 현재 레코드로 만들어서 사용자나 애플리케이션 코드가 레코드에 대한 필드 값을 입력할 수 있도록 합니다.

애플리케이션이 *Post*를 호출하거나 캐시된 업데이트를 사용하는 경우, *ApplyUpdates*를 호출하면 다음 세 가지 방법 중 하나를 이용해서 데이터베이스에 새로 추가된 레코드를 씁니다.

- 인덱스된 Paradox 및 dBASE 테이블의 경우, 레코드는 해당 인덱스에 따라 데이터셋의 특정 위치에 삽입됩니다.
- 인덱싱되지 않은 Paradox 및 dBASE 테이블의 경우, 레코드는 데이터셋의 끝에 추가됩니다.
- SQL 데이터베이스에서는 실제 추가 위치가 구현에 따라 다릅니다. 테이블이 인덱싱된 경우, 인덱스는 새로운 레코드 정보를 가지고 업데이트됩니다.

레코드 삭제

Delete 메소드를 사용하여 활성 데이터셋의 현재 레코드를 삭제합니다. *Delete* 메소드를 호출하면 다음이 발생합니다.

- 데이터셋은 *BeforeDelete* 이벤트를 받습니다.
- 데이터셋은 현재 레코드를 삭제하려고 합니다.

- 데이터셋은 *dsBrowse* 상태로 반환됩니다.
- 데이터셋은 *AfterDelete* 이벤트를 받습니다.

BeforeDelete 이벤트 핸들러에서 삭제를 방지하기 위해 전역 *Abort* 프로시저를 호출할 수 있습니다.

```
procedure TForm1.TableBeforeDelete (Dataset:TDataset)begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```

*Delete*에 실패하면 *OnDeleteError* 이벤트가 생성됩니다. *OnDeleteError* 이벤트 핸들러로 문제를 수정할 수 없는 경우, 데이터셋은 *dsEdit* 상태로 남습니다. *Delete*에 성공하면 데이터셋은 *dsBrowse* 상태로 되돌아가고 삭제된 레코드 다음에 오는 레코드가 현재 레코드가 됩니다.

업데이트를 캐싱하는 경우, 삭제된 레코드는 *ApplyUpdates*를 호출할 때까지 원본으로 사용하는 데이터베이스 테이블에서 제거되지 않습니다.

폼에 탐색기 컨트롤트를 두면 사용자가 탐색기의 Delete 버튼을 클릭하여 현재 레코드를 삭제할 수 있습니다. 코드에서 현재 레코드를 제거하려면 *Delete*를 명시적으로 호출해야 합니다.

데이터 포스트

레코드 편집을 완료한 후에 *Post* 메소드를 호출하여 변경 내용을 작성해야 합니다. *Post* 메소드는 업데이트 캐싱 여부와 데이터셋의 상태에 따라 다르게 작동합니다.

- 업데이트를 캐싱하지 않는 경우, 데이터셋이 *dsEdit* 또는 *dsInsert* 상태에 있으면 *Post*는 현재 레코드를 데이터베이스로 쓰고 데이터셋을 *dsBrowse* 상태로 반환합니다.
- 업데이트를 캐싱하는 경우, 데이터셋이 *dsEdit*나 *dsInsert* 상태에 있으면 *Post*는 현재 레코드를 내부 캐시로 쓰고 데이터셋을 *dsBrowse* 상태로 반환합니다. 편집 내용은 *ApplyUpdates*를 호출해야 데이터베이스에 씁니다.
- 데이터셋이 *dsSetKey* 상태에 있으면 *Post*는 데이터셋을 *dsBrowse* 상태로 반환합니다.

데이터셋의 초기 상태에 관계 없이 *Post*는 현재의 변경 내용을 쓰기 전후에 *BeforePost* 및 *AfterPost* 이벤트를 생성합니다. 이러한 이벤트를 사용하면 사용자 인터페이스를 업데이트하거나 또는 *Abort* 프로시저를 호출하여 데이터셋이 변경 내용을 포스트할 수 없게 합니다. *Post* 호출에 실패하면 데이터셋은 *OnPostError* 이벤트를 받아 사용자에게 문제를 알려 주거나 문제를 수정하도록 할 수 있습니다.

포스트는 다른 프로시저의 일부로 명시적으로 또는 암시적으로 수행될 수 있습니다. 애플리케이션이 현재 레코드에서 떠날 때는 *Post*가 암시적으로 호출됩니다. 테이블이 *dsEdit* 또는 *dsInsert* 모드인 경우, *First*, *Next*, *Prior* 및 *Last* 메소드를 호출하면 *Post*가 수행됩니다. 또한 *Append* 및 *Insert* 메소드는 모든 보류 중인 데이터를 암시적으로 포스트합니다.

경고 *Close* 메소드는 *Post*를 암시적으로 호출하지 않습니다. *BeforeClose* 이벤트를 사용하여 모든 보류 중인 편집 내용을 명시적으로 포스트합니다.

변경 취소

아직 *Post*를 직접 또는 간접적으로 호출하지 않은 경우, 애플리케이션은 현재 레코드의 변경 내용을 언제라도 취소할 수 있습니다. 예를 들어, 데이터셋이 *dsEdit* 모드이고 사용자가 하나 이상의 필드에서 데이터를 변경한 경우, 애플리케이션은 데이터셋에 대한 *Cancel* 메소드를 호출하여 레코드를 원래 값으로 다시 돌려 놓을 수 있습니다. *Cancel*을 호출하면 항상 데이터셋을 *dsBrowse* 상태로 돌려 놓습니다.

애플리케이션에서 *Cancel*을 호출했을 때 데이터셋이 *dsEdit*나 *dsInsert* 모드에 있으면 애플리케이션은 현재 레코드를 원래 값으로 복원하기 전에 *BeforeCancel* 및 *AfterCancel* 이벤트를 받습니다.

폼에서 데이터셋에 연결된 탐색기 컴포넌트에 *Cancel* 버튼을 포함시켜 사용자가 편집, 삽입 또는 추가 작업을 취소할 수 있게 하거나 폼의 *Cancel* 버튼에 대한 코드를 직접 작성할 수 있습니다.

전체 레코드 수정

폼에서 그리드와 탐색기를 제외한 모든 data-aware 컨트롤은 레코드의 단일 필드에 대한 액세스를 제공합니다.

그러나 코드에서 데이터셋을 원본으로 하는 데이터베이스 테이블의 구조가 안정적이고 변경되지 않는다면 전체 레코드 구조에 적용되는 메소드를 사용할 수 있습니다. 다음 표는 이들 레코드의 개별 필드가 아닌 전체 레코드에 사용할 수 있는 메소드를 요약해서 보여 줍니다.

표 18.8 전체 레코드에 적용되는 메소드

메소드	설명
<i>AppendRecord</i> ([array of values])	테이블의 끝에 지정된 열 값의 레코드를 추가합니다. <i>Append</i> 와 유사합니다. 암시적 <i>Post</i> 를 수행합니다.
<i>InsertRecord</i> ([array of values])	레코드로 지정된 값을 테이블의 현재 커서 위치 앞에 삽입합니다. <i>Insert</i> 와 유사합니다. 암시적 <i>Post</i> 를 수행합니다.
<i>SetFields</i> ([array of values])	해당 필드의 값을 설정합니다. <i>TField</i> 에 값을 할당하는 것과 유사합니다. 애플리케이션은 명시적 <i>Post</i> 를 수행해야 합니다.

이러한 메소드 각각은 인수로 값의 배열을 취합니다. 각각의 배열 값은 원본으로 사용하는 데이터셋의 열에 해당합니다. 이 값은 리터럴, 변수 또는 Null일 수 있습니다. 인수 값의 수가 데이터셋 열의 수보다 작으면 나머지 값들은 Null로 처리됩니다.

인덱싱되지 않은 데이터셋에서 *AppendRecord*는 데이터셋 끝에 레코드를 추가하고, *InsertRecord*는 현재 커서 위치 다음에 레코드를 삽입합니다. 인덱싱된 테이블의 경우, 두 메소드는 모두 인덱스를 기반으로 테이블의 올바른 위치에 레코드를 둡니다. 두 경우 모두 메소드는 커서를 레코드의 위치로 이동시킵니다.

*SetFields*는 매개변수 배열에 지정된 값을 데이터셋의 필드에 할당합니다. *SetFields*를 사용하려면 애플리케이션이 먼저 *Edit*를 호출하여 데이터셋을 *dsEdit* 모드로 두어야 합니다. 현재 레코드에 대한 변경 내용을 적용하려면 *Post*를 수행해야 합니다.

*SetFields*를 사용하여 기존 레코드에서 모든 필드가 아닌 일부 필드를 수정하는 경우, 변경을 원하지 않는 필드에 대해 Null 값을 전달할 수 있습니다. 레코드의 모든 필드에 대한 충분한 값을 제공하지 않으면 *SetFields*는 필드에 Null 값을 할당합니다. Null 값은 이미 필드에 있는 모든 기존 값을 덮어씁니다.

예를 들어, 데이터베이스에 Name, Capital, Continent, Area 및 Population 열을 갖는 COUNTRY 테이블이 있다고 가정합니다. *CountryTable*이라는 *TTable* 컴포넌트가 COUNTRY 테이블에 연결되어 있으면 다음 문장은 레코드를 COUNTRY 테이블에 삽입합니다.

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

이 문장은 Area 및 Population에 대한 값을 지정하지 않으므로 이들 열에 대해서는 Null 값이 삽입됩니다. 테이블이 Name을 기반으로 인덱싱되어 있으므로 위의 문장은 "Japan"의 알파벳 순서에 따라 레코드를 삽입합니다.

레코드를 업데이트하기 위해서 애플리케이션은 다음의 코드를 사용할 수 있습니다.

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then:
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

이 코드는 Area 및 Population 필드에 값을 할당하고 난 다음 데이터베이스에 포스트합니다. 세 개의 NULL 포인터는 처음 세 개의 열에 대한 위치 표시자 역할을 하여 현재의 내용을 유지합니다.

필드 계산

Fields 편집기를 사용하여 데이터셋의 계산된 필드를 정의할 수 있습니다. 데이터셋에 계산된 필드가 포함되어 있으면 *OnCalcFields* 이벤트 핸들러에서 그러한 필드의 값을 계산하는 코드를 작성합니다. Fields 편집기를 사용하여 계산된 필드를 정의하는 방법에 대한 자세한 내용은 19-7 페이지의 "계산된 필드 정의"를 참조하십시오.

AutoCalcFields 속성은 *OnCalcFields*를 호출할 시기를 결정합니다. *AutoCalcFields*가 *True*이면 다음의 경우에 *OnCalcFields*가 호출됩니다.

- 데이터셋이 열릴 때
- 데이터셋이 편집 모드가 될 때
- 데이터베이스에서 레코드를 검색할 때
- 포커스가 한 비주얼 컴포넌트에서 다른 비주얼 컴포넌트로 이동하거나 data-aware 그리드 컨트롤의 한 열에서 다른 열로 이동하고 현재 레코드가 수정되었을 때

*AutoCalcFields*가 *False*이면 레코드 내의 개별 필드를 편집할 때(위의 네 번째 조건) *OnCalcFields*를 호출하지 않습니다.

주의 *OnCalcFields*는 자주 호출되므로 이에 대해 작성된 코드는 짧아야 합니다. 또한 *AutoCalcFields*가 *True*이면 *OnCalcFields*는 데이터셋 (또는 마스터/디테일 관계의 일부인 경우 연결된 데이터셋)을 수정하는 어떤 작업도 수행하지 않아야 합니다. 왜냐하면 이 경우는 재귀 호출을 초래하기 때문입니다. 예를 들어, *OnCalcFields*가 *Post*를 수행하고 *AutoCalcFields*가 *True*인 경우, *OnCalcFields*가 재호출되어 *Post*를 재수행하게 됩니다.

*OnCalcFields*를 실행하면 데이터셋은 *dsCalcFields* 모드가 됩니다. 이 상태는 핸들러에서 수정하도록 디자인된 계산된 필드를 제외한 레코드를 수정 또는 추가하지 못하게 합니다. 다른 수정 작업을 못 하게 막는 이유는 *OnCalcFields*는 계산된 필드 값을 파생할 때 다른 필드의 값을 사용하기 때문입니다. 다른 필드에 변경이 있으면 계산된 필드에 할당된 값을 무효화합니다. *OnCalcFields*가 완료된 후 데이터셋은 *dsBrowse* 상태로 돌아옵니다.

데이터셋 형식

18-2 페이지의 "TDataSet 자손 사용"에서는 자손의 데이터를 액세스하는 데 사용하는 방법에 따라 *TDataSet* 자손을 분류합니다. *TDataSet* 자손을 분류하는 또 다른 유용한 방법은 자손이 나타내는 서버 데이터의 타입을 고려하는 것입니다. 이러한 방식으로 보면 데이터셋에는 세 가지 기본 클래스가 있습니다.

- **테이블 타입 데이터셋.** 테이블 타입 데이터셋은 테이블의 모든 행과 열을 포함하여 데이터베이스 서버의 단일 테이블을 나타냅니다. 테이블 타입 데이터셋에는 *TTable*, *TADOTable*, *TSQLTable* 및 *TIBTable*이 있습니다.

테이블 타입 데이터셋은 서버에 정의된 인덱스를 사용할 수 있게 합니다. 왜냐하면 데이터베이스 테이블과 데이터셋은 일대일로 대응하므로 데이터베이스 테이블에 정의된 서버 인덱스를 사용할 수 있습니다. 인덱스를 사용하여 애플리케이션은 테이블의 레코드를 정렬하고, 검색 및 조회 속도를 향상시키며, 마스터/디테일 관계의 기초를 형성할 수 있습니다. 또한 일부 테이블 타입 데이터셋은 데이터셋과 데이터베이스 테이블 사이의 일대일 관계를 이용하여 데이터베이스 테이블 작성 및 삭제와 같은 테이블 수준의 작업을 수행할 수 있습니다.

- **쿼리 타입 데이터셋.** 쿼리 타입 데이터셋은 단일 SQL 명령 또는 쿼리를 나타냅니다. 쿼리는 명령 (일반적으로 SELECT 문)을 실행하여 결과 집합을 표시하거나 또는 레코드를 반환하지 않는 명령 (예를 들어, UPDATE 문)을 실행할 수 있습니다. 쿼리 타입 데이터셋에는 *TQuery*, *TADOQuery*, *TSQLQuery* 및 *TIBQuery*가 있습니다.

쿼리 타입 데이터셋을 효과적으로 사용하려면 SQL-92 표준 제한 및 확장을 포함하여 SQL 및 서버의 SQL 구현에 익숙해야 합니다. SQL을 처음 사용하는 경우에는 SQL에 관해 자세하게 다루어 놓은 타사의 책을 구입할 수도 있습니다. 추천할 만한 책으로는 Morgan Kaufmann Publishers에서 발행된 Jim Melton, Alan R. Simpson 공저 *Understanding the New SQL: A Complete Guide*가 있습니다.

- **내장 프로시저 타입 데이터셋.** 내장 프로시저 타입 데이터셋은 데이터베이스 서버에 내장 프로시저를 표시합니다. 내장 프로시저 타입 데이터셋에는 *TStoredProc*, *TADOStoredProc*, *TSQLStoredProc* 및 *TIBStoredProc*가 있습니다.

내장 프로시저는 사용되는 데이터베이스 시스템에 특징적인 프로시저 및 트리거 랭귀지로 작성된 독립적인 프로그램입니다. 일반적으로 내장 프로시저는 자주 반복되는 데이터베이스 관련 작업을 처리하고, 특히 많은 레코드에서 작업을 하거나 집계 또는 수학적 함수를 사용하는 작업에 유용합니다. 일반적으로 내장 프로시저를 사용하면 다음과 같은 이유에서 데이터베이스 애플리케이션의 성능이 향상됩니다.

- 서버의 높은 처리 능력과 속도를 이용합니다.
- 서버에서 처리하도록 하여 네트워크 트래픽을 줄입니다.

내장 프로시저는 데이터를 반환할 수도 있고 반환하지 않을 수도 있습니다. 데이터를 반환하는 내장 프로시저는 커서(SELECT 쿼리의 결과와 유사), 다중 커서(여러 데이터셋을 효과적으로 반환)로 반환하거나 출력 매개변수에 데이터를 반환합니다. 이러한 차이는 부분적으로 서버에 따라 다르게 나타납니다. 일부 서버는 내장 프로시저의 데이터 반환을 허용하지 않거나 출력 매개변수만 허용합니다. 내장 프로시저를 아예 지원하지 않는 서버도 있습니다. 서버 설명서를 참조하여 사용 가능한 작업을 결정하십시오.

참고 대부분의 서버는 내장 프로시저를 사용할 때 SQL에 확장을 제공하기 때문에 일반적으로 쿼리 타입 데이터셋을 사용하여 내장 프로시저를 실행할 수 있습니다. 하지만 각 서버는 자체의 고유한 구문을 사용합니다. 내장 프로시저 타입 데이터셋 대신 쿼리 타입 데이터셋을 사용하려고 선택한 경우 필요한 구문을 보려면 서버 설명서를 참조하십시오.

*TDataSet*에는 이 세 가지 범주에 알맞는 데이터셋 외에도 둘 이상의 범주에 적합한 일부 자손이 있습니다.

- *TADODataSet* 및 *TSQLDataSet*에는 그 타입이 테이블인지, 쿼리인지, 내장 프로시저인지 지정할 수 있는 *CommandType* 속성이 있습니다. *TADODataSet*으로 테이블 타입 데이터셋과 같은 인덱스를 지정할 수 있지만 속성 및 메소드 이름은 쿼리 타입 데이터셋과 가장 유사합니다.
- *TClientDataSet*은 다른 데이터셋의 데이터를 나타냅니다. 마찬가지로 테이블, 쿼리 또는 내장 프로시저를 나타낼 수 있습니다. *TClientDataSet*은 인덱스를 지원하지 때문에 테이블 타입 데이터셋과 가장 유사하게 작동합니다. 그러나 쿼리 및 내장 프로시저의 기능도 일부 가지고 있는데 매개변수의 관리 기능과 결과 집합을 검색하지 않고 실행하는 기능이 여기에 포함됩니다.
- 일부 다른 클라이언트 데이터셋 (*TBDEClientDataSet* 및 *TSQLClientDataSet*)에는 그 타입이 테이블인지, 쿼리인지, 내장 프로시저인지 지정할 수 있는 *CommandType* 속성이 있습니다. 속성 및 메소드 이름은 매개변수 지원, 인덱스 및 결과 집합을 검색하지 않고 실행하는 기능을 포함하여 *TClientDataSet*과 비슷합니다.
- *TIBDataSet*은 쿼리 및 내장 프로시저를 나타낼 수 있습니다. 실제로 다중 쿼리 및 내장 프로시저를 각각에 대한 개별 속성으로 동시에 나타낼 수 있습니다.

테이블 타입 데이터셋 사용

다음과 같이 테이블 타입 데이터셋을 사용합니다.

- 1 적절한 데이터셋 컴포넌트를 데이터 모듈이나 폼에 두고 그 컴포넌트의 *Name* 속성을 사용자의 애플리케이션에 적합한 고유한 값으로 설정합니다.
- 2 사용하려는 테이블이 들어 있는 데이터베이스 서버를 식별합니다. 테이블 타입 데이터셋은 각기 다르게 수행하지만 일반적으로 데이터베이스 연결 컴포넌트를 지정합니다.
 - *TTable*의 경우 *TDatabase* 컴포넌트나 *DatabaseName* 속성을 사용하는 BDE 알리아스를 지정합니다.
 - *TADOTable*의 경우 *Connection* 속성을 사용하여 *TADODConnection* 컴포넌트를 지정합니다.
 - *TSQLTable*의 경우 *SQLConnection* 속성을 사용하여 *TSQLConnection* 컴포넌트를 지정합니다.
 - *TIBTable*의 경우 *Database* 속성을 사용하여 *TIBConnection* 컴포넌트를 지정합니다.

데이터베이스 연결 컴포넌트 사용에 대한 내용은 17장 "데이터베이스에 연결"을 참조하십시오.
- 3 *TableName* 속성을 데이터베이스에 있는 테이블의 이름으로 설정합니다. 데이터베이스 연결 컴포넌트를 이미 식별한 경우에는 드롭다운 목록에서 테이블을 선택할 수 있습니다.
- 4 데이터 소스 컴포넌트를 데이터 모듈이나 폼에 두고 *DataSet* 속성을 데이터셋의 이름으로 설정합니다. 데이터 소스 컴포넌트를 사용하여 결과 집합을 데이터셋에서 data-aware 컴포넌트로 전달하고 표시합니다.

테이블 타입 데이터셋 사용의 장점

테이블 타입 데이터셋을 사용하는 주요 장점은 인덱스를 사용할 수 있다는 점입니다. 인덱스를 사용하면 애플리케이션은 다음 작업을 수행할 수 있습니다.

- 데이터셋에서 레코드를 정렬합니다.
- 레코드를 빠르게 찾습니다.
- 표시되는 레코드를 제한합니다.
- 마스터/디테일 관계를 설정합니다.

또한 많은 데이터셋의 다음과 같은 경우에 테이블 타입 데이터셋과 데이터베이스 테이블 사이의 일대일 관계를 사용할 수 있습니다.

- 테이블에 대한 읽기/쓰기 권한 제어
- 테이블 생성 및 삭제
- 테이블 비우기
- 테이블 동기화

인덱스로 레코드 정렬

인덱스는 테이블에서 레코드 표시 순서를 결정합니다. 일반적으로 레코드는 기본 인덱스를 기초로 하여 오름차순으로 표시됩니다. 이러한 기본 동작은 애플리케이션에서 개입할 필요가 없습니다. 하지만 다른 정렬 순서를 원하는 경우에는 다음 두 가지 중의 하나를 지정해야 합니다.

- 대체 인덱스
- 정렬할 열 목록(SQL 기반 서버가 아닌 경우에는 사용할 수 없음)

인덱스를 사용하면 테이블의 데이터를 다른 순서로 나타낼 수 있습니다. SQL 기반 테이블에서 이러한 정렬 순서는 테이블의 레코드를 폐치하는 쿼리에서 ORDER BY 절을 생성하는 인덱스를 사용하여 구현됩니다. Paradox 및 dBASE 테이블과 같은 다른 테이블에서는 데이터 액세스 메커니즘에서 인덱스를 사용하여 원하는 순서로 레코드를 표시합니다.

인덱스 관련 정보 얻기

애플리케이션은 모든 테이블 타입 데이터셋에서 서버 정의 인덱스에 대한 정보를 얻을 수 있습니다. 데이터셋에 사용할 수 있는 인덱스 목록을 얻으려면 *GetIndexNames* 메소드를 호출합니다. *GetIndexNames*는 유효한 인덱스 이름으로 문자열 목록을 채웁니다. 예를 들어, 다음 코드는 *CustomersTable* 데이터셋에 정의된 모든 인덱스의 이름으로 리스트 박스를 채웁니다.

```
CustomersTable.GetIndexNames(ListBox1.Items);
```

참고 Paradox 테이블의 경우 기본 인덱스는 이름이 없으므로 *GetIndexNames*에 의해 반환되지 않습니다. 하지만 *IndexName* 속성을 빈 문자열로 설정하면 대체 인덱스를 사용한 후에도 Paradox 테이블의 인덱스를 기본 인덱스로 다시 변경할 수 있습니다.

현재 인덱스의 필드에 대한 정보를 얻으려면 다음과 같이 합니다.

- *IndexFieldCount* 속성을 사용하여 인덱스의 열 수를 확인합니다.
- *IndexFields* 속성을 사용하여 인덱스를 구성하는 열에 대한 필드 컴포넌트 목록을 조사합니다.

다음 코드는 *IndexFieldCount* 및 *IndexFields*를 사용하여 애플리케이션의 열 이름 목록을 통해 반복하는 방법을 보여 줍니다.

```
var
  I:Integer;
  ListOfIndexFields:array[0 to 20] of string;
begin
  with CustomersTable do
    begin
      for I := 0 to IndexFieldCount - 1 do
        ListOfIndexFields[I] := IndexFields[I].FieldName;
      end;
    end;
end;
```

참고 *IndexFieldCount*는 표현식 인덱스에 열려 있는 dBASE 테이블에 대해 유효하지 않습니다.

IndexName으로 인덱스 지정

IndexName 속성을 사용하여 인덱스가 활성화되게 합니다. 활성화되면 인덱스는 데이터셋의 레코드 순서를 결정합니다. 또한 마스터/디테일 연결, 인덱스 기반 검색 또는 인덱스 기반 필터링의 기초로 사용할 수 있습니다.

인덱스를 활성화하려면 *IndexName* 속성을 인덱스 이름으로 설정합니다. 일부 데이터베이스 시스템에서는 기본 인덱스에 이름이 없습니다. 이러한 인덱스를 활성화하려면 *IndexName*을 빈 문자열로 설정합니다.

디자인 타임에 Object Inspector에서 속성의 생략 버튼을 클릭하면 사용 가능한 인덱스의 목록에서 인덱스를 선택할 수 있습니다. 런타임 시 **String** 리터럴 또는 변수를 사용하여 *IndexName*을 설정합니다. *GetIndexNames* 메소드를 호출하면 사용 가능한 인덱스 목록을 얻을 수 있습니다.

다음 코드는 *CustomersTable*의 인덱스를 *CustDescending*으로 설정합니다.

```
CustomersTable.IndexName := 'CustDescending';
```

IndexFieldNames로 인덱스 작성

원하는 정렬 순서를 구현하는 정의된 인덱스가 없으면 *IndexFieldNames* 속성을 사용하여 의사(pseudo) 인덱스를 만들 수 있습니다.

참고 *IndexName*과 *IndexFieldNames*는 함께 사용할 수 없습니다. 한 속성을 설정하면 다른 속성에 설정된 값이 지워집니다.

*IndexFieldNames*의 값은 문자열입니다. 정렬 순서를 지정하려면 사용할 각 열 이름을 사용할 순서대로 나열하고 세미콜론으로 그 이름들을 구분합니다. 오름차순으로만 정렬됩니다. 정렬의 대소문자 구분은 서버의 기능에 따라 달라집니다. 자세한 내용은 서버 설명서를 참조하십시오.

다음 코드는 *LastName*을 기반으로 한 다음 *FirstName*을 기반으로 하여 *PhoneTable*의 정렬 순서를 설정합니다.

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

참고 Paradox 및 dBASE 테이블에서 *IndexFieldNames*를 사용하는 경우, 데이터셋은 지정한 열을 사용하는 인덱스를 찾으려고 합니다. 지정한 열을 사용하는 인덱스를 찾을 수 없으면 예외가 발생합니다.

인덱스를 사용하여 레코드 검색

*TDataSet*의 *Locate* 및 *Lookup* 메소드를 사용하여 데이터셋을 검색할 수 있습니다. 하지만 인덱스를 명시적으로 사용함으로써 일부 테이블 타입 데이터셋은 *Locate* 및 *Lookup* 메소드에서 제공하는 검색 성능을 향상시킬 수 있습니다.

ADO 데이터셋은 모두 *Seek* 메소드를 지원하며 현재 인덱스의 필드에 대한 필드 값 집합을 기반으로 하는 레코드로 이동합니다. *Seek*을 사용하여 일치하는 첫 번째 레코드 또는 마지막 레코드와 관련하여 커서를 이동할 위치를 지정할 수 있습니다.

TTable 및 모든 타입의 클라이언트 데이터셋은 유사한 인덱스 기반 검색을 지원하지만 관련 메소드를 조합하여 사용합니다. 다음 표는 *TTable*이 제공하는 여섯 개의 관련 메소드와 인덱스 기반 검색을 지원하는 클라이언트 데이터베이스에 대해 요약한 것입니다.

표 18.9 인덱스 기반 검색 메소드

메소드	용도
<i>EditKey</i>	검색 키 버퍼의 현재 내용을 보관하고 데이터셋을 <i>dsSetKey</i> 상태로 두어 사용하는 애플리케이션에서 검색을 수행하기 전에 기존 검색 조건을 수정할 수 있습니다.
<i>FindKey</i>	<i>SetKey</i> 메소드와 <i>GotoKey</i> 메소드를 하나의 메소드로 결합합니다.
<i>FindNearest</i>	<i>SetKey</i> 메소드와 <i>GotoNearest</i> 메소드를 하나의 메소드로 결합합니다.
<i>GotoKey</i>	데이터셋에서 검색 조건에 정확하게 일치하는 첫 번째 레코드를 검색하고 찾은 레코드로 커서를 이동합니다.
<i>GotoNearest</i>	문자열 기반 필드에서 부분 키 값에 기반하여 가장 가깝게 일치하는 레코드를 검색하고 해당 레코드로 커서를 이동합니다.
<i>SetKey</i>	검색 키 버퍼를 지우고 데이터셋을 <i>dsSetKey</i> 상태로 두어 애플리케이션에서 검색을 수행하기 전에 새로운 검색 조건을 지정할 수 있습니다.

부울 함수 *GotoKey* 및 *FindKey*는 검색이 성공하면 일치하는 레코드로 커서를 이동한 다음 *True*를 반환합니다. 검색이 실패하는 경우, 커서를 이동하지 않고 *False*를 반환합니다.

GotoNearest 및 *FindNearest*는 정확하게 일치하는 첫 번째 레코드나 일치하는 레코드가 없는 경우에는 지정한 검색 조건보다 큰 첫 번째 레코드로 커서를 이동합니다.

Goto 메소드로 검색 수행

Goto 메소드를 사용하여 검색을 수행하려면 다음과 같은 일반적인 단계를 따릅니다.

- 1 검색하는 데 사용할 인덱스를 지정합니다. 이것은 데이터셋의 레코드를 정렬하는 것과 동일한 인덱스입니다(18-26 페이지의 "인덱스로 레코드 정렬" 참조). 인덱스를 지정하려면 *IndexName* 또는 *IndexFieldNames* 속성을 사용합니다.
- 2 데이터셋을 엽니다.
- 3 *SetKey* 메소드를 호출하여 데이터셋을 *dsSetKey* 상태에 둡니다.

- 4 검색할 값을 *Fields* 속성에 지정합니다. *Fields*는 *TFields* 객체로서 열에 해당하는 서수를 지정함으로써 액세스 가능한 필드 컴포넌트의 인덱스 목록을 유지 관리합니다. 데이터셋의 첫 번째 열 번호는 0입니다.
- 5 *GotoKey* 또는 *GotoNearest*로 찾은 첫 번째 일치하는 레코드를 검색하여 이동합니다.

예를 들어, 버튼의 *OnClick* 이벤트에 연결된 다음 코드는 *GotoKey* 메소드를 사용하여 인덱스의 첫 번째 필드에 편집 상자의 텍스트와 정확하게 일치하는 값이 있는 첫 번째 레코드로 이동합니다.

```
procedure TSearchDemo.SearchExactClick(Sender:TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
```

*GotoNearest*도 이와 비슷하며 부분 필드 값과 가장 가깝게 일치하는 값을 검색합니다. 문자열 필드에서만 사용될 수 있습니다. 예를 들면 다음과 같습니다.

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
```

인덱싱된 첫 필드 값의 처음 두 글자가 "Sm"인 레코드가 있는 경우 이 레코드에 커서가 위치합니다. 해당 레코드가 없는 경우 커서의 위치는 변하지 않고 *GotoNearest*는 *False*를 반환합니다.

Find 메소드로 검색 수행

Find 메소드는 검색할 키 필드 값을 지정하기 위해 명시적으로 데이터셋을 *dsSetKey* 상태로 둘 필요가 없다는 점을 제외하면 *Goto* 메소드와 동일합니다. *Find* 메소드를 사용하여 검색을 수행하려면 다음과 같은 일반적인 단계를 따릅니다.

- 1 검색하는 데 사용할 인덱스를 지정합니다. 이것은 데이터셋의 레코드를 정렬하는 것과 동일한 인덱스입니다(18-26 페이지의 "인덱스로 레코드 정렬" 참조). 인덱스를 지정하려면 *IndexName* 또는 *IndexFieldNames* 속성을 사용합니다.
- 2 데이터셋을 엽니다.
- 3 *FindKey* 또는 *FindNearest*를 사용하여 첫 번째 레코드 또는 가장 가까운 레코드를 검색하여 이동합니다. 두 메소드에는 하나의 매개변수, 필드 값의 쉼표로 분리된 목록이 필요하며 각 필드 값은 원본으로 사용하는 테이블의 인덱싱된 열에 해당합니다.

참고 *FindNearest*는 문자열 필드에서만 사용할 수 있습니다.

성공적인 검색 이후 현재 레코드 지정

기본적으로는 검색이 성공하면 검색 조건에 일치하는 첫 번째 레코드로 커서가 이동합니다. 원한다면 *KeyExclusive* 속성을 *True*로 설정하여 첫 번째 일치하는 레코드의 다음 레코드로 커서를 이동할 수도 있습니다.

기본적으로 *KeyExclusive*는 *False*이며, 이는 검색이 성공하면 첫 번째 일치하는 레코드에 커서가 이동한다는 의미입니다.

부분 키 검색

데이터셋에 두 개 이상의 키 열이 있고 이 키의 서브셋에서 값을 검색하려는 경우에는 *KeyFieldCount*를 검색할 열의 번호로 설정합니다. 예를 들어, 데이터셋의 현재 인덱스에 세 개의 열이 있고 첫 번째 열만 사용하여 값을 검색하려면 *KeyFieldCount*를 1로 설정합니다.

여러 열 키를 갖는 테이블 타입 데이터셋의 경우 첫 번째 열부터 시작하여 이웃하는 열의 값만을 검색할 수 있습니다. 예를 들면, 3열 키의 경우 첫 번째 열에서 검색하거나, 첫 번째와 두 번째 열에서 검색하거나, 첫 번째, 두 번째, 세 번째 열에서 값을 검색할 수 있지만 첫 번째와 세 번째 열에서는 검색할 수 없습니다.

검색 반복 또는 확장

SetKey 또는 *FindKey*를 호출할 때마다 메소드는 *Fields* 속성에 있는 이전 값을 지웁니다. 이전에 설정된 필드를 사용하여 검색을 반복하거나 또는 검색에 사용되는 필드를 추가하려면 *SetKey*와 *FindKey* 대신 *EditKey*를 호출하십시오.

예를 들어, "CityIndex" 인덱스의 City 필드를 기반으로 Employee 테이블에 대한 검색을 이미 실행했다고 가정합니다. 계속해서 "CityIndex"는 *City* 필드와 *Company* 필드를 모두 포함한다고 가정해 보십시오. 특정 도시에 있는 특정 회사 이름으로 레코드를 찾으려면 다음과 같은 코드를 사용합니다.

```
Employee.KeyFieldCount := 2;
Employee.EditKey;
Employee['Company'] := Edit2.Text;
Employee.GotoNearest;
```

범위로 레코드 제한

필터를 사용(18-12 페이지의 "필터를 사용한 데이터 서브셋 표시 및 편집" 참조)하면 모든 데이터셋 데이터의 서브셋을 임시로 보고 편집할 수 있습니다. 일부 테이블 타입 데이터셋은 사용 가능한 레코드의 서브셋에 액세스하기 위해 범위라는 추가적인 방법을 지원합니다.

범위는 *TTable* 및 클라이언트 데이터셋에만 적용됩니다. 범위와 필터는 서로 유사하지만 다른 용도를 가집니다. 다음 항목에서는 범위와 필터의 차이점과 범위를 사용하는 방법에 대해 설명합니다.

범위와 필터의 차이점 이해

범위와 필터는 보이는 레코드를 모든 사용 가능 레코드의 서브셋으로 제한하지만 이를 수행하는 방법은 서로 다릅니다. 범위는 지정된 경계 값들 사이의 연속적으로 인덱싱된 레코드 집합입니다. 예를 들어, 성에 따라 인덱싱된 직원 데이터베이스에서 "Jones"와 "Smith" 사이에 있는 성을 가진 직원들을 모두 표시하도록 범위를 적용할 수 있습니다.

범위는 인덱스에 따라 다르기 때문에 현재 인덱스를 범위를 정의하는 데 사용할 수 있는 인덱스로 설정해야 합니다. 레코드를 정렬할 인덱스를 지정할 때와 마찬가지로 *IndexName* 또는 *IndexFieldNames* 속성을 사용하여 범위를 정의할 인덱스를 할당할 수 있습니다.

반면 필터는 인덱스에 상관 없이 특정 데이터를 공유하는 레코드 집합을 말합니다. 예를 들어, 캘리포니아에 살면서 회사에 근무한 지 5년 이상 되는 직원들을 모두 표시하도록 직원 데이터베이스를 필터링할 수 있습니다. 필터에서 인덱스를 사용할 수 있지만 필터가 인덱스에 의존하지는 않습니다. 필터는 애플리케이션에서 데이터셋을 스크롤할 때 한 레코드씩 적용됩니다.

일반적으로 필터가 범위보다 더 유연합니다. 하지만 데이터셋이 크고 애플리케이션에서 관심있는 레코드가 연속적으로 인덱싱된 그룹으로 이미 블록화되어 있는 경우는 범위가 더 효과적일 수 있습니다. 매우 큰 데이터셋의 경우, 쿼리 타입 데이터셋의 WHERE 절을 사용하여 데이터를 선택하는 것이 훨씬 더 효율적일 수 있습니다. 쿼리 지정에 대한 자세한 내용은 18-42 페이지의 "쿼리 타입 데이터셋 사용"을 참조하십시오.

범위 지정

범위를 지정하는 데는 다음과 같은 두 가지 상호 배타적인 방법이 있습니다.

- *SetRangeStart*와 *SetRangeEnd*를 사용하여 시작과 끝을 각각 지정합니다.
- *SetRange*를 사용하여 시작과 끝을 한꺼번에 지정합니다.

범위의 시작 설정

SetRangeStart 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 시작 값 목록을 만들기 시작합니다. 일단 *SetRangeStart*를 호출하면 *Fields* 속성에 대한 다음 할당은 범위를 적용할 때 사용할 시작 인덱스 값으로 간주됩니다. 지정된 필드는 현재 인덱스에 적용해야 합니다.

예를 들어, 애플리케이션에서 CUSTOMER 테이블에 연결된 *Customers*라는 이름의 *TSQLClientDataSet* 컴포넌트를 사용하며 사용자가 *Customers* 데이터셋의 각 필드에 대해 영구적 (persistent) 필드 컴포넌트를 작성했다고 가정해 보십시오. CUSTOMER 는 첫 번째 열 (*CustNo*)에 인덱싱되어 있습니다. 애플리케이션의 폼에는 범위의 시작과 끝 값을 지정하는 데 사용되는 *StartVal*과 *EndVal*이라는 두 개의 편집 컴포넌트가 있습니다. 다음 코드는 범위를 생성하고 적용하는 데 사용할 수 있습니다.

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

이 코드는 *Fields*에 값을 할당하기 전에 *EndVal*에 입력된 텍스트가 Null이 아님을 확인합니다. *StartVal*에 입력된 텍스트가 Null인 경우 어떠한 값도 Null보다는 크기 때문

에 데이터셋의 시작부터 모든 레코드가 포함됩니다. 그러나 *EndVal*에 입력된 텍스트가 Null인 경우 어떠한 값도 Null보다는 크므로 레코드가 하나도 포함되지 않습니다.

여러 열 인덱스의 경우 인덱스의 모든 필드나 일부 필드에 대해 시작 값을 지정할 수 있습니다. 인덱스에 사용된 필드에 값을 지정하지 않으면 범위를 적용할 때 Null로 간주됩니다. 인덱스에 없는 필드에 값을 설정하려고 하면 데이터셋은 예외를 발생시킵니다.

팁 데이터셋의 처음에서 시작하려면 *SetRangeStart* 호출을 생략합니다.

범위의 시작 지정을 완료하기 위해서는 *SetRangeEnd*를 호출하거나 범위를 적용 또는 취소하십시오. 범위 적용 및 취소에 대한 자세한 내용은 18-34 페이지의 "범위의 적용 또는 취소"를 참조하십시오.

범위의 끝 설정

SetRangeEnd 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 끝 값 목록을 만들기 시작합니다. 일단 *SetRangeEnd*를 호출하면 *Fields* 속성에 대한 다음 할당은 범위를 적용할 때 사용하는 끝 인덱스 값으로 간주됩니다. 지정된 필드는 현재 인덱스에 적용해야 합니다.

경고 범위의 끝을 데이터셋의 마지막 레코드로 정하고자 하더라도 항상 범위의 끝 값을 지정하십시오. 끝 값을 지정하지 않으면 Delphi에서는 범위의 끝 값을 Null 값으로 간주합니다. 끝 값이 Null인 범위는 항상 비어 있습니다.

끝 값을 할당하는 가장 쉬운 방법은 *FieldByName* 메소드를 호출하는 것입니다. 예를 들면, 다음과 같습니다.

```
with Contacts do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

범위 값의 시작을 지정할 때 인덱스에 없는 필드에 대해 값을 설정하고자 하면 데이터셋은 예외를 발생시킵니다.

범위의 끝 지정을 완료하려면 범위를 적용하거나 취소하십시오. 범위 적용 및 취소에 대한 자세한 내용은 18-34 페이지의 "범위의 적용 또는 취소"를 참조하십시오.

시작 및 끝 범위 값 설정

*SetRangeStart*와 *SetRangeEnd*를 각각 호출하여 범위 경계를 지정하는 대신 *SetRange* 프로시저를 한 번 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 시작과 끝 값을 설정할 수 있습니다.

*SetRange*는 시작 값 집합과 끝 값 집합의 두 개의 상수 배열 매개변수를 사용합니다. 예를 들어, 다음 문장은 두 개의 열 인덱스를 기반으로 범위를 설정합니다.

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

여러 열 인덱스의 경우 인덱스의 모든 필드나 일부 필드에 시작과 끝 값을 지정할 수 있습니다. 인덱스에 사용된 필드에 값을 지정하지 않으면 범위를 적용할 때 Null로 간주됩니다. 인덱스의 첫 번째 필드에 대한 값을 생략하려면 생략된 필드에 Null을 전달하십시오.

범위의 끝이 데이터셋의 마지막 레코드이더라도 범위의 끝 값을 항상 지정하십시오. 끝 값을 지정하지 않으면 데이터셋은 범위의 끝 값을 Null로 간주합니다. 끝 값이 Null인 범위는 시작 값이 끝 값보다 크거나 같기 때문에 항상 비어 있습니다.

부분 키에 기반하여 범위 지정

키가 하나 이상의 문자열 필드로 구성되는 경우, *SetRange* 메소드는 부분 키를 지원합니다. 예를 들어, 인덱스가 *LastName*과 *FirstName* 열에 기반하는 경우 다음과 같은 범위 지정이 유효합니다.

```
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Zzzzzz';
Contacts.ApplyRange;
```

이 코드는 *LastName*이 "Smith"보다 크거나 같은 범위의 모든 레코드를 포함시킵니다. 다음과 같은 값 지정도 가능합니다.

```
Contacts['LastName'] := 'Sm';
```

이 문장은 *LastName*이 "Sm"보다 크거나 같은 레코드를 포함시킵니다.

경계 값과 일치하는 레코드 포함 또는 제외

기본적으로 범위에는 지정된 시작 범위보다 크거나 같은 레코드와 지정된 끝 범위보다 작거나 같은 모든 레코드가 포함됩니다. 이 행동은 *KeyExclusive* 속성에 의해 제어됩니다. *KeyExclusive*는 기본값이 *False*입니다.

원한다면 클라이언트 데이터셋의 *KeyExclusive* 속성을 *True*로 설정하여 끝 범위와 동일한 레코드를 제외할 수 있습니다. 예를 들면, 다음과 같습니다.

```
Contacts.KeyExclusive := True;
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Tyler';
Contacts.ApplyRange;
```

이 코드는 *LastName*의 범위가 "Smith"보다 크거나 같고 "Tyler"보다 작은 모든 레코드를 포함시킵니다.

범위 수정

다음 두 함수를 사용하면 범위에 대한 기존의 경계 조건을 수정할 수 있습니다.

*EditRangeStart*는 범위의 시작 값을 변경할 경우에 사용하고 *EditRangeEnd*는 범위의 끝 값을 변경할 경우에 사용합니다.

범위를 편집하고 적용하기 위한 프로세스에는 다음과 같은 일반적인 단계가 있습니다.

- 1 데이터셋을 *ds.SetKey* 상태로 두고 범위의 시작 인덱스 값을 수정합니다.
- 2 범위의 끝 인덱스 값을 수정합니다.
- 3 데이터셋에 범위를 적용합니다.

범위의 시작 또는 끝 값을 수정하거나 또는 두 가지 경계 조건을 모두 수정할 수 있습니다. 데이터셋에 현재 적용된 범위의 경계 조건을 수정하는 경우, *ApplyRange*를 다시 호출하기 전까지는 변경 내용이 적용되지 않습니다.

범위의 시작 편집

EditRangeStart 프로시저를 호출하여 데이터셋을 *ds.SetKey* 상태로 만들고 범위 시작 값의 현재 목록을 수정하기 시작합니다. 일단 *EditRangeStart*를 호출하면 *Fields* 속성에 대한 다음 할당은 범위를 적용할 때 사용하는 현재 인덱스 값을 덮어씁니다.

팁 처음에 부분 키에 기반하여 시작 범위를 만든 경우, *EditRangeStart*를 사용하여 범위의 시작 값을 확장할 수 있습니다. 부분 키에 기반한 범위에 대한 자세한 내용은 18-33 페이지의 "부분 키에 기반하여 범위 지정"을 참조하십시오.

범위의 끝 편집

EditRangeEnd 프로시저를 호출하여 데이터셋을 *ds.SetKey* 상태로 두고 범위의 끝 값 목록을 만들기 시작합니다. 일단 *EditRangeEnd*를 호출하면 *Fields* 속성에 대한 다음 할당은 범위를 적용할 때 사용하는 끝 인덱스 값으로 간주됩니다.

범위의 적용 또는 취소

범위의 시작을 지정하기 위해 *SetRangeStart* 또는 *EditRangeStart*를 호출하거나 범위의 끝을 지정하기 위해 *SetRangeEnd* 또는 *EditRangeEnd*를 호출할 경우, 데이터셋은 *ds.SetKey* 상태가 됩니다. 사용자가 범위를 적용 또는 취소할 때까지 계속 이 상태를 유지합니다.

범위 적용

범위를 지정할 때 사용자가 정의하는 경계 조건은 사용자가 범위를 적용하기 전까지는 효력이 없습니다. 범위가 효력을 발생하도록 하려면 *ApplyRange* 프로시저를 호출하십시오. *ApplyRange*는 사용자가 지정된 데이터셋의 서브셋에 있는 데이터를 보거나 액세스하는 것을 즉시 제한합니다.

범위 취소

CancelRange 메소드는 범위의 애플리케이션을 종료하고 전체 데이터셋에 대한 액세스를 복구합니다. 범위를 취소하면 데이터셋의 모든 레코드에 대한 액세스가 복구되더라도 해당 범위의 경계 범위 조건은 계속 유지되어 사용자가 나중에 범위를 다시 적용할 수 있습니다. 범위 경계는 사용자가 새로운 범위 경계를 제시하거나 기존 경계를 수정하기 전까지는 계속 유지됩니다. 예를 들어, 다음과 같은 코드는 유효합니다.

```

:
MyTable.CancelRange;

```

```

:
{later on, use the same range again. No need to call SetRangeStart, etc.}
MyTable.ApplyRange;
:

```

마스터/디테일 관계 생성

테이블 타입 데이터셋은 마스터/디테일 관계로 연결될 수 있습니다. 사용자가 마스터/디테일 관계를 설정할 경우, 한 테이블(디테일)의 모든 레코드가 다른 테이블(마스터)의 현재 레코드 하나에 일치하도록 두 데이터셋을 연결합니다.

테이블 타입 데이터셋은 매우 다른 다음 두 가지 방식 중 하나를 이용해서 마스터/디테일 관계를 지원합니다.

- 모든 테이블 타입 데이터셋은 커서를 연결하여 다른 데이터셋의 디테일처럼 작동할 수 있습니다. 이 프로세스는 아래에 있는 "테이블을 다른 데이터셋의 디테일로 만들기"에서 설명합니다.
- *TTable*, *TSQLTable* 및 모든 클라이언트 데이터셋은 중첩 디테일 테이블을 사용하는 마스터/디테일 관계에서 마스터처럼 작동할 수 있습니다. 이 프로세스는 18-37 페이지의 "중첩 디테일 테이블 사용"에서 다룹니다.

이러한 방법은 저마다 고유한 장점이 있습니다. 커서를 연결하면 마스터 테이블이 데이터셋인 마스터/디테일 관계를 만들 수 있습니다. 중첩 디테일을 사용하면 디테일 테이블처럼 작동할 수 있는 데이터셋의 타입이 제한되지만 데이터를 표시하는 방법에 대한 더 많은 옵션을 제공합니다. 마스터가 클라이언트 데이터셋이면 중첩 디테일은 캐시된 업데이트를 적용하는 더 강력한 메커니즘을 제공합니다.

테이블을 다른 데이터셋의 디테일로 만들기

테이블 타입 데이터셋의 *MasterSource* 및 *MasterFields* 속성은 두 데이터셋 간에 일대다 관계를 설정하는 데 사용할 수 있습니다.

MasterSource 속성은 테이블에서 마스터 테이블의 데이터를 가져오는 데이터 소스를 지정하는 데 사용됩니다. 이 데이터 소스는 모든 타입의 데이터셋에 연결할 수 있습니다. 예를 들어, 이 속성에 쿼리의 데이터 소스를 지정하면 클라이언트 데이터셋이 쿼리에서 발생하는 이벤트를 추적하도록 쿼리의 디테일로서 클라이언트 데이터셋을 연결할 수 있습니다.

데이터셋은 현재 인덱스를 기반으로 마스터 테이블에 연결됩니다. 디테일 데이터셋에서 추적하는 마스터 데이터셋의 필드를 지정하기 전에 해당 필드로 시작하는 디테일 데이터셋의 인덱스를 먼저 지정하십시오. *IndexName* 속성 또는 *IndexFieldNames* 속성을 사용할 수 있습니다.

사용할 인덱스를 지정하고 나면 *MasterFields* 속성을 사용하여 디테일 테이블의 인덱스 필드에 해당하는 마스터 데이터셋의 열을 나타냅니다. 여러 열 이름으로 데이터셋을 연결하려면 다음과 같이 세미콜론으로 필드 이름을 구분합니다.

```
Parts.MasterFields := 'OrderNo;ItemNo';
```

두 데이터셋 사이의 의미 있는 연결을 돕는 Field Link 디자이너를 사용할 수 있습니다. Field Link 디자이너를 사용하려면 *MasterSource*와 인덱스를 할당한 후 Object Inspector의 *MasterFields* 속성을 더블 클릭하십시오.

다음 단계를 따라 사용자가 고객 레코드를 둘러보고 현재 고객에 대한 모든 순서를 표시할 수 있는 일반 폼이 작성됩니다. 마스터 테이블은 *CustomersTable*이고 디테일 테이블은 *OrdersTable*입니다. 예제에서는 BDE 기반 *TTable* 컴포넌트를 사용하지만 동일한 메소드를 사용하여 모든 테이블 타입 데이터셋을 연결할 수 있습니다.

- 1 두 개의 *TTable* 컴포넌트와 두 개의 *TDataSource* 컴포넌트를 데이터 모듈에 둡니다.
- 2 첫 번째 *TTable* 컴포넌트의 속성을 다음과 같이 설정합니다.
 - *DatabaseName*: DBDEMOS
 - *TableName*: CUSTOMER
 - *Name*: CustomersTable
- 3 다음과 같이 두 번째 *TTable* 컴포넌트의 속성을 설정합니다.
 - *DatabaseName*: DBDEMOS
 - *TableName*: ORDERS
 - *Name*: OrdersTable
- 4 첫 번째 *TDataSource* 컴포넌트의 속성을 다음과 같이 설정합니다.
 - *Name*: CustSource
 - *DataSet*: CustomersTable
- 5 다음과 같이 두 번째 *TDataSource* 컴포넌트의 속성을 설정합니다.
 - *Name*: OrdersSource
 - *DataSet*: OrdersTable
- 6 두 개의 *TDBGrid* 컴포넌트를 폼에 놓습니다.
- 7 File|Use Unit을 선택하여 폼에서 데이터를 사용한다고 지정합니다.
- 8 첫 번째 그리드 컴포넌트의 *DataSource* 속성은 "CustSource"로 설정하고 두 번째 그리드의 *DataSource* 속성은 "OrdersSource"로 설정합니다.
- 9 *OrdersTable*의 *MasterSource* 속성을 "CustSource"로 설정합니다. 이렇게 하면 CUSTOMER 테이블(마스터 테이블)이 ORDERS 테이블(디테일 테이블)에 연결됩니다.
- 10 Object Inspector의 *MasterFields* 속성 값 상자를 더블 클릭하여 Field Link 디자이너를 호출하고 다음 속성을 설정합니다.
 - Available Indexes 필드에서 *CustNo*를 선택하여 *CustNo* 필드로 두 테이블을 연결합니다.
 - Detail Fields 필드 목록과 Master Fields 필드 목록 모두에서 *CustNo*를 선택합니다.
 - Add 버튼을 클릭하여 조인 조건을 추가합니다. Joined Fields 목록에 "CustNo -> CustNo"가 나타납니다.

- 선택한 내용을 커밋하려면 OK를 선택하고 Field Link 디자이너를 종료합니다.
- 11** 폼의 그리드에 데이터를 표시하도록 *CustomersTable*과 *OrdersTable*의 *Active* 속성을 *True*로 설정합니다.
- 12** 애플리케이션을 컴파일하여 실행합니다.

이제 애플리케이션을 실행하면 테이블이 함께 연결되었는지 확인할 수 있으며 CUSTOMER 테이블의 새 레코드로 이동할 때 현재의 고객에 속한 ORDERS 테이블의 해당 레코드만 볼 수 있습니다.

중첩 디테일 테이블 사용

중첩 테이블은 다른(마스터) 데이터셋에서 단일 데이터셋 필드의 값이 되는 디테일 데이터셋입니다. 서버 데이터를 나타내는 데이터셋의 경우, 중첩 디테일 데이터셋은 서버의 데이터셋 필드에만 사용할 수 있습니다. *TClientDataSet* 컴포넌트는 서버 데이터를 나타내지 않지만 중첩 디테일을 포함하는 컴포넌트용 데이터셋을 만들거나 마스터/디테일 관계의 마스터 테이블에 연결된 프로바이더로부터 데이터를 받는 경우에는 데이터셋 필드를 포함할 수도 있습니다.

참고 *TClientDataSet*의 경우, 마스터 테이블과 디테일 테이블에서 데이터베이스 서버로 업데이트 내용을 적용하려면 중첩 디테일 셋을 사용해야 합니다.

중첩 디테일 셋을 사용하려면 마스터 데이터셋의 *ObjectView* 속성이 *True*여야 합니다. 테이블 타입 데이터셋에 중첩 디테일 데이터셋이 포함된 경우, *TDBGrid*는 중첩 디테일을 팝업 메뉴에 표시하도록 지원합니다. 이 방법은 19-26 페이지의 "데이터셋 필드 표시"에서 다룹니다.

그 대신 디테일 셋에 대해 별도의 데이터셋 컴포넌트를 사용하여 data-aware 컨트롤에 디테일 데이터셋을 표시하거나 편집할 수 있습니다. 디자인 타임에 Fields 편집기를 사용하여 (마스터) 데이터셋의 필드에 영구적인 필드를 만듭니다. 마스터 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Fields 편집기를 선택합니다. Add Fields를 마우스 오른쪽 버튼으로 클릭하여 사용자의 클라이언트 데이터셋에 새로운 영구적 필드를 추가합니다. DataSet Field 타입의 새로운 필드를 정의합니다. Fields 편집기에서 디테일 테이블의 구조를 정의합니다. 또한 마스터 데이터셋에서 사용되는 모든 다른 필드에 대해서도 영구적인 필드를 추가해야 합니다.

디테일 테이블의 데이터셋 컴포넌트는 마스터 테이블에서 허용되는 타입의 데이터셋 자손입니다. *TTable* 컴포넌트만 *TNestedDataSet* 컴포넌트를 중첩 데이터셋으로 허용합니다. *TSQLTable* 컴포넌트는 다른 *TSQLTable* 컴포넌트를 허용합니다. *TClientDataSet* 컴포넌트는 다른 클라이언트 데이터셋을 허용합니다. 컴포넌트 팔레트에서 적당한 타입의 데이터셋을 선택하고 폼이나 데이터 모듈에 추가합니다. 이 디테일 데이터셋의 *DataSetField* 속성을 마스터 데이터셋의 영구적 데이터셋 필드로 설정합니다. 마지막으로 데이터 소스 컴포넌트를 폼이나 데이터 모듈에 놓고 *DataSet* 속성을 디테일 데이터셋으로 설정합니다. data-aware 컨트롤은 이 데이터 소스를 사용하여 디테일 셋의 데이터에 액세스할 수 있습니다.

테이블에 대한 읽기/쓰기 권한 제어

기본적으로 테이블 타입 데이터셋을 열면 원본으로 사용하는 데이터베이스 테이블에 읽기 및 쓰기 권한을 요청합니다. 원본으로 사용하는 데이터베이스 테이블의 특성에 따라 요청한 쓰기 권한이 허용되지 않을 수도 있습니다. 예를 들면, 원격 서버에 있는 SQL 테이블에 대해 쓰기 권한을 요청하는 경우와 서버에서 테이블의 권한을 읽기 전용으로 제한한 경우입니다.

참고 데이터셋 프로바이더가 데이터 패킷으로 제공하는 정보에서 사용자가 데이터를 편집할 수 있는지 결정하는 *TClientDataSet*에는 적용되지 않습니다. 단방향 데이터셋으로 항상 읽기 전용인 *TSQLTable*에도 적용되지 않습니다.

테이블이 열리면 *CanModify* 속성을 검사하여 원본으로 사용하는 데이터베이스 또는 데이터셋 프로바이더를 통해 사용자가 테이블의 데이터를 편집할 수 있는지 여부를 확인할 수 있습니다. *CanModify*가 *False*인 경우, 애플리케이션은 데이터베이스에 쓸 수 없습니다. *CanModify*가 *True*인 경우, 테이블의 *ReadOnly* 속성이 *False*이면 애플리케이션은 데이터베이스에 쓸 수 있습니다.

*ReadOnly*는 사용자가 데이터를 보기 및 편집할 수 있는지 여부를 결정합니다. *ReadOnly*가 *False*(기본값)이면 사용자는 데이터를 보기 및 편집할 수 있습니다. 사용자가 데이터를 볼 수만 있도록 제한하려면 테이블을 열기 전에 *ReadOnly*를 *True*로 설정합니다.

참고 *ReadOnly*는 항상 읽기 전용인 *TSQLTable*을 제외한 모든 테이블 타입 데이터셋에서 구현됩니다.

테이블 생성 및 삭제

일부 테이블 타입 데이터셋을 사용하면 디자인 타임이나 런타임에 원본으로 사용하는 테이블을 만들고 삭제할 수 있습니다. 일반적으로 데이터베이스 테이블은 데이터베이스 관리자가 만들고 삭제합니다. 하지만 이것은 애플리케이션에서 사용할 수 있는 데이터베이스 테이블을 만들고 삭제하는 애플리케이션 개발 및 테스트 기간에 유용하게 사용됩니다.

테이블 생성

*TTable*과 *TIBTable*을 둘 다 사용하면 SQL을 사용하지 않고 원본으로 사용하는 데이터베이스 테이블을 만들 수 있습니다. 마찬가지로 *TClientDataSet*을 사용하면 데이터셋 프로바이더를 사용하지 않을 때도 데이터셋을 만들 수 있습니다. *TTable*과 *TClientDataSet*을 사용하면 디자인 타임이나 런타임에 테이블을 만들 수 있습니다. *TIBTable*만 사용하면 런타임에 테이블을 만들 수 있습니다.

테이블을 만들기 전에 만들려는 테이블의 구조를 지정하는 속성을 설정해야 합니다. 구체적으로 다음과 같은 내용을 지정해야 합니다.

- 새 테이블을 호스트할 데이터베이스. *TTable*의 경우에는 *DatabaseName* 속성을 사용하여 데이터베이스를 지정합니다. *TIBTable*의 경우에는 *Database* 속성에 할당된 *TIBDataBase* 컴포넌트를 사용해야 합니다. 클라이언트 데이터셋은 데이터베이스를 사용하지 않습니다.

- 데이터베이스의 타입 (*TTable*에만 해당). *TableType* 속성을 원하는 테이블 타입으로 설정합니다. Paradox, dBASE 또는 ASCII 테이블의 경우에는 *TableType*을 *ttParadox*, *ttDBase*, *ttASCII*로 각각 설정합니다. 다른 모든 테이블 타입의 경우에는 *TableType*을 *ttDefault*로 설정합니다.
- 만들려는 테이블의 이름. *TTable*과 *TIBTable* 모두에는 새 테이블의 이름을 지정할 수 있는 *TableName* 속성이 있습니다. 클라이언트 데이터셋은 테이블 이름을 사용하지 않지만 새 테이블을 저장하기 전에 *FileName* 속성을 지정해야 합니다. 기존 테이블의 이름과 중복되는 테이블을 만들면 새로 만든 테이블이 기존 테이블과 모든 데이터를 덮어씁니다. 이전 테이블과 데이터는 복구할 수 없습니다. 기존 테이블을 덮어쓰지 않기 위해 런타임에 *Exists* 속성을 검사할 수 있습니다. *Exists*는 *TTable*과 *TIBTable*에서만 사용할 수 있습니다.
- 새 테이블의 필드. 다음과 같은 두 가지 방법으로 지정합니다.
 - *FieldDefs* 속성에 필드 정의를 추가할 수 있습니다. 디자인 타임에 Object Inspector의 *FieldDefs* 속성을 더블 클릭하여 컬렉션 에디터를 엽니다. 컬렉션 에디터를 사용하여 필드 정의의 속성을 추가, 제거 또는 변경합니다. 런타임에 기존의 모든 필드 정의를 지운 다음 *AddFieldDef* 메소드를 사용하여 새로운 필드 정의를 추가합니다. 각각의 새로운 필드 정의에 대해 필드의 원하는 속성을 지정하도록 *TFieldDef* 객체의 속성을 설정합니다.
 - 영구적인 필드 컴포넌트를 대신 사용할 수 있습니다. 디자인 타임에 데이터셋을 더블 클릭하여 *Fields* 편집기를 엽니다. *Fields* 편집기에서 마우스 오른쪽 버튼을 클릭하고 New Field 명령을 선택합니다. 필드의 기본 속성을 설명합니다. 필드가 생성되면 *Fields* 편집기에서 필드를 선택하여 Object Inspector에서 속성을 변경할 수 있습니다.
- 새 테이블의 인덱스 (옵션). 디자인 타임에 Object Inspector의 *IndexDefs* 속성을 더블 클릭하여 컬렉션 에디터를 엽니다. 컬렉션 에디터를 사용하여 인덱스 정의의 속성을 추가, 제거 또는 변경합니다. 런타임에 기존의 모든 인덱스 정의를 지운 다음 *AddIndexDef* 메소드를 사용하여 새로운 인덱스 정의를 추가합니다. 각각의 새로운 인덱스 정의에 대해 인덱스의 원하는 속성을 지정하도록 *TIndexDef* 객체의 속성을 설정합니다.

참고 필드 정의 객체 대신 영구적인 필드 컴포넌트를 사용하는 경우에는 새 테이블의 인덱스를 정의할 수 없습니다.

디자인 타임에 테이블을 만들려면 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Create Table (*TTable*) 이나 Create Data Set (*TClientDataSet*) 을 선택합니다. 이 명령은 필요한 정보를 모두 지정한 후에 컨텍스트 메뉴에 나타납니다.

런타임에 테이블을 만들려면 *CreateTable* 메소드 (*TTable* 및 *TIBTable*) 또는 *CreateDataSet* 메소드 (*TClientDataSet*) 를 호출합니다.

참고 디자인 타임에 정의를 설정한 다음 런타임에 *CreateTable*(또는 *CreateDataSet*) 메소드를 호출하여 테이블을 만듭니다. 하지만 이렇게 하려면 런타임에 지정된 정의가 데이터셋 컴포넌트로 저장되도록 지정해야 합니다. 기본적으로 필드와 인덱스 정의는 런타임에 동적으로 생성됩니다. *StoreDefs* 속성을 *True*로 설정하여 정의를 데이터셋으로 저장하도록 지정합니다.

팁 *TTable*을 사용하는 경우에는 디자인 타임에 기존 테이블의 필드 정의와 인덱스 정의를 미리 로드할 수 있습니다. 기존 테이블을 지정하도록 *DatabaseName* 및 *TableName* 속성을 설정합니다. 테이블 컴포넌트를 마우스 오른쪽 버튼으로 클릭하고 Update Table Definition을 선택합니다. 이렇게 하면 기존 테이블의 필드와 인덱스를 설명하도록 *FieldDefs*와 *IndexDefs* 속성의 값이 자동으로 설정됩니다. 그런 다음 기존 테이블의 이름을 바꿀지 확인하는 메시지를 취소하고 만들려는 테이블을 지정하도록 *DatabaseName*과 *TableName*을 재설정합니다.

참고 Oracle8 테이블을 만들면 객체 필드(ADT 필드, 배열 필드 및 데이터셋 필드)를 만들 수 없습니다.

다음 코드는 런타임에 새 테이블을 만들어 DBDEMOS 알리아스에 연결합니다. 새 테이블을 만들기 전에 기존 테이블의 이름과 일치하지 않는지 확인합니다.

```

var
  TableFound:Boolean;
begin
  with TTable.Create(nil) do // create a temporary TTable component
  begin
    try
      { set properties of the temporary TTable component }
      Active := False;
      DatabaseName := 'DBDEMOS';
      TableName := Edit1.Text;
      TableType := ttDefault;
      { define fields for the new table }
      FieldDefs.Clear;
      with FieldDefs.AddFieldDef do begin
        Name := 'First';
        DataType := ftString;
        Size := 20;
        Required := False;
      end;
      with FieldDefs.AddFieldDef do begin
        Name := 'Second';
        DataType := ftString;
        Size := 30;
        Required := False;
      end;
      { define indexes for the new table }
      IndexDefs.Clear;
      with IndexDefs.AddIndexDef do begin
        Name := '';
        Fields := 'First';
        Options := [ixPrimary];
      end;
      TableFound := Exists; // check whether the table already exists
      if TableFound then
        if MessageDlg('Overwrite existing table ' + Edit1.Text + '?',
          mtConfirmation, mbYesNoCancel, 0) = mrYes then
          TableFound := False;
      if not TableFound then
        CreateTable; // create the table
    finally

```

```

    Free; // destroy the temporary TTable when done
end;
end;
end;

```

테이블 삭제

TTable 및 *TIBTable*을 사용하면 SQL을 사용하지 않고도 원본으로 사용하는 데이터베이스 테이블에서 테이블을 삭제할 수 있습니다. 런타임에 테이블을 삭제하려면 데이터셋의 *DeleteTable* 메소드를 호출합니다. 예를 들어, 다음 문장은 원본으로 사용하는 데이터셋 테이블을 제거합니다.

```
CustomersTable.DeleteTable;
```

주의 *DeleteTable*로 테이블을 삭제하면 테이블과 테이블의 데이터가 영구히 삭제됩니다.

*TTable*을 사용하면 디자인 타임에 테이블을 삭제할 수도 있습니다. 테이블 컴포넌트를 마우스 오른쪽 버튼으로 클릭하고 컨텍스트 메뉴에서 Delete Table을 선택합니다. Delete Table 메뉴는 테이블 컴포넌트가 기존의 데이터베이스 테이블을 나타내는 경우에만 표시됩니다. *DatabaseName* 및 *TableName* 속성으로 기존 테이블을 지정합니다.

테이블 비우기

많은 테이블 타입 데이터셋은 테이블에 있는 데이터의 모든 행을 삭제할 수 있는 단일 메소드를 제공합니다.

- *TTable* 및 *TIBTable*의 경우에는 런타임에 *EmptyTable* 메소드를 호출하여 모든 레코드를 삭제할 수 있습니다.

```
PhoneTable.EmptyTable;
```

- *TADOTable*의 경우에는 *DeleteRecords* 메소드를 사용할 수 있습니다.

```
PhoneTable.DeleteRecords;
```

- *TSQLTable*의 경우에는 *DeleteRecords* 메소드를 사용할 수도 있습니다. 하지만 *DeleteRecords*의 *TSQLTable* 버전은 매개변수를 결코 취할 수 없다는 점에 유의합니다.

```
PhoneTable.DeleteRecords;
```

- 클라이언트 데이터셋의 경우에는 *EmptyDataSet* 메소드를 사용할 수 있습니다.

```
PhoneTable.EmptyDataSet;
```

참고 SQL 서버에 있는 테이블의 경우, 이러한 메소드는 테이블에 대해 DELETE 권한이 있는 경우에만 성공합니다.

주의 데이터셋을 비우면 삭제한 데이터가 영구히 삭제됩니다.

테이블 동기화

동일한 데이터베이스 테이블을 나타내는 둘 이상의 데이터셋이 있지만 데이터 소스 컴포넌트를 공유하지 않은 경우 각 데이터셋은 데이터에 대한 고유한 뷰와 고유한 현재 레

코드를 갖습니다. 사용자가 각 데이터셋을 통해 레코드에 액세스하면 컴포넌트의 현재 레코드가 다릅니다.

데이터셋이 *TTable*의 모든 인스턴스, *TIBTable*의 모든 인스턴스 또는 모든 클라이언트 데이터셋이면 *GotoCurrent* 메소드를 호출하여 각 데이터셋의 현재 레코드가 같아지게 할 수 있습니다. *GotoCurrent*는 고유한 데이터셋의 현재 레코드를 일치하는 데이터셋의 현재 레코드로 설정합니다. 예를 들어, 다음 코드는 *CustomerTableOne*의 현재 레코드가 *CustomerTableTwo*의 현재 레코드와 같아지도록 설정합니다.

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

팁 이런 방식으로 애플리케이션이 데이터셋을 동기화해야 하는 경우에는 데이터셋을 데이터 모듈에 두고 테이블을 액세스하는 각 유닛의 *uses* 절에 데이터 모듈의 유닛을 추가합니다.

별도의 폼에서 데이터셋을 동기화하려면 폼의 유닛을 다른 폼의 *uses* 절에 추가하고 적어도 하나 이상의 데이터셋 이름을 폼 이름으로 한정해야 합니다. 예를 들면 다음과 같습니다.

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

쿼리 타입 데이터셋 사용

다음과 같은 방법으로 쿼리 타입 데이터셋을 사용합니다.

- 1 적절한 데이터셋 컴포넌트를 데이터 모듈이나 폼에 두고 그 컴포넌트의 *Name* 속성을 사용자의 애플리케이션에 적합한 고유한 값으로 설정합니다.
- 2 쿼리할 데이터베이스 서버를 식별합니다. 각 쿼리 타입 데이터셋은 이 작업을 다르게 수행하지만 일반적으로 데이터베이스 연결 컴포넌트를 지정합니다.
 - *TQuery*의 경우에는 *DatabaseName* 속성을 사용하여 *TDatabase* 컴포넌트나 BDE 알리아스를 지정합니다.
 - *TADOQuery*의 경우에는 *Connection* 속성을 사용하여 *TADOConnection* 컴포넌트를 지정합니다.
 - *TSQLQuery*의 경우에는 *SQLConnection* 속성을 사용하여 *TSQLConnection* 컴포넌트를 지정합니다.
 - *TIBQuery*의 경우에는 *Database* 속성을 사용하여 *TIBConnection* 컴포넌트를 지정합니다.

데이터베이스 연결 컴포넌트 사용에 대한 내용은 17장 "데이터베이스에 연결"을 참조하십시오.

- 3 데이터셋의 *SQL* 속성에서 SQL 문을 지정하고 옵션으로 해당 문의 모든 매개변수를 지정합니다. 자세한 내용은 18-43 페이지의 "쿼리 지정" 및 18-45 페이지의 "쿼리의 매개변수 사용"을 참조하십시오.
- 4 쿼리 데이터를 비주얼 데이터 컨트롤과 함께 사용하려면 데이터 소스 컴포넌트를 데이터 모듈에 추가하고 *DataSet* 속성을 쿼리 타입 데이터셋으로 설정합니다. 데이터 소스 컴포넌트는 *결과 집합*이라는 쿼리의 결과를 data-aware 컴포넌트로 전달하여

표시합니다. *DataSource* 및 *DataField* 속성을 사용하여 data-aware 컴포넌트를 데이터 소스에 연결합니다.

- 5 쿼리 컴포넌트를 활성화합니다. 결과 집합을 반환하는 쿼리의 경우, *Active* 속성이거나 *Open* 메소드를 사용합니다. 테이블에서 작업을 수행하고 결과 집합을 반환하지 않는 쿼리를 실행하려면 런타임에 *ExecSQL* 메소드를 사용합니다. 쿼리를 한 번 이상 실행하는 경우, *Prepare*를 호출하여 데이터 액세스 계층을 초기화하고 매개변수 값을 쿼리로 바인딩할 수 있습니다. 쿼리 준비에 대한 내용은 18-48 페이지의 "쿼리 준비"를 참조하십시오.

쿼리 지정

트루 쿼리 타입 데이터셋의 경우에는 *SQL* 속성을 사용하여 데이터셋이 실행할 SQL 문을 지정합니다. *TADODataSet*, *TSQLDataSet* 및 클라이언트 데이터셋과 같은 일부 데이터셋은 *CommandText* 속성을 사용하여 동일한 작업을 수행합니다.

레코드를 반환하는 대부분의 쿼리가 *SELECT* 명령입니다. 일반적으로 이런 쿼리는 포함할 필드, 이들 필드를 선택할 테이블, 포함할 레코드를 제한하는 조건 및 결과 데이터셋의 순서를 정의합니다. 예를 들면, 다음과 같습니다.

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

레코드를 반환하지 않는 쿼리에는 *SELECT* 문이 아닌 *DDL* (Data Definition Language) 이나 *DML* (Data Manipulation Language) 문을 사용하는 문장이 포함됩니다. 예를 들어 *INSERT*, *DELETE*, *UPDATE*, *CREATE INDEX* 및 *ALTER TABLE* 명령은 레코드를 반환하지 않습니다. 명령에 사용되는 랭귀지는 서버 특정적이지만 일반적으로 *SQL* 랭귀지의 *SQL-92* 표준과 호환됩니다.

실행하는 *SQL* 명령은 사용 중인 서버에서 수용 가능해야 합니다. 데이터셋은 *SQL* 을 평가하지 않고 이를 실행하지도 않습니다. 데이터셋은 단지 실행을 위해 서버에 명령을 전달합니다. 필요에 따라 해당 문장이 복잡할 수도 있지만(예를 들면, *AND* 및 *OR* 같은 여러 개의 중첩 논리 연산자를 사용하는 *WHERE* 절이 있는 *SELECT* 문의 경우) 대부분의 경우 *SQL* 명령은 단 하나의 완전한 *SQL* 문이 되어야 합니다. 일부 서버는 또한 여러 문장을 허용하는 "일괄" 구문을 지원합니다. 서버가 이러한 구문을 지원하면 쿼리를 지정할 때 여러 문장을 입력할 수 있습니다.

쿼리에 사용되는 *SQL* 문은 축어적일 수 있으며 교체 가능한 매개변수를 포함할 수 있습니다. 매개변수를 사용하는 쿼리는 *매개변수화된 쿼리*라고 합니다. 매개변수화된 쿼리를 사용할 때 매개변수에 할당된 실제 값은 쿼리를 실행하거나 수행하기 전에 쿼리에 삽입됩니다. 런타임 시 *SQL* 문을 변경하지 않고 데이터에 대한 액세스 및 사용자의 뷰를 즉시 변경할 수 있으므로 매개변수화된 쿼리 사용이 매우 유연합니다. 매개변수화된 쿼리에 대한 자세한 내용은 18-45 페이지의 "쿼리의 매개변수 사용"을 참조하십시오.

SQL 속성을 사용하여 쿼리 지정

트루 쿼리 타입 데이터셋 (*TQuery*, *TADOQuery*, *TSQLQuery* 또는 *TIBQuery*)을 사용할 때는 쿼리를 *SQL* 속성에 할당합니다. *SQL* 속성은 *TStrings* 객체입니다. 이 *TStrings* 객체의 분리된 각 문자열은 쿼리의 분리된 행입니다. 여러 줄을 사용해도 쿼리가 서버에서 실행되는 방법에는 영향을 주지 않지만 문장을 논리적인 단위로 나누면 쿼리를 더 쉽게 수정하고 디버그할 수 있습니다.

```
MyQuery.Close;
MyQuery.SQL.Clear;
MyQuery.SQL.Add('SELECT CustNo, OrderNO, SaleDate');
MyQuery.SQL.Add(' FROM Orders');
MyQuery.SQL.Add('ORDER BY SaleDate');
MyQuery.Open;
```

아래 코드는 기존의 SQL 문에서 한 줄로만 수정하는 방법을 보여 줍니다. 이 경우 ORDER BY 절은 문장의 세 번째 줄에 이미 있으며 인덱스 2를 사용하여 *SQL* 속성을 통해 참조됩니다.

```
MyQuery.SQL[2] := 'ORDER BY OrderNo';
```

참고 SQL 속성을 지정하거나 수정할 때는 데이터셋을 닫아야 합니다.

디자인 타임에 String List 에디터를 사용하여 쿼리를 지정합니다. Object Inspector의 *SQL* 속성의 생략 버튼을 클릭해서 String List 에디터를 표시합니다.

참고 Delphi의 일부 버전에서는 *TQuery*를 사용하는 경우 SQL 빌더를 사용하여 데이터베이스에서 테이블과 필드가 표시되는 방식을 기반으로 쿼리를 생성할 수도 있습니다. SQL Builder를 사용하려면 쿼리 컴포넌트를 선택하고 마우스 오른쪽 버튼으로 클릭하여 컨텍스트 메뉴를 호출한 다음 Graphical Query Editor를 선택합니다. SQL Builder를 사용하는 방법을 익히려면 SQL Builder를 열고 온라인 도움말을 사용하십시오.

SQL 속성이 *TStrings* 객체이므로 *TStrings.LoadFromFile* 메소드를 호출하면 파일에서 쿼리의 텍스트를 로드할 수 있습니다.

```
MyQuery.SQL.LoadFromFile('custquery.sql');
```

SQL 속성의 *Assign* 메소드를 사용하면 문자열 목록 객체의 내용을 *SQL* 속성으로 복사할 수도 있습니다. *Assign* 메소드는 새 문장을 복사하기 전에 *SQL* 속성의 현재 내용을 자동으로 지웁니다.

```
MyQuery.SQL.Assign(Memo1.Lines);
```

CommandText 속성을 사용하여 쿼리 지정

TADODataSet, *TSQLDataSet* 또는 클라이언트 데이터셋을 사용할 때는 쿼리 문의 텍스트를 *CommandText* 속성에 할당합니다.

```
MyQuery.CommandText := 'SELECT CustName, Address FROM Customer';
```

디자인 타임에 Object Inspector에 직접 쿼리를 입력할 수 있습니다. 또한 데이터셋의 데이터베이스에 대한 연결이 이미 활성화된 경우에는 *CommandText* 속성의 생략 버튼을 클릭해서 Command Text 에디터를 표시할 수 있습니다. Command Text 에디터는 사용 가능한 테이블과 테이블의 필드를 나열하므로 쿼리를 쉽게 작성할 수 있습니다.

쿼리의 매개변수 사용

매개변수화된 SQL 문은 매개변수, 변수, 디자인 타임이나 런타임에 달라질 수 있는 매개변수의 값을 포함합니다. SQL 문에 나타나는 매개변수는 WHERE 절에서 비교를 위해 사용되는 값처럼 데이터 값으로 대체될 수 있습니다. 일반적으로 매개변수는 문장에 전달된 데이터 값을 대신합니다. 예를 들면, 다음의 INSERT 문에서는 삽입할 값을 매개변수로 전달합니다.

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

이 SQL 문에서 *:Name*, *:Capital* 및 *:Population*은 런타임 시 애플리케이션이 문장에 제공되는 실제 값에 대한 위치 표시자입니다. 매개변수 이름은 콜론으로 시작합니다. 콜론은 매개변수 값을 리터럴 값과 구분할 수 있도록 하는 데 필요합니다. 또한 쿼리에 물음표(?)를 추가하여 명명되지 않은 매개변수를 포함시킬 수 있습니다. 명명되지 않은 매개변수는 고유한 이름이 없기 때문에 위치로 식별합니다.

데이터셋이 쿼리를 실행하기 전에 쿼리 텍스트의 매개변수에 대한 값을 제공해야 합니다. *TQuery*, *TIBQuery*, *TSQLQuery* 및 클라이언트 데이터셋은 *Params* 속성을 사용하여 이 값을 저장합니다. *TADOQuery*는 대신 *Parameters* 속성을 사용합니다. *Params* 또는 *Parameters*는 매개변수 객체 (*TParam* 또는 *TParameter*)의 컬렉션이며 여기서 각 객체는 하나의 매개변수를 나타냅니다. 쿼리에 텍스트를 지정하면 데이터셋은 이러한 매개변수 객체 집합을 생성하고 데이터셋 타입에 따라 쿼리에서 추론할 수 있는 모든 속성을 초기화합니다.

참고 *ParamCheck* 속성을 *False*로 설정함으로써 쿼리 텍스트의 변경에 응답하는 매개변수 객체의 자동 생성을 억제할 수 있습니다. 이것은 쿼리 자체에 대한 매개변수가 아닌 DDL 문의 일부로 매개변수를 포함하는 DDL(Data Definition Language) 문에 유용합니다. 예를 들어, 내장 프로시저를 생성하는 DDL 문은 내장 프로시저의 일부인 매개변수를 정의할 수 있습니다. *ParamCheck*를 *False*로 설정하면 이러한 매개변수가 쿼리의 매개변수로 잘못 인식되지 않게 할 수 있습니다.

매개변수 값은 처음 실행되기 전에 SQL 문으로 바인딩되어야 합니다. 쿼리 컴포넌트는 쿼리를 실행하기 전에 사용자가 명시적으로 *Prepare* 메소드를 호출하지 않는 경우에도 자동으로 이러한 작업을 수행합니다.

팁 연결되어 있는 열의 실제 이름에 해당하는 매개변수에 변수 이름을 제공하는 것은 좋은 프로그래밍 방법입니다. 예를 들어, 열 이름이 "Number"면 해당하는 매개변수는 ":Number"가 됩니다. 일치하는 이름을 사용하는 것은 데이터셋이 데이터 소스를 사용하여 다른 데이터셋으로부터 매개변수 값을 가져오는 경우에 특히 중요합니다. 이 프로세스는 18-47 페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"에서 다룹니다.

디자인 타임에 매개변수 제공

디자인 타임에 매개변수 컬렉션 에디터를 사용하여 매개변수 값을 지정할 수 있습니다. 매개변수 컬렉션 에디터를 표시하려면 Object Inspector에서 *Params* 또는 *Parameters* 속성의 생략 버튼을 클릭합니다. SQL 문이 매개변수를 포함하지 않으면 컬렉션 에디터에 어떠한 객체도 나열되지 않습니다.

참고 매개변수 컬렉션 에디터는 다른 컬렉션 속성에 나타나는 것과 동일한 컬렉션 에디터입니다. 편집기가 다른 속성과 공유되기 때문에 오른쪽 버튼을 클릭해서 표시되는 컨텍스트 메뉴에는 Add 및 Delete 명령이 포함됩니다. 하지만 이 명령들은 쿼리 매개변수에 대해서는 사용할 수 없습니다. 매개변수를 추가하거나 삭제할 수 있는 유일한 방법은 SQL 문 자체에 있습니다.

각 매개변수는 매개변수 컬렉션 에디터에서 선택합니다. 그런 다음 Object Inspector를 사용하여 속성을 수정합니다.

Params 속성(*TParam* 객체)을 사용할 때는 다음을 확인하거나 수정해야 합니다.

- *DataType* 속성은 매개변수 값의 데이터 타입을 나열합니다. 일부 데이터셋의 경우에는 이 값이 올바르게 초기화될 수 있습니다. 데이터셋이 타입을 추론할 수 없을 경우 *DataType*은 *ftUnknown*이며 매개변수 값의 타입을 나타내도록 변경해야 합니다.

DataType 속성은 매개변수에 대한 논리 데이터 타입을 나열합니다. 일반적으로 이 데이터 타입은 서버 데이터 타입에 따릅니다. 특정 논리 타입에서 서버 데이터 타입으로의 매핑에 대해서는 설명서에서 데이터 액세스 메커니즘 (BDE, dbExpress, InterBase)을 참조하십시오.

- *ParamType* 속성은 선택된 매개변수의 타입을 나열합니다. 쿼리의 경우에는 입력 매개변수만 포함할 수 있기 때문에 항상 *ptInput*으로 초기화됩니다. *ParamType*의 값이 *ptUnknown*이면 *ptInput*으로 변경합니다.
- *Value* 속성은 선택한 매개변수의 값을 지정합니다. 애플리케이션이 런타임 시 매개변수 값을 제공하는 경우에는 *Value*를 공백으로 둘 수 있습니다.

Parameters 속성(*TParameter* 객체)을 사용할 때는 다음을 확인하거나 수정해야 합니다.

- *DataType* 속성은 매개변수 값의 데이터 타입을 나열합니다. 일부 데이터 타입에는 다음과 같은 추가 정보를 제공해야 합니다.
 - *NumericScale* 속성은 숫자 매개변수의 소수점 자릿수를 나타냅니다.
 - *Precision* 속성은 숫자 매개변수의 총 자릿수를 나타냅니다.
 - *Size* 속성은 문자열 매개변수의 문자 수를 나타냅니다.
- *Direction* 속성은 선택한 매개변수의 타입을 나열합니다. 쿼리에는 입력 매개변수만 포함할 수 있으므로 쿼리의 경우에는 항상 *pdInput*입니다.
- *Attributes* 속성은 매개변수가 받아들이는 값의 타입을 제어합니다. *Attributes*는 *psSigned*, *psNullable* 및 *psLong*의 조합으로 설정할 수도 있습니다.
- *Value* 속성은 선택한 매개변수의 값을 지정합니다. 애플리케이션이 런타임 시 매개변수 값을 제공하는 경우에는 *Value*를 공백으로 둘 수 있습니다.

런타임 시 매개변수 제공

런타임에 매개변수를 작성하기 위해 다음을 사용할 수 있습니다.

- 이름을 기반으로 매개변수에 값을 할당하는 *ParamByName* 메소드 (*TADOQuery*에는 사용할 수 없음)

- SQL 문에서의 매개변수 순서 위치를 기반으로 매개변수에 값을 할당하기 위한 *Params* 또는 *Parameters* 속성
- *Params.ParamValues* 또는 *Parameters.ParamValues* 속성은 설정된 각 매개변수 집합의 이름을 기반으로 단일 명령줄의 하나 이상의 매개변수에 대한 값을 할당합니다.

다음 코드에서는 *ParamByName*을 사용하여 *:Capital* 매개변수에 편집 상자의 텍스트를 할당합니다.

```
SQLQuery1.ParamByName('Capital').AsString := Edit1.Text;
```

인덱스를 0으로 하여 (*:Capital* 매개변수가 SQL 문의 첫 번째 매개변수라고 가정) *Params* 속성을 사용하는 동일한 코드를 다시 작성할 수 있습니다.

```
SQLQuery1.Params[0].AsString := Edit1.Text;
```

아래의 명령줄은 *Params.ParamValues* 속성을 사용하여 한 번에 세 개의 매개변수를 설정합니다.

```
Query1.Params.ParamValues['Name;Capital;Continent'] :=  
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

*ParamValues*가 값을 변환하지 않아도 되는 *Variants*를 사용한다는 점에 유의하십시오.

매개변수를 사용하여 마스터/디테일 관계 설정

디테일 셋이 쿼리 타입 데이터셋인 마스터/디테일 관계를 설정하려면 매개변수를 사용하는 쿼리를 지정해야 합니다. 이들 매개변수는 마스터 데이터셋의 현재 필드 값을 참조합니다. 마스터 데이터셋의 현재 필드 값은 런타임에 동적으로 변경되기 때문에 마스터 레코드가 변경될 때마다 디테일 셋의 매개변수를 다시 바인딩되어야 합니다. 이벤트 핸들러를 사용하여 이 작업을 수행하는 코드를 작성할 수 있지만 *TIBQuery*를 제외한 모든 쿼리 타입 데이터셋은 *DataSource* 속성을 사용하는 더 쉬운 메커니즘을 제공합니다.

매개변수화된 쿼리의 매개변수 값이 디자인 타임에 바인딩되지 않거나 런타임에 지정되지 않을 경우, 쿼리 타입 데이터셋은 *DataSource* 속성에 따라 이에 대한 값을 제공하려고 합니다. *DataSource*는 바인딩되지 않은 매개변수의 이름과 일치하는 필드 이름으로 검색된 다른 데이터셋을 식별합니다. 이 검색 데이터셋은 모든 데이터셋 타입일 수 있습니다. 검색 데이터셋은 이를 사용하는 디테일 데이터셋을 작성하기 전에 작성되고 채워져야 합니다. 검색 데이터셋에서 일치하는 항목을 찾으려면 디테일 데이터셋은 매개변수 값을 데이터 소스에 의해 지시된 현재 레코드의 필드 값에 연결합니다.

작동하는 방법을 보여 주기 위해 고객 테이블 및 주문 테이블의 두 테이블을 생각해 봅니다. 모든 고객에 대해 주문 테이블은 고객이 만드는 주문 집합을 포함합니다. 고객 테이블은 고유한 고객 ID를 지정하는 ID 필드를 포함합니다. 주문 테이블은 주문을 작성한 고객의 ID를 지정하는 *CustID* 필드를 포함합니다.

첫 번째 단계는 Customer 데이터셋을 설정하는 것입니다.

- 1 테이블 타입 데이터셋을 애플리케이션에 추가하고 Customer 테이블로 바인딩합니다.

- 이름이 *CustomerSource*인 *TDataSource*를 추가합니다. *DataSet* 속성을 1단계에서 추가한 데이터셋으로 설정합니다. 이 데이터 소스는 이제 Customer 데이터셋을 나타냅니다.
- 쿼리 타입 데이터셋을 추가하고 *SQL* 속성을 다음과 같이 설정합니다.

```
SELECT CustID, OrderNo, SaleDate
FROM Orders
WHERE CustID = :ID
```

매개변수 이름이 마스터(Customer) 테이블의 필드 이름과 동일하다는 것에 유의하십시오.

- 디테일 데이터셋의 *DataSource* 속성을 *CustomerSource*로 설정합니다. 이 속성을 설정하면 디테일 데이터셋이 연결된 쿼리로 됩니다.

런타임 시 디테일 데이터셋의 SQL 문에 있는 *:ID* 매개변수는 값이 할당되지 않습니다. 따라서 데이터셋은 *CustomersSource*로 식별되는 데이터셋의 열에 대하여 이름별로 매개변수를 일치시키려고 시도합니다. *CustomersSource*는 Customer 테이블에서 데이터를 차례로 파생하는 마스터 데이터셋에서 데이터를 가져옵니다. Customer 테이블이 "ID"라는 열을 포함하고 있기 때문에 마스터 데이터셋의 현재 레코드에 있는 *ID* 필드의 값은 디테일 데이터셋의 SQL 문의 *:ID* 매개변수에 할당됩니다. 데이터셋은 마스터/디테일 관계로 연결됩니다. 현재 레코드가 Customers 데이터셋에서 변경될 때마다 디테일 데이터셋의 SELECT 문은 현재 고객의 ID에 기반한 모든 주문을 검색하기 위해 실행됩니다.

쿼리 준비

쿼리 준비는 쿼리 실행에 선행하는 옵션 단계입니다. 쿼리 준비에서는 데이터 액세스 계층 및 데이터베이스 서버에 SQL 문과 매개변수를 제출하여 구문 분석하고, 리소스를 할당하며, 최적화합니다. 일부 데이터셋에서는 쿼리를 준비할 때 데이터셋이 추가로 설정 작업을 수행할 수도 있습니다. 이러한 작업은 쿼리 성능을 향상시키고 특히 업데이트할 수 있는 쿼리로 작업할 때 애플리케이션 속도가 더 빨라지게 합니다.

애플리케이션은 *Prepared* 속성을 *True*로 설정하여 쿼리를 준비할 수 있습니다. 쿼리를 실행하기 전에 준비하지 않으면 데이터셋은 *Open* 또는 *ExecSQL*을 호출할 때마다 자동으로 쿼리를 준비합니다. 데이터셋이 쿼리를 준비하는 경우에도 처음 데이터셋을 열기 전에 명시적으로 데이터셋을 준비하여 성능을 향상시킬 수 있습니다.

```
CustQuery.Prepared := True;
```

데이터셋을 명시적으로 준비하면 *Prepared*를 *False*로 설정하기 전에는 구문 실행을 위해 할당된 리소스가 해제되지 않습니다.

예를 들어, 매개변수를 추가한 경우 데이터셋이 실행되기 전에 다시 준비되어 있는지 확인하려면 *Prepared* 속성을 *False*로 설정합니다.

참고 쿼리에 대한 *SQL* 속성의 텍스트를 변경하면 데이터셋은 자동으로 쿼리를 닫고 쿼리를 준비하지 않습니다.

결과 집합(Result set)을 반환하지 않는 쿼리 실행

쿼리에서 SELECT 쿼리와 같은 레코드 집합을 반환하면 *Active*를 *True*로 설정하거나 *Open* 메소드를 호출하여 레코드로 모든 데이터셋을 채우는 것과 같은 방법으로 쿼리를 실행합니다.

하지만 SQL 명령은 종종 레코드를 반환하지 않습니다. 이러한 명령은 SELECT 문(예를 들면, INSERT, DELETE, UPDATE, CREATE INDEX 및 ALTER TABLE 명령은 어떤 레코드도 반환하지 않습니다.)이 아닌 DDL(Data Definition Language) 또는 DML(Data Manipulation Language) 문을 사용하는 문장을 포함합니다.

모든 쿼리 타입 데이터셋의 경우 *ExecSQL*을 호출하여 결과 집합을 반환하지 않는 쿼리를 실행할 수 있습니다.

```
CustomerQuery.ExecSQL; { query does not return a result set }
```

팁 쿼리를 여러 번 실행하려는 경우에는 *Prepared* 속성을 *True*로 설정하는 것이 좋습니다.

쿼리에서 레코드를 반환하지 않는 경우에도 DELETE 쿼리에서 삭제한 레코드 수처럼 쿼리의 영향을 받는 레코드 수를 알 수 있습니다. *RowsAffected* 속성은 *ExecSQL*의 호출에 영향을 받는 레코드 수를 제공합니다.

팁 사용자가 런타임 시에 쿼리를 동적으로 제공하는 경우처럼 쿼리가 결과 집합을 반환하는지 여부를 디자인 타임에 알 수 없으면 **try...except** 블록에서 두 가지 쿼리 실행 문을 코딩할 수 있습니다. *Open* 메소드를 호출하여 **try** 절에 둡니다. 실행 쿼리는 *Open* 메소드로 쿼리를 활성화할 때 실행되지만 예외도 발생합니다. 예외를 확인하고 단순하게 결과 집합이 부족하다는 메시지만 표시하는 경우에는 무시합니다. 예를 들어, *TQuery*는 이런 경우를 *ENoResultSet* 예외로 나타냅니다.

단방향 결과 집합 사용

쿼리 타입 데이터셋이 결과 집합을 반환하면 커서 또는 결과 집합의 첫 번째 레코드에 대한 포인터도 받습니다. 커서가 가리키는 레코드는 현재 활성 레코드입니다. 현재 레코드는 결과 집합의 데이터 소스와 연결된 data-aware 컴포넌트에 결과 값이 표시되는 레코드입니다. dbExpress를 사용하지 않으면 기본적으로 이 커서는 양방향입니다. 양방향 커서는 레코드를 앞뒤로 탐색할 수 있습니다. 양방향 커서를 지원하면 프로세싱 오버헤드가 추가될 수 있으며 일부 쿼리가 느려질 수 있습니다.

결과 집합을 뒤로 탐색할 필요가 없으면 *TQuery*와 *TIBQuery*로 단방향 커서를 요청하여 쿼리 성능을 향상시킬 수 있습니다. 단방향 커서를 요청하려면 *UniDirectional* 속성을 *True*로 설정합니다.

쿼리를 준비하고 실행하기 전에 *UniDirectional*을 설정합니다. 다음 코드는 쿼리를 준비하고 실행하기 전의 *UniDirectional* 설정을 보여 줍니다.

```
if not (CustomerQuery.Prepared) then
begin
    CustomerQuery.UniDirectional := True;
    CustomerQuery.Prepared := True;
end;
CustomerQuery.Open; { returns a result set with a one-way cursor }
```

참고 *UniDirectional* 속성을 단방향 데이터셋과 혼동하지 마십시오. 단방향 데이터셋 (*TSQLDataSet*, *TSQLTable*, *TSQLQuery* 및 *TSQLStoredProc*)은 dbExpress를 사용하는데 이는 단방향 커서만 반환합니다. 단방향 데이터셋은 뒤로 탐색하는 기능을 제한할 뿐만 아니라 레코드를 버퍼링하지 않으므로 필터를 사용할 수 없는 기능과 같은 추가적인 제한이 있습니다.

내장 프로시저 타입 데이터셋 사용

애플리케이션이 내장 프로시저를 사용하는 방법은 내장 프로시저가 코딩된 방법, 데이터를 반환하는지 여부와 반환하는 방법, 사용되는 특정 데이터베이스 서버 또는 이러한 요소의 조합에 따라 달라집니다.

일반적으로 서버의 내장 프로시저를 액세스하려면 애플리케이션은 다음을 수행해야 합니다.

- 1 적절한 데이터셋 컴포넌트를 데이터 모듈이나 폼에 두고 그 컴포넌트의 *Name* 속성을 사용자의 애플리케이션에 적합한 고유한 값으로 설정합니다.
- 2 내장 프로시저를 정의하는 데이터베이스 서버를 식별합니다. 각 내장 프로시저 타입 데이터셋은 이 작업을 다르게 수행하지만 일반적으로 데이터베이스 연결 컴포넌트를 지정합니다.
 - *TStoredProc*의 경우에는 *DatabaseName* 속성을 사용하여 *TDatabase* 컴포넌트 또는 BDE 알리아스를 지정합니다.
 - *TADOStoredProc*의 경우에는 *Connection* 속성을 사용하여 *TADOConnection* 컴포넌트를 지정합니다.
 - *TSQLStoredProc*의 경우에는 *SQLConnection* 속성을 사용하여 *TSQLConnection* 컴포넌트를 지정합니다.
 - *TIBStoredProc*의 경우에는 *Database* 속성을 사용하여 *TIBConnection* 컴포넌트를 지정합니다.

데이터베이스 연결 컴포넌트 사용에 대한 내용은 17장 "데이터베이스에 연결"을 참조하십시오.

- 3 내장 프로시저를 지정하여 실행합니다. 대부분의 내장 프로시저 타입 데이터셋의 경우에는 *StoredProcName* 속성을 설정하여 이 작업을 수행합니다. *ProcedureName* 속성을 가진 *TADOStoredProc*는 예외입니다.
- 4 내장 프로시저가 비주어 데이터 컨트롤과 함께 사용할 커서를 반환하면 데이터 소스 컴포넌트를 데이터 모듈에 추가하고 *DataSet* 속성을 내장 프로시저 타입 데이터셋으로 설정합니다. *DataSource* 및 *DataField* 속성을 사용하여 data-aware 컴포넌트를 데이터 소스에 연결합니다.
- 5 필요하면 내장 프로시저에 입력 매개변수 값을 제공합니다. 서버가 모든 내장 프로시저 매개변수에 대한 정보를 제공하지 않으면 매개변수 이름 및 데이터 타입과 같은 입력 매개변수 정보를 추가로 제공해야 할 수 있습니다. 내장 프로시저 작업에 대한 자세한 내용은 18-51 페이지의 "내장 프로시저 매개변수 작업"을 참조하십시오.

- 6 내장 프로시저를 실행합니다. 커서를 반환하는 내장 프로시저의 경우에는 *Active* 속성이나 *Open* 메소드를 사용합니다. 결과를 반환하지 않거나 출력 매개변수만 반환하는 내장 프로시저를 실행하려면 런타임 시 *ExecProc* 메소드를 사용합니다. 내장 프로시저를 한 번 이상 실행하려는 경우에는 *Prepare*를 호출하여 데이터 액세스 계층을 초기화하고 매개변수 값을 내장 프로시저로 바인딩하려고 할 수 있습니다. 쿼리 준비에 대한 내용은 18-54 페이지의 "결과 집합 (Result set)을 반환하지 않는 내장 프로시저 실행"을 참조하십시오.
- 7 결과를 처리합니다. 이러한 결과는 결과 및 출력 매개변수로 반환되거나 내장 프로시저 타입 데이터셋을 채우는 결과 집합으로 반환될 수 있습니다. 일부 내장 프로시저는 여러 커서를 반환합니다. 추가 커서를 액세스하는 방법에 대한 자세한 내용은 18-54 페이지의 "여러 결과 집합 페치"를 참조하십시오.

내장 프로시저 매개변수 작업

내장 프로시저와 연결될 수 있는 매개변수 타입에는 네 가지가 있습니다.

- **입력 매개변수**는 처리를 위해 내장 프로시저에 값을 전달할 때 사용됩니다.
- **출력 매개변수**는 내장 프로시저에서 애플리케이션에 반환 값을 전달하기 위해 사용됩니다.
- **입/출력 매개변수**는 처리를 위해 내장 매개변수에 값을 전달할 때 사용되며, 내장 프로시저에서 애플리케이션에 반환 값을 전달하기 위해 사용됩니다.
- **결과 매개변수**는 일부 내장 프로시저에서 애플리케이션에 오류나 상태 값을 반환하기 위해 사용됩니다. 내장 프로시저는 하나의 결과 매개변수만 반환할 수 있습니다.

내장 프로시저가 특정 타입의 매개변수를 사용할지는 데이터베이스 서버에 있는 내장 프로시저의 일반적인 랭귀지 구현과 내장 프로시저의 특정 인스턴스에 따라 달라집니다. 서버에서는 각각의 내장 프로시저가 입력 매개변수를 사용할 수도 있고 사용하지 않을 수도 있습니다. 반면 일부 매개변수는 서버마다 다르게 사용됩니다. 예를 들어, MS-SQL Server 및 Sybase 내장 프로시저는 항상 결과 매개변수를 반환하지만 내장 프로시저의 InterBase 구현은 결과 매개변수를 반환하지 않습니다.

내장 프로시저 매개변수에 대한 액세스는 *TStoredProc*, *TSQLStoredProc*, *TIBStoredProc*의 *Params* 속성이나 *TADOStoredProc*의 *Parameters* 속성으로 제공됩니다. *StoredProcName*이나 *ProcedureName* 속성에 값을 할당하면 데이터셋은 내장 프로시저의 각 매개변수에 객체를 자동으로 생성합니다. 일부 데이터셋의 경우 런타임까지 내장 프로시저 이름을 지정하지 않으면 각 매개변수의 객체가 런타임 시 프로그램에서 작성됩니다. 내장 프로시저를 지정하지 않고 수동으로 *TParam* 또는 *TParameter* 객체를 작성하면 단일 데이터셋은 사용 가능한 모든 내장 프로시저를 사용할 수 있습니다.

참고 일부 내장 프로시저는 출력 및 결과 매개변수뿐만 아니라 데이터셋을 반환합니다. 애플리케이션이 data-aware 컨트롤에 데이터셋 레코드를 표시할 수 있지만 출력 및 결과 매개변수를 별도로 처리해야 합니다.

디자인 타임 시 매개변수 설정

매개변수 컬렉션 에디터를 사용하여 디자인 타임에 내장 프로시저 매개변수 값을 지정할 수 있습니다. 매개변수 컬렉션 에디터를 표시하려면 Object Inspector에서 *Params* 또는 *Parameters* 속성의 생략 버튼을 클릭합니다.

중요 매개변수 컬렉션 에디터에서 값을 선택하고 Object Inspector를 사용하여 *Value* 속성을 설정하여 입력 매개변수에 값을 할당할 수 있습니다. 그러나 서버에서 보고된 입력 매개변수에 대한 이름이나 데이터 타입은 변경하지 마십시오. 이를 변경할 경우 내장 프로시저를 실행할 때 예외가 발생할 수 있습니다.

일부 서버는 매개변수 이름 또는 데이터 타입을 보고하지 않습니다. 이러한 경우에는 매개변수 컬렉션 에디터를 사용하여 매개변수를 직접 설정해야 합니다. 오른쪽 버튼을 클릭하고 Add를 선택하여 매개변수를 추가합니다. 추가한 각 매개변수에 대해서는 완벽하게 설명해야 합니다. 매개변수를 추가할 필요가 없는 경우에도 개별 매개변수 객체의 속성을 검토하여 올바른지 확인해야 합니다.

데이터셋에 *Params* 속성 (*TParam* 객체)이 있으면 다음 속성을 올바르게 지정해야 합니다.

- *Name* 속성은 내장 프로시저에 의해 정의된 매개변수의 이름을 나타냅니다.
- *DataType* 속성은 매개변수 값의 데이터 타입을 제공합니다. *TSQLStoredProc* 사용 시 일부 데이터 타입에는 추가 정보가 필요합니다.
 - *NumericScale* 속성은 숫자 매개변수의 소수점 자릿수를 나타냅니다.
 - *Precision* 속성은 숫자 매개변수의 총 자릿수를 나타냅니다.
 - *Size* 속성은 문자열 매개변수의 문자 수를 나타냅니다.
- *ParamType* 속성은 선택된 매개변수의 타입을 나열합니다. 입력 매개변수의 경우에는 *ptInput*, 출력 매개변수의 경우에는 *ptOutput*, 입/출력 매개변수의 경우에는 *ptInputOutput*, 결과 매개변수의 경우에는 *ptResult*가 될 수 있습니다.
- *Value* 속성은 선택된 매개변수에 대한 값을 지정합니다. 출력 매개변수와 결과 매개변수에 대한 값은 설정할 수 없습니다. 이러한 타입의 매개변수는 내장 프로시저 실행에 의해 설정된 값을 가집니다. 입력 및 입/출력 매개변수에서 애플리케이션이 런타임 시 매개변수 값을 제공하는 경우 *Value*를 공백으로 둘 수 있습니다.

데이터셋이 *Parameters* 속성 (*TParameter* 객체)을 사용하면 다음 속성을 올바르게 지정해야 합니다.

- *Name* 속성은 내장 프로시저에 의해 정의된 매개변수의 이름을 나타냅니다.
- *DataType* 속성은 매개변수 값의 데이터 타입을 제공합니다. 일부 데이터 타입에는 다음과 같은 추가 정보를 제공해야 합니다.
 - *NumericScale* 속성은 숫자 매개변수의 소수점 자릿수를 나타냅니다.
 - *Precision* 속성은 숫자 매개변수의 총 자릿수를 나타냅니다.
 - *Size* 속성은 문자열 매개변수의 문자 수를 나타냅니다.

- *Direction* 속성은 선택한 매개변수의 타입을 제공합니다. 입력 매개변수의 경우에는 *pdInput*, 출력 매개변수의 경우에는 *pdOutput*, 입/출력 매개변수의 경우에는 *pdInputOutput*, 결과 매개변수의 경우에는 *pdReturnValue*가 될 수 있습니다.
- *Attributes* 속성은 매개변수가 받아 들이는 값의 타입을 제어합니다. *Attributes*는 *psSigned*, *psNullable* 및 *psLong*의 조합으로 설정할 수도 있습니다.
- *Value* 속성은 선택된 매개변수에 대한 값을 지정합니다. 출력 및 결과 매개변수의 값은 설정하지 마십시오. 입력 및 입/출력 매개변수에서 애플리케이션이 런타임 시 매개변수 값을 제공하는 경우 *Value*를 공백으로 둘 수 있습니다.

런타임 시 매개변수 사용

일부 데이터셋을 사용하는 경우 런타임까지 내장 프로시저 이름을 지정하지 않으면 매개변수의 *TParam* 객체는 자동으로 만들어지지 않으므로 프로그램에서 만들어야 합니다. 이 작업은 *TParam.Create* 메소드 또는 *TParams.AddParam* 메소드를 사용하여 수행할 수 있습니다.

```
var
    P1, P2:Tparam;
begin
    ...
    with StoredProc1 do begin
        StoredProcName := 'GET_EMP_PROJ';
        Params.Clear;
        P1 := TParam.Create(Params, ptInput);
        P2 := TParam.Create(Params, ptOutput);
        try
            Params[0].Name := ?MP_NO?
            Params[1].Name := ?ROJ_ID?
            ParamByName(êEMP_NOí).AsSmallInt := 52;
            ExecProc;
            Edit1.Text := ParamByName(êPROJ_IDí).AsString;
        finally
            P1.Free;
            P2.Free;
        end;
    end;
    ...
end;
```

런타임 시 개별 매개변수 객체를 추가할 필요가 없는 경우에는 개별 매개변수 객체를 액세스하여 입력 매개변수에 값을 할당하고 출력 매개변수에서 값을 가져올 수 있습니다. 데이터셋의 *ParamByName* 메소드를 사용하면 이름을 기반으로 하는 개별 매개변수에 액세스할 수 있습니다. 예를 들어, 다음 코드는 입/출력 매개변수의 값을 설정하고, 내장 프로시저를 실행하고, 반환된 값을 검색합니다.

```
with SQLStoredProc1 do
begin
    ParamByName('IN_OUTVAR').AsInteger := 103;
    ExecProc;
    IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

내장 프로시저 준비

쿼리 타입 데이터셋을 사용할 때와 마찬가지로 내장 프로시저를 실행하기 전에 내장 프로시저 타입 데이터셋을 준비해야 합니다. 내장 프로시저 준비에서는 내장 프로시저의 리소스를 할당하고 매개변수를 바인딩한다는 사실을 데이터 액세스 계층 및 데이터베이스 서버에 알려 줍니다. 이 작업을 통해 성능이 향상될 수 있습니다.

내장 프로시저를 준비하기 전에 실행하려고 하면 데이터셋은 자동으로 내장 프로시저를 준비한 다음 실행한 후에는 준비하지 않습니다. 내장 프로시저를 여러 번 실행하려면 Prepared 속성을 *True*로 설정하여 명시적으로 준비하는 것이 더 효율적입니다.

```
MyProc.Prepared := True;
```

데이터셋을 명시적으로 준비하면 *Prepared*를 *False*로 설정하기 전에는 내장 프로시저 실행을 위해 할당된 리소스가 해제되지 않습니다.

데이터셋이 실행되기 전에 재준비되는지 확인하려면(예를 들어, Oracle 오버로드된 프로시저 사용 시 매개변수 값을 변경하는 경우) *Prepared* 속성을 *False*로 설정합니다.

결과 집합(Result set)을 반환하지 않는 내장 프로시저 실행

내장 프로시저가 커서를 반환하면 *Active*를 *True*로 설정하거나 *Open* 메소드를 호출하여 레코드로 데이터셋을 채우는 것과 같은 방법으로 내장 프로시저를 실행합니다.

그러나 내장 프로시저는 종종 데이터를 반환하지 않거나 출력 매개변수에 결과만 반환합니다. *ExecProc*를 호출하면 결과 집합을 반환하지 않는 내장 프로시저를 실행할 수 있습니다. 내장 프로시저를 실행한 후에는 *ParamByName* 메소드를 사용하여 결과 매개변수나 출력 매개변수의 값을 읽을 수 있습니다.

```
MyStoredProcedure.ExecProc; { does not return a result set }
Edit1.Text := MyStoredProcedure.ParamByName('OUTVAR').AsString;
```

참고 *TADOStoredProc*에는 *ParamByName* 메소드가 없습니다. ADO를 사용할 때 출력 매개변수 값을 가져오려면 *Parameters* 속성을 사용하여 매개변수 객체에 액세스합니다.

팁 프로시저를 여러 번 실행 중인 경우에는 *Prepared* 속성을 *True*로 설정하는 것이 좋습니다.

여러 결과 집합 폐치

일부 내장 프로시저는 레코드의 여러 집합을 반환합니다. 데이터셋을 열 때 첫 번째 집합만 폐치합니다. *TSQLStoredProc* 또는 *TADOStoredProc*를 사용하는 경우 *NextRecordSet* 메소드를 사용하여 다른 레코드 집합에 액세스할 수 있습니다.

```
var
  DataSet2:TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```


*TSQLStoredProc*에서 *NextRecordSet*은 다음 레코드 집합에 대한 액세스를 제공하는 새로 생성된 *TCustomSQLDataSet* 컴포넌트를 반환합니다.

*TADOStoredProc*에서 *NextRecordset*은 기존 ADO 데이터셋의 *RecordSet* 속성에 할당할 수 있는 인터페이스를 반환합니다. 두 클래스에 대해 메소드는 반환된 데이터셋의 레코드 수를 출력 매개변수로 반환합니다.

*NextRecordSet*을 처음 호출하면 두 번째 레코드 집합이 반환됩니다. *NextRecordSet*을 다시 호출하면 세 번째 데이터셋을 반환하며 더 이상 남은 레코드 집합이 없을 때까지 반복됩니다. 더 이상 데이터셋이 없으면 *NextRecordSet*은 **nil**을 반환합니다.

19

필드 컴포넌트 사용

이 장은 *TField* 객체와 그 자손에 공통적인 속성, 이벤트 및 메소드를 설명합니다. 필드 컴포넌트는 데이터셋의 개별 필드(열)를 나타냅니다. 이 장은 또한 애플리케이션의 데이터 표시 및 편집을 조정하기 위해 필드 컴포넌트를 사용하는 방법을 설명합니다.

필드 컴포넌트는 항상 데이터셋과 연결되어 있습니다. *TField* 객체는 사용자의 애플리케이션에서 직접 사용되지 않습니다. 그 대신 사용자 애플리케이션의 각 필드 컴포넌트는 데이터셋의 열 데이터 타입별 *TField* 자손입니다. 필드 컴포넌트는 연결된 데이터셋의 특정 열의 데이터에 대한 *TDBEdit* 및 *TDBGrid* 액세스와 같이 data-aware 컨트롤을 제공합니다.

일반적으로 단일 필드 컴포넌트는 데이터 타입 및 크기와 같은 데이터셋의 단일 열이나 필드의 특성을 나타냅니다. 또한 정렬, 표시 형식 및 편집 형식과 같은 필드의 표시 특성을 나타냅니다. 예를 들어, *TFloatField*에는 데이터의 모습에 직접적으로 영향을 주는 속성이 네 가지 있습니다.

표 19.1 데이터 표시에 영향을 주는 *TFloatField* 속성

속성	용도
Alignment	왼쪽, 가운데 또는 오른쪽 정렬의 데이터 표시 방법을 지정합니다.
DisplayWidth	한 번에 컨트롤에 표시할 수 있는 숫자의 수를 지정합니다.
DisplayFormat	소수점 이하 자릿수를 얼마나 표시할 것인가와 같은 표시할 데이터 서식을 지정합니다.
EditFormat	편집 중에 값을 표시하는 방법을 지정합니다.

데이터셋의 레코드에서 레코드로 스크롤할 때 필드 컴포넌트를 사용하면 현재 레코드의 필드 값을 보거나 변경할 수 있습니다.

필드 컴포넌트는 서로 공통적인 많은 속성(예를 들어, *DisplayWidth* 및 *Alignment*)을 갖고 있으며, 데이터 타입별 속성(예를 들어, *TFloatField*의 *Precision*)도 갖고 있습니다. 이들 각 속성은 데이터가 폼에서 애플리케이션의 사용자에게 표시되는 방식에 영향을 줍니다. *Precision*과 같은 일부 속성은 또한 데이터 수정이나 입력 시 사용자가 컨트롤에 입력할 수 있는 데이터 값에도 영향을 줍니다.

데이터셋의 모든 필드 컴포넌트는 동적(원본으로 사용한 데이터베이스 테이블의 구조에 따라 자동으로 생성)이거나 영구적(Fields Editor에서 설정하는 특정 필드 이름과 속성으로 생성)입니다. 동적 필드와 영구적 필드는 각기 다른 장점을 갖고 있으며, 각기 다른 애플리케이션 타입에 적합합니다. 다음 단원에서는 동적 필드와 영구적 필드를 좀 더 자세히 설명하고 이들의 선택에 대한 조언을 제공합니다.

동적(Dynamic) 필드 컴포넌트

동적으로 생성된 필드 컴포넌트는 기본값입니다. 실제로 데이터셋의 모든 필드 컴포넌트는 데이터 모듈에 데이터셋을 처음 둘 때 동적 필드로 시작되고, 데이터셋이 데이터를 폐치하는 방법을 지정하며, 데이터셋을 엽니다. 데이터셋에서 나타나는 원본으로 사용한 데이터의 실제 특성에 따라 자동으로 생성되는 필드 컴포넌트는 동적 필드 컴포넌트입니다. 데이터셋은 원본으로 사용한 데이터의 각 열에 대해 하나의 필드 컴포넌트를 작성합니다. 각 열에 대해 생성된 정확한 *TField* 자손은 *TClientDataSet*에 대한 데이터베이스나 프로바이더 컴포넌트에서 받은 필드 타입 정보에 의해 결정됩니다.

동적 필드는 일시적입니다. 동적 필드는 데이터셋이 열려 있는 동안에만 존재합니다. 데이터셋은 데이터셋을 원본으로 하는 데이터의 현재 구조에 따라 동적 필드를 사용하는 데이터셋을 다시 열 때마다 완전히 새로운 동적 필드 컴포넌트 집합을 다시 만듭니다. 원본으로 사용한 데이터의 열이 변경되고 동적 필드 컴포넌트를 사용하는 데이터셋을 연 경우, 자동으로 생성되는 필드 컴포넌트는 또한 이를 반영하기 위해 변경됩니다.

데이터 표시 및 편집에 대해 유동적이어야 하는 애플리케이션에서 동적 필드를 사용합니다. 예를 들어, SQL explorer와 같은 데이터베이스 찾아보기 툴을 만들려면 모든 데이터베이스 테이블은 열의 숫자와 타입이 다르기 때문에 동적 필드를 사용해야 합니다. 또한 사용자의 데이터와의 상호 작용이 대부분 그리드 컴포넌트 내에서 발생하고 애플리케이션에서 사용하는 데이터셋이 자주 변경되는 애플리케이션의 경우에도 동적 필드를 사용합니다.

애플리케이션에서 동적 필드를 사용하려면 다음과 같이 합니다.

- 1 데이터셋과 데이터 소스를 데이터 모듈에 둡니다.
- 2 데이터셋을 데이터에 연결합니다. 이렇게 하면 데이터 소스에 연결하기 위해 연결 컴포넌트나 프로바이더를 사용할 수 있으며 데이터셋이 나타내는 데이터를 지정하는 속성을 설정할 수 있습니다.
- 3 데이터 소스를 데이터셋에 연결합니다.
- 4 data-aware 컨트롤을 애플리케이션의 폼에 두고, 데이터 모듈을 각 폼의 유닛에 대한 각각의 사용 절에 추가하고, 각 data-aware 컨트롤을 모듈의 데이터 소스에 연결합니다. 또한 이를 필요로 하는 각각의 data-aware 컨트롤에 필드를 연결합니다. 동적 필드 컴포넌트를 사용하기 때문에 데이터셋을 열 때 지정한 필드 이름이 계속 남아 있지 않을 수도 있습니다.
- 5 데이터셋을 엽니다.

동적 필드는 사용이 간편한 반면 제한적으로 사용될 수 있습니다. 코드를 작성하지 않으면 동적 필드에 대한 표시 및 편집 기본값을 변경할 수 없고, 동적 필드가 표시되는 순서를 안전하게 변경할 수 없으며, 데이터셋의 모든 필드에 대한 액세스를 방지할 수 없습니다. 계산된 필드나 조회 필드와 같이 데이터셋에 대한 추가 필드를 생성할 수 없고 동적 필드의 기본 데이터 타입을 오버라이드할 수 없습니다. 데이터베이스 애플리케이션에서 필드에 대한 조정 및 유동성을 얻으려면 Fields Editor를 불러서 데이터셋에 대한 영구적 필드 컴포넌트를 생성해야 합니다.

영구적(Persistent) 필드 컴포넌트

기본적으로 데이터셋 필드는 동적입니다. 필드의 속성과 사용 가능성은 자동으로 설정되고 어떠한 방식으로든 변경될 수 없습니다. 필드의 속성과 이벤트를 제어하려면 데이터셋에 대한 영구적 필드를 생성해야 합니다. 영구적 필드를 사용하면 다음 작업을 수행할 수 있습니다.

- 필드의 표시를 설정 또는 변경하거나 디자인 타임이나 런타임 시 특성을 편집합니다.
- 조회 필드, 계산된 필드 및 집계 필드와 같은 새 필드를 생성할 때 데이터셋의 기존 필드의 값을 기준으로 합니다.
- 데이터 입력을 검증합니다.
- 영구적 컴포넌트 목록에서 필드 컴포넌트를 제거하여 애플리케이션 원본으로 사용하는 데이터베이스의 특정 열에 액세스하지 못하도록 합니다.
- 새 필드를 정의하여 데이터셋을 원본으로 사용하는 테이블이나 쿼리의 열에 따라 기존 필드를 교체합니다.

디자인 타임에 Fields Editor를 사용하여 애플리케이션의 데이터셋에서 사용하는 필드 컴포넌트의 영구적 목록을 만들 수 있습니다. 영구적 필드 컴포넌트 목록은 애플리케이션에 저장되는데 원본으로 사용한 데이터셋의 데이터베이스 구조가 변경되더라도 변경되지 않습니다. 일단 Fields Editor로 영구적 필드를 생성하면 데이터 값의 변경에 응답하고 데이터 입력을 검증하는 이벤트 핸들러도 생성할 수 있습니다.

참고 데이터셋에 대한 영구적 필드를 생성할 때 사용자가 선택하는 필드만 디자인 타임 및 런타임 시 사용자 애플리케이션에서 사용될 수 있습니다. 디자인 타임에 항상 Fields Editor를 사용하여 데이터셋에 대한 영구적 필드를 추가하거나 삭제할 수 있습니다.

단일 데이터셋에서 사용하는 모든 필드는 영구적이거나 동적입니다. 단일 데이터셋의 필드 타입들을 혼합할 수 없습니다. 데이터셋에 대한 영구적 필드를 작성한 다음 동적 필드로 변환하려면 데이터셋에서 모든 영구적 필드를 제거해야 합니다. 동적 필드에 대한 자세한 내용은 19-2 페이지의 "동적(Dynamic) 필드 컴포넌트"를 참조하십시오.

참고 영구적 필드의 일차적인 용도 중 하나는 데이터의 모습과 표시를 조정하는 것입니다. 또한 data-aware 그리드의 열 모습을 조정할 수도 있습니다. 그리드의 열 모양 조정에 대한 내용은 15-17 페이지의 "사용자 지정 그리드 생성"을 참조하십시오.

영구적 필드 생성

Fields Editor로 생성된 영구적 필드 컴포넌트는 원본으로 사용한 데이터에 대해 효과적이고 읽기 가능하며 타입을 고려하지 않아도 되는(type-safe) 프로그램 액세스를 제공합니다. 영구적 필드 컴포넌트를 사용하면 애플리케이션이 실행될 때마다 원본으로 사용한 데이터의 실제 구조가 변경되더라도 항상 동일한 순서로 동일한 열이 사용되고 표시됩니다. 특정 필드에 따라 달라지는 data-aware 컴포넌트와 프로그램 코드는 항상 의도하는 대로 작동합니다. 영구적 필드 컴포넌트를 기반으로 하는 열이 삭제되거나 변경되면 Delphi는 존재하지 않는 열이나 일치하지 않는 데이터에 대해 애플리케이션을 실행하지 않고 예외를 발생시킵니다.

다음과 같은 방법으로 데이터셋에 대한 영구적 필드를 생성합니다.

- 1 데이터 모듈에 데이터셋을 둡니다.
- 2 데이터셋을 원본으로 사용한 데이터에 결합시킵니다. 이렇게 하면 일반적으로 데이터셋이 연결 컴포넌트나 프로바이더에 연결되고 데이터를 설명하는 모든 속성이 지정됩니다. 예를 들어, *TADODataSet*을 사용하는 경우에는 적절히 구성된 *TADOConnection* 컴포넌트에 대한 *Connection* 속성을 설정하고, 유효한 쿼리에 대한 *CommandText* 속성을 설정할 수 있습니다.
- 3 데이터 모듈의 데이터셋 컴포넌트를 더블 클릭하면 Fields Editor를 불러옵니다. Fields Editor에는 제목 표시줄, 탐색기 버튼 및 리스트 박스가 있습니다.

Fields Editor의 제목 표시줄은 데이터 모듈 또는 데이터셋을 포함하는 폼의 이름과 데이터셋의 이름을 직접 표시합니다. 예를 들어, *CustomerData* 데이터 모듈의 *Customers* 데이터셋을 열면 제목 표시줄은 'CustomerData.Customers' 또는 적합한 만큼의 이름을 표시합니다.

제목 표시줄 아래에는 디자인 타임에 활성 데이터셋의 레코드를 하나씩 스크롤하고 첫 번째나 마지막 레코드로 이동할 수 있도록 하는 탐색 버튼 집합이 있습니다. 데이터셋이 활성이 아니거나 비어 있으면 탐색 버튼이 흐리게 표시됩니다. 데이터셋이 단방향인 경우에는 마지막 레코드와 이전 레코드로 이동하는 버튼이 항상 흐리게 표시됩니다.

리스트 박스에는 데이터셋에 대한 영구적 필드 컴포넌트의 이름을 표시합니다. 새 데이터셋에 대해 Fields Editor를 처음 불러오면 데이터셋의 필드 컴포넌트가 영구적이지 않고 동적이기 때문에 목록이 비어 있습니다. 이미 영구적 필드 컴포넌트가 있는 데이터셋에 대해 Fields Editor를 실행하면 리스트 박스에 필드 컴포넌트 이름이 표시됩니다.

- 4 Fields Editor 컨텍스트 메뉴에서 Add Fields를 선택합니다.
- 5 Add Fields 대화 상자에서 영구적으로 만들 필드를 선택합니다. 기본적으로 대화 상자가 열릴 때 모든 필드가 선택됩니다. 사용자가 선택하는 모든 필드는 영구적 필드가 됩니다.

Add Fields 대화 상자가 닫히면 사용자가 선택한 필드가 Fields Editor 리스트 박스에 나타납니다. Fields Editor 리스트 박스의 필드는 영구적입니다. 또한 데이터셋이 활성 상태이면 리스트 박스 위의 다음 및 마지막 탐색 버튼(데이터셋이 단방향이 아닌 경우)을 사용할 수 있습니다.

그러면 이제부터는 데이터셋을 열 때마다 원본으로 사용한 데이터베이스의 모든 열에 대한 동적 필드 컴포넌트를 더 이상 생성하지 않습니다. 그 대신 사용자가 지정한 필드에 대한 영구적 컴포넌트만 생성합니다.

데이터셋을 열 때마다 각각의 계산되지 않은 영구적 필드가 데이터베이스에 있거나 데이터베이스의 데이터에서 생성될 수 있는지 검증합니다. 그렇지 않으면 데이터셋은 필드가 유효하지 않아 열리지 않는다는 것을 사용자에게 경고하는 예외를 발생시킵니다.

영구적 필드 정렬

영구적 필드 컴포넌트가 Fields Editor 리스트 박스에 나열되는 순서는 필드가 data-aware 그리드 컴포넌트에 나타나는 기본 순서입니다. 리스트 박스에서 필드를 끌어다 놓으면 필드 순서를 변경할 수 있습니다.

다음과 같은 방법으로 필드 순서를 변경합니다.

- 1 필드를 선택합니다. 한 번에 하나 이상의 필드를 선택하고 순서를 바꿀 수 있습니다.
- 2 필드를 새 위치로 끌어 놓습니다.

여러 비연속적인 필드 집합을 선택하여 새로운 위치로 끌어가면 연속적인 블록으로 삽입됩니다. 이 블록 내에서 필드의 순서는 변경되지 않습니다.

다른 방법으로는 필드를 선택하고 *Ctrl+Up* 및 *Ctrl+Dn*을 사용하여 목록에서 개별 필드의 순서를 변경할 수 있습니다.

새 영구적 필드 정의

기존 데이터셋 필드를 영구적 필드로 만드는 것 외에 데이터셋의 다른 영구적 필드에 대한 교체 또는 추가로 특별한 영구적 필드를 생성할 수도 있습니다.

사용자가 생성하는 새 영구적 필드는 표시 목적으로만 사용합니다. 이 필드가 런타임 시 포함하는 데이터는 이미 데이터베이스의 다른 곳에 있거나 임시적이기 때문에 유지되지 않습니다. 데이터셋의 원본으로 사용한 데이터의 실제 구조는 어떤 방식으로든 변경되지 않습니다.

새 영구적 필드 컴포넌트를 작성하려면 Fields Editor에서 컨텍스트 메뉴를 불러서 New field를 선택합니다. New Field 대화 상자가 나타납니다.

New Field 대화 상자에는 세 개의 그룹 상자, Field properties, Field type 및 Lookup definition이 있습니다.

- Field properties 그룹 상자를 사용하면 일반적인 필드 컴포넌트 정보를 입력할 수 있습니다. Name 편집 상자에 필드 이름을 입력합니다. 여기서 입력하는 이름은 필드 컴포넌트의 *FieldName* 속성과 일치합니다. New Field 대화 상자는 이 이름을 사용하여 Component 편집 상자의 컴포넌트 이름을 만듭니다. Component 편집 상자에 표시되는 이름은 필드 컴포넌트의 *Name* 속성과 일치하고 정보 목적을 위해서만 제공됩니다 (*Name*은 사용자 소스 코드의 필드 컴포넌트를 참조하는 식별자입니다). 대화 상자는 Component 편집 상자에 직접 입력하는 모든 내용을 버립니다.

- Field properties 그룹의 Type 콤보 박스를 사용하면 필드 컴포넌트의 데이터 타입을 지정할 수 있습니다. 생성하는 모든 새로운 필드 컴포넌트에 대해 데이터 타입을 제공해야 합니다. 예를 들어, 필드에 부동 소수점 화폐 값을 표시하려면 드롭다운 목록에서 *Currency*를 선택합니다. Size 편집 상자를 사용하여 문자열 기반 필드에 표시 또는 입력될 수 있는 문자의 최대 수나 *Bytes* 및 *VarBytes* 필드의 크기를 지정합니다. 다른 모든 데이터 타입에서 Size는 무의미합니다.
- Field type 라디오 그룹을 사용하면 작성할 새로운 필드 컴포넌트의 타입을 지정할 수 있습니다. 기본 타입은 Data입니다. Lookup을 선택하면 Lookup definition 그룹 상자의 Dataset 및 Source Fields 편집 상자를 사용할 수 있습니다. 또한 Calculated 필드를 작성할 수 있으며 클라이언트 데이터셋에서 작업하는 경우에는 InternalCalc 필드나 Aggregate 필드를 작성할 수 있습니다. 다음 표는 작성할 수 있는 필드 타입을 설명한 것입니다.

표 19.2 특수한 영구적 필드 종류

필드 종류	용도
Data	기존의 필드를 대체합니다(예를 들어, 데이터 타입 변경).
Calculated	런타임 시 데이터셋의 <i>OnCalcFields</i> 이벤트 핸들러로 계산된 값을 표시합니다.
Lookup	런타임 시 사용자가 지정한 검색 기준에 따라 지정된 데이터셋에서 값을 검색합니다. (단방향 데이터셋에서는 지원되지 않음)
InternalCalc	런타임 시 클라이언트 데이터셋으로 계산되고 데이터가 저장된 값을 표시합니다.
Aggregate	클라이언트 데이터셋에서 레코드 집합의 데이터를 요약하는 값을 표시합니다.

Lookup definition 그룹 상자는 조회 필드를 작성하기 위해서만 사용됩니다. 보다 자세한 설명은 19-8 페이지의 "조회 필드 정의"를 참조합니다.

데이터 필드 정의

데이터 필드는 데이터셋의 기존 필드를 교체합니다. 예를 들어, 프로그램적인 이유로 인해 *TSmallIntField*를 *TIntegerField*로 교체해야 할 수도 있습니다. 필드의 데이터 타입은 직접 변경할 수 없기 때문에 이를 교체할 새로운 필드를 정의해야 합니다.

중요 기존의 필드를 교체할 새로운 필드를 정의하는 경우에도 정의하는 필드는 원본으로 사용한 데이터셋의 테이블에 있는 기존의 열에서 데이터 값을 파생해야 합니다.

원본으로 사용한 데이터셋의 테이블에 있는 필드에 대한 교체 데이터 필드를 작성하려면 다음 단계를 따르십시오.

- 1 데이터셋에 할당된 영구적 필드 목록에서 필드를 제거하고 컨텍스트 메뉴에서 New Field를 선택합니다.
- 2 New Field 대화 상자에서 Name 편집 상자의 데이터베이스 테이블에 있는 기존 필드의 이름을 입력합니다. 새로운 필드 이름을 입력하지 마십시오. 실제로는 새로운 필드가 데이터를 파생할 필드의 이름을 지정하는 것입니다.
- 3 Type 콤보 박스에서 필드의 데이터 타입을 선택합니다. 선택하는 데이터 타입은 교체하는 필드의 데이터 타입과 달라야 합니다. 크기가 다른 문자열 필드는 서로 교체할 수 없습니다. 데이터 타입은 달라야 하지만 원본 테이블의 필드의 실제 데이터 타입과는 호환되어야 합니다.

- 4 Size 편집 상자에 적당한 필드의 크기를 입력합니다. 크기는 *TStringField*, *TBytesField*, *TVarBytesField* 등과 같은 필드 타입에만 적용됩니다.
- 5 아직 선택되지 않았으면 Field type 라디오 그룹에서 Data를 선택합니다.
- 6 OK를 선택합니다. New Field 대화 상자가 닫히고, 새로 정의된 데이터 필드가 1 단계에서 지정한 기존 필드를 교체하고, 데이터 모듈의 컴포넌트 선언이나 폼의 **type** 선언이 업데이트됩니다.

필드 컴포넌트와 연결된 속성 또는 이벤트를 편집하려면 Field Editor 리스트 박스에서 컴포넌트 이름을 선택한 다음 Object Inspector를 사용하여 속성 또는 이벤트를 편집합니다. 필드 컴포넌트 속성 및 이벤트의 편집에 대한 자세한 내용은 19-11 페이지의 "영구적 필드 속성 및 이벤트 설정"을 참조하십시오.

계산된 필드 정의

계산된 필드는 런타임 시 데이터셋의 *OnCalcFields* 이벤트 핸들러로 계산된 값을 표시합니다. 예를 들어, 다른 필드에서 연결된 값을 표시하는 문자열 필드를 작성할 수 있습니다.

다음과 같은 방법으로 New Field 대화 상자에서 계산된 필드를 작성합니다.

- 1 Name 편집 상자에 계산된 필드의 이름을 입력합니다. 기존 필드의 이름을 입력하지 마십시오.
- 2 Type 콤보 박스에서 필드의 데이터 타입을 선택합니다.
- 3 Size 편집 상자에 적당한 필드의 크기를 입력합니다. 크기는 *TStringField*, *TBytesField*, *TVarBytesField* 등과 같은 필드 타입에만 적용됩니다.
- 4 Field type 라디오 그룹에서 Calculated 또는 InternalCalc를 선택합니다. InternalCalc는 클라이언트 데이터셋에서 작업하는 경우에만 사용할 수 있습니다. 이들 계산된 필드의 타입 사이의 중요한 차이점은 InternalCalc 필드에 대해 계산된 값은 클라이언트 데이터셋의 데이터의 일부로서 저장 및 검색된다는 점입니다.
- 5 OK를 선택합니다. 새로 정의된 계산된 필드는 Field Editor 리스트 박스의 영구적 필드 목록 끝에 자동으로 추가되며, 컴포넌트 선언은 폼 또는 데이터 모듈의 **type** 선언에 자동으로 추가됩니다.
- 6 필드에 대해 값을 계산하는 코드를 데이터셋에 대한 *OnCalcFields* 이벤트 핸들러에 둡니다. 필드 값 계산을 위한 코드 작성에 대한 자세한 내용은 19-8 페이지의 "계산된 필드 프로그래밍"을 참조하십시오.

참고 필드 컴포넌트와 연결된 속성 또는 이벤트를 편집하려면 Field Editor 리스트 박스에서 컴포넌트 이름을 선택한 다음 Object Inspector를 사용하여 속성 또는 이벤트를 편집합니다. 필드 컴포넌트 속성 및 이벤트 편집에 대한 자세한 내용은 19-11 페이지의 "영구적 필드 속성 및 이벤트 설정"을 참조하십시오.

계산된 필드 프로그래밍

계산된 필드를 정의하고 나면 값을 계산하기 위한 코드를 작성해야 합니다. 그렇지 않으면 항상 Null 값을 가집니다. 계산된 필드의 코드는 데이터셋에 대한 *OnCalcFields* 이벤트에 둡니다.

다음과 같은 방법으로 계산된 필드에 대한 값을 프로그래밍합니다.

- 1 Object Inspector 드롭다운 목록에서 데이터셋 컴포넌트를 선택합니다.
- 2 Object Inspector Events 페이지를 선택합니다.
- 3 *OnCalcFields* 속성을 더블 클릭하여 데이터셋 컴포넌트에 대한 *CalcFields* 프로시저를 불러오거나 생성합니다.
- 4 계산된 필드의 값과 속성을 설정하는 코드를 원하는 대로 작성합니다.

예를 들어, *CustomerData* 데이터 모듈의 *Customers* 테이블에 대한 *CityStateZip*이라는 계산된 필드를 생성했다고 가정합니다. *CityStateZip*은 data-aware 컨트롤에서 한 줄로 회사의 주소와 우편 번호를 표시합니다.

Customers 테이블에 대한 *CalcFields* 프로시저에 코드를 추가하려면 Object Inspector 드롭다운 목록에서 *Customers* 테이블을 선택하고 Events 페이지로 변환하여 *OnCalcFields* 속성을 더블 클릭합니다.

TCustomerData.CustomersCalcFields 프로시저는 유닛의 소스 코드 창에 나타납니다. 다음 코드를 프로시저에 추가하여 필드를 계산합니다.

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value  
+ ' ' + CustomersZip.Value;
```

참고 내부적으로 계산된 필드에 대한 *OnCalcFields* 이벤트 핸들러를 작성할 때 *State*가 *dsInternalCalc*인 경우에 클라이언트 데이터셋의 *State* 속성을 확인하고 그 값을 단지 다시 계산하는 것만으로 성능을 향상시킬 수 있습니다. 자세한 내용은 23-11 페이지의 "클라이언트 데이터셋에서 내부적으로 계산된 필드 사용"을 참조하십시오.

조회 필드 정의

조회 필드는 지정한 검색 기준에 따라 런타임 시 값을 표시하는 읽기 전용 필드입니다. 가장 간단한 폼에서는 조회 필드에 검색할 기준 필드의 이름과 검색할 필드 값, 값을 표시해야 하는 조회 데이터셋의 다른 필드를 전달합니다.

예를 들어, 작업자가 조회 필드를 사용하여 고객이 제공하는 우편 번호와 일치하는 주소를 자동으로 결정할 수 있는 메일 주문 애플리케이션을 가정하십시오. 검색할 열은 *ZipTable.Zip*이고, 검색할 값은 *Order.CustZip*에 입력된 고객의 우편 번호이고, 반환할 값은 레코드의 *ZipTable.City* 및 *ZipTable.State* 열입니다. 여기서 *ZipTable.Zip*의 값은 *Order.CustZip* 필드의 현재 값과 일치합니다.

참고 단방향 데이터셋은 조회 필드를 지원하지 않습니다.

다음과 같은 방법으로 New Field 대화 상자에서 조회 필드를 생성합니다.

- 1 Name 편집 상자에 조회 필드의 이름을 입력합니다. 기존 필드의 이름을 입력하지 마십시오.
- 2 Type 콤보 박스에서 필드의 데이터 타입을 선택합니다.
- 3 Size 편집 상자에 적당한 필드의 크기를 입력합니다. 크기는 *TStringField*, *TBytesField*, *TVarBytesField* 등과 같은 필드 타입에만 적용됩니다.
- 4 Field type 라디오 그룹에서 Lookup을 선택합니다. Lookup을 선택하면 Dataset 및 Key Fields 콤보 박스를 사용할 수 있습니다.
- 5 Dataset 콤보 박스 드롭다운 목록에서 필드 값을 조회할 데이터셋을 선택합니다. 조회 데이터셋은 필드 컴포넌트 자체에 대한 데이터셋과는 달라야 합니다. 그렇지 않으면 런타임 시 순환 참조 예외가 발생합니다. 조회 데이터셋을 지정하면 Lookup Keys 및 Result Field 콤보 박스를 사용할 수 있습니다.
- 6 Key Fields 드롭다운 목록에서 값을 일치시킬 현재 데이터셋의 필드를 선택합니다. 하나 이상의 필드를 일치시키려면 드롭다운 목록에서 선택하는 대신 필드 이름을 직접 입력합니다. 세미콜론을 사용하여 여러 필드 이름을 구분합니다. 하나 이상의 필드를 사용하는 경우에는 영구적 필드 컴포넌트를 사용해야 합니다.
- 7 Lookup Keys 드롭다운 목록에서 6 단계에서 지정한 Source Fields 필드에 대해 일치시킬 조회 데이터셋의 필드를 선택합니다. 하나 이상의 키 필드를 지정한 경우에는 동일한 수의 조회 키를 지정해야 합니다. 하나 이상의 필드를 지정하려면 필드 이름을 직접 입력하여 세미콜론으로 여러 필드 이름을 구분합니다.
- 8 Result Field 드롭다운 목록에서 생성 중인 조회 필드의 값으로 반환할 조회 데이터셋의 필드를 선택합니다.

애플리케이션을 디자인하고 실행할 때 조회 필드 값은 계산된 필드 값이 계산되기 전에 결정됩니다. 계산된 필드는 조회 필드에 기준을 둘 수 있지만 조회 필드는 계산된 필드에 기준을 둘 수가 없습니다.

LookupCache 속성을 사용하여 조회 필드가 결정되는 방식을 조정할 수 있습니다. *LookupCache*는 데이터셋이 처음으로 열릴 때 조회 필드의 값이 메모리에 캐시로 저장되는지 또는 데이터셋의 현재 레코드가 변경될 때마다 조회 필드의 값이 동적으로 조회되는지의 여부를 결정합니다. *LookupCache*를 *True*로 설정하면 *LookupDataSet*이 거의 변경되지 않으며 다른 조회 값의 수가 작은 경우 조회 필드 값을 캐시로 저장합니다. 조회 값을 캐시로 저장하면 *DataSet*이 열릴 때 *LookupKeyFields* 값의 모든 집합에 대한 조회 값이 미리 로드되기 때문에 수행 속도가 빨라집니다. *DataSet*의 현재 레코드가 변경되면 필드 객체는 *LookupDataSet*에 액세스하지 않고 캐시에서 *Value*를 찾을 수 있습니다. 이러한 성능 향상은 *LookupDataSet*이 액세스가 느린 네트워크 상에 있는 경우 특히 두드러집니다.

팁 조회 캐시를 사용하면 보조 데이터셋에서가 아닌 프로그램으로 조회 값을 제공할 수 있습니다. *LookupDataSet* 속성이 *nil*인지 확인합니다. 그런 다음 *LookupList* 속성의 *Add* 메소드를 사용하여 조회 값으로 채웁니다. *LookupCache* 속성을 *True*로 설정합니다. 필드는 조회 데이터셋의 값으로 겹쳐 쓰지 않고 제공된 조회 목록을 사용합니다.

*DataSet*의 모든 레코드가 *KeyFields*에 대해 각기 다른 값을 가지면 캐시의 값을 찾는 오버헤드가 캐시에서 제공하는 성능 상의 장점보다 더 클 수 있습니다. 캐시의 값을 찾는 오버헤드는 *KeyFields*에서 취할 수 있는 다른 값의 수로 증가합니다.

*LookupDataSet*이 순간적인 경우, 조회 값을 캐시로 저장하면 부정확한 결과를 초래할 수 있습니다. *RefreshLookupList*를 호출하여 조회 캐시의 값을 업데이트합니다. *RefreshLookupList*는 모든 *LookupKeyFields* 값의 집합에 대한 *LookupResultField*의 값을 포함하는 *LookupList* 속성을 다시 생성합니다.

런타임 시 *LookupCache*를 설정할 때 *RefreshLookupList*를 호출하여 캐시를 초기화합니다.

집계 필드(Aggregates field) 정의

집계 필드는 클라이언트 데이터셋의 추출된 집계로부터의 값을 표시합니다. 집계는 레코드 집합의 데이터를 요약하는 계산입니다. 추출된 집계에 대한 자세한 사항은 23-12 페이지의 "유지 관리되는 집계(Maintained Aggregates) 속성 사용"을 참조하십시오.

다음과 같은 방법으로 New Field 대화 상자에서 집계 필드를 생성합니다.

- 1 Name 편집 상자에 집계 필드의 이름을 입력합니다. 기존 필드의 이름을 입력하지 마십시오.
- 2 Type 콤보 박스에서 필드의 데이터 타입을 선택합니다.
- 3 Field 타입 라디오 그룹에서 Aggregate를 선택합니다.
- 4 OK를 선택합니다. 새로 정의된 집계 필드가 클라이언트 데이터셋에 자동으로 추가되고 클라이언트 데이터셋의 *Aggregates* 속성이 적절한 집계 사양을 포함하도록 자동으로 업데이트됩니다.
- 5 새로 생성된 집계 필드의 *ExprText* 속성에 집계에 대한 계산을 둡니다. 집계 정의에 대한 자세한 내용은 23-12 페이지의 "Aggregates 지정"을 참조하십시오.

일단 영구적 *TAggregateField*를 생성하면 *TDBText* 컨트롤을 집계 필드에 연결할 수 있습니다. 그러면 *TDBText* 컨트롤에서 원본으로 사용한 클라이언트 데이터셋의 현재 레코드와 관계 있는 집계 필드의 값을 표시합니다.

영구적 필드 컴포넌트 삭제

영구적 필드 컴포넌트를 삭제하면 테이블의 사용 가능한 열의 하위 집합을 액세스하고 테이블의 열을 교체하기 위해 자신의 영구적 필드를 정의할 경우에 유용합니다. 다음과 같은 방법으로 데이터셋에 대한 하나 이상의 영구적 필드 컴포넌트를 삭제합니다.

- 1 Fields 편집 상자에서 삭제할 필드를 선택합니다.
- 2 *Del*을 누릅니다.

참고 또한 컨텍스트 메뉴의 Delete를 선택하여 선택된 필드를 삭제할 수 있습니다.

삭제하는 필드는 더 이상 데이터셋에서 사용할 수 없으며 data-aware 컨트롤로 표시할 수도 없습니다. 실수로 삭제한 영구적 필드 컴포넌트는 항상 다시 생성할 수 있지만 속성이나 이벤트에 이전의 변경 내용은 없어집니다. 자세한 내용은 19-4 페이지의 "영구적 필드 생성"을 참조하십시오.

참고 데이터셋의 모든 영구적 필드 컴포넌트를 삭제하면 데이터셋은 원본으로 사용한 데이터베이스 테이블의 모든 열에 동적 필드 컴포넌트를 사용하는 상태로 변환됩니다.

영구적 필드 속성 및 이벤트 설정

디자인 타임에 영구적 필드 컴포넌트에 대한 속성을 설정하고 이벤트를 사용자 지정할 수 있습니다. 속성은 data-aware 컴포넌트에서, 예를 들어 필드가 *TDBGrid*에 나타날 수 있는지의 여부나 값을 수정할 수 있는지의 여부와 같은 필드 표시 방식을 조정합니다. 이벤트는 필드의 데이터가 폐치, 변경, 설정 또는 검증될 때 발생하는 내용들을 조정합니다.

필드 컴포넌트의 속성을 설정하거나 그에 대한 사용자 지정 이벤트 핸들러를 작성하려면 Fields Editor에서 컴포넌트를 선택하거나 Object Inspector의 컴포넌트 목록에서 컴포넌트를 선택합니다.

디자인 타임 시 표시 및 편집 속성 설정

선택된 필드 컴포넌트의 표시 속성을 편집하려면 Object Inspector 창의 Properties 페이지로 변환합니다. 다음 표는 편집될 수 있는 표시 속성을 요약한 것입니다.

표 19.3 필드 컴포넌트 속성

속성	용도
<i>Alignment</i>	data-aware 컴포넌트 내에서 필드 내용을 왼쪽, 오른쪽 또는 가운데 정렬합니다.
<i>ConstraintErrorMessage</i>	제약 조건으로 충돌을 편집할 때 표시할 텍스트를 지정합니다.
<i>CustomConstraint</i>	편집 중에 데이터에 적용할 지역 제약 조건을 지정합니다.
<i>Currency</i>	Numeric 필드에만 해당됩니다. <i>True</i> : 화폐 값을 표시합니다. <i>False</i> (기본값): 화폐 값을 표시하지 않습니다.
<i>DisplayFormat</i>	data-aware 컴포넌트에 표시되는 데이터 형식을 지정합니다.
<i>DisplayLabel</i>	data-aware 그리드 컴포넌트의 필드에 대한 열 이름을 지정합니다.
<i>DisplayWidth</i>	이 필드를 표시하는 그리드 열의 문자 폭을 지정합니다.
<i>EditFormat</i>	data-aware 컴포넌트의 데이터의 편집 형식을 지정합니다.
<i>EditMask</i>	문자의 지정된 타입 및 범위로 편집 가능한 필드의 데이터 입력을 제한하고, 필드 내에 나타나는 모든 특수한 편집 불가능한 문자(하이픈, 괄호 등)를 지정합니다.
<i>FieldKind</i>	만들려는 필드의 타입을 지정합니다.
<i>FieldName</i>	필드가 값과 데이터 타입을 파생시키는 테이블의 열 이름을 지정합니다.
<i>HasConstraints</i>	필드에 주어진 제약 조건이 있는지 나타냅니다.
<i>ImportedConstraint</i>	데이터 사전이나 SQL 서버에서 import 한 SQL 제약 조건을 지정합니다.
<i>Index</i>	데이터셋의 필드 순서를 지정합니다.
<i>LookupDataSet</i>	<i>Lookup</i> 이 <i>True</i> 일 때 필드 값을 조회하는 데 사용되는 테이블을 지정합니다.
<i>LookupKeyFields</i>	조회를 수행할 때 일치시킬 조회 데이터셋의 필드를 지정합니다.
<i>LookupResultField</i>	이 필드로 값을 복사할 조회 데이터셋의 필드를 지정합니다.
<i>MaxValue</i>	Numeric 필드에만 해당됩니다. 사용자가 필드에 입력할 수 있는 최대값을 지정합니다.

표 19.3 필드 컴포넌트 속성 (계속)

속성	용도
<i>Min Value</i>	Numeric 필드에만 해당됩니다. 사용자가 필드에 입력할 수 있는 최소값을 지정합니다.
<i>Name</i>	Delphi 내에서 필드 컴포넌트의 컴포넌트 이름을 지정합니다.
<i>Origin</i>	원본으로 사용한 데이터베이스에 나타나는 것과 똑같이 필드의 이름을 지정합니다.
<i>Precision</i>	Numeric 필드에만 해당됩니다. 유효 자리수를 지정합니다.
<i>ReadOnly</i>	<i>True</i> : data-aware 컨트롤에 필드 값을 표시하지만 편집할 수는 없습니다. <i>False</i> (기본값): 필드 값을 표시하고 편집할 수 있습니다.
<i>Size</i>	문자열 기반 필드에 표시하거나 입력할 수 있는 문자의 최대 수 또는 <i>TBytesField</i> 및 <i>TVarBytesField</i> 필드의 크기(바이트)를 지정합니다.
<i>Tag</i>	필요에 따라 모든 컴포넌트에서 프로그래머가 사용할 수 있는 Long 정수입니다.
<i>Transliterate</i>	<i>True</i> (기본값): 데이터셋과 데이터베이스 사이로 데이터가 전송될 때 해당 로케일에 대한 번역이 일어나도록 지정합니다. <i>False</i> : 로케일 번역이 일어나지 않습니다.
<i>Visible</i>	<i>True</i> (기본값): data-aware 그리드에 필드를 표시할 수 있습니다. <i>False</i> : data-aware 그리드 컴포넌트의 필드를 표시할 수 없습니다. 사용자 지정 컴포넌트는 이 속성에 따라 표시 여부를 결정할 수 있습니다.

모든 필드 컴포넌트가 다 모든 속성을 사용할 수 있는 것은 아닙니다. 예를 들어, *TStringField* 타입의 필드 컴포넌트는 *Currency*, *MaxValue* 또는 *DisplayFormat* 속성이 없으며 *TFloatField* 타입의 컴포넌트는 *Size* 속성이 없습니다.

대부분의 속성의 목적은 직관적이지만 *Calculated*와 같은 일부 속성은 추가적인 프로그래밍 단계를 거쳐야 유용해질 수 있습니다. *DisplayFormat*, *EditFormat* 및 *EditMask* 와 같은 기타 속성은 상호 관계를 가지며 설정이 조정되어야 합니다. *DisplayFormat*, *EditFormat* 및 *EditMask* 사용에 대한 자세한 내용은 19-14 페이지의 "사용자 입력 조정 및 마스킹"을 참조하십시오.

런타임 시 필드 컴포넌트 속성 설정

런타임 시 필드 컴포넌트의 속성을 사용하고 조작할 수 있습니다. 영구적 필드 컴포넌트를 이름순으로 액세스합니다. 여기서 이름은 필드 이름을 데이터셋 이름에 연결하여 가져올 수 있습니다.

예를 들어, 다음 코드는 *Customers* 테이블의 *CityStateZip* 필드에 대한 *ReadOnly* 속성을 *True*로 설정합니다.

```
CustomersCityStateZip.ReadOnly := True;
```

그리고 다음 문장은 *Customers* 테이블에 있는 *CityStateZip* 필드의 *Index* 속성을 3으로 설정하여 필드 순서를 변경합니다.

```
CustomersCityStateZip.Index := 3;
```

필드 컴포넌트의 속성 집합 만들기

애플리케이션에서 사용하는 데이터셋의 여러 필드가 일반적인 서식 속성(예: *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue* 등)을 공유하면 한 필드에 속성을 설정한 다음 해당 속성을 데이터 사전에 속성 집합으로 저장하는 것이 훨씬 더 편리합니다. 데이터 사전에 저장된 속성 집합은 다른 필드에 쉽게 적용할 수 있습니다.

참고 속성 집합 및 데이터 사전은 BDE 활성 데이터셋에만 사용할 수 있습니다.

데이터셋의 필드 컴포넌트를 기반으로 하는 속성 집합을 만들려면

- 1 데이터셋을 더블 클릭하여 Fields Editor를 불러옵니다.
- 2 속성을 설정할 필드를 선택합니다.
- 3 Object Inspector의 필드에 원하는 속성을 설정합니다.
- 4 Fields Editor 리스트 박스를 마우스 오른쪽 버튼으로 클릭하여 컨텍스트 메뉴를 불러옵니다.
- 5 속성 저장을 선택하여 현재 필드의 속성 설정을 데이터 사전의 속성 집합으로 저장합니다.

속성 집합의 이름은 현재 필드의 이름으로 기본값을 설정합니다. 컨텍스트 메뉴에서 속성 저장 대신 다른 이름으로 속성 저장을 선택하면 속성 집합에 다른 이름을 지정할 수 있습니다.

새 속성 집합을 만들고 데이터 사전에 추가하면 다른 영구적 필드 컴포넌트와 연결할 수 있습니다. 나중에 연결을 제거하는 경우에도 속성 집합은 데이터 사전에 정의되어 있습니다.

참고 SQL Explorer에서 직접 속성 집합을 만들 수도 있습니다. SQL Explorer를 사용하여 속성 집합을 만들면 그 속성 집합이 데이터 사전에 추가되지만 모든 필드에 적용되지는 않습니다. SQL Explorer를 사용하면 속성 집합을 기반으로 하는 필드를 폼으로 끌어올 때 자동으로 폼에 놓이는 두 가지 속성, 필드 타입(*TFloatField*, *TStringField* 등)과 data-aware 컨트롤(*TDBEdit*, *TDBCheckBox* 등)을 추가로 지정할 수 있습니다. 자세한 내용은 SQL Explorer의 온라인 도움말을 참조하십시오.

필드 컴포넌트와 속성 집합 연결

애플리케이션에서 사용하는 데이터셋의 여러 필드가 일반적인 서식 속성(예: *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue* 등)을 공유하고 해당 속성 설정을 데이터 사전의 속성 집합으로 저장하면 각 필드에 대한 설정을 수동으로 다시 만들지 않고 속성 집합을 쉽게 필드에 적용할 수 있습니다. 또한 나중에 데이터 사전에서 속성 설정을 변경하면 다음에 필드 컴포넌트를 데이터셋에 추가할 때 해당 변경 내용은 속성 집합에 연결된 모든 필드에 자동으로 적용됩니다.

다음과 같이 속성 집합을 필드 컴포넌트에 적용합니다.

- 1 데이터셋을 더블 클릭하여 Fields Editor를 불러옵니다.
- 2 속성 집합을 적용할 필드를 선택합니다.

- 3 컨텍스트 메뉴를 불러오고 Associate Attributes를 선택합니다.
- 4 Associate Attributes 상자에서 적용할 속성 집합을 선택하거나 입력합니다. 데이터 사전에 현재 필드와 이름이 같은 속성 집합이 있으면 그 집합 이름은 편집 상자에 나타냅니다.

중요 나중에 데이터 사전의 속성 집합이 변경되면 속성 집합을 사용하는 각 필드 컴포넌트에 그 집합을 다시 적용해야 합니다. 속성을 다시 적용할 때 데이터셋 내에서 Field Editor를 불러오고 필드 컴포넌트를 여러 개 선택할 수 있습니다.

속성 연결 제거

속성 집합을 필드와 연결하지 않으려는 경우에는 다음 단계를 수행하여 연결을 제거할 수 있습니다.

- 1 필드를 포함하는 데이터셋에 대한 Fields Editor를 불러옵니다.
- 2 속성 연결을 제거할 필드를 선택합니다.
- 3 Fields Editor의 컨텍스트 메뉴를 불러오고 Unassociate Attributes를 선택합니다.

중요 속성 집합을 연결하지 않아도 모든 필드 속성이 변경되지 않습니다. 필드는 속성 집합이 적용될 때의 설정을 그대로 유지합니다. 이러한 속성을 변경하려면 Fields Editor에서 필드를 선택한 다음 Object Inspector에서 그 속성을 설정합니다.

사용자 입력 조정 및 마스킹

EditMask 속성은 사용자가 *TStringField*, *TDateField*, *TTimeField*, *TDateTimeField* 및 *TSQLTimeStampField* 컴포넌트와 연결된 data-aware 컴포넌트에 입력할 수 있는 값의 형식과 범위를 조정하는 방법을 제공합니다. 기존의 마스크를 사용하거나 자신의 마스크를 생성할 수 있습니다. 가장 쉽게 편집 마스크를 사용하고 작성하는 방법은 Input Mask 에디터를 사용하는 것입니다. 그러나 Object Inspector의 *EditMask* 필드에 직접 마스크를 입력할 수도 있습니다.

참고 *TStringField* 컴포넌트에서 *EditMask* 속성 또한 표시 형식입니다.

다음과 같은 방법으로 필드 컴포넌트에 대한 Input Mask 에디터를 실행합니다.

- 1 Fields Editor나 Object Inspector에서 컴포넌트를 선택합니다.
- 2 Object Inspector의 Properties 페이지를 클릭합니다.
- 3 Object Inspector의 EditMask 필드에 대한 값 열을 더블 클릭하거나 생략 버튼을 클릭합니다. Input Mask 에디터가 열립니다.

Input Mask 편집 상자를 사용하면 마스크 형식을 만들고 편집할 수 있습니다. Sample Masks 그리드를 사용하면 이미 정의된 마스크에서 선택할 수 있습니다. 예제 마스크를 선택하면 사용자가 수정하거나 있는 그대로 사용할 수 있는 Input Mask 편집 상자에 마스크 형식이 나타납니다. Test Input 편집 상자에서 마스크에 대해 허용할 수 있는 사용자 입력을 테스트할 수 있습니다.

사용자가 사용자 지정 마스크 집합을 생성한 경우에는 Masks 버튼을 사용하면 선택을 쉽게 하기 위해 그 마스크 집합을 Sample Masks 그리드로 로드할 수 있습니다.

숫자, 날짜 및 시간 필드에 대한 기본 서식 사용

Delphi는 *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, *TTimeField* 및 *TSQLTimeStampField* 컴포넌트에 기본 제공된 표시 및 편집 형식 루틴과 지능적인 기본 서식을 제공합니다. 이 루틴들을 사용하기 위해 해야 할 작업은 없습니다.

기본 서식은 다음과 같은 루틴으로 수행됩니다.

표 19.4 필드 컴포넌트 서식 루틴

루틴	필드 컴포넌트
<i>FormatFloat</i>	<i>TFloatField</i> , <i>TCurrencyField</i>
<i>FormatDateTime</i>	<i>TDateField</i> , <i>TTimeField</i> , <i>TDateTimeField</i>
<i>SQLTimeStampToString</i>	<i>TSQLTimeStampField</i>
<i>FormatCurr</i>	<i>TCurrencyField</i> , <i>TBCDField</i>
<i>BcdToStrF</i>	<i>TFMTBcdField</i>

필드 컴포넌트의 데이터 타입에만 적합한 형식 속성은 지정된 컴포넌트에 대해 사용될 수 있습니다.

날짜, 시간, 통화 및 숫자 값에 대한 기본 서식 규칙은 제어판의 지역별 설정 등록 정보를 기반으로 합니다. 예를 들어, 미국의 기본 설정을 사용하면 *Currency* 속성이 *True* 로 설정된 *TFloatField* 열은 값 1234.56에 대한 *DisplayFormat* 속성을 \$1234.56로 설정하는 반면, *EditFormat*은 1234.56입니다.

디자인 타임이나 런타임 시 필드 컴포넌트의 *DisplayFormat* 및 *EditFormat* 속성을 편집하면 필드에 대한 기본 표시 설정을 오버라이드할 수 있습니다. 또한 *OnGetText* 및 *OnSetText* 이벤트 핸들러를 작성하여 런타임 시 필드 컴포넌트에 대한 서식을 사용자 지정할 수 있습니다.

이벤트 처리

대부분의 컴포넌트처럼 필드 컴포넌트도 연결된 이벤트가 있습니다. 메소드는 이벤트에 대한 핸들러로서 할당될 수 있습니다. 이들 핸들러를 작성하면 data-aware 컨트롤을 통해 필드에 입력된 데이터에 영향을 주는 이벤트의 발생에 응답하고 사용자가 의도한 대로 동작을 수행할 수 있습니다. 다음 표는 필드 컴포넌트에 연결된 이벤트를 나열한 것입니다.

표 19.5 필드 컴포넌트 이벤트

이벤트	용도
<i>OnChange</i>	필드 값이 변경될 때 호출됩니다.
<i>OnGetText</i>	필드 컴포넌트 값을 표시하거나 편집하기 위해 검색할 때 호출됩니다.

표 19.5 필드 컴포넌트 이벤트 (계속)

이벤트	용도
<i>OnSetText</i>	필드 컴포넌트 값이 설정될 때 호출됩니다.
<i>OnValidate</i>	편집이나 삽입 작업으로 인해 값이 변경될 때마다 필드 컴포넌트 값을 확인하기 위해 호출됩니다.

OnGetText 및 *OnSetText* 이벤트는 주로 기본 제공 서식 함수보다 정교한 사용자 지정 서식을 원하는 프로그래머에게 유용합니다. *OnChange*는 메뉴나 비주얼 컨트롤의 사용 가능 또는 사용 불가능과 같은 데이터 변경에 연결된 애플리케이션별 작업을 수행하는 경우에 유용합니다. *OnValidate*는 데이터베이스 서버에 값을 반환하기 전에 애플리케이션에서 데이터 입력 검증을 조정하고자 하는 경우에 유용합니다.

다음과 같은 방법으로 필드 컴포넌트에 대한 이벤트 핸들러를 작성합니다.

- 1 컴포넌트를 선택합니다.
- 2 Object Inspector에서 Events 페이지를 선택합니다.
- 3 이벤트 핸들러의 Value 필드를 더블 클릭하여 소스 코드 창을 표시합니다.
- 4 핸들러 코드를 생성하거나 편집합니다.

런타임 시 필드 컴포넌트 메소드 사용

런타임 시 사용 가능한 필드 컴포넌트 메소드를 사용하면 한 데이터 타입에서 다른 데이터 타입으로 필드 값을 변환할 수 있고 필드 컴포넌트에 연결된 폼의 첫 번째 data-aware 컨트롤에 포커스를 설정할 수 있습니다.

필드에 연결된 data-aware 컴포넌트의 포커스를 조정하는 것은 애플리케이션이 *BeforePost*와 같은 데이터셋 이벤트 핸들러에서 레코드 지향 데이터 검증을 수행할 때 중요합니다. 검증은 연결된 data-aware 컨트롤이 포커스를 갖고 있는지 여부와 상관없이 레코드의 필드에서 수행될 수 있습니다. 레코드의 특정 필드에서 검증이 실패하면 잘못된 데이터를 포함하는 data-aware 컨트롤에 포커스를 두어 수정 사항을 입력할 수 있습니다.

필드의 *FocusControl* 메소드를 사용하여 필드의 data-aware 컴포넌트에 대한 포커스를 조정합니다. *FocusControl*은 포커스를 필드에 연결된 폼의 첫 번째 data-aware 컨트롤로 설정합니다. 이벤트 핸들러는 필드를 검증하기 전에 필드의 *FocusControl* 메소드를 호출할 수 있습니다. 다음 코드는 *Customers* 테이블의 *Company* 필드에 대한 *FocusControl* 메소드를 호출하는 방법을 설명합니다.

```
CustomersCompany.FocusControl;
```

다음 표는 다른 필드 컴포넌트 메소드와 그 용도를 나열한 것입니다. 각 메소드에 대한 완전한 목록과 자세한 내용은 온라인 *VCL Reference*에서 *TField*와 자손에 대한 항목을 참조하십시오.

표 19.6 선택된 필드 컴포넌트 메소드

메소드	용도
AssignValue	필드의 타입에 기반한 자동 변환 기능을 사용하여 필드 값을 지정된 값으로 설정합니다.
Clear	필드를 해제하고 필드 값을 Null로 설정합니다.
GetData	필드에서 서식이 없는 데이터를 검색합니다.
IsValidChar	값을 설정하기 위해 data-aware 컨트롤에서 사용자가 입력한 문자가 이 필드에 허용되는지 결정합니다.
SetData	이 필드에 서식이 없는 데이터를 할당합니다.

필드 값 표시, 변환 및 액세스

TDBEdit 및 *TDBGrid*와 같은 data-aware 컨트롤은 필드 컴포넌트에 연결된 값을 자동으로 표시합니다. 데이터셋과 컨트롤에서 편집이 가능하면 data-aware 컨트롤 또한 데이터베이스에 새로운 값과 변경된 값을 전송할 수 있습니다. 일반적으로 data-aware 컨트롤의 기본 제공 속성과 메소드를 사용하면 사용자가 추가 프로그래밍 없이도 데이터셋에 연결하고 값을 표시하며 업데이트를 할 수 있습니다. 데이터베이스 애플리케이션에서 사용 가능한 경우에는 언제든지 사용하십시오. data-aware 컨트롤에 대한 자세한 내용은 15장 "데이터 컨트롤 사용"을 참조하십시오.

표준 컨트롤 또한 필드 컴포넌트에 연결된 데이터베이스 값을 표시하고 편집할 수 있습니다. 그러나 표준 컨트롤을 사용하려면 사용자가 추가로 프로그래밍해야 합니다. 예를 들어, 애플리케이션에서는 표준 컨트롤을 사용할 때 필드 값이 변경되기 때문에 컨트롤을 업데이트할 시기를 추적해야 합니다. 데이터셋에 데이터소스 컴포넌트가 있을 경우, 이벤트를 사용하여 이러한 작업을 할 수 있습니다. 특히 *OnDataChange* 이벤트를 사용하면 컨트롤의 값을 업데이트해야 할 시기를 알 수 있고 *OnStateChange* 이벤트를 사용하면 컨트롤의 사용 가능 및 사용 불가능 시기를 결정할 수 있습니다. 이들 이벤트에 대한 자세한 내용은 15-4 페이지의 "데이터 소스에서 변경 사항에 대한 응답"을 참조하십시오.

다음 항목에서는 표준 컨트롤에 표시하기 위해 필드 값을 작동하는 방법을 설명합니다.

표준 컨트롤의 필드 컴포넌트 값 표시

애플리케이션은 필드 컴포넌트의 *Value* 속성을 통해 데이터셋 열의 값을 액세스할 수 있습니다. 예를 들어, *CustomersCompany* 필드의 값이 변경될 수도 있기 때문에 다음과 같은 *OnDataChange* 이벤트 핸들러는 *TEdit* 컨트롤의 텍스트를 업데이트합니다.

```
procedure TForm1.CustomersDataChange(Sender:TObject, Field:TField);
begin
    Edit3.Text := CustomersCompany.Value;
end;
```

이 메소드는 문자열 값에는 잘 수행되지만 다른 데이터 타입에 대해서는 변환을 처리할 추가 프로그램이 필요할 수도 있습니다. 다행히 필드 컴포넌트에는 기본 제공된 변환 처리 기능이 있습니다.

참고 또한 가변 타입을 사용하여 필드 값을 액세스하고 설정할 수 있습니다. 필드 값 액세스 및 설정을 위한 가변 타입 사용에 대한 자세한 내용은 19-19 페이지의 "기본 데이터셋 속성으로 필드 값 액세스"를 참조하십시오.

필드 값 변환

변환 속성은 한 데이터 타입에서 다른 데이터 타입으로 변환하려고 시도합니다. 예를 들어, *AsString* 속성은 숫자 및 부울 값을 문자열 표시로 변환합니다. 다음 표는 필드 컴포넌트 변환 속성을 나열하고, 필드 컴포넌트 클래스에 의해 필드 컴포넌트에 권장되는 속성을 설명한 것입니다.

	AsVariant	AsString	AsInteger	AsFloat AsCurrency AsBCD	AsDateTime AsSQLTimeStamp	AsBoolean
TStringField;	예	해당 없음	예	예	예	예
TWideStringField	예	예	예	예	예	예
TIntegerField	예	예	해당 없음	예		
TSmallIntField	예	예	예	예		
TWordField	예	예	예	예		
TLargeintField	예	예	예	예		
TFloatField	예	예	예	예		
TCurrencyField	예	예	예	예		
TBCDField	예	예	예	예		
TFMTBCDField	예	예	예	예		
TDateTimeField	예	예		예	예	
TDateField	예	예		예	예	
TTimeField	예	예		예	예	
TSQLTimeStampField	예	예		예	예	
TBooleanField	예	예				
TBytesField	예	예				
TVarBytesField	예	예				
TBlobField	예	예				
TMemoField	예	예				
TGraphicField	예	예				
TVariantField	해당 없음	예	예	예	예	예
TAggregateField	예	예				

테이블의 일부 열들은 하나 이상의 변환 속성(예를 들어, *AsFloat*, *AsCurrency* 및 *AsBCD*)을 참조한다는 것에 유의하십시오. 이는 이들 속성 중 하나를 지원하는 모든 필드 데이터 타입이 항상 다른 속성도 함께 지원하기 때문입니다.

또한 *AsVariant* 속성은 모든 데이터 타입을 변환할 수 있다는 것에 유의하십시오. 위어나 열되지 않은 다른 데이터 타입에서 실제로는 유일한 옵션이라고 할 수 있는 *AsVariant*도 사용할 수 있습니다. 확실하지 않을 경우에는 *AsVariant*를 사용합니다.

어떤 경우에 대해서는 변환이 항상 가능하지는 않습니다. 예를 들어, *AsDateTime*은 문자열 값이 인식할 수 있는 날짜 시간 형식으로 되어 있는 경우에만 문자열이 날짜, 시간 또는 날짜 시간 형식으로 변환하는 데 사용될 수 있습니다. 변환 시도가 실패하면 예외가 발생합니다.

일부 다른 경우에 변환은 가능하지만 변환의 결과가 항상 직관적이지는 않습니다. 예를 들어, *TDateTimeField* 값을 부동 소수점 형식으로 변환한다는 것은 *AsFloat*는 필드의 날짜 부분을 12/31/1899 이후의 일 수로 변환하고, 필드의 시간 부분은 24시간의 단위로 변환한다는 것을 의미합니다. 표 19.7은 특수한 결과를 보여 주는 허용 가능한 변환을 나열한 것입니다.

표 19.7 특수한 변환 결과

변환	결과
문자열에서 부울로 변환	"True", "False", "Yes" 및 "No"를 부울로 변환합니다. 다른 값은 예외를 발생시킵니다.
부동 소수점에서 정수로 변환	부동 소수점 값을 가장 가까운 정수 값으로 반올림합니다.
DateTime 또는 SQLTimeStamp에서 부동 소수점으로 변환	날짜를 12/31/1899 이후의 일 수로 변환하고, 시간을 24시간의 단위로 변환합니다.
부울에서 문자열로 변환	모든 부울 값을 "True"나 "False"로 변환합니다.

그 외의 경우에는 변환이 전혀 가능하지 않습니다. 이러한 경우에 변환을 시도하면 예외가 발생합니다.

변환은 항상 할당되기 전에 발생합니다. 예를 들어, 다음 문장은 *CustomersCustNo* 값을 문자열로 변환하고 문자열을 편집 컨트롤의 텍스트에 할당합니다.

```
Edit1.Text := CustomersCustNo.AsString;
```

이와 반대로 다음 문장은 편집 컨트롤의 텍스트를 정수로 *CustomersCustNo* 필드에 할당합니다.

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

기본 데이터셋 속성으로 필드 값 액세스

필드 값 액세스를 위한 가장 일반적인 메소드는 *FieldValues* 속성에 가변 타입을 사용하는 것입니다. 예를 들어, 다음 문장은 편집 상자의 값을 *Customers* 테이블의 *CustNo* 필드에 둡니다.

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

FieldValues 속성은 가변 타입이므로 다른 데이터 타입을 가변 값으로 자동 변환합니다.

가변 타입에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

데이터셋의 Fields 속성으로 필드 값 액세스

필드가 속하는 데이터셋 컴포넌트의 *Fields* 속성으로 필드 값을 액세스할 수 있습니다. *Fields*는 데이터셋의 모든 필드의 인덱스된 목록을 유지합니다. 필드 값을 *Fields* 속성으로 액세스하면 많은 열을 반복해야 할 경우나 애플리케이션에서 디자인 타임에는 사용할 수 없는 테이블 작업을 할 경우에 유용합니다.

Fields 속성을 사용하려면 데이터셋의 필드의 순서와 데이터 타입을 알아야 합니다. 순서 번호를 사용하여 액세스할 필드를 지정합니다. 데이터셋의 첫 번째 필드는 번호가 0입니다. 필드 값은 각 필드 컴포넌트의 변환 속성을 사용하여 적절하게 변환되어야 합니다. 필드 컴포넌트 변환 속성에 대한 자세한 내용은 19-18 페이지의 "필드 값 변환"을 참조하십시오.

예를 들어, 다음 문장은 *Customers* 테이블의 일곱 번째 열(Country)의 현재 값을 편집 컨트롤에 할당합니다.

```
Edit1.Text := CustTable.Fields[6].AsString;
```

이와 반대로 데이터셋의 *Fields* 속성을 원하는 필드로 설정하여 값을 필드에 할당할 수 있습니다. 예를 들면, 다음과 같습니다.

```
begin
    Customers.Edit;
    Customers.Fields[6].AsString := Edit1.Text;
    Customers.Post;
end;
```

데이터셋의 FieldByName 메소드로 필드 값 액세스

또한 데이터셋의 *FieldByName* 메소드로 필드 값을 액세스할 수 있습니다. 이 메소드는 액세스하려는 필드 이름은 알지만 디자인 타임 시 원본으로 사용한 테이블에 액세스할 수 없을 경우에 유용합니다.

*FieldByName*을 사용하려면 액세스하려는 데이터셋과 필드 이름을 알아야 합니다. 필드의 이름을 인수로서 메소드에 전달합니다. 필드 값을 액세스하거나 변경하려면 *AsString*이나 *AsInteger*와 같은 적절한 필드 컴포넌트 변환 속성을 사용하여 결과를 변환합니다. 예를 들어, 다음 문장은 *Customers* 데이터셋의 *CustNo* 필드 값을 편집 컨트롤에 할당합니다.

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

이와 반대로 필드에 값을 할당할 수 있습니다.

```
begin
    Customers.Edit;
    Customers.FieldByName('CustNo').AsString := Edit2.Text;
    Customers.Post;
```

```
end;
```

필드에 대한 기본값 설정

클라이언트 데이터셋이나 BDE 활성 데이터셋에 있는 필드의 기본값이 런타임 시에 *DefaultExpression* 속성을 사용하여 계산되는 방법을 지정할 수 있습니다.

*DefaultExpression*은 필드 값을 참조하지 않는 유효한 SQL 값 표현식이 될 수 있습니다. 표현식에 숫자 값이 아닌 리터럴이 포함되는 경우에는 인용 부호로 나타냅니다. 예를 들어, 시간에 대한 정오의 기본값은 다음과 같이 표시됩니다.

```
'12:00:00'
```

리터럴 값 주위에 인용 부호가 포함됩니다.

참고 원본으로 사용한 데이터베이스 테이블이 필드에 대한 기본값을 정의하는 경우, 사용자가 *DefaultExpression*에 지정하는 기본값이 우선권을 가집니다. 이는 데이터셋이 필드를 포함한 레코드를 포스트할 때 *DefaultExpression*이 적용되고 이것이 편집된 레코드가 데이터베이스 서버에 적용되기 전에 발생하기 때문입니다.

제약 조건 작업

클라이언트 데이터셋이나 BDE 활성 데이터셋의 필드 컴포넌트는 SQL 서버 제약 조건을 사용할 수 있습니다. 또한 애플리케이션은 애플리케이션에 로컬인 이러한 데이터셋에 대해 사용자 지정 제약 조건을 만들어 사용할 수도 있습니다. 모든 제약 조건은 필드가 저장할 수 있는 값의 범위에 제한을 부과하는 규칙이나 조건입니다.

사용자 지정 제약 조건 만들기

사용자 지정 제약 조건은 다른 제약 조건 같이 서버에서 import할 수 없습니다. 사용자가 로컬 애플리케이션에서 선언하고 구현하며 적용하는 제약 조건입니다. 이처럼 사용자 지정 제약 조건은 데이터 항목의 사전 유효성 검사 강제 실행을 제공하는 데 유용하지만 서버 애플리케이션에서 받거나 보낸 데이터에 적용할 수는 없습니다.

사용자 지정 제약 조건을 만들려면 *CustomConstraint* 속성을 설정하여 제약 조건을 지정하고, 사용자가 런타임 시 제약 조건을 위반할 때 *ConstraintErrorMessage*를 표시하는 메시지로 설정합니다.

*CustomConstraint*는 필드 값에 부여된 애플리케이션별 제약 조건을 지정하는 SQL 문자열입니다. 사용자가 필드에 입력할 수 있는 값을 제한하도록 *CustomConstraint*를 설정합니다. *CustomConstraint*는 다음과 같은 유효한 SQL 검색 표현식이 될 수 있습니다.

```
x > 0 and x < 100
```

필드 값을 참조하는 데 사용되는 이름은 제약 조건 표현식에서 지속적으로 사용되는 한 예약된 SQL 키워드가 아닌 모든 문자열이 될 수 있습니다.

참고 사용자 지정 제약 조건은 BDE 활성 데이터셋과 클라이언트 데이터셋에서만 사용할 수 있습니다.

사용자 지정 제약 조건은 서버에서 오는 필드의 값에 대한 제약 조건과 함께 부과됩니다. 서버에서 부과한 제약 조건을 보려면 *ImportedConstraint* 속성을 읽어 보십시오.

서버 제약 조건 사용

대부분의 SQL 데이터베이스는 제약 조건을 사용하여 필드에 대한 가능한 값에 조건을 부여합니다. 예를 들어, 어떤 필드는 Null 값을 허용하지 않을 수도 있고, 값이 해당 열에 대해 고유해야 하거나 0보다 크고 150보다 작아야 할 수도 있습니다. 클라이언트 애플리케이션에서 이러한 조건을 복제할 수 있는 동안 클라이언트 데이터셋과 BDE 활성 데이터셋은 서버의 제약 조건을 로컬로 전달하는 *ImportedConstraint* 속성을 제공합니다.

*ImportedConstraint*는 어떤 방식으로든 필드 값을 제한하는 SQL 절을 지정하는 읽기 전용 속성입니다. 예를 들면, 다음과 같습니다.

값 > 0 및 값 < 100

데이터베이스 엔진에서 해석할 수 없으므로 설명으로 import한 서버 관련 SQL이나 표준이 아닌 SQL을 편집하는 경우를 제외하고는 *ImportedConstraint*의 값을 변경하지 마십시오.

필드 값에 제약 조건을 추가하려면 *CustomConstraint* 속성을 사용하십시오. 사용자 지정 제약 조건은 import된 제약 조건에 추가로 부과됩니다. 서버 제약 조건이 변경되면 *ImportedConstraint*의 값도 변경되지만 *CustomConstraint* 속성에 도입된 제약 조건은 유지됩니다.

ImportedConstraint 속성에서 제약 조건을 제거해도 이 제약 조건을 위반하는 필드 값의 유효성은 변경되지 않습니다. 제약 조건을 제거하면 로컬이 아닌 서버에서 제약 조건을 검사합니다. 제약 조건을 로컬로 검사하면 서버에서 오류 메시지를 표시하지 않고 위반이 발견될 때 *ConstraintErrorMessage* 속성이 표시하는 대로 오류 메시지가 나타납니다.

객체 필드 사용

객체 필드는 보다 단순한 다른 데이터 타입의 복합물을 나타내는 필드입니다. 여기에는 ADT(Abstract Data Type) 필드, 배열 필드, DataSet 필드 및 Reference 필드가 포함됩니다. 이러한 필드 타입은 모두 자식 필드나 다른 데이터셋을 포함하거나 참조합니다.

ADT 필드와 배열 필드는 자식 필드를 포함하는 필드입니다. ADT 필드의 자식 필드는 다른 필드 타입인 스칼라 또는 객체 타입이 될 수 있습니다. 이들 자식 필드는 서로 타입이 다를 수 있습니다. 배열 필드에는 동일한 타입의 자식 필드 배열이 포함됩니다.

데이터셋 및 참조 필드는 다른 데이터 집합을 액세스하는 필드입니다. 데이터셋 필드는 중첩된 (디테일) 데이터셋에 액세스하고, 참조 필드는 포인터(참조)를 다른 영구적 객체(ADT)에 저장합니다.

표 19.8 객체 필드 컴포넌트의 타입

컴포넌트 이름	용도
TADTField	ADT(Abstract Data Type) 필드를 나타냅니다.
TArrayField	배열 필드를 나타냅니다.
TDataSetField	중첩된 데이터셋 참조를 포함하는 필드를 나타냅니다.
TReferenceField	ADT에 대한 포인터인 REF 필드를 나타냅니다.

Fields Editor를 사용하여 객체 필드가 있는 데이터셋에 필드를 추가할 때 올바른 타입의 영구적 객체 필드가 자동으로 생성됩니다. 데이터셋에 영구적 객체 필드를 추가하면 데이터셋의 *ObjectView* 속성이 *True*로 자동으로 설정됩니다. 이는 구성 자식 필드가 구분된 독립 필드인 것처럼 이들 필드를 모두 평평하게(flatten out) 하기보다는 계층적으로 저장하도록 데이터셋에 알립니다.

다음 속성들은 모든 객체 필드에 공통적이며, 자식 필드와 데이터셋을 처리할 수 있는 기능을 제공합니다.

표 19.9 공통 객체 필드 자손 속성

속성	용도
Fields	객체 필드에 속하는 자식 필드를 포함합니다.
ObjectType	객체 필드를 분류합니다.
FieldCount	객체 필드에 속한 자식 필드의 수입니다.
FieldValues	자식 필드 값에 대한 액세스를 제공합니다.

ADT 및 배열 필드 표시

ADT 및 배열 필드는 모두 data-aware 컨트롤을 통해 표시될 수 있는 자식 필드를 포함합니다.

단일 필드 값을 표시하는 *TDBEdit*와 같은 data-aware 컨트롤은 쉽표로 구분된 편집할 수 없는 문자열에 자식 필드 값을 표시합니다. 또한 컨트롤의 *DataField* 속성을 객체 필드 자체 대신에 자식 필드로 설정하면 자식 필드는 다른 일반 데이터 필드와 같이 편집되어 보여질 수 있습니다.

TDBGrid 컨트롤은 데이터셋의 *ObjectView* 속성 값에 따라 ADT와 배열 필드를 다르게 표시합니다. *ObjectView*가 *False*이면 각 자식 필드는 단일 열에 나타납니다. *ObjectView*가 *True*이면 ADT 또는 배열 필드는 열의 제목 표시줄에 있는 화살표를 클릭하여 확장하거나 축소시킬 수 있습니다. 필드가 확장되면 각 자식 필드가 ADT 또는 배열 필드 자체의 제목 표시줄 아래에 자신의 열과 제목 표시줄에 나타납니다. ADT 또는 배열을 축소하면 단지 하나의 열만이 자식 필드가 포함된 쉽표로 구분된 편집할 수 없는 문자열로 나타납니다.

ADT 필드 작업

ADT는 서버에서 만들어진 사용자 지정 타입이며 레코드 타입과 유사합니다. ADT는 대부분의 스칼라 필드 타입, 배열 필드, 참조 필드 및 중첩 ADT를 포함할 수 있습니다.

ADT 필드 타입의 데이터에 액세스하는 방법은 매우 다양합니다. 다음 예제에서 자식 필드 값을 *CityEdit*라는 편집 상자에 할당하고 다음과 같은 ADT 구조를 사용하는 것에 대해 설명합니다.

```
Address
  Street
  City
  State
  Zip
```

영구적 필드 컴포넌트 사용

ADT 필드 값에 액세스하는 가장 쉬운 방법은 영구적 필드 컴포넌트를 사용하는 것입니다. 위의 ADT 구조에서 다음과 같은 영구적 필드는 Fields Editor를 사용하여 *Customer* 테이블에 추가될 수 있습니다.

```
CustomerAddress:TADTField;
CustomerAddrStreet:TStringField;
CustomerAddrCity:TStringField;
CustomerAddrState:TStringField;
CustomerAddrZip:TStringField;
```

이러한 영구적 필드의 경우에는 이름으로 ADT 필드의 자식 필드에 간단히 액세스할 수 있습니다.

```
CityEdit.Text := CustomerAddrCity.AsString;
```

영구적 필드는 ADT 자식 필드에 액세스하기 위한 가장 쉬운 방법이지만 데이터셋 구조를 디자인 타임 시 알 수 없는 경우에는 영구적 필드를 사용할 수 없습니다. 영구적 필드 없이 ADT 자식 필드에 액세스할 때는 데이터셋의 *ObjectView* 속성을 *True*로 설정해야 합니다.

데이터셋의 FieldByName 메소드 사용

자식 필드의 이름을 ADT 필드의 이름으로 한정하면 데이터셋의 *FieldByName* 메소드를 사용하여 ADT 필드의 자식 필드에 액세스할 수 있습니다.

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

데이터셋의 FieldValues 속성 사용

또한 데이터셋의 *FieldValues* 속성으로 한정된 필드 이름을 사용할 수도 있습니다.

```
CityEdit.Text := Customer['Address.City'];
```

*FieldValues*는 데이터셋의 기본 속성이기 때문에 속성 이름 (*FieldValues*)은 생략할 수 있습니다.

참고 ADT 자식 필드 값에 액세스하기 위한 다른 런타임 메소드와 달리 *FieldValues* 속성은 데이터셋의 *ObjectView* 속성이 *False*인 경우에도 작동합니다.

ADT 필드의 FieldValues 속성 사용

*TADTField*의 *FieldValues* 속성으로 자식 필드의 값을 액세스할 수 있습니다. *FieldValues*는 *Variant*를 수용하고 반환하여 모든 타입의 필드를 처리하고 변환할 수 있습니다. 인덱스 매개변수는 필드의 오프셋을 지정하는 정수 값입니다.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

*FieldValues*는 *TADTField*의 기본 속성이기 때문에 속성 이름 (*FieldValues*)은 생략할 수 있습니다. 따라서 다음 문장은 위의 문장과 동일합니다.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

ADT 필드의 Fields 속성 사용

각 ADT 필드에는 데이터셋의 *Fields* 속성과 유사한 *Fields* 속성이 있습니다. 데이터셋의 *Fields* 속성처럼 위치별로 자식 필드에 액세스하는 데 사용할 수 있습니다.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

이름별로 자식 필드에 액세스할 수도 있습니다.

```
CityEdit.Text :=
  TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

배열 필드(Array fields) 작업

배열 필드는 동일한 타입의 필드 집합으로 구성됩니다. 필드 타입은 스칼라(예를 들어, 부동 소수점 및 문자열) 또는 스칼라가 아닐(ADT) 수 있지만 배열에서는 배열 필드를 사용할 수 없습니다. *TDataSet*의 *SparseArrays* 속성은 배열 필드의 각 요소에 대해 고유한 *TField* 객체가 생성되는지 결정합니다.

배열 필드 타입의 데이터에 액세스하는 방법은 매우 다양합니다. 영구적 필드를 사용하지 않는 경우, 배열 필드의 요소에 액세스하기 전에 데이터셋의 *ObjectView* 속성을 *True*로 설정해야 합니다.

영구적 필드 사용

영구적 필드는 배열 필드의 배열 요소에 개별적으로 매핑할 수 있습니다. 예를 들어, 문자열의 6가지 요소 배열인 *TelNos_Array* 배열 필드를 고려해 보십시오. *Customer* 테이블 컴포넌트에 생성된 다음과 같은 영구적 필드는 *TelNos_Array* 필드와 이 필드의 6가지 요소를 나타냅니다.

```
CustomerTelNos_Array:TArrayField;
CustomerTelNos_Array0:TStringField;
CustomerTelNos_Array1:TStringField;
CustomerTelNos_Array2:TStringField;
CustomerTelNos_Array3:TStringField;
CustomerTelNos_Array4:TStringField;
CustomerTelNos_Array5:TStringField;
```

이들 영구적 필드의 경우, 다음 코드는 영구적 필드를 사용하여 배열 요소 값을 *TelEdit* 라고 하는 편집 상자에 할당합니다.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

배열 필드의 FieldValues 속성 사용

배열 필드의 *FieldValues* 속성으로 자식 필드 값에 액세스할 수 있습니다. *FieldValues*는 *Variant*를 수용하고 반환하므로 모든 타입의 자식 필드를 처리하고 변환할 수 있습니다. 예를 들면, 다음과 같습니다.

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

*FieldValues*는 *TArrayField*의 기본 속성이기 때문에 또한 다음과 같이 작성할 수 있습니다.

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

배열 필드의 Fields 속성 사용

*TArrayField*에는 개별 하위 필드를 액세스하는 데 사용할 수 있는 *Fields* 속성이 있습니다. 이것은 배열 필드(*OrderDates*)가 Null이 아닌 배열 요소로 리스트 박스를 채우는 데 사용되는 다음 예제에서 설명됩니다.

```
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;
```

데이터셋 필드 작업

데이터셋 필드는 중첩된 데이터셋에 저장된 데이터에 대한 액세스를 제공합니다. *NestedDataSet* 속성은 중첩된 데이터셋을 참조합니다. 그런 다음 중첩된 데이터셋의 데이터가 중첩된 데이터셋의 필드 객체를 통해 액세스됩니다.

데이터셋 필드 표시

TDBGrid 컨트롤을 사용하면 데이터셋 필드에 저장된 데이터를 표시할 수 있습니다. *TDBGrid* 컨트롤에서 데이터셋 필드는 "(DataSet)" 문자열을 사용하여 데이터셋 열의 각 셀에 나타나고, 또한 런타임 시 생략 버튼이 오른쪽에 표시됩니다. 생략 버튼을 클릭하면 현재 레코드의 데이터셋 필드에 연결된 데이터셋이 표시된 그리드가 들어 있는 새로운 폼이 나타납니다. 또한 이 폼은 프로그램에서 DB 그리드의 *ShowPopupEditor* 메소드를 사용하여 불러올 수 있습니다. 예를 들어, 그리드의 일곱 번째 열이 데이터셋 필드를 나타낸다면 다음 코드는 현재 레코드의 필드에 연결된 데이터셋을 표시합니다.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

중첩된 데이터셋의 데이터 액세스

데이터셋 필드는 일반적으로 data-aware 컨트롤에 직접 연결되지 않습니다. 그보다 중첩된 데이터셋은 바로 데이터셋 자체이므로 *TDataSet* 자손을 통해 데이터를 얻는다는 의미입니다. 사용하는 데이터셋의 타입은 부모 데이터셋(데이터셋 필드가 있는 데이터셋)에 의해 결정됩니다. 예를 들어, BDE 활성 데이터셋은 *TNestedTable*을 사용하여 데이터셋 필드의 데이터를 나타내고 클라이언트 데이터셋은 다른 클라이언트 데이터셋을 사용합니다.

다음과 같은 방법으로 데이터셋 필드의 데이터에 액세스합니다.

- 1 부모 데이터셋용 Fields Editor를 호출하여 영구적 *TDataSetField* 객체를 만듭니다.
- 2 데이터셋 필드의 값을 나타내는 데이터셋을 생성합니다. 부모 데이터셋과 호환되는 타입이어야 합니다.
- 3 2 단계에서 만든 데이터셋의 *DataSetField* 속성을 1 단계에서 만든 영구적 데이터셋 필드로 설정합니다.

현재 레코드의 중첩된 데이터셋 필드에 값이 있는 경우에는 디테일 데이터셋 컴포넌트가 중첩된 데이터를 가진 레코드를 포함할 것이며, 그렇지 않을 경우에는 디테일 데이터셋이 비어 있을 것입니다.

레코드가 방금 삽입된 경우에는 중첩된 데이터셋에 레코드를 삽입하기 전에 해당 레코드를 마스터 테이블에 포스트해야 합니다. 삽입된 레코드를 포스트하지 않으면 중첩된 데이터셋이 포스트되기 전에 자동으로 포스트됩니다.

참조 필드(Reference fields) 작업

참조 필드는 포인터나 참조를 다른 ADT 객체에 저장합니다. 이 ADT 객체는 다른 객체 테이블의 단일 레코드입니다. 참조 필드는 항상 데이터셋(객체 테이블)의 단일 레코드를 참조합니다. 참조되는 객체의 데이터는 중첩된 데이터셋으로 반환되지만, 또한 *TReferenceField*의 *Fields* 속성을 통해 액세스될 수도 있습니다.

참조 필드 표시

TDBGrid 컨트롤에서 참조 필드는 런타임 시 오른쪽에 생략 버튼이 있는 데이터셋 열의 각 셀에(Reference)가 지정됩니다. 런타임 시 생략 버튼을 클릭하면 그리드가 들어 있는 새로운 폼이 나타나면서 현재 레코드의 참조 필드에 연결된 객체를 표시합니다.

또한 이 폼은 프로그램에서 DB 그리드의 *ShowPopupEditor* 메소드를 사용하여 불러올 수 있습니다. 예를 들어, 그리드의 일곱 번째 열이 참조 필드를 나타낸다면 다음 코드는 현재 레코드의 필드에 연결된 객체를 표시합니다.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

참조 필드의 데이터 액세스

중첩된 데이터셋에 액세스하는 것과 동일한 방법으로 참조 필드의 데이터에 액세스할 수 있습니다.

- 1 부모 데이터셋용 Fields Editor를 호출하여 영구적 *TDataSetField* 객체를 만듭니다.
- 2 해당 데이터셋 필드의 값을 나타내는 데이터셋을 생성합니다.
- 3 2 단계에서 만든 데이터셋의 *DataSetField* 속성을 1 단계에서 만든 영구적 데이터셋 필드로 설정합니다.

참조가 할당되면 참조 데이터셋은 참조된 데이터를 가진 단일 레코드를 갖습니다. 참조가 Null이면 참조 데이터셋은 비게 됩니다.

또한 참조 필드의 Fields 속성을 사용하여 참조 필드의 데이터를 액세스할 수도 있습니다. 예를 들어, 다음 두 줄은 동일하게 참조 필드 *CustomerRefCity*의 데이터를 *CityEdit*라고 하는 편집 상자에 할당합니다.

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

참조 필드의 데이터를 편집하면 실제로는 참조된 데이터가 수정됩니다.

참조 필드를 지정하려면 먼저 SELECT 문을 사용하여 테이블에서 참조를 선택한 다음 지정해야 합니다. 예를 들면, 다음과 같습니다.

```
var
  AddressQuery:TQuery
  CustomerAddressRef:TReferenceField
begin
  AddressQuery.SQL.Text := 'SELECT REF(A) FROM AddressTable A WHERE A.City = ''San
  Francisco''';
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
```

20

Borland Database Engine 사용

BDE(Borland Database Engine)는 여러 애플리케이션에서 공유할 수 있는 데이터 액세스 메커니즘입니다. BDE는 데이터를 생성, 재구성, 페치(fetch), 업데이트하거나 로컬 및 원격 데이터베이스 서버를 처리할 수 있는 API 호출의 강력한 라이브러리를 정의합니다. BDE는 다른 데이터베이스에 연결할 드라이버를 사용하여 다양한 데이터베이스 서버에 액세스할 수 있는 단일 인터페이스를 제공합니다. Delphi의 버전에 따라 로컬 데이터베이스(Paradox, dBASE, FoxPro 및 Access)용 드라이버, InterBase, Oracle, Sybase, Informix, Microsoft SQL 서버, DB2 등의 원격 데이터베이스 서버용 SQL 연결 드라이버, 사용자 공유의 ODBC 드라이버를 제공할 수 있게 해주는 ODBC 어댑터를 사용할 수 있습니다.

BDE 기반 애플리케이션을 배포할 때에는 애플리케이션에 BDE를 포함시켜야 합니다. 그렇게 함으로써 애플리케이션의 크기와 배포의 복잡성은 증가되는 반면, BDE는 다른 BDE 기반 애플리케이션에서 공유될 수 있으며 데이터베이스를 처리하기 위한 광범위한 지원을 제공합니다. 애플리케이션에서 직접 BDE의 API를 사용할 수도 있지만 컴포넌트 팔레트의 BDE 페이지에 있는 컴포넌트에 이 기능의 대부분이 포함되어 있습니다.

참고 BDE API에 대한 내용은 BDE를 설치한 디렉토리에 설치되어 있는 BDE32.hlp 온라인 도움말 파일을 참조하십시오.

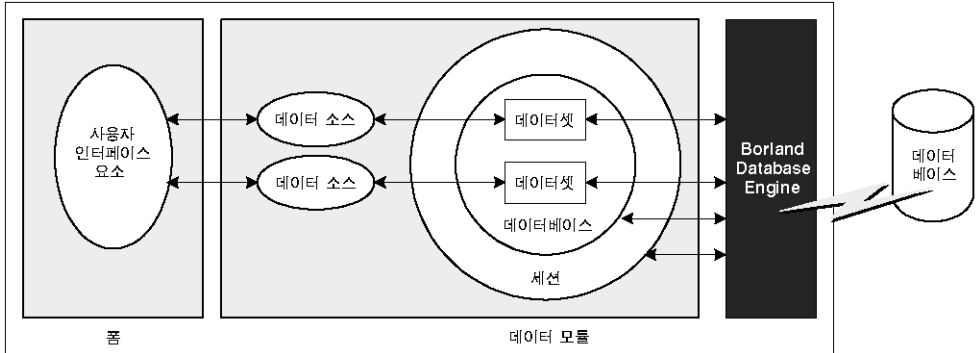
BDE 기반 아키텍처

BDE 사용 시 애플리케이션에서는 14-6 페이지의 "데이터베이스 아키텍처"에서 설명한 일반 데이터베이스 아키텍처의 변형을 사용합니다. BDE 기반 애플리케이션은 모든 Delphi 데이터베이스 애플리케이션에서 공통적으로 사용되는 사용자 인터페이스 요소, 데이터소스 및 데이터셋 외에도 다음과 같은 내용을 포함합니다.

- 트랜잭션을 제어하고 데이터베이스 연결을 관리하는 하나 이상의 데이터베이스 컴포넌트.
- 데이터베이스 연결과 같은 데이터 액세스 작업을 분리하고 데이터베이스 그룹을 관리하는 하나 이상의 세션 컴포넌트.

BDE 기반 애플리케이션의 컴포넌트 사이의 관계는 그림 20.1에 설명되어 있습니다.

그림 20.1 BDE 기반 애플리케이션의 컴포넌트



BDE 활성 데이터셋 사용

BDE 활성 데이터셋은 BDE (Borland Database Engine)를 사용하여 데이터에 액세스합니다. BDE 활성 데이터셋은 구현을 제공하는 BDE를 사용하여 18장 "데이터셋 이해"에서 설명한 공통적인 데이터셋 기능을 상속합니다. 또한 모든 BDE 데이터셋은 다음 기능들을 위한 속성, 이벤트 및 메소드를 추가합니다.

- 데이터베이스 및 세션 연결과 데이터셋의 연결
- BLOB 캐시
- BDE 핸들 얻기

BDE 활성 데이터셋에는 다음 세 가지가 있습니다.

- *TTable*은 단일 데이터베이스 테이블의 모든 행과 열을 나타내는 테이블 타입 데이터셋입니다. 테이블 타입 데이터셋의 공통적인 기능에 대한 설명은 18-25 페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오. *TTable*의 고유한 기능에 대한 설명은 20-4 페이지의 "TTable 사용"을 참조하십시오.
- *TQuery*는 SQL 문을 캡슐화하고 애플리케이션에서 결과 레코드에 액세스할 수 있게 해주는 쿼리 타입 데이터셋입니다. 쿼리 타입 데이터셋의 공통적인 기능에 대한 설명은 18-42 페이지의 "쿼리 타입 데이터셋 사용"을 참조하십시오. *TQuery*에 고유한 기능에 대한 설명은 20-9 페이지의 "TQuery 사용"을 참조하십시오.
- *TStoredProc*는 데이터베이스 서버에 정의되어 있는 내장 프로시저를 실행하는 내장 프로시저 타입 데이터셋입니다. 내장 프로시저 타입 데이터셋의 공통적인 기능에 대한 설명은 18-50 페이지의 "내장 프로시저 타입 데이터셋 사용"을 참조하십시오. *TStoredProc*의 고유한 기능에 대한 설명은 20-12 페이지의 "TStoredProc 사용"을 참조하십시오.

참고 BDE 활성 데이터셋의 세 가지 타입 외에도 업데이트를 캐시하는 데 사용할 수 있는 BDE 기반 클라이언트 데이터셋 (*TBDEClientDataSet*)이 있습니다. 업데이트 캐시에 대한 내용은 23-16 페이지의 "클라이언트 데이터셋을 사용하여 업데이트 캐시"를 참조하십시오.

데이터베이스 및 세션 연결과 데이터셋의 연결

BDE 활성 데이터셋이 데이터베이스에서 데이터를 페치(fetch)하려면 데이터베이스와 세션을 모두 사용해야 합니다.

- 데이터베이스는 특정 데이터베이스 서버로의 연결을 나타냅니다. 데이터베이스는 BDE 드라이버, BDE 드라이버를 사용하는 특정 데이터베이스 서버, 이 특정 데이터베이스 서버에 연결하기 위한 연결 매개변수 집합을 식별합니다. 각 데이터베이스는 *TDatabase* 컴포넌트로 표시합니다. 폼 또는 데이터 모듈에 추가하는 *TDatabase* 컴포넌트와 사용자의 데이터셋을 연결하거나 데이터베이스 서버를 이름별로 식별하여 Delphi에서 암시 데이터베이스 컴포넌트를 생성하도록 할 수도 있습니다. 데이터베이스 컴포넌트를 통해 로그인 프로세스 등 연결 설정 방법을 제어하여 트랜잭션을 생성하고 사용할 수 있기 때문에 명시적으로 생성된 *TDatabase* 컴포넌트를 사용하는 것이 대부분의 애플리케이션에서 권장됩니다.

BDE 활성 데이터셋과 데이터베이스를 연결하려면 *DatabaseName* 속성을 사용합니다. *DatabaseName*은 명시적 데이터베이스 컴포넌트 사용 여부에 따라, 그렇지 않은 경우 사용 중인 데이터베이스 타입에 따라 다른 정보를 갖고 있는 문자열입니다.

- 명시적 *TDatabase* 컴포넌트를 사용하는 경우 *DatabaseName*은 데이터베이스 컴포넌트의 *DatabaseName* 속성 값입니다.
- 암시적 데이터베이스 컴포넌트를 사용하고 데이터베이스에 BDE 별칭이 있는 경우 BDE 별칭을 *DatabaseName* 값으로 지정할 수 있습니다. BDE 별칭은 데이터베이스와 그 데이터베이스의 구성 정보를 나타냅니다. 별칭에 관련된 구성 정보는 데이터베이스 타입 (Oracle, Sybase, InterBase, Paradox, dBASE 등)에 따라 다릅니다. BDE 관리 도구 또는 SQL 탐색기를 사용하여 BDE 별칭을 생성하고 관리합니다.
- Paradox 또는 dBASE 데이터베이스용 암시적 데이터베이스 컴포넌트를 사용할 경우에도 *DatabaseName*을 사용하여 데이터베이스 테이블이 있는 디렉토리를 지정할 수 있습니다.
- 세션을 통해 애플리케이션의 데이터베이스 연결 그룹을 전역적으로 관리할 수 있습니다. BDE 활성 데이터셋을 애플리케이션에 추가하면 애플리케이션에 *Session*이라는 세션 컴포넌트가 자동으로 포함됩니다. 데이터베이스와 데이터셋 컴포넌트를 애플리케이션에 추가하면 이 데이터베이스와 데이터셋 컴포넌트가 해당 기본 세션에 자동으로 연결됩니다. 이를 통해 암호가 보호되는 Paradox 파일에 대한 액세스를 제어하고 네트워크 상에서 Paradox 파일을 공유하기 위한 디렉토리 위치를 지정합니다. 세션의 속성, 이벤트 및 메소드를 사용하여 데이터베이스 연결을 제어하고 Paradox 파일에 대한 액세스를 제어할 수 있습니다.

기본 세션을 사용하여 애플리케이션의 모든 데이터베이스 연결을 제어할 수 있습니다. 또 다른 방법으로는 디자인 타임 시 추가 세션 컴포넌트를 추가하거나 런타임 시 추가 세션 컴포넌트를 동적으로 생성하여 애플리케이션에서 데이터베이스 연결의 하위 집합을 제어할 수 있습니다. 사용자의 데이터셋과 명시적으로 생성된 세션 컴포넌트를 연결하려면 *SessionName* 속성을 사용합니다. 애플리케이션의 명시적 세션 컴포넌트를 사용하지 않는 경우 이 속성에 값을 제공하지 않아도 됩니다. 기본 세션을 사용하거나 *SessionName* 속성을 사용하여 세션을 명시적으로 지정하는지에 상관 없이 *DBSession* 속성을 읽어 데이터셋에 연결된 세션에 액세스할 수 있습니다.

참고 세션 컴포넌트를 사용하는 경우 데이터셋의 *SessionName* 속성은 데이터셋이 연결되어 있는 데이터베이스 컴포넌트의 *SessionName* 속성과 일치해야 합니다.

*TDatabase*와 *TSession*에 대한 자세한 내용은 20-13 페이지의 "TDatabase를 갖는 데이터베이스에 연결"과 20-17 페이지의 "데이터베이스 세션 관리"를 참조하십시오.

BLOB 캐시

BDE 활성 데이터셋은 모두 애플리케이션에서 BLOB 레코드를 읽을 때 BLOB 필드가 BDE를 기준으로 로컬로 캐시되는지 여부를 제어하는 *CacheBlobs* 속성을 가집니다. 기본적으로 *CacheBlobs*는 *True*이며 이는 BDE가 BLOB 필드의 로컬 복사본을 캐시한다는 것을 의미합니다. BLOB를 캐시하면 사용자가 레코드를 스크롤하면서 데이터베이스 서버에서 로컬 복사본을 반복적으로 페치(fetch)하는 대신 BDE에 BLOB의 로컬 복사본을 저장할 수 있으므로 애플리케이션 성능이 향상됩니다.

BLOB가 자주 업데이트하거나 바뀌며 BLOB 데이터의 새로 보기가 애플리케이션 성능보다 더 중요한 경우, 애플리케이션에서 항상 최신 버전의 BLOB 필드를 볼 수 있도록 *CacheBlobs*를 *False*로 설정할 수 있습니다.

BDE 핸들 얻기

Borland Database Engine에 API를 직접 호출할 필요 없이 BDE 활성 데이터셋을 사용할 수 있습니다. BDE 활성 데이터셋은 데이터베이스 및 세션 컴포넌트와 함께 BDE 기능 대부분을 캡슐화합니다. 그러나 BDE에 API를 직접 호출해야 하는 경우에는 BDE에서 관리되는 리소스용 BDE 핸들이 필요합니다. 대부분의 BDE API는 매개변수로 이러한 핸들을 필요로 합니다.

모든 BDE 활성 데이터셋은 런타임 시 BDE 핸들을 액세스하기 위한 세 개의 읽기 전용 속성을 포함합니다.

- *Handle*은 데이터셋의 레코드에 액세스하는 BDE 커서에 대한 핸들입니다.
- *DBHandle*은 원본으로 사용하는 테이블 또는 내장 프로시저를 포함하는 데이터베이스에 대한 핸들입니다.
- *DBLocale*은 데이터셋의 BDE 랭귀지 드라이버에 대한 핸들입니다. 로케일은 문자열 데이터에 사용되는 정렬 순서와 문자 집합을 제어합니다.

이 속성들은 BDE를 통해 데이터베이스 서버로 연결될 때 데이터셋에 자동으로 지정됩니다. BDE API에 대한 자세한 내용은 BDE32.HLP 온라인 도움말 파일을 참조하십시오.

TTable 사용

*TTable*은 원본으로 사용하는 데이터베이스 테이블의 전체 구조 및 데이터를 캡슐화합니다. *TTable*은 *TDataSet*에 의해 초기화된 모든 기본 기능은 물론 테이블 타입 데이터셋의 일반적인 모든 특수 기능을 구현합니다. *TTable*에 의해 초기화되는 고유 기능을 살펴 보기 전에 18-25 페이지에서 시작하는 테이블 타입 데이터셋에 대한 단원과 "데이터셋 이해"에서 설명한 일반 데이터베이스 기능을 잘 알고 있어야 합니다.

*TTable*은 BDE 활성 데이터셋이므로 데이터베이스와 세션에 연결되어야 합니다. 20-3 페이지의 "데이터베이스 및 세션 연결과 데이터셋의 연결"에서 이러한 연결 형성 방법에 대해 설명합니다. 데이터셋이 데이터베이스와 세션에 연결되면 *TableName* 속성을 설정하여 이 데이터셋을 특정 데이터베이스 테이블에 바인드할 수 있으며 Paradox, dBASE, FoxPro 또는 쉘표로 구분된 ASCII 텍스트 테이블을 사용하는 경우에는 *TableType* 속성을 설정합니다.

참고 데이터베이스, 세션 또는 데이터베이스 테이블에 대한 해당 연결을 변경하거나 *TableType* 속성을 설정할 경우에는 테이블을 닫아야 합니다. 하지만 테이블을 닫고 이 속성들을 변경하기 전에 먼저 대기 중인 변경을 모두 게시하거나 삭제합니다. 캐시된 업데이트가 활성화되어 있는 경우 *ApplyUpdates* 메소드를 호출하여 데이터베이스의 게시된 변경을 기록합니다.

TTable 컴포넌트는 로컬 데이터베이스 테이블(Paradox, dBASE, FoxPro 및 쉘표로 구분된 ASCII 텍스트 테이블)에 제공하는 지원에 있어서 특이합니다. 다음 항목에서는 이러한 지원을 구현하는 특수한 속성과 메소드에 대해 설명합니다.

또한 *TTable* 컴포넌트는 배치 작업의 BDE 지원을 활용합니다(전체 레코드 그룹을 추가, 업데이트, 삭제 또는 복사하는 테이블 레벨 작업). 이러한 지원 내용은 20-8 페이지의 "다른 테이블에서 데이터 가져오기"에 설명되어 있습니다.

로컬 테이블의 테이블 타입 지정

애플리케이션에서 Paradox, dBASE, FoxPro 또는 쉘표로 구분된 ASCII 텍스트 테이블에 액세스하는 경우 BDE는 *TableType* 속성을 사용하여 테이블 타입(테이블의 예상 구조)을 결정합니다. *TTable*이 데이터베이스 서버에서 SQL 기반 테이블을 나타낼 때 *TableType*은 사용하지 않습니다.

기본적으로 *TableType*은 *ttDefault*로 설정됩니다. *TableType*이 *ttDefault*일 때 BDE는 파일 이름 확장자로부터 테이블 타입을 결정합니다. 표 20.1은 BDE에 의해 인식되는 파일 확장자와 테이블 타입에 대한 가정을 요약한 것입니다.

표 20.1 파일 확장자에 기반한 BDE 에 의해 인식되는 테이블 타입

확장자	테이블 타입
파일 확장자 없음	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	ASCII 텍스트

로컬 Paradox, dBASE 및 ASCII 텍스트 테이블이 표 20.1에서 설명된 대로 파일 확장자를 사용하는 경우 *TableType*이 *ttDefault*로 설정된 채로 유지할 수 있습니다. 그렇지 않은 경우에는 정확한 테이블 타입을 나타내도록 *TableType*을 설정해야 합니다. 표 20.2는 *TableType*에 지정할 수 있는 값을 나타냅니다.

표 20.2 TableType 값

값	테이블 타입
ttDefault	BDE에 의해 자동으로 결정되는 테이블 타입
ttParadox	Paradox
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	컴표로 구분된 ASCII 텍스트

로컬 테이블에 대한 읽기/쓰기 액세스 제어

테이블 타입 데이터셋과 마찬가지로 *TTable*을 통해 *ReadOnly* 속성을 사용하여 애플리케이션에서 읽기 및 쓰기 액세스를 제어할 수 있습니다.

뿐만 아니라 Paradox, dBASE 및 FoxPro 테이블에서 *TTable*을 통해 다른 애플리케이션의 테이블에 대한 읽기 및 쓰기 액세스를 제어할 수도 있습니다. *Exclusive* 속성은 애플리케이션이 Paradox, dBASE 또는 FoxPro 테이블에 대한 고유 읽기/쓰기 액세스를 얻을지 여부를 제어합니다. 이 테이블 타입에 대한 고유 읽기/쓰기 액세스를 얻으려면 테이블을 열기 전에 테이블 컴포넌트의 *Exclusive* 속성을 *True*로 설정합니다. 배타적 액세스로 테이블을 여는 경우에는 다른 애플리케이션에서 테이블로부터 데이터를 읽거나 테이블에 데이터를 쓸 수 없습니다. 테이블이 이미 사용 중일 때 테이블을 열려고 시도하면 배타적 액세스에 대한 요청이 허용되지 않습니다.

다음 문장은 배타적 액세스로 테이블을 엽니다.

```
CustomersTable.Exclusive := True; {Set request for exclusive lock}
CustomersTable.Active := True; {Now open the table}
```

참고 SQL 테이블에 *Exclusive*를 설정할 수는 있지만 일부 서버는 배타적 테이블 레벨의 잠금 기능을 지원하지 않습니다. 그 외의 서버는 배타적 잠금 기능을 제공하지만 다른 애플리케이션에서 테이블로부터 데이터를 읽는 것을 허용합니다. 서버에 있는 데이터베이스 테이블의 배타적 잠금에 대한 자세한 내용은 해당 서버 설명서를 참조하십시오.

dBASE 인덱스 파일 지정

대부분의 서버에서 모든 데이터 타입 데이터셋에 일반적인 메소드를 사용하여 인덱스를 지정합니다. 이러한 메소드는 18-26 페이지의 "인덱스로 레코드 정렬"에서 설명합니다.

비생성 인덱스 파일 또는 dBASE III PLUS-style 인덱스 (*.NDX)를 사용하는 dBASE 테이블에서는 *IndexFiles*와 *IndexName* 속성을 대신 사용해야 합니다. *IndexFiles* 속성을 비생성 인덱스 파일의 이름으로 설정하거나 .NDX 파일을 나열합니다. 그런 다음 데이터셋을 능동적으로 정렬할 *IndexName* 속성의 인덱스 하나를 지정합니다.

디자인 타임 시 Object Inspector에서 *IndexFiles* 속성 값의 생략 버튼을 클릭하여 Index Files 편집기를 호출합니다. 다음과 같은 방법으로 비생성 인덱스 파일 또는 .NDX 파일을 추가합니다. Index Files 대화 상자의 Add 버튼을 클릭하고 Open 대화 상자에서 파일을 선택합니다. 각 비생성 인덱스 파일 또는 .NDX 파일마다 이 프로세스를 반복합니다. 원하는 인덱스를 모두 추가한 후 Index Files 대화 상자의 OK 버튼을 클릭합니다.

런타임 시 이와 동일한 작업을 프로그래밍 방식으로 수행할 수 있습니다. 이 작업을 수행하려면 문자열 목록의 속성과 메소드를 사용하여 *IndexFiles* 속성에 액세스합니다. 새로운 인덱스 집합을 추가할 때 먼저 테이블의 *IndexFiles* 속성의 *Clear* 메소드를 호출하여 기존 항목을 제거합니다. *Add* 메소드를 호출하여 각각의 비생성 인덱스 파일 또는 .NDX 파일을 추가합니다.

```
with Table2.IndexFiles do begin
  Clear;
  Add('Bystate.ndx');
  Add('Byzip.ndx');
  Add('Fullname.ndx');
  Add('St_name.ndx');
end;
```

원하는 비생성 인덱스 파일이나 .NDX 인덱스 파일을 추가하고 나면 인덱스 파일의 각 인덱스 이름들을 사용할 수 있고 *IndexName* 속성에 할당할 수 있습니다. *GetIndexNames* 메소드 사용 시 또는 *IndexDefs* 속성의 *TIndexDef* 객체를 통해 인덱스 정의 검사 시 인덱스 태그가 나열됩니다. 제대로 나열된 .NDX 파일은 지정된 인덱스가 *IndexName* 속성에서 사용되는지 여부에 상관 없이 테이블에서 데이터가 추가, 변경 또는 삭제될 때 자동으로 업데이트됩니다.

아래 예제에서 *AnimalsTable* 테이블 컴포넌트의 *IndexFiles*는 비생성 인덱스 파일 ANIMALS.MDX로 설정되며 그 *IndexName* 속성은 "NAME"이라는 인덱스 태그로 설정됩니다.

```
AnimalsTable.IndexFiles.Add('ANIMALS.MDX');
AnimalsTable.IndexName := 'NAME';
```

인덱스 파일을 지정한 후 비생성 인덱스 또는 .NDX 인덱스를 사용하면 다른 인덱스와 마찬가지로 작동합니다. 인덱스 이름을 지정하면 테이블에서 데이터가 정렬되고 인덱스 기반 검색, 범위 및 비생성 인덱스 마스터 디테일 연결이 가능해집니다. 이러한 인덱스 사용에 대한 자세한 내용은 18-25 페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오.

TTable 컴포넌트가 있는 dBASE III PLUS-style .NDX 인덱스를 사용할 때 고려할 특수 사항이 두 가지 있습니다. 첫째, .NDX 파일은 마스터/디테일 연결의 기준으로 사용할 수 없습니다. 둘째, *IndexName* 속성이 있는 .NDX 인덱스를 활성화할 때 .NDX 확장자를 인덱스 이름의 일부로 속성 값에 포함시켜야 합니다.

```
with Table1 do begin
  IndexName := 'ByState.NDX';
  FindKey(['CA']);
end;
```

로컬 테이블 이름 재지정

디자인 타임 시 Paradox 또는 dBASE 테이블의 이름을 재지정하려면 테이블 컴포넌트를 마우스 오른쪽 버튼으로 클릭하고 컨텍스트 메뉴에서 Rename Table을 선택합니다.

런타임 시 Paradox 또는 dBASE 테이블의 이름을 재지정하려면 테이블의 *RenameTable* 메소드를 호출합니다. 예를 들어, 다음 문장은 Customer 테이블의 이름을 CustInfo:로 재지정합니다.

```
Customer.RenameTable('CustInfo');
```

다른 테이블에서 데이터 가져오기

테이블 컴포넌트의 *BatchMove* 메소드를 사용하여 다른 테이블에서 데이터를 가져올 수 있습니다. *BatchMove*는 다음을 수행할 수 있습니다.

- 다른 테이블에서 이 테이블로 레코드를 복사합니다.
- 다른 테이블에서 발생하는 레코드를 이 테이블에서 업데이트합니다.
- 다른 테이블의 레코드를 이 테이블의 끝에 추가합니다.
- 다른 테이블에서 발생하는 레코드를 이 테이블에서 삭제합니다.

*BatchMove*는 두 개의 매개변수를 사용하며 데이터를 가져올 테이블 이름과 수행할 가져오기 작업을 결정하는 모드 지정이 이에 해당됩니다. 표 20.3은 모드를 지정하기 위한 가능한 설정에 대해 설명한 것입니다.

표 20.3 BatchMove 의 import 모드

값	의미
batAppend	소스 테이블의 모든 레코드를 이 테이블의 끝에 추가합니다.
batAppendUpdate	소스 테이블의 모든 레코드를 이 테이블의 끝에 추가하고 이 테이블의 기존 레코드를 소스 테이블의 일치하는 레코드로 업데이트합니다.
batCopy	소스 테이블의 모든 레코드를 이 테이블에 복사합니다.
batDelete	소스 테이블에도 나타나는 이 테이블의 모든 레코드를 삭제합니다.
batUpdate	이 테이블의 기존 레코드를 소스 테이블의 일치하는 레코드로 업데이트합니다.

예를 들어, 다음 코드는 현재 테이블의 모든 레코드를 현재 인덱스에 있는 필드의 동일한 값을 가지는 *Customer* 테이블의 레코드로 업데이트합니다.

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

*BatchMove*는 성공적으로 import한 레코드 수를 반환합니다.

주의 *batCopy* 모드를 사용하여 레코드를 import하면 기존 레코드를 겹쳐 씁니다. 기존 레코드를 유지하려면 *batAppend*를 대신 사용합니다.

*BatchMove*는 BDE에 의해 지원되는 배치 작업 중 일부만 수행합니다. 추가 기능은 *TBatchMove* 컴포넌트를 통해 사용할 수 있습니다. 테이블 간에 대량의 데이터를 이동해야 하는 경우 테이블의 *BatchMove* 메소드를 호출하는 대신 *TBatchMove*를 사용합니다. *TBatchMove* 사용에 대한 내용은 20-49 페이지의 "TBatchMove 사용"을 참조하십시오.

TQuery 사용

*TQuery*는 단일 DDL(Data Definition Language) 또는 DML(Data Manipulation Language) 문(예: SELECT, INSERT, DELETE, UPDATE, CREATE INDEX 또는 ALTER TABLE 명령)을 나타냅니다. 명령에 사용되는 랭귀지는 서버 특정적이지만 일반적으로 SQL 랭귀지의 SQL-92 표준과 호환됩니다. *TQuery*는 *TDataSet*에 의해 초기화되는 모든 기본 기능뿐만 아니라 쿼리 타입 데이터셋의 일반적인 모든 특수 기능도 구현합니다. *TQuery*에 의해 초기화된 고유 기능을 살펴 보기 전에 18-42 페이지에서 시작하는 쿼리 타입 데이터셋에 대한 단원과 "데이터셋 이해"에서 설명한 일반 데이터베이스 기능을 잘 알고 있어야 합니다.

*TQuery*는 BDE 활성 데이터셋이므로 데이터베이스와 세션에 연결되어 있어야 합니다. (이중 쿼리에 *TQuery*를 사용하는 경우는 예외입니다.) 20-3 페이지의 "데이터베이스 및 세션 연결과 데이터셋의 연결"에서는 이러한 연결 형성하는 방법을 설명합니다. SQL 속성을 설정하여 쿼리에 대한 SQL 문을 지정합니다.

TQuery 컴포넌트는 다음과 같이 데이터에 액세스할 수 있습니다.

- BDE의 일부인 로컬 SQL을 통해 Paradox 또는 dBASE 테이블에서 데이터에 액세스합니다. 로컬 SQL은 SQL-92 지정의 하위 집합입니다. 대부분의 DML이 지원되며 많은 DDL 구문이 이러한 테이블 타입을 사용합니다. 지원되는 SQL 구문에 대한 자세한 내용은 로컬 SQL 도움말, LOCALSQL.HLP를 참조하십시오.
- InterBase 엔진을 통해 Local InterBase Server 데이터베이스에서 데이터에 액세스합니다. InterBase의 SQL-92 표준 SQL 구문 지원과 확장 구문 지원에 대한 내용은 InterBase *Language Reference*를 참조하십시오.
- Oracle, Sybase, MS-SQL Server, Informix, DB2 및 InterBase 등 원격 데이터베이스 서버의 데이터베이스에서 데이터에 액세스합니다. 적절한 SQL Link 드라이버와 데이터베이스 서버에 고유한 클라이언트 소프트웨어(공급업체 제공)를 설치하여 원격 서버에 액세스해야 합니다. 이러한 서버에서 지원하는 표준 SQL 구문은 허용됩니다. SQL 구문, 제한 및 확장에 대한 내용은 해당 서버의 설명서를 참조하십시오.

이중 쿼리 생성

*TQuery*는 두 개 이상의 서버 또는 Oracle 테이블과 Paradox 테이블의 데이터와 같은 테이블 타입에 대한 이중 쿼리를 지원합니다. 이중 쿼리를 실행할 때 BDE는 로컬 SQL을 사용하여 쿼리를 분석하고 처리합니다. BDE는 로컬 SQL을 사용하기 때문에 확장적, 서버 특정적 SQL 구문은 지원되지 않습니다.

다음 단계에 따라 이중 쿼리를 수행합니다.

- 1 BDE 관리 도구 또는 SQL 탐색기를 사용하여 쿼리에서 액세스한 각 데이터베이스에서 별도의 BDE 별칭을 정의합니다.
- 2 *TQuery*의 *DatabaseName* 속성을 공백으로 두면 사용한 데이터베이스의 이름이 SQL 문에서 지정됩니다.
- 3 SQL 속성에서 실행할 SQL 문을 지정합니다. 문의 각 테이블 이름 앞에 테이블 데이터베이스의 BDE 별칭을 두고 별칭 앞뒤에 콜론(:)을 넣습니다. 그런 다음 이 전체 참조를 인용 부호로 묶습니다.
- 4 *Params* 속성에서 쿼리의 매개변수를 설정합니다.
- 5 *Prepare*를 호출하여 첫 실행에 앞서 쿼리 실행을 준비합니다.
- 6 실행할 쿼리 타입에 따라 *Open* 또는 *ExecSQL*을 호출합니다.

예를 들어, CUSTOMER 테이블을 가지는 Oracle 데이터베이스용 *Oracle1*이라는 별칭과 ORDERS 테이블을 가지는 Sybase 데이터베이스용 *Sybase1*이라는 별칭을 정의한다고 가정합니다. 이러한 두 테이블에 대한 간단한 쿼리는 다음과 같습니다.

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
JOIN ":Sybase1:ORDERS"
ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

이중 쿼리의 데이터베이스를 지정하기 위해 BDE 별칭을 사용하는 대신 *TDatabase* 컴포넌트를 사용할 수 있습니다. *TDatabase*를 보통으로 구성하여 데이터베이스를 가리키고 *TDatabase*를 설정합니다. *DatabaseName*은 고유한 임의 값으로 설정한 다음 BDE 별칭 대신 SQL 문에서 이 값을 사용합니다.

편집 가능한 결과 집합(Result set) 얻기

사용자가 data-aware 컨트롤에서 편집할 수 있는 결과 집합을 요청하려면 쿼리 컴포넌트의 *RequestLive* 속성을 *True*로 설정합니다. *RequestLive*를 *True*로 설정하는 것이 라이브 결과 집합을 보장하지는 않지만 BDE는 가능할 때마다 요청을 받아들이고자 합니다. 쿼리에서 로컬 SQL 파서 또는 서버의 SQL 파서를 사용할지 여부에 따라 라이브 결과 집합 요청에 대해 몇 가지 제한 사항이 있습니다.

- 이중 쿼리에서처럼 테이블 이름 앞에 BDE 데이터베이스 별칭이 오는 쿼리와 Paradox 또는 dBASE에 대해 실행되는 쿼리는 로컬 SQL을 사용하여 BDE에 의해 분석됩니다. 쿼리에서 로컬 SQL 파서를 사용할 때 BDE는 단일 테이블과 다중 테이블 쿼리의 업데이트 가능한 라이브 결과 집합에 대한 확장된 지원을 제공합니다. 로컬 SQL 사용 시 쿼리에 다음 중 어떤 것도 포함되지 않은 경우 단일 테이블이나 뷰에 대한 쿼리의 라이브 결과 집합이 반환됩니다.
 - SELECT 문의 DISTINCT 절
 - 조인(내부, 외부 또는 UNION)
 - GROUP BY 또는 HAVING 절을 포함하거나 포함하지 않는 집계 함수
 - 업데이트할 수 없는 기준 테이블 또는 뷰
 - 하위 집합
 - 인덱스에 기반하지 않는 ORDER BY 절

- 원격 데이터베이스 서버에 대한 쿼리는 서버에서 분석됩니다. *RequestLive* 속성이 *True*로 설정되어 있는 경우, BDE에서 테이블에 데이터 변경 내용을 전달하는 데이터 속성을 사용하므로 SQL 문은 로컬 SQL 표준과 서버 특정 제한 사항을 따라야 합니다. 쿼리에 다음 중 어떤 것도 포함되지 않은 경우 단일 테이블이나 뷰에 대한 쿼리의 라이브 결과 집합이 반환됩니다.
 - SELECT 문의 DISTINCT 절
 - GROUP BY 또는 HAVING 절을 포함하거나 포함하지 않는 집계 함수
 - 두 개 이상의 기준 테이블 또는 업데이트 가능한 뷰에 대한 참조(조인)
 - FROM 절의 테이블 또는 다른 테이블을 참조하는 하위 쿼리

애플리케이션에서 라이브 결과 집합을 요청하고 수신하는 경우 쿼리 컴포넌트의 *CanModify* 속성은 *True*로 설정됩니다. 쿼리에서 라이브 결과 집합을 반환해도 쿼리에 연결된 필드가 있거나 업데이트를 시도하기 전에 인덱스를 전환하는 경우에는 결과 집합을 직접 업데이트할 수 없습니다. 이러한 경우 결과 집합을 읽기 전용 결과 집합으로 취급하여 그에 맞게 업데이트합니다.

애플리케이션에서 라이브 결과 집합을 요청하지만 SELECT 문 구문에서 이를 허용하지 않는 경우 BDE는 다음 중 하나를 반환합니다.

- Paradox 또는 dBASE에 대해 생성된 쿼리의 읽기 전용 결과 집합.
- 원격 서버에 대해 생성된 SQL 쿼리의 오류 코드.

읽기 전용 결과 집합 업데이트

애플리케이션에서 캐시된 업데이트를 사용하는 경우, 읽기 전용 결과 집합에서 반환된 데이터를 업데이트할 수 있습니다.

클라이언트 데이터셋을 사용하여 업데이트를 캐시하는 경우, 클라이언트 데이터셋 또는 그 관련된 프로바이더는 쿼리에서 다중 테이블을 나타내지 않는 한 업데이트를 적용하기 위해 SQL을 자동으로 생성할 수 있습니다. 쿼리에서 다중 테이블을 나타내는 경우 업데이트 적용 방법을 표시해야 합니다.

- 모든 업데이트 내용이 단일 데이터베이스 테이블에 적용되는 경우 *OnGetTableName* 이벤트 핸들러에서 업데이트할 원본 테이블을 표시할 수 있습니다.
- 업데이트 적용에 추가 제어가 필요한 경우 쿼리를 update object (*TUpdateSQL*)와 연결할 수 있습니다. 프로바이더는 이 업데이트 객체를 자동으로 사용하여 업데이트를 적용합니다.
 - 1 쿼리의 *UpdateObject* 속성을 사용 중인 *TUpdateSQL* 객체로 설정하여 업데이트 객체를 쿼리에 연결합니다.
 - 2 업데이트 객체의 *ModifySQL*, *InsertSQL* 및 *DeleteSQL* 속성을 쿼리 데이터에 적합한 업데이트를 수행하는 SQL 문으로 설정합니다.

BDE를 사용하여 업데이트를 캐시하는 경우 업데이트 객체를 사용해야 합니다.

참고 업데이트 객체 사용에 대한 자세한 내용은 20-40 페이지의 "업데이트 객체를 사용하여 데이터셋 업데이트"를 참조하십시오.

TStoredProc 사용

*TStoredProc*는 내장 프로시저를 나타냅니다. *TStoredProc*는 *TDataSet*에 의해 초기화되는 모든 기본 기능뿐만 아니라 내장 프로시저 타입 데이터셋의 일반적인 대부분의 특수 기능을 구현합니다. *TStoredProc*에 의해 초기화되는 고유 기능을 살펴 보기 전에 18-50 페이지에서 시작하는 내장 프로시저 타입 데이터셋에 대한 단원과 "데이터셋 이해"에서 설명한 일반적인 데이터베이스 기능을 잘 알고 있어야 합니다.

*TStoredProc*는 BDE 활성 데이터셋이므로 데이터베이스와 세션에 연결되어 있어야 합니다. 20-3 페이지의 "데이터베이스 및 세션 연결과 데이터셋의 연결"에서 이러한 연결 형성 방법에 대해 설명합니다. 데이터셋이 데이터베이스와 세션에 연결되면 *StoredProcName* 속성을 설정하여 이 데이터셋을 특정 내장 프로시저에 바인딩할 수 있습니다.

*TStoredProc*는 다음과 같은 면에서 다른 내장 프로시저 타입 데이터셋과 다릅니다.

- *TStoredProc*는 매개변수 바인딩 방법에 대해 제어할 수 있게 해줍니다.
- 또한 Oracle 오버로드 내장 프로시저에 대한 지원을 제공합니다.

매개변수 바인딩

내장 프로시저를 준비하고 실행할 때 그 입력 매개변수는 서버의 매개변수에 자동으로 바인딩됩니다.

*TStoredProc*를 통해 *ParamBindMode* 속성을 사용하여 매개변수를 서버의 매개변수에 바인딩하는 방법을 지정할 수 있습니다. 기본적으로 *ParamBindMode*는 *pbByName*으로 설정되므로 내장 프로시저 컴포넌트의 매개변수는 서버의 매개변수와 이름이 일치합니다. 이 방법은 가장 쉬운 매개변수 바인딩 메소드입니다.

일부 서버는 매개변수가 내장 프로시저에 나타나는 순서인 순서 값 기준의 매개변수 바인딩을 지원하기도 합니다. 이 경우 매개변수 컬렉션 편집기의 매개변수를 지정하는 순서는 의미를 가집니다. 지정한 첫 번째 매개변수는 서버의 첫 번째 입력 매개변수에 상응하고, 두 번째 매개변수는 서버의 두 번째 입력 매개변수에 상응합니다. 서버가 순서 값 기준의 매개변수 바인딩을 지원하는 경우 *ParamBindMode*를 *pbByNumber*로 설정할 수 있습니다.

- 팁** *ParamBindMode*를 *pbByNumber*로 설정하려면 올바른 매개변수 타입을 올바른 순서로 지정해야 합니다. SQL 탐색기에서 서버의 내장 프로시저 소스 코드를 보고 지정할 매개변수의 올바른 순서와 타입을 확인할 수 있습니다.

Oracle 오버로드 내장 프로시저 사용

Oracle 서버를 통해 내장 프로시저를 오버로드할 수 있습니다. 오버로드된 프로시저는 동일한 이름을 갖는 다른 프로시저입니다. 내장 프로시저 컴포넌트의 *Overload* 속성을 통해 애플리케이션에서 실행할 프로시저를 지정할 수 있습니다.

*Overload*가 0(기본값)이면 오버로딩이 없는 것으로 간주됩니다. *Overload*가 1이면 내장 프로시저 컴포넌트는 오버로드된 이름을 갖는 Oracle 서버에 있는 첫 번째 내장 프로시저를 실행합니다. *Overload*가 2이면 두 번째 내장 프로시저를 실행합니다.

참고 오버로드된 내장 프로시저는 다른 입력 및 출력 매개변수를 취할 수도 있습니다. 자세한 내용은 Oracle 서버 설명서를 참조하십시오.

TDatabase를 갖는 데이터베이스에 연결

Delphi 애플리케이션에서 BDE(Borland Database Engine)를 사용하여 데이터베이스에 연결할 때 그 연결은 *TDatabase* 컴포넌트에 의해 캡슐화됩니다. 데이터베이스 컴포넌트는 BDE 세션의 컨텍스트로 단일 데이터베이스에 대한 연결을 나타냅니다.

*TDatabase*는 다양한 공통적인 속성, 메소드 및 이벤트와 같은 동일한 작업을 많이 수행하고 이를 다른 데이터베이스 연결 컴포넌트와 공유합니다. 이러한 공통적 내용은 17장 "데이터베이스에 연결"에서 설명합니다.

공통적인 속성, 메소드 및 이벤트 외에도 *TDatabase*는 다양한 BDE 특정 기능을 제공합니다. 이러한 기능은 다음 항목에서 설명합니다.

데이터베이스 컴포넌트와 세션 연결

모든 데이터베이스 컴포넌트는 BDE 세션과 연결되어야 합니다. *SessionName*을 사용하고 이 연결을 설정합니다. 디자인 타임 시 처음으로 데이터베이스 컴포넌트를 만들 때 *SessionName*이 "기본값"으로 설정되어 있으면 전역 *Session* 변수가 참조하는 기본 세션 컴포넌트와 연결되어 있다는 것을 의미합니다.

다중 스레드 또는 재진입(reentrant) BDE 애플리케이션이 세션을 두 개 이상 필요로 할 수 있습니다. 다중 세션을 사용해야 하는 경우 각 세션에서 *TSession* 컴포넌트를 추가합니다. 그런 다음 *SessionName* 속성을 세션 컴포넌트의 *SessionName* 속성에 설정하여 데이터셋을 세션 컴포넌트에 연결합니다.

런타임 시 *Session* 속성을 읽어서 데이터베이스가 연결된 세션 컴포넌트에 액세스할 수 있습니다. *SessionName*이 공백이거나 "기본값"인 경우 *Session* 속성은 전역 *Session* 변수가 참조하는 동일한 *TSession* 인스턴스를 참조합니다. *Session*을 통해 세션의 실제 이름을 알지 못해도 애플리케이션에서 데이터베이스 컴포넌트의 부모 세션 컴포넌트의 속성, 메소드 및 이벤트에 액세스할 수 있습니다.

BDE 세션에 대한 자세한 내용은 20-17 페이지의 "데이터베이스 세션 관리"를 참조하십시오.

암시적 데이터베이스 컴포넌트를 사용하는 경우 그 데이터베이스 컴포넌트의 세션은 데이터셋의 *SessionName* 속성에 의해 지정된 세션입니다.

데이터베이스와 세션 컴포넌트 상호 작용 이해

일반적으로 세션 컴포넌트 속성은 런타임 시 생성된 모든 암시적 데이터베이스 컴포넌트에 적용되는 전역 기본 동작을 제공합니다. 예를 들어, 제어 세션의 *KeepConnections* 속성은 그 연결된 데이터셋이 닫혀 있거나(기본값) 해당 데이터셋이 모두 닫혀있을 때 연결이 끊어질 경우에도 데이터베이스 연결이 유지되는지 여부를 결정합니다. 마찬가지로 세션의 기본 *OnPassword* 이벤트는 애플리케이션에서 암호를 필요로 하는 서버의 데이터베이스에 연결을 시도할 때 표준 암호를 요구하는 대화 상자를 표시하도록 합니다.

Session 메소드는 다소 다르게 적용됩니다. *TSession* 메소드는 명시적으로 생성되었는지 또는 데이터셋에 의해 암시적으로 인스턴스화되었는지 여부에 상관 없이 모든 데이터베이스 컴포넌트에 영향을 미칩니다. 예를 들어, 세션 메소드 *DropConnections*는 각 데이터베이스 컴포넌트의 *KeepConnection* 속성이 *True*인 경우에도 세션의 데이터베이스 컴포넌트에 속하는 모든 데이터셋을 닫은 다음 모든 데이터베이스 연결을 끊습니다.

데이터베이스 컴포넌트 메소드는 지정된 데이터베이스 컴포넌트에 연결된 데이터셋에만 적용됩니다. 예를 들어, 데이터베이스 컴포넌트 *Database1*이 기본 세션과 연결되어 있다고 가정합니다. *Database1.CloseDataSets()*는 *Database1*에 연결된 데이터셋만 닫습니다. 기본 세션 내의 다른 데이터베이스 컴포넌트에 속하는 개방 데이터셋은 열린 채로 유지됩니다.

데이터베이스 식별

*AliasName*과 *DriverName*은 *TDatabase* 컴포넌트가 연결된 데이터베이스 서버를 식별하는 상호 배타적인 속성입니다.

- *AliasName*은 기존 BDE 별칭의 이름을 지정하여 데이터베이스 컴포넌트에 사용합니다. 별칭이 데이터셋 컴포넌트의 연속 드롭다운 목록에 나타나므로 이러한 별칭을 특정 데이터베이스 컴포넌트에 연결할 수 있습니다. 데이터베이스 컴포넌트의 *AliasName*을 지정하는 경우 드라이버 이름은 항상 BDE 별칭의 일부이므로 *DriverName*에 이미 할당된 값이 지워집니다.

Database Explorer 또는 BDE Administration 유틸리티를 사용하여 BDE 별칭을 생성하고 편집합니다. BDE 별칭 생성과 유지에 대한 자세한 내용은 이러한 유틸리티의 온라인 설명서를 참조하십시오.

- *DriverName*은 BDE 드라이버의 이름입니다. 드라이버 이름은 BDE 별칭의 한 매개 변수이지만 *DatabaseName* 속성을 사용하여 데이터베이스 컴포넌트의 로컬 BDE 별칭 생성 시 별칭 대신 드라이버 이름을 지정할 수도 있습니다. *DriverName*을 지정하는 경우 *AliasName*에 이미 할당된 값은 지정한 드라이버 이름과 *AliasName*에서 식별되는 BDE 별칭의 일부인 드라이버 이름 사이의 잠재적 충돌을 방지하기 위해 지워집니다.

*DatabaseName*을 통해 데이터베이스 연결에 대한 사용자 고유의 이름을 제공할 수 있습니다. 제공하는 이름은 *AliasName* 또는 *DriverName*이며 애플리케이션에 대해 로컬입니다. *DatabaseName*은 BDE 별칭이거나 Paradox 및 dBASE 파일에서는 전체 경로 이름입니다. *AliasName*과 마찬가지로 *DatabaseName*은 데이터셋 컴포넌트의 연속 드롭다운 목록에 나타나므로 이러한 이름을 데이터베이스 컴포넌트에 연결할 수 있습니다.

디자인 타임 시 BDE 별칭을 지정하려면 BDE 드라이버를 할당하거나 로컬 BDE 별칭을 만들고 데이터베이스 컴포넌트를 더블 클릭하여 Database Properties 에디터를 호출합니다.

속성 편집기의 Name 편집 상자에 *DatabaseName*을 입력할 수 있습니다. *Alias* 속성의 Alias 이름 콤보 상자에 기존 BDE 별칭 이름을 입력하거나 드롭다운 목록의 기존 별칭에서 선택할 수 있습니다. Driver 이름 콤보 상자에서 *DriverName* 속성에 대한 기존 BDE 드라이버 이름을 입력하거나 드롭다운 목록의 기존 드라이버 이름에서 선택할 수 있습니다.

참고 또한 Database Properties 에디터를 통해 BDE 연결 매개변수를 보고 설정한 후 *LoginPrompt* 및 *KeepConnection* 속성의 상태를 설정할 수도 있습니다. 연결 매개변수에 대한 내용은 아래의 "BDE 별칭 매개변수 설정"을 참조하십시오. *LoginPrompt*에 대한 내용은 17-4 페이지의 "서버 로그인 제어"를 참조하십시오. *KeepConnection*에 대한 내용은 20-15 페이지의 "TDatabase를 사용하여 연결 열기"를 참조하십시오.

BDE 별칭 매개변수 설정

디자인 타임 시 다음 세 가지 방법으로 연결 매개변수를 생성하거나 편집할 수 있습니다.

- Database Explorer 또는 BDE Administration 유틸리티를 사용하여 매개변수를 포함하여 BDE 별칭을 생성하거나 수정합니다. 이러한 유틸리티에 대한 자세한 내용은 해당 온라인 도움말 파일을 참조하십시오.
- Object Inspector의 *Params* 속성을 더블 클릭하여 String List 편집기를 호출합니다.
- 데이터 모듈 또는 폼의 데이터베이스 컴포넌트를 더블 클릭하여 Database Properties 에디터를 호출합니다.

이러한 모든 메소드는 데이터베이스 컴포넌트의 *Params* 속성을 편집합니다. *Params* 는 데이터베이스 컴포넌트와 연결된 BDE 별칭의 데이터베이스 연결 매개변수를 포함하는 문자열 목록입니다. 몇 가지 일반적인 연결 매개변수로는 경로 문, 서버 이름, 스키마 캐싱 크기, 랜커지 드라이버 및 SQL 쿼리 모드가 있습니다.

처음으로 Database Properties 에디터를 호출할 때 BDE 별칭의 매개변수는 보이지 않습니다. 현재 설정을 보려면 Defaults를 클릭합니다. Parameter 오버라이드 메모 상자에 현재 매개변수가 표시됩니다. 기존 항목을 편집하거나 새 항목을 추가할 수 있습니다. 기존 매개변수를 지우려면 Clear를 클릭합니다. OK를 클릭하면 변경 내용이 적용됩니다.

런타임 시 애플리케이션에서 *Params* 속성을 직접 편집해서만 별칭 매개변수를 설정할 수 있습니다. BDE를 갖는 SQL 연결 드라이버 사용 고유의 매개변수에 대한 자세한 내용은 온라인 SQL 연결 도움말 파일을 참조하십시오.

TDatabase를 사용하여 연결 열기

모든 데이터베이스 연결 컴포넌트에서 *TDatabase*를 사용하여 데이터베이스에 연결하려면 *Connected* 속성을 *True*로 설정하거나 *Open* 메소드를 호출합니다. 이 프로세스는 17-3 페이지의 "데이터베이스 서버에 연결"에서 다룹니다. 데이터베이스가 일단 연결 되면 적어도 하나의 활성 데이터셋이 있는 한 연결이 유지됩니다. 더 이상 활성 데이터셋이 없으면 데이터베이스 컴포넌트의 *KeepConnection* 속성이 *True*가 아닌 경우 삭제됩니다.

애플리케이션의 원격 데이터베이스 서버로 연결할 때 애플리케이션에서 BDE와 Borland SQL 연결 드라이버를 사용하여 연결합니다. (BDE는 제공하는 ODBC 드라이버와 통신할 수도 있습니다.) 연결하기 전에 애플리케이션의 SQL 연결 또는 ODBC 드라이버를 구성해야 합니다. SQL 연결과 ODBC 매개변수는 데이터베이스 컴포넌트의 *Params* 속성에 저장됩니다. SQL 연결 매개변수에 대한 내용은 온라인 *SQL Links User's Guide*를 참조하십시오. *Params* 속성을 편집하려면 20-15 페이지의 "BDE 별칭 매개변수 설정"을 참조하십시오.

네트워크 프로토콜 사용

적절한 SQL 연결 또는 ODBC 드라이버 구성 시 드라이버의 구성 옵션에 따라 SPX/IPX 또는 TCP/IP와 같은 서버에서 사용하는 네트워크 프로토콜을 지정해야 합니다. 대부분의 경우 네트워크 프로토콜 구성은 서버의 클라이언트 설정 소프트웨어를 통해 처리됩니다. ODBC에서는 ODBC 드라이버 관리자를 사용하여 드라이버 설정을 확인해야 할 수도 있습니다.

클라이언트와 서버 사이의 초기 연결에 문제가 생길 수 있습니다. 문제 발생 시 다음의 문제 해결 점검 목록을 참조하십시오.

- 서버의 클라이언트측 연결이 제대로 구성되어 있습니까?
- 검색 경로의 연결과 데이터베이스 드라이버에 DLL이 있습니까?
- TCP/IP를 사용하는 경우:
 - TCP/IP 통신 소프트웨어가 설치되어 있습니까? 적절한 WINSOCK.DLL이 설치되어 있습니까?
 - 서버의 IP 주소가 클라이언트의 HOSTS 파일에 등록되어 있습니까?
 - DNS(Domain Name Services)가 제대로 구성되어 있습니까?
 - 서버를 핑(ping)할 수 있습니까?

자세한 문제 해결 정보는 온라인 *SQL Links User's Guide*와 서버 설명서를 참조하십시오.

ODBC 사용

애플리케이션에서는 Btrieve와 같은 ODBC 데이터 소스를 사용할 수 있습니다. ODBC 드라이버 연결 요구 사항은 다음과 같습니다.

- 공급 업체가 제공하는 ODBC 드라이버.
- Microsoft ODBC Driver Manager.
- BDE Administration 유틸리티.

ODBC 드라이버 연결에 대한 BDE 별칭을 설정하려면 BDE Administration 유틸리티를 사용합니다. 자세한 내용은 BDE Administration 유틸리티의 온라인 도움말 파일을 참조하십시오.

데이터 모듈의 데이터베이스 컴포넌트 사용

데이터 모듈에 데이터베이스 컴포넌트를 안전하게 둘 수 있습니다. Object Repository에 데이터베이스 컴포넌트를 갖고 있는 데이터 모듈을 두면서 다른 사용자가 그 데이터 모듈에서 상속할 수 있도록 하려면 데이터베이스 컴포넌트의 *HandleShared* 속성을 *True*로 설정하여 전역 네임스페이스 충돌을 방지해야 합니다.

데이터베이스 세션 관리

BDE 기반 애플리케이션의 데이터베이스 연결, 드라이버, 커서, 쿼리 등은 하나 이상의 BDE 세션 컨텍스트 내에서 유지됩니다. 세션은 애플리케이션의 다른 인스턴스를 시작할 필요 없이 데이터베이스 연결과 같은 데이터베이스 액세스 작업 집합을 분리합니다.

모든 BDE 기반 데이터베이스 애플리케이션에는 기본 BDE 세션을 캡슐화하는 *Session*이라는 기본 세션 컴포넌트를 자동으로 포함됩니다. 애플리케이션에 데이터베이스 컴포넌트 추가 시 기본 세션에 자동으로 연결됩니다(*SessionName*은 "Default"). 기본 세션은 암시적(생성한 데이터베이스 컴포넌트에 연결되지 않은 데이터셋을 열 때 런타임 시 세션에서 생성됨)인지 또는 영구적(애플리케이션에서 명시적으로 생성됨)인지 여부에 상관 없이 다른 세션에 연결되지 않는 모든 데이터베이스 컴포넌트에 대한 전역 제어를 제공합니다. 디자인 타임 시 데이터 모듈 또는 폼에 기본 세션이 보이지 않지만 런타임 시 코드에서 그 속성과 메소드에 액세스할 수 있습니다.

기본 세션을 사용하려면 애플리케이션에 다음 내용이 해당되지 않을 경우 코드를 쓸 필요가 없습니다.

- 명시적으로 세션을 활성화 또는 활성화 해제하여 열리는 세션의 데이터베이스 기능을 사용 가능 또는 사용 불가능하게 합니다.
- 암시적으로 생성된 데이터베이스 컴포넌트의 기본 속성을 지정하는 등 세션의 속성을 수정합니다.
- 데이터베이스 연결 관리 (예를 들면, 사용자의 작업에 대한 응답으로 데이터베이스 연결을 열거나 닫기)와 같은 세션 메소드 하나를 실행합니다.
- 애플리케이션에서 암호 보호되는 Paradox 또는 dBASE 테이블에 액세스를 시도하는 경우처럼 세션 이벤트에 응답합니다.
- 네트워크에서 Paradox 테이블에 액세스하기 위한 *NetFileDir* 속성과 성능을 높이기 위한 로컬 하드 드라이브에 대한 *PrivateDir* 속성 등 Paradox 디렉토리 위치를 설정합니다.
- 세션을 사용하는 데이터베이스와 데이터셋의 가능한 데이터베이스 연결 구성을 나타내는 BDE 별칭을 관리합니다.

디자인 타임 시 애플리케이션에 데이터베이스 컴포넌트를 추가하거나 런타임 시 데이터베이스 컴포넌트를 동적으로 생성하는지 여부에 상관 없이 다른 세션에 특정하게 할당하지 않는 한 데이터베이스 컴포넌트는 기본 세션에 자동으로 연결됩니다. 데이터베이스 컴포넌트에 연결되지 않은 데이터셋을 열면 Delphi는 자동으로 다음을 수행합니다.

- 런타임 시 데이터베이스 컴포넌트를 생성합니다.
- 데이터베이스 컴포넌트를 기본 세션에 연결합니다.
- 기본 세션의 속성에 따라 데이터베이스 컴포넌트의 일부 핵심 속성을 초기화합니다. 가장 중요한 속성 중에서 *KeepConnections*는 애플리케이션에서 데이터베이스 연결이 유지 또는 삭제되는 시기를 결정합니다.

기본 세션은 대부분의 애플리케이션에서 사용할 수 있는 적용 범위가 넓은 기본 집합을 제공합니다. 컴포넌트가 기본 세션에서 이미 열린 데이터베이스에 대해 동시 쿼리를 수행하는 경우 데이터베이스 컴포넌트를 명시적으로 명명된 세션에만 연결해야 합니다. 이 경우 각 동시 쿼리는 그 고유의 세션에서 실행되어야 합니다. 또한 다중 스레드 데이터베이스 애플리케이션은 각 스레드가 그 고유의 세션을 갖는 다중 세션을 필요로 합니다.

필요할 때 애플리케이션에서 추가 세션 컴포넌트를 생성할 수 있습니다. BDE 기반 데이터베이스 애플리케이션에는 모든 세션 컴포넌트를 관리하는 데 사용할 수 있는 *Sessions*라는 이름의 세션 목록 컴포넌트가 자동으로 포함됩니다. 다중 세션에 대한 자세한 내용은 20-29 페이지의 "다중 세션 관리"를 참조하십시오.

데이터 모듈에 세션 컴포넌트를 안전하게 둘 수 있습니다. Object Repository에 세션 컴포넌트를 하나 이상 포함하는 데이터 모듈을 두는 경우 *AutoSessionName* 속성을 *True*로 설정하여 그 데이터 모듈에서 상속 시 네임스페이스 충돌을 방지해야 합니다.

세션 활성화

*Active*는 세션과 연결된 데이터베이스와 데이터셋 컴포넌트가 열려 있는지 결정하는 부울 속성입니다. 이 속성을 사용하여 세션의 데이터베이스와 데이터셋 연결의 현재 상태를 읽거나 상태를 변경할 수 있습니다. *Active*가 *False*(기본값)이면 세션과 연결된 모든 데이터베이스와 데이터셋은 닫히고, *True*이면 데이터베이스와 데이터셋이 열립니다.

처음 생성 시 세션은 활성화되어 있으며 그 이후로 세션의 *Active* 속성이 *False*에서 *True*로 변경될 때마다 활성화됩니다(예를 들어, 세션과 연결된 데이터베이스나 데이터셋이 열려 있고 다른 데이터베이스나 데이터셋이 현재 열려 있지 않을 때). *Active*를 *True*로 설정하면 세션의 *OnStartup* 이벤트가 트리거되고, BDE에서 Paradox 디렉토리 위치가 등록되며, 세션에서 사용 가능한 BDE 별칭을 결정하는 *ConfigMode* 속성이 등록됩니다. BDE에서 등록하거나 다른 특정 세션 시작 활동을 수행하기 전에 *OnStartup* 이벤트 핸들러를 작성하여 *NetFileDir*, *PrivateDir* 및 *ConfigMode* 속성을 초기화할 수 있습니다. *NetFileDir* 및 *PrivateDir* 속성에 대한 내용은 20-24 페이지의 "Paradox 디렉토리 위치 지정"을 참조하십시오. *ConfigMode*에 대한 내용은 20-25 페이지의 "BDE 별칭 사용"을 참조하십시오.

세션이 활성화되면 *OpenDatabase* 메소드를 호출하여 세션의 데이터베이스 연결을 열 수 있습니다.

데이터 모듈 또는 폼에 두는 세션 컴포넌트에서 데이터베이스나 데이터셋이 열려 있을 때 *Active*를 *False*로 설정하면 그 데이터베이스나 데이터셋이 닫힙니다. 런타임 시 데이터베이스와 데이터셋을 닫으면 그와 연결된 이벤트를 트리거할 수도 있습니다.

참고 디자인 타임 시 기본 세션에서 *Active*를 *False*로 설정할 수 없습니다. 런타임 시 기본 세션을 닫을 수는 있지만 닫지 않는 것이 좋습니다.

세션의 *Open* 및 *Close* 메소드를 사용하여 런타임 시 기본 세션이 아닌 세션을 활성화하거나 활성화 해제할 수도 있습니다. 예를 들어, 다음과 같은 코드 행은 세션의 열려 있는 데이터베이스와 데이터셋을 모두 닫습니다.

```
Session1.Close;
```


이 코드는 Session1의 *Active* 속성을 *False*로 설정합니다. 세션의 *Active* 속성이 *False*이면 뒤이어 데이터베이스나 데이터셋을 열려는 애플리케이션의 시도는 *Active*를 *True*로 재설정하고 세션의 *OnStartup* 이벤트 핸들러를 호출합니다. 런타임 시 세션 재활성을 명시적으로 코드화할 수도 있습니다. 다음 코드는 *Session1*을 재활성화합니다.

```
Session1.Open;
```

참고 세션이 활성화되면 데이터베이스 연결을 개별적으로 열고 닫을 수 있습니다. 자세한 내용은 20-20 페이지의 "데이터베이스 연결 닫기"를 참조하십시오.

기본 데이터베이스 연결 동작 지정

*KeepConnections*는 런타임 시 생성된 암시적 데이터베이스 컴포넌트의 *KeepConnection* 속성에 대한 기본값을 제공합니다. *KeepConnection*은 모든 데이터셋이 닫혀 있을 때 데이터베이스 컴포넌트에 설정된 데이터베이스 연결에 발생하는 내용을 지정합니다. *True* (기본값)로 설정되었다면 데이터셋이 활성화되어 있지 않아도 일정한 또는 영구적인 데이터베이스 연결은 유지됩니다. *False*인 경우 그 데이터셋이 모두 닫히는 즉시 데이터베이스 연결이 끊어집니다.

참고 데이터 모듈 또는 폼에 명시적으로 두는 데이터베이스 컴포넌트의 영구적인 연결은 그 데이터베이스 컴포넌트의 *KeepConnection* 속성에 의해 제어됩니다. 이와 다르게 설정되어 있을 경우 데이터베이스 컴포넌트의 *KeepConnection*은 세션의 *KeepConnections* 속성을 항상 오버라이드합니다. 세션 내의 개별 데이터베이스 연결 제어에 대한 자세한 내용은 20-19 페이지의 "데이터베이스 연결 관리"를 참조하십시오.

원격 서버의 데이터베이스에 연결된 모든 데이터셋을 자주 열고 닫는 애플리케이션에서 *KeepConnections*는 *True*로 설정되어야 합니다. 이 설정은 세션의 수명 동안 연결이 한 번만 열리고 닫힌다는 것을 의미하므로 이 설정을 통해 네트워크 소동량을 줄이고 데이터 액세스 속도를 높일 수 있습니다. 그렇지 않을 경우 애플리케이션에서 연결을 닫거나 재구축할 때마다 데이터베이스에 연결하고 연결을 끊어야 하는 오버헤드가 발생합니다.

참고 세션에서 *KeepConnections*가 *True*인 경우에도 *DropConnections* 메소드를 호출하여 모든 암시적 데이터베이스 컴포넌트의 비활성 데이터베이스 연결을 닫고 해제할 수 있습니다. *DropConnections*에 대한 자세한 내용은 20-21 페이지의 "비활성 데이터베이스 연결 끊기"를 참조하십시오.

데이터베이스 연결 관리

세션 컴포넌트를 사용하여 그 세션 컴포넌트 내에서 데이터베이스 연결을 관리할 수 있습니다. 세션 컴포넌트에 포함된 속성과 메소드를 사용하여 다음을 수행할 수 있습니다.

- 데이터베이스 연결을 엽니다.
- 데이터베이스 연결을 닫습니다.
- 모든 비활성 임시 데이터베이스 연결을 닫고 해제합니다.
- 특정 데이터베이스 연결을 찾습니다.
- 열려 있는 모든 데이터베이스 연결을 통해 반복적으로 사용합니다.

데이터베이스 연결 열기

세션 내의 데이터베이스 연결을 열려면 *OpenDatabase* 메소드를 호출합니다. *OpenDatabase*는 열리는 데이터베이스 이름인 매개변수 한 개를 사용합니다. 이 이름은 BDE 별칭 또는 데이터베이스 컴포넌트의 이름입니다. Paradox 또는 dBASE에서는 이름이 전체 경로 이름일 수도 있습니다. 예를 들어, 다음 문장은 기본 세션을 사용하고 DBDEMOS 별칭이 가리키는 데이터베이스의 데이터베이스 연결을 엽니다.

```
var
  DBDemosDatabase:TDatabase;
begin
  DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
  ...
```

*OpenDatabase*는 세션이 이미 활성화되어 있지 않은 경우 세션을 활성화한 다음 지정된 데이터베이스 이름이 세션의 데이터베이스 컴포넌트의 *DatabaseName* 속성에 일치하는지 확인합니다. 이름이 기존 데이터베이스 컴포넌트에 일치하지 않는 경우 *OpenDatabase*는 지정된 이름을 사용하여 임시 데이터베이스 컴포넌트를 생성합니다. 마지막으로 *OpenDatabase*는 데이터베이스 컴포넌트의 *Open* 메소드를 호출하여 서버에 연결합니다. *OpenDatabase*를 호출할 때마다 데이터베이스의 참조 카운트가 1씩 증가합니다. 이 참조 카운트가 0보다 크면 데이터베이스는 열려 있습니다.

데이터베이스 연결 닫기

데이터베이스 연결을 개별적으로 닫으려면 *CloseDatabase* 메소드를 호출합니다. *OpenDatabase*를 호출할 때마다 증가하는 데이터베이스의 참조 카운트는 *CloseDatabase*를 호출할 때마다 1씩 감소합니다. 데이터베이스의 참조 카운트가 0이면 데이터베이스가 닫힙니다. *CloseDatabase*는 닫을 데이터베이스인 매개변수 한 개를 사용합니다. *OpenDatabase* 메소드를 사용하여 데이터베이스를 열면 이 매개변수는 *OpenDatabase*의 반환 값으로 설정될 수 있습니다.

```
Session.CloseDatabase(DBDemosDatabase);
```

지정된 데이터베이스 이름이 임시(암시적) 데이터베이스 컴포넌트에 연결되어 있고 세션의 *KeepConnections* 속성이 *False*로 설정되어 있는 경우 데이터베이스 컴포넌트가 해제되어 실제로 연결을 닫습니다.

참고 *KeepConnections*가 *False*이면 임시 데이터베이스 컴포넌트는 데이터베이스 컴포넌트에 연결된 마지막 데이터셋이 닫힐 때 자동으로 닫히고 해제됩니다. 강제로 닫기 전에 애플리케이션에서 *CloseDatabase*를 항상 호출할 수 있습니다. *KeepConnections*가 *True*일 때 임시 데이터베이스 컴포넌트를 해제하려면 데이터베이스 컴포넌트의 *Close* 메소드를 호출한 다음 세션의 *DropConnections* 메소드를 호출합니다.

참고 영구적 데이터베이스 컴포넌트에 *CloseDatabase*를 호출해도 실제로 연결을 닫지는 않습니다. 연결을 닫으려면 데이터베이스 컴포넌트의 *Close* 메소드를 직접 호출합니다.

다음 두 방법을 사용하여 세션 내 모든 데이터베이스 연결을 닫을 수 있습니다.

- 세션의 *Active* 속성을 *False*로 설정합니다.
- 세션에 *Close* 메소드를 호출합니다.

*Active*를 *False*로 설정하면 Delphi에서 *Close* 메소드를 자동으로 호출합니다. *Close*는 임시 데이터베이스 컴포넌트를 해제하고 각 영구적 데이터베이스 컴포넌트의 *Close* 메소드를 호출하여 모든 활성 데이터베이스와의 연결을 끊습니다. 마지막으로 *Close*는 세션의 BDE 핸들을 *nil*로 설정합니다.

비활성 데이터베이스 연결 끊기

세션의 *KeepConnections* 속성이 *True*(기본값)이면 임시 데이터베이스 컴포넌트의 데이터베이스 연결은 컴포넌트에서 사용하는 모든 데이터셋을 닫는 경우에도 유지됩니다. *DropConnections* 메소드를 호출하여 이러한 연결을 제거하고 세션의 모든 비활성 임시 데이터베이스 컴포넌트를 해제할 수 있습니다. 예를 들어, 다음 코드는 기본 세션의 모든 비활성 임시 데이터베이스 컴포넌트를 해제합니다.

```
Session.DropConnections;
```

데이터셋이 하나 이상 활성화되어 있는 임시 데이터베이스 컴포넌트는 이 호출로 삭제되거나 해제되지 않습니다. 이러한 컴포넌트를 해제하려면 *Close*를 호출합니다.

데이터베이스 연결 검색

세션의 *FindDatabase* 메소드를 사용하여 지정된 데이터베이스 컴포넌트가 세션에 이미 연결되어 있는지 여부를 확인합니다. *FindDatabase*는 검색할 데이터베이스 이름인 매개변수 한 개를 사용합니다. 이 이름은 BDE 별칭 또는 데이터베이스 컴포넌트 이름입니다. Paradox 또는 dBASE에서는 전체 경로 이름일 수도 있습니다.

*FindDatabase*는 일치하는 것을 찾으면 데이터베이스 컴포넌트를 반환합니다. 그렇지 않을 경우에는 *nil*을 반환합니다.

다음 코드는 DBDEMOS 별칭을 사용하여 데이터베이스 컴포넌트의 기본 세션을 검색하며 기본 세션이 없으면 기본 세션을 만든 후에 엽니다.

```
var
  DB:TDatabase;
begin
  DB := Session.FindDatabase('DBDEMOS');
  if (DB = nil) then { database doesn't exist for session so,
    DB := Session.OpenDatabase('DBDEMOS'); { create and open it}
  if Assigned(DB) and DB.Connected then begin
    DB.StartTransaction;
    ...
  end;
end;
```

세션의 데이터베이스 컴포넌트를 통해 반복 사용

두 개의 세션 컴포넌트 속성, 즉 *Databases*와 *DatabaseCount*를 사용하여 세션에 연결된 모든 활성 데이터베이스 컴포넌트를 반복합니다.

*Databases*는 세션에 연결된 현재의 활성 데이터베이스 컴포넌트 전부의 배열입니다. *DatabaseCount*는 배열의 데이터베이스 수입니다. 세션 수명 동안 연결이 열리거나 닫힐 때 *Databases*와 *DatabaseCount*의 값이 변경됩니다. 예를 들어, 세션의 *KeepConnections* 고유한 데이터베이스가 열릴 때마다 *DatabaseCount*는 1씩 증가합니다. 고유한 데이터

베이스가 닫힐 때마다 *DatabaseCount*는 1씩 감소합니다. *DatabaseCount*가 0이면 해당 세션에 현재 활성 데이터베이스 컴포넌트가 없습니다.

다음 예제 코드는 기본 세션에서 각 활성 데이터베이스의 *KeepConnection* 속성을 *True*로 설정합니다.

```
var
  MaxDbCount:Integer;
begin
  with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
        Databases[MaxDbCount].KeepConnection := True;
      end;
    end;
```

암호 보호되는 Paradox 및 dBASE 테이블 사용

세션 컴포넌트는 암호 보호되는 Paradox 및 dBASE 테이블의 암호를 저장할 수 있습니다. 세션에 암호를 추가하면 애플리케이션에서 그 암호에 의해 보호되는 테이블을 열 수 있습니다. 세션에서 암호를 제거하면 암호를 다시 추가할 때까지 애플리케이션에서 암호를 사용하는 테이블을 열 수 없습니다.

AddPassword 메소드 사용

AddPassword 메소드는 애플리케이션에 옵션 방법을 제공하여 액세스 암호를 필요로 하는 암호화된 Paradox 또는 dBASE 테이블을 열기 전에 세션에 암호를 제공합니다. 세션에 암호를 추가하지 않는 경우 애플리케이션에서 암호 보호되는 테이블을 열려고 시도하면 암호를 요구하는 메시지의 대화 상자가 표시됩니다.

*AddPassword*는 사용할 암호를 포함하는 문자열인 매개변수 한 개를 사용합니다. 필요할 때마다 *AddPassword*를 호출하여 다른 암호로 보호되는 테이블에 액세스하기 위한 암호를 한 번에 하나씩 추가할 수 있습니다.

```
var
  Passwrд:String;
begin
  Passwrд := InputBox('Enter password', 'Password:', '');
  Session.AddPassword(Passwrд);
  try
    Table1.Open;
  except
    ShowMessage('Could not open table!');
    Application.Terminate;
  end;
end;
```

참고 위의 *InputBox* 함수 사용은 예시용입니다. 실제 애플리케이션에서 입력 시 암호를 표시하는 *PasswordDialog* 함수나 사용자 지정 폼과 같은 암호 입력 기능을 사용합니다.

PasswordDialog 함수 대화 상자의 Add 버튼은 *AddPassword* 메소드와 동일한 결과를 갖습니다.

```
if PasswordDialog(Session) then
```

```

Table1.Open
else
  ShowMessage('No password given, could not open table!');
end;

```

RemovePassword 및 RemoveAllPasswords 메소드 사용

*RemovePassword*는 이전에 추가한 암호를 메모리에서 삭제합니다. *RemovePassword*는 삭제할 암호를 포함하는 문자열인 매개변수 한 개를 사용합니다.

```
Session.RemovePassword('secret');
```

*RemoveAllPasswords*는 이전에 추가한 암호를 모두 메모리에서 삭제합니다.

```
Session.RemoveAllPasswords;
```

GetPassword 메소드와 OnPassword 이벤트 사용

OnPassword 이벤트를 통해 필요할 때 애플리케이션에서 Paradox 및 dBASE 테이블에 암호를 제공하는 방법을 제어할 수 있습니다. 기본 암호 처리 동작을 오버라이드하려는 경우에도 *OnPassword* 이벤트에 핸들러를 제공합니다. 핸들러를 제공하지 않으면 Delphi에서 암호를 입력할 수 있는 기본 대화 상자가 표시되고 특수한 동작은 나타나지 않습니다. 테이블 열기 시도가 성공하거나 예외가 발생합니다.

OnPassword 이벤트에 핸들러를 제공하는 경우 이벤트 핸들러에서 다음 두 가지를 수행합니다. *AddPassword* 메소드를 호출하고 이벤트 핸들러의 *Continue* 매개변수를 *True*로 설정합니다. *AddPassword* 메소드는 세션에 문자열을 전달하여 테이블의 암호로 사용합니다. *Continue* 매개변수는 이 테이블 열기 시도에 대해 더 이상 암호를 요구하는 메시지가 필요하지 않음을 Delphi에 나타냅니다. *Continue*의 기본값은 *False*이므로 이 값을 *True*로 명시적으로 설정해야 합니다. 이벤트 핸들러가 실행을 마친 후 *Continue*가 *False*이면 *AddPassword*를 사용하여 올바른 암호가 전달된 경우에도 *OnPassword* 이벤트가 다시 발생합니다. 이벤트 핸들러가 실행을 마친 후 *Continue*가 *True*이고 *AddPassword*에 전달된 문자열이 올바른 암호가 아니면 테이블 열기 시도는 실패하고 예외가 발생합니다.

*OnPassword*는 두 가지 상황에서 트리거됩니다. 한 가지 상황은 올바른 암호가 세션에 아직 제공되지 않았을 때 암호 보호되는 테이블(dBASE 또는 Paradox)을 열려고 시도하는 경우입니다. (해당 테이블의 올바른 암호가 이미 제공되었으면 *OnPassword* 이벤트가 발생하지 않습니다.)

다른 상황은 *GetPassword* 메소드를 호출하는 경우입니다. *GetPassword*는 *OnPassword* 이벤트를 생성하거나, 세션에 *OnPassword* 이벤트 핸들러가 없는 경우 기본 암호 대화 상자를 표시합니다. *OnPassword* 이벤트 핸들러 또는 기본 대화 상자에서 세션에 암호를 추가하면 *True*를 반환하고 항목이 생성되지 않았으면 *False*를 반환합니다.

다음 예제에서 전역 *Session* 객체의 *OnPassword* 속성에 할당하여 *Password* 메소드를 기본 세션의 *OnPassword* 이벤트 핸들러로 지정합니다.

```

procedure TForm1.FormCreate(Sender:TObject);
begin
    Session.OnPassword := Password;
end;

```

Password 메소드에서 *InputBox* 함수는 사용자에게 암호를 요구하는 메시지를 표시합니다. 그런 다음 *AddPassword* 메소드는 프로그래밍적으로 대화 상자에 입력한 암호를 세션에 제공합니다.

```

procedure TForm1.Button1Click(Sender:TObject; var Action:Boolean);
var
    Passwr:String;
begin
    Passwr := InputBox('Enter password', 'Password:', '');
    Continue := (Passwr > '');
    Session.AddPassword(Passwr);
end;

```

OnPassword 이벤트(및 *Password* 이벤트 핸들러)는 아래와 같이 암호 보호되는 테이블을 열려는 시도에 의해 트리거됩니다. *OnPassword* 이벤트의 핸들러에 암호를 요구하는 메시지가 표시된 경우에도 사용자가 잘못된 암호를 입력하거나 그 외의 문제가 발생하면 테이블 열기 시도가 실패합니다.

```

procedure TForm1.OpenTableBtnClick(Sender:TObject);
const
    CRLF = #13 + #10;
begin
    try
        Table1.Open;                                { this line triggers the OnPassword event }
    except
        on E:Exception do begin                        { exception if cannot open table }
            ShowMessage('Error!' + CRLF +             { display error explaining what happened }
                E.Message + CRLF +
                'Terminating application...');
            Application.Terminate;                    { end the application }
        end;
    end;
end;

```

Paradox 디렉토리 위치 지정

두 개의 세션 컴포넌트 속성, 즉 *NetFileDir*과 *PrivateDir*은 Paradox 테이블을 사용하는 애플리케이션에 대해 고유합니다.

*NetFileDir*은 Paradox 네트워크 제어 파일, PDOXUSRS.NET을 포함하는 디렉토리를 지정합니다. 이 파일은 네트워크 드라이브의 Paradox 테이블 공유를 관리합니다. Paradox 테이블을 공유해야 하는 모든 애플리케이션에서 네트워크 제어 파일에 동일한 디렉토리를 지정해야 합니다(보통 네트워크 파일 서버의 디렉토리). Delphi는 지정된 데이터베이스 별칭의 BDE(Borland Database Engine) 구성 파일에서 *NetFileDir*의 값을 파생합니다. 사용자가 직접 *NetFileDir*을 설정하는 경우 제공하는 값이 BDE 구성 설정을 오버라이드하므로 새로운 값을 확인해야 합니다.

디자인 타임 시 Object Inspector에서 *NetFileDir*의 값을 지정할 수 있습니다. 런타임 시 코드의 *NetFileDir*을 설정하거나 변경할 수도 있습니다. 다음 코드는 기본 세션의 *NetFileDir*을 애플리케이션에서 실행하는 디렉토리 위치로 설정합니다.

```
Session.NetFileDir := ExtractFilePath(Application.EXEName);
```

참고 *NetFileDir*은 애플리케이션에 열려 있는 Paradox 파일이 없을 때만 변경할 수 있습니다. 런타임 시 *NetFileDir*을 변경하는 경우 네트워크 사용자들이 공유하는 올바른 네트워크 디렉토리를 가리키는지 확인합니다.

*PrivateDir*은 로컬 SQL 문 처리를 위해 BDE에서 생성한 것과 같은 임시 테이블 처리 파일을 저장하기 위한 디렉토리를 지정합니다. *PrivateDir* 속성에 지정된 값이 없으면 BDE는 초기화될 때 현재 디렉토리를 자동으로 사용합니다. 애플리케이션이 네트워크 파일 서버에서 직접 실행되는 경우 데이터베이스를 열기 전에 *PrivateDir*을 사용자의 로컬 하드 드라이브로 설정하여 런타임 시 애플리케이션 성능을 향상시킬 수 있습니다.

참고 디자인 타임 시 *PrivateDir*을 설정한 다음 IDE에서 세션을 열지 마십시오. 그렇게 하면 IDE에서 애플리케이션 실행 시 Directory 사용 중 오류가 발생합니다.

다음 코드는 기본 세션의 *PrivateDir* 속성이 사용자의 C:\TEMP 디렉토리로 설정된 것을 변경합니다.

```
Session.PrivateDir := 'C:\TEMP';
```

중요 *PrivateDir*을 드라이브의 루트 디렉토리로 설정하지 마십시오. 항상 하위 디렉토리를 지정하십시오.

BDE 별칭 사용

세션에 연결된 각 데이터베이스 컴포넌트에는 BDE 별칭이 있습니다 (Paradox 및 dBASE 테이블 액세스 시 옵션으로 전체 경로 이름 대신 별칭을 사용할 수 있습니다). 세션에서 세션 수명 동안 별칭을 생성, 수정 및 삭제할 수 있습니다.

AddAlias 메소드는 SQL 데이터베이스 서버의 새 BDE 별칭을 만듭니다. *AddAlias*는 세 개의 매개변수를 사용하며 별칭 이름을 포함하는 문자열, 사용할 SQL 연결 드라이버를 지정하는 문자열, 별칭의 매개변수로 구성된 문자열 목록이 이에 해당됩니다. 예를 들어, 다음 문은 *AddAlias*를 사용하여 InterBase 서버에 액세스하기 위한 새 별칭을 기본 세션에 추가합니다.

```
var
  AliasParams:TStringList;
begin
  AliasParams := TStringList.Create;
  try
    with AliasParams do begin
      Add('OPEN MODE=READ');
      Add('USER NAME=TOMSTOPPARD');
      Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
    end;
    Session.AddAlias('CATS', 'INTRBASE', AliasParams);
    ...
  finally
    AliasParams.Free;
  end;
end;
```

*AddStandardAlias*는 Paradox, dBASE 또는 ASCII 테이블에 새 BDE 별칭을 생성합니다. *AddStandardAlias*는 세 개의 매개변수를 사용하며 별칭 이름, 액세스할 Paradox 또는 dBASE 테이블에 대한 전체 경로, 확장자가 없는 테이블을 열려고 할 때 사용할 기본 드라이버의 이름이 이에 해당됩니다. 예를 들어, 다음 문은 *AddStandardAlias*를 사용하여 Paradox 테이블에 액세스하기 위한 새 별칭을 만듭니다.

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

세션에 별칭 추가 시 BDE는 별칭의 복사본을 메모리에 저장하며 이 복사본은 이 세션과 *ConfigMode* 속성에 포함된 *cfmPersistent*가 있는 다른 세션에 대해서만 사용 가능합니다. *ConfigMode*는 세션의 데이터베이스에서 사용할 수 있는 별칭 타입을 결정하는 집합입니다. 기본 설정은 *cmAll*이며 [*cfmVirtual*, *cfmPersistent*, *cfmSession*]으로 번역됩니다. *ConfigMode*가 *cmAll*이면 세션은 세션에서 생성된 모든 별칭 (*cfmSession*), 사용자 시스템에 있는 BDE 구성 파일의 모든 별칭 (*cfmPersistent*), BDE가 메모리에 유지하는 모든 별칭 (*cfmVirtual*)을 볼 수 있습니다. *ConfigMode*를 변경하여 세션의 데이터베이스에서 사용할 수 있는 BDE 별칭을 제한할 수 있습니다. 예를 들어, *ConfigMode*를 *cfmSession*으로 설정하면 해당 세션에서 생성된 별칭에 대한 뷰가 제한됩니다. BDE 구성 파일과 메모리의 다른 별칭은 모두 사용할 수 없습니다.

모든 세션과 다른 애플리케이션에서 새로 만든 별칭을 사용할 수 있도록 하려면 세션의 *SaveConfigFile* 메소드를 사용합니다. *SaveConfigFile*은 다른 BDE 활성 애플리케이션에서 읽거나 사용할 수 있는 BDE 구성 파일에 메모리의 별칭을 씁니다.

별칭을 생성한 후 *ModifyAlias*를 호출하여 해당 매개변수를 변경할 수 있습니다. *ModifyAlias*는 두 개의 매개변수를 사용하며 수정할 별칭 이름, 변경할 매개변수와 그 값을 포함하는 문자열 목록이 이에 해당됩니다. 예를 들어, 다음 문은 *ModifyAlias*를 사용하여 기본 세션에서 CATS 별칭의 OPEN MODE 매개변수를 READ/WRITE로 변경합니다.

```
var
  List:TStringList;
begin
  List := TStringList.Create;
  with CustSource do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...
```

세션에서 이전에 생성한 별칭을 삭제하려면 *DeleteAlias* 메소드를 호출합니다. *DeleteAlias*는 삭제할 별칭 이름을 가진 매개변수 한 개를 사용합니다. *DeleteAlias*는 세션에서 별칭을 사용할 수 없도록 합니다.

참고 *DeleteAlias*는 *SaveConfigFile*에 대한 이전 호출에 의해 별칭이 파일에 쓰여지면 BDE 구성 파일에서 별칭을 제거하지 않습니다. *DeleteAlias*를 호출한 후 구성 파일에서 별칭을 제거하려면 *SaveConfigFile*을 다시 호출합니다.

세션 컴포넌트는 매개변수 정보와 드라이버 정보 등 BDE 별칭에 대한 정보를 검색하기 위한 다섯 개의 메소드를 제공합니다. 해당 메소드는 다음과 같습니다.

- *GetAliasNames*는 세션이 액세스한 별칭을 나열합니다.
- *GetAliasParams*는 지정된 별칭의 매개변수를 나열합니다.
- *GetAliasDriverName*은 별칭이 사용한 BDE 드라이버의 이름을 반환합니다.
- *GetDriverNames*는 세션에 사용할 수 있는 모든 BDE 드라이버를 나열합니다.
- *GetDriverParams*는 지정된 드라이버의 드라이버 매개변수를 반환합니다.

세션의 정보 메소드 사용에 대한 자세한 내용은 아래의 "세션에 대한 정보 검색"을 참조하십시오. 사용할 BDE 별칭과 SQL 연결에 대한 자세한 내용은 BDE 온라인 도움말인 BDE32.HLP를 참조하십시오.

세션에 대한 정보 검색

세션의 정보 메소드를 사용하여 세션 및 그 데이터베이스 컴포넌트에 대한 정보를 검색할 수 있습니다. 예를 들어, 한 메소드는 세션에 알려진 모든 별칭 이름을 검색하고, 다른 메소드는 세션에서 사용한 특정 데이터베이스 컴포넌트에 연결된 테이블 이름을 검색합니다. 표 20.4는 세션 컴포넌트에 대한 정보 메소드를 요약한 것입니다.

표 20.4 세션 컴포넌트의 데이터베이스 관련 정보 메소드

메소드	용도
<i>GetAliasDriverName</i>	데이터베이스의 지정된 별칭에 대한 BDE 드라이버를 검색합니다.
<i>GetAliasNames</i>	데이터베이스의 BDE 별칭 목록을 검색합니다.
<i>GetAliasParams</i>	데이터베이스의 지정된 BDE 별칭에 대한 매개변수 목록을 검색합니다.
<i>GetConfigParams</i>	BDE 구성 파일의 구성 정보를 검색합니다.
<i>GetDatabaseNames</i>	BDE 별칭 목록과 현재 사용 중인 <i>TDatabase</i> 컴포넌트 이름을 검색합니다.
<i>GetDriverNames</i>	현재 설치되어 있는 모든 BDE 드라이버의 이름을 검색합니다.
<i>GetDriverParams</i>	지정된 BDE 드라이버의 매개변수 목록을 검색합니다.
<i>GetStoredProcNames</i>	지정된 데이터베이스의 모든 내장 프로시저 이름을 검색합니다.
<i>GetTableNames</i>	지정된 데이터베이스의 지정된 패턴과 일치하는 모든 테이블 이름을 검색합니다.
<i>GetFieldNames</i>	지정된 데이터베이스의 지정된 테이블에 있는 모든 필드 이름을 검색합니다.

*GetAliasDriverName*을 제외하고 이러한 메소드는 애플리케이션에서 선언 및 유지된 문자열 목록에 대한 값 집합을 반환합니다. (*GetAliasDriverName*은 단일 문자열, 세션에서 사용하는 특정 데이터베이스 컴포넌트의 현재 BDE 드라이버 이름을 반환합니다.)

예를 들어, 다음 코드는 기본 세션에 알려진 모든 데이터베이스 컴포넌트와 별칭의 이름을 검색합니다.

```
var
  List:TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
```

```
List.Free;
end;
end;
```

추가 세션 생성

세션을 생성하고 기본 세션을 보충할 수 있습니다. 디자인 타임 시 데이터 모듈 또는 폼에 추가 세션을 두고, Object Inspector에 세션의 속성을 설정하고, 세션에 대한 이벤트 핸들러를 작성하고, 세션의 메소드를 호출하는 코드를 쓸 수 있습니다. 런타임 시 세션을 생성하고, 세션의 속성을 설정하고, 세션의 메소드를 호출할 수도 있습니다.

참고 애플리케이션에서 데이터베이스에 대해 동시 쿼리를 실행하거나 애플리케이션이 다중 스레드가 아니면 추가 세션 생성은 옵션입니다.

런타임 시 세션 컴포넌트를 동적으로 생성할 수 있게 하려면 다음 단계를 수행합니다.

- 1 *TSession* 변수를 선언합니다.
- 2 *Create* 메소드를 호출하여 새 세션을 인스턴스화합니다. 생성자는 세션에 대한 데이터베이스 컴포넌트의 빈 목록을 설정하고, *KeepConnections* 속성을 *True*로 설정하며, 애플리케이션의 세션 목록 컴포넌트에 의해 유지되는 세션 목록에 세션을 추가합니다.
- 3 새 세션의 *SessionName* 속성을 고유한 이름으로 설정합니다. 이 속성은 데이터베이스 컴포넌트를 세션에 연결하는 데 사용됩니다. *SessionName* 속성에 대한 자세한 내용은 20-29 페이지의 "세션 이름 지정"을 참조하십시오.
- 4 세션을 활성화하고 그 속성을 옵션으로 조정합니다.

*TSessionList*의 *OpenSession* 메소드를 사용하여 세션을 생성하고 열 수도 있습니다. 세션이 없는 경우 *OpenSession*만 세션을 생성하므로 *OpenSession*을 사용하는 것이 *Create*를 호출하는 것보다 안전합니다. *OpenSession*에 대한 세션은 20-29 페이지의 "다중 세션 관리"를 참조하십시오.

다음 코드는 새로운 세션 컴포넌트를 생성하고, 그 이름을 지정하며, 이어지는 데이터베이스 작업의 세션을 엽니다(여기서는 표시 안 함). 사용 후 *Free* 메소드를 호출하면 해당 세션이 소멸됩니다.

참고 기본 세션은 절대 삭제하지 마십시오.

```
var
  SecondSession:TSession;
begin
  SecondSession := TSession.Create(Form1);
  with SecondSession do
    try
      SessionName := 'SecondSession';
      KeepConnections := False;
      Open;
      ...
    finally
      SecondSession.Free;
    end;
  end;
end;
```

세션 이름 지정

세션의 *SessionName* 속성은 데이터베이스와 데이터셋을 세션에 연결할 수 있도록 세션의 이름을 지정하는 데 사용됩니다. 기본 세션에서 *SessionName*은 "Default"입니다. 생성하는 각 추가 세션 컴포넌트에서 *SessionName* 속성을 고유한 값으로 설정해야 합니다.

데이터베이스와 데이터셋 컴포넌트는 세션 컴포넌트의 *SessionName* 속성에 해당하는 *SessionName* 속성을 가집니다. 데이터베이스 또는 데이터셋 컴포넌트의 *SessionName* 속성을 공백으로 두면 기본 세션에 자동으로 연결됩니다. 데이터베이스 또는 데이터셋 컴포넌트의 *SessionName*을 생성되는 세션 컴포넌트의 *SessionName*에 해당하는 이름으로 설정할 수도 있습니다.

다음 코드는 기본 *TSessionList* 컴포넌트인 *Sessions*의 *OpenSession* 메소드를 사용하여 그 *SessionName*을 "InterBaseSession"으로 설정하고 기존 데이터베이스 컴포넌트 *Database1*을 해당 세션에 연결합니다.

```
var
  IBSession:TSession;
  ...
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;
```

다중 세션 관리

다중 스레드를 사용하여 데이터베이스 작업을 수행하는 단일 애플리케이션을 생성하는 경우 각 스레드에 대한 추가 세션을 하나 만들어야 합니다. 컴포넌트 팔레트의 BDE 페이지에는 디자인 타임 시 데이터 모듈 또는 폼에 둘 수 있는 세션 컴포넌트가 포함됩니다.

중요 세션 컴포넌트를 둘 때 기본 세션의 *SessionName* 속성과 충돌하지 않도록 그 *SessionName* 속성을 고유한 값으로 설정해야 합니다.

디자인 타임 시 세션 컴포넌트를 두면 런타임 시 애플리케이션에서 필요로 하는 스레드 및 세션의 수는 정적인 것으로 가정합니다. 이는 무엇보다도 애플리케이션에서 세션을 동적으로 생성해야 한다는 것을 의미합니다. 세션을 동적으로 생성하려면 런타임 시 전역 *Sessions* 객체의 *OpenSession* 메소드를 호출합니다.

*OpenSession*은 애플리케이션의 모든 세션 이름에서 고유한 세션의 이름인 단일 매개 변수를 필요로 합니다. 다음 코드는 고유하게 생성된 이름의 새 세션을 동적으로 생성하고 활성화합니다.

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

이 문은 세션의 현재 수를 검색하고 그 값에 하나를 추가하여 새 세션의 고유 이름을 생성합니다. 런타임 시 세션을 동적으로 생성하고 소멸하는 경우 이 예제 코드가 예상한 대로 작동하지 않았지만 이 예제는 다중 세션을 관리하기 위해 *Sessions*의 속성과 메소드를 사용하는 방법을 보여 줍니다.

*Sessions*는 BDE 기반 데이터베이스 애플리케이션에 대해 자동으로 인스턴스화되는 *TSessionList* 타입의 변수입니다. *Sessions*의 속성과 메소드를 사용하여 다중 스레드 데이터베이스 애플리케이션의 여러 세션을 추적합니다. 표 20.5는 *TSessionList* 컴포넌트의 속성과 메소드를 요약한 것입니다.

표 20.5 TSessionList 속성과 메소드

속성 또는 메소드	용도
<i>Count</i>	세션 목록에서 활성화와 비활성의 세션 수를 반환합니다.
<i>FindSession</i>	지정된 이름의 세션을 검색하고 해당 세션에 대한 포인터를 반환하거나, 지정된 이름의 세션이 없는 경우 nil 을 반환합니다. 빈 세션 이름을 전달하면 <i>FindSession</i> 은 기본 세션인 <i>Session</i> 에 대한 포인터를 반환합니다.
<i>GetSessionNames</i>	현재 인스턴스화되어 있는 모든 세션 컴포넌트의 이름으로 문자열 목록을 구성합니다. 이 프로시저는 기본 세션에서 적어도 하나의 문자열, "Default"를 추가합니다.
<i>List</i>	지정된 세션 이름의 세션 컴포넌트를 반환합니다. 지정된 이름의 세션이 없는 경우 예외가 발생합니다.
<i>OpenSession</i>	새 세션을 생성하고 활성화하거나 지정된 세션 이름의 기존 세션을 재 활성화합니다.
<i>Sessions</i>	순서 값 기준으로 세션에 액세스합니다.

다중 스레드 애플리케이션에서 *Sessions* 속성과 메소드를 사용하는 예로 데이터베이스 연결을 열 때 발생할 일을 고려합니다. 이미 연결이 있는지 확인하려면 *Sessions* 속성을 사용하여 세션 목록에서 기본 세션부터 시작하여 각 세션을 살펴 봅니다. 각 세션 컴포넌트에서 그 *Databases* 속성을 점검하여 해당 데이터베이스가 열려 있는지 확인합니다. 다른 스레드가 이미 원하는 데이터베이스를 사용하고 있는 경우에는 목록에서 다음 세션을 검사합니다.

기존 스레드가 데이터베이스를 사용하지 않는 경우에는 그 세션 내에서 연결을 열 수 있습니다.

반면, 기존 스레드가 모두 데이터베이스를 사용하는 경우 다른 데이터베이스 연결을 열 새로운 세션을 열어야 합니다.

각 스레드에 데이터 모듈의 고유 복사본이 있는 다중 스레드 애플리케이션에 세션을 포함하는 데이터 모듈을 복제하는 경우, *AutoSessionName* 속성을 사용하여 데이터 모듈의 모든 데이터셋이 올바른 세션을 사용하는지 확인할 수 있습니다. *AutoSessionName*을 *True*로 설정하면 런타임 시 생성될 때 세션에서 그 고유의 이름이 동적으로 생성됩니다. 그러면 이 이름이 데이터 모듈의 모든 데이터셋에 할당되어 명시적으로 설정된 세션 이름을 오버라이드합니다. 이렇게 하면 각 스레드가 그 고유의 세션을 가지며 각 데이터셋이 그 고유의 데이터 모듈에서 해당 세션을 사용하게 됩니다.

BDE에서 트랜잭션 사용

기본적으로 BDE는 애플리케이션에 암시적 트랜잭션 제어를 제공합니다. 애플리케이션이 암시적 트랜잭션 제어 하에 있을 때 원본으로 사용하는 데이터베이스에 기록되는 데이터셋의 각 기록에 별도의 트랜잭션을 사용합니다. 암시적 트랜잭션은 레코드 업데이트 충돌 최소화 및 일관성 있는 데이터베이스 뷰를 보장합니다. 반면, 데이터베이스에 기록되는 각 데이터 행이 그 고유의 트랜잭션에 생기고, 암시적 트랜잭션 제어로 인해 네트워크 소용량이 과도해지고, 애플리케이션 성능이 저하됩니다. 또한 암시적 트랜잭션 제어는 레코드를 두 개 이상 스캔하는 논리 연산을 보호하지 않습니다.

명시적으로 트랜잭션을 제어하는 경우 트랜잭션을 시작, 커밋 및 롤백할 가장 효과적인 시간을 선택할 수 있습니다. 다중 사용자 환경에서 애플리케이션을 개발할 때, 특히 애플리케이션에서 원격 SQL 서버에 대해 실행할 때 명시적으로 트랜잭션을 제어해야 합니다.

BDE 기반 데이터베이스 애플리케이션에서 명시적으로 트랜잭션을 제어하는 데에는 두 가지의 상호 배타적인 방법이 있습니다.

- 데이터베이스 컴포넌트를 사용하여 트랜잭션을 제어합니다. 데이터베이스 컴포넌트의 메소드와 속성을 사용할 경우의 주요 이점은 특정 데이터베이스 또는 서버에 종속적이지 않은 명백하고 인식 가능한 애플리케이션을 제공한다는 것입니다. 이 트랜잭션 제어 타입은 모든 데이터베이스 연결 컴포넌트에 의해 지원되며, 17-6 페이지의 "트랜잭션 관리"에서 설명됩니다.
- 쿼리 컴포넌트에서 Passthrough SQL을 사용하여 SQL 문을 원격 SQL 또는 ODBC 서버로 직접 전달합니다. Passthrough SQL의 주요 이점은 스키마 캐싱과 같은 특정 데이터베이스 서버의 고급 트랜잭션 관리 기능을 사용할 수 있다는 것입니다. 서버 트랜잭션 관리 모델의 이점에 대해 알아보려면 데이터베이스 서버 설명서를 참조하십시오. Passthrough SQL에 대한 자세한 내용은 아래 "Passthrough SQL 사용"을 참조하십시오.

로컬 데이터베이스를 사용할 때 데이터베이스 컴포넌트를 사용해서만 명시적 트랜잭션을 생성할 수 있습니다(로컬 데이터베이스는 Passthrough SQL을 지원하지 않습니다). 하지만 로컬 트랜잭션을 사용하는 데에는 몇 가지 제한이 있습니다. 로컬 트랜잭션 사용에 대한 자세한 내용은 20-32 페이지의 "로컬 트랜잭션 사용"을 참조하십시오.

참고 업데이트를 캐시하여 필요한 트랜잭션 수를 최소화할 수 있습니다. 캐시된 업데이트에 대한 자세한 내용은 23-16 페이지의 "클라이언트 데이터셋을 사용하여 업데이트 캐시"와 20-33 페이지의 "BDE 사용하여 업데이트 캐시"를 참조하십시오.

Passthrough SQL 사용

Passthrough SQL를 통해 *TQuery*, *TStoredProc* 또는 *TUpdateSQL* 컴포넌트를 사용하여 SQL 트랜잭션 제어 문을 원격 데이터베이스 서버에 직접 보냅니다. BDE는 SQL 문을 처리하지 않습니다. Passthrough SQL을 사용하면 특히 해당 제어가 비표준일 때 서버에서 제공하는 트랜잭션 제어를 직접 이용할 수 있습니다.

Passthrough SQL을 사용하여 트랜잭션을 제어하려면 다음을 수행해야 합니다.

- 적절한 SQL 연결 드라이버를 설치합니다. Delphi 설치 시 "Typical" 설치를 선택한 경우 모든 SQL 연결 드라이버는 이미 제대로 설치되어 있습니다.
- 네트워크 프로토콜을 구성합니다. 자세한 내용은 네트워크 관리자에게 문의하십시오.
- 원격 서버의 데이터베이스에 액세스합니다.
- SQL 탐색기를 사용하여 SQLPASSTHRU MODE를 NOT SHARED로 설정합니다. SQLPASSTHRU MODE는 BDE 와 passthrough SQL 문이 동일한 데이터베이스 연결을 공유할 수 있는지 여부를 지정합니다. 대부분의 경우 SQLPASSTHRU MODE는 SHARED AUTOCOMMIT으로 설정됩니다. 하지만 트랜잭션 제어 문 사용 시 데이터베이스 연결을 공유할 수 없습니다. SQLPASSTHRU 모드에 대한 자세한 내용은 BDE Administration 유틸리티의 도움말 파일을 참조하십시오.

참고 SQLPASSTHRU MODE가 NOT SHARED이면 SQL 트랜잭션 문을 서버에 전달하거나 전달하지 않는 데이터셋에 대해 별도의 데이터베이스 컴포넌트를 사용해야 합니다.

로컬 트랜잭션 사용

BDE는 Paradox, dBASE, Access 및 FoxPro 테이블에 대해 로컬 트랜잭션을 지원합니다. 코드 관점에서 로컬 트랜잭션과 원격 데이터베이스 서버에 대한 트랜잭션 간에는 별 차이가 없습니다.

참고 로컬 Paradox, dBASE, Access 및 FoxPro 테이블에서 트랜잭션 사용 시 *tiReadCommitted*의 기본값을 사용하는 대신 *TransIsolation*을 *tiDirtyRead*로 설정합니다. *TransIsolation*을 로컬 테이블의 *tiDirtyRead* 이외의 것으로 설정하는 경우 BDE 오류가 반환됩니다.

로컬 테이블에 대해 트랜잭션을 시작하면 테이블에 대해 수행된 업데이트가 로그됩니다. 각 로그 레코드는 레코드의 이전 레코드 버퍼를 포함합니다. 트랜잭션을 활성화하면 트랜잭션이 커밋 또는 롤백될 때까지 업데이트된 레코드를 잠급니다. 롤백에서 이전 레코드 버퍼가 업데이트된 레코드에 적용되어 해당 레코드를 이전 업데이트 상태로 복구합니다.

로컬 트랜잭션은 SQL 서버 또는 ODBC 드라이버에 대한 트랜잭션보다 더 제한됩니다. 특히 로컬 트랜잭션에는 다음과 같은 제한 사항이 있습니다.

- 자동 손상 복구가 제공되지 않습니다.
- 데이터 정의 문은 지원되지 않습니다.
- 임시 테이블에 대해 트랜잭션을 실행할 수 없습니다.
- *TransIsolation* 레벨은 *tiDirtyRead*로만 설정해야 합니다.
- Paradox에서 로컬 트랜잭션은 유효한 인덱스가 있는 테이블에서만 수행될 수 있습니다. 데이터는 인덱스가 없는 Paradox 테이블에서는 롤백될 수 없습니다.
- 제한된 수의 레코드만을 잠그거나 수정할 수 있습니다. Paradox 테이블에서는 255개의 레코드로 제한됩니다. dBASE에서는 100개의 레코드로 제한됩니다.
- BDE ASCII 드라이버에 대해 트랜잭션을 실행할 수 없습니다.

- 트랜잭션 도중 테이블에서 커서를 닫으면 다음 두 경우를 제외하고 트랜잭션이 롤백됩니다.
 - 여러 테이블이 열려 있는 경우.
 - 변경되지 않은 테이블에서 커서가 닫힌 경우.

BDE 사용하여 업데이트 캐시

업데이트를 캐시하는 데 권장되는 방법은 클라이언트 데이터셋(*TBDEClientDataSet*)을 사용하거나 데이터셋 프로바이더를 통해 BDE 데이터셋을 클라이언트 데이터셋에 연결하는 것입니다. 클라이언트 데이터셋 사용의 이점에 대해서는 23-16 페이지의 "클라이언트 데이터셋을 사용하여 업데이트 캐시"에서 설명합니다.

하지만 간단한 경우에는 대신 BDE를 사용하여 업데이트를 캐시할 수도 있습니다. BDE 활성 데이터셋과 *TDatabase* 컴포넌트는 캐시된 업데이트를 처리하는 데 내장 속성, 메소드 및 이벤트를 제공합니다. 이들 대부분은 클라이언트 데이터셋을 사용하여 업데이트를 캐시할 때 클라이언트 데이터셋과 데이터셋 프로바이더에서 사용되는 속성, 메소드 및 이벤트에 직접 상응합니다. 다음 표에는 이러한 속성, 이벤트 및 메소드와 *TBDEClientDataSet*에서 상응하는 속성, 메소드 및 이벤트가 나열되어 있습니다.

표 20.6 캐시된 업데이트를 위한 속성, 메소드 및 이벤트

BDE 활성 데이터셋 (또는 TDatabase)	TBDEClientDataSet	용도
<i>CachedUpdates</i>	항상 업데이트를 캐시하는 클라이언트 데이터셋에는 필요하지 않습니다.	캐시된 업데이트가 데이터셋에 적용되는지 여부를 결정합니다.
<i>UpdateObject</i>	<i>BeforeUpdateRecord</i> 이벤트 핸들러를 사용하거나 <i>TClientDataSet</i> 을 사용 중인 경우 BDE 활성 소스 데이터셋의 <i>UpdateObject</i> 속성을 사용합니다.	읽기 전용 데이터셋을 업데이트하기 위한 업데이트 객체를 지정합니다.
<i>UpdatesPending</i>	<i>ChangeCount</i>	로컬 캐시가 데이터베이스에 적용하는 데 필요한 업데이트된 레코드를 포함하는지 나타냅니다.
<i>UpdateRecordTypes</i>	<i>StatusFilter</i>	업데이트된 레코드의 종류를 나타내어 캐시된 업데이트를 적용할 때 보이게 합니다.
<i>UpdateStatus</i>	<i>UpdateStatus</i>	레코드가 변경되지 않았는지, 수정되었는지, 삽입되었는지 또는 삭제되었는지 나타냅니다.
<i>OnUpdateError</i>	<i>OnReconcileError</i>	레코드별로 업데이트 오류를 처리하기 위한 이벤트입니다.
<i>OnUpdateRecord</i>	<i>BeforeUpdateRecord</i>	레코드별로 업데이트를 처리하기 위한 이벤트입니다.

표 20.6 캐시된 업데이트를 위한 속성, 메소드 및 이벤트 (계속)

BDE 활성 데이터셋 (또는 TDatabase)	TBDEClientDataSet	용도
<i>ApplyUpdates</i> <i>ApplyUpdates</i> (database)	<i>ApplyUpdates</i>	레코드를 로컬 캐시의 데이터베이스에 적용합니다.
<i>CommitUpdates</i>	<i>Reconcile</i>	성공적으로 업데이트된 애플리케이션 다음에 이어지는 업데이트 캐시를 지웁니다.
<i>FetchAll</i>	<i>GetNextPacket</i> (및 <i>PacketRecords</i>)	데이터베이스 레코드를 편집 및 업데이트할 목적으로 로컬 캐시에 복사합니다.
<i>RevertRecord</i>	<i>RevertRecord</i>	업데이트가 아직 적용되지 않았으면 현재 레코드에 대한 업데이트를 취소합니다.

캐시된 업데이트 프로세스의 개요에 관해서는 23-17 페이지의 "캐시된 업데이트 사용 개요"를 참조하십시오.

참고 클라이언트 데이터셋을 사용하여 업데이트를 캐시하는 경우라도 20-40 페이지의 업데이트 객체에 대한 단원을 참조할 수 있습니다. *TBDEClientDataSet* 또는 *TDataSetProvider*의 *BeforeUpdateRecord* 이벤트 핸들러에 있는 업데이트 객체를 사용하여 내장 프로시저나 다중 테이블 쿼리의 업데이트를 적용할 수 있습니다.

BDE 기반 캐시된 업데이트 활성화

캐시된 업데이트에 BDE를 사용하려면 BDE 활성 데이터셋에서 업데이트를 캐시해야 함을 나타내야 합니다. 이는 *CachedUpdates* 속성을 *True*로 설정하여 지정됩니다. 캐시된 업데이트를 활성화할 때 모든 레코드의 복사본이 로컬 메모리에 캐시됩니다. 사용자가 이러한 데이터의 로컬 복사본을 보고 편집할 수 있습니다. 변경, 삽입 또는 삭제도 메모리에 캐시됩니다. 이러한 내용들은 애플리케이션에서 변경 내용을 데이터베이스 서버에 적용할 때까지 메모리에 축적됩니다. 변경된 레코드가 데이터베이스에 적용되면 해당 변경의 레코드는 캐시에서 해제됩니다.

데이터셋은 *CachedUpdates*를 *False*로 설정할 때까지 모든 업데이트를 캐시합니다. 캐시된 업데이트를 적용해도 이후 캐시된 업데이트를 사용 불가능하게 만들지 않으며 단지 현재의 변경 집합을 데이터베이스에 기록하고 그 내용을 메모리에서 지웁니다. *CancelUpdates*를 호출하여 업데이트를 취소하면 캐시에 현재 있는 모든 변경이 제거되지만 데이터셋에서 이후의 변경을 캐시하는 것을 막지는 않습니다.

참고 *CachedUpdates*를 *False*로 설정하여 캐시된 업데이트를 사용 불가능하게 하면 아직 적용하지 않은 대기 중인 변경이 통보 없이 삭제됩니다. 변경 내용 손실을 방지하려면 캐시된 업데이트를 사용 불가능하게 만들기 전에 *UpdatesPending* 속성을 테스트합니다.

BDE 기반 캐시된 업데이트 적용

업데이트 적용은 애플리케이션에서 오류를 원만하게 복구할 수 있도록 데이터베이스 컴포넌트 트랜잭션의 컨텍스트에서 발생할 수 있는 두 단계 프로세스입니다. 데이터베이스 컴포넌트를 처리하는 트랜잭션에 대한 내용은 17-6 페이지의 "트랜잭션 관리"를 참조하십시오.

데이터베이스 트랜잭션 제어 하에서 업데이트를 적용하면 다음 이벤트가 발생합니다.

- 1 데이터베이스 트랜잭션이 시작됩니다.
- 2 캐시된 업데이트가 데이터베이스에 쓰여집니다(1 단계). 이를 제공하면 데이터베이스에 쓰여진 각 레코드마다 *OnUpdateRecord* 이벤트가 한 번 트리거됩니다. 레코드가 데이터베이스에 적용될 때 오류가 발생하는 경우 이를 제공하면 *OnUpdateError* 이벤트가 트리거됩니다.
- 3 트랜잭션이 데이터베이스 쓰기가 성공적인 경우 커밋되고 데이터베이스 쓰기가 성공적이지 않은 경우 롤백됩니다.

데이터베이스 쓰기가 성공적인 경우:

- 데이터베이스 변경이 커밋되어 데이터베이스 트랜잭션이 종료됩니다.
- 캐시된 업데이트가 커밋되어 내부 캐시 버퍼가 지워집니다(2 단계).

데이터베이스 쓰기가 성공적이지 않은 경우:

- 데이터베이스 변경이 롤백되어 데이터베이스 트랜잭션이 종료됩니다.
- 캐시된 업데이트가 커밋되지 않아 내부 캐시에 그대로 유지됩니다.

OnUpdateRecord 이벤트 핸들러 생성과 사용에 대한 내용은 20-37 페이지의 "OnUpdateRecord 이벤트 핸들러 생성"을 참조하십시오. 캐시된 업데이트 적용 시 발생하는 업데이트 오류 처리에 대한 내용은 20-38 페이지의 "캐시된 업데이트 오류 처리"를 참조하십시오.

참고 각 데이터셋에 업데이트를 적용하는 순서가 중요하기 때문에 마스터/디테일 관계로 연결된 다중 데이터셋을 사용하는 경우에는 캐시된 업데이트 적용이 특히 까다롭습니다. 일반적으로 이 순서를 바꾸어 삭제된 레코드를 처리하는 경우를 제외한 경우에는 마스터 테이블을 디테일 테이블보다 먼저 업데이트해야 합니다. 이러한 어려움 때문에 마스터/디테일 폼에서 업데이트를 캐시할 때 클라이언트 데이터셋을 사용하는 것이 좋습니다. 클라이언트 데이터셋은 마스터/디테일 관계로 문제 순서 지정을 자동으로 처리합니다.

BDE 기반 업데이트를 적용하는 방법에는 두 가지가 있습니다.

- *ApplyUpdates* 메소드를 호출하면 데이터베이스 컴포넌트를 사용하여 업데이트를 적용할 수 있습니다. 업데이트가 완료되면 데이터베이스에서 업데이트 프로세스의 트랜잭션을 관리하고 데이터셋의 캐시를 지우는 것에 대한 모든 세부 사항을 처리하므로 이 메소드가 가장 간단한 방법입니다.
- 데이터셋의 *ApplyUpdates*와 *CommitUpdates* 메소드를 호출하여 단일 데이터셋의 업데이트를 적용할 수 있습니다. 데이터셋 레벨에서 업데이트 적용 시 업데이트 프로세스를 랩하는 트랜잭션을 명시적으로 코드화하고 *CommitUpdates*를 명시적으로 호출하여 캐시에서 업데이트를 커밋해야 합니다.

중요 내장 프로시저 또는 라이브 결과 집합을 반환하지 않는 SQL 쿼리의 업데이트를 적용하려면 *TUpdateSQL*을 사용하여 업데이트 수행 방법을 지정해야 합니다. 조인(두 개 이상의 테이블에 대한 쿼리)에 대한 업데이트에서 관련 테이블마다 *TUpdateSQL* 객체를 하나씩 제공하고 *OnUpdateRecord* 이벤트 핸들러를 사용하여 업데이트를 수행하는 객체를 호출해야 합니다. 자세한 내용은 20-40 페이지의 "업데이트 객체를 사용하여 데이터셋 업데이트"를 참조하십시오.

데이터베이스를 사용하여 캐시된 업데이트 적용

데이터베이스 연결의 컨텍스트에서 하나 이상의 데이터셋에 캐시된 업데이트를 적용하려면 데이터베이스 컴포넌트의 *ApplyUpdates* 메소드를 호출합니다. 다음 코드는 버튼 클릭 이벤트에 응답하여 *CustomersQuery* 데이터셋에 업데이트를 적용합니다.

```

procedure TForm1.ApplyButtonClick(Sender:TObject);
begin
    // for local databases such as Paradox, dBASE, and FoxPro
    // set TransIsolation to DirtyRead
    if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
        Database1.TransIsolation := tiDirtyRead;
        Database1.ApplyUpdates([CustomersQuery]);
    end;

```

위의 시퀀스는 자동 생성된 트랜잭션의 컨텍스트에서 데이터베이스에 캐시된 업데이트를 씁니다. 이 작업이 성공하면 트랜잭션을 커밋한 다음 캐시된 업데이트를 커밋합니다. 이 작업이 성공하지 못하면 트랜잭션을 롤백하고 업데이트 캐시를 변경하지 않고 그대로 둡니다. 후자의 경우 데이터셋의 *OnUpdateError* 이벤트를 통해 캐시된 업데이트 오류를 처리해야 합니다. 업데이트 오류 처리에 대한 자세한 내용은 20-38 페이지의 "캐시된 업데이트 오류 처리"를 참조하십시오.

데이터베이스 컴포넌트의 *ApplyUpdates* 메소드 호출에 대한 주요 이점은 데이터베이스에 연결된 데이터셋 컴포넌트의 수를 업데이트할 수 있다는 것입니다. 데이터베이스의 *ApplyUpdates* 메소드에 대한 매개변수는 *TDBDataSet*의 배열입니다. 예를 들어, 다음 코드는 두 가지 쿼리의 업데이트를 적용합니다.

```

if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
    Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);

```

데이터셋 컴포넌트 메소드로 캐시된 업데이트 적용

데이터셋의 *ApplyUpdates* 및 *CommitUpdates* 메소드를 사용하여 직접 개별 BDE 환경 데이터셋의 업데이트를 적용할 수 있습니다. 이러한 메소드들은 각각 업데이트 프로세스 중 한 단계를 캡슐화합니다.

- 1 *ApplyUpdates*는 캐시된 변경을 데이터베이스에 씁니다(1 단계).
- 2 *CommitUpdates*는 데이터베이스 쓰기가 성공적일 경우 내부 캐시를 지웁니다(2 단계).

다음 코드는 *CustomerQuery* 데이터셋의 트랜잭션 내에서 업데이트를 적용하는 방법을 보여 줍니다.

```

procedure TForm1.ApplyButtonClick(Sender:TObject)
begin
    Datasel1.StartTransaction;
    try
        if not (Databasel.IsSQLBased) and not (Databasel.TransIsolation = tiDirtyRead) then
            Databasel.TransIsolation := tiDirtyRead;
        CustomerQuery.ApplyUpdates;           { try to write the updates to the database }
        Databasel.Commit;                     { on success, commit the changes }
    except
        Databasel.Rollback;                   { on failure, undo any changes }
        raise;                               { raise the exception again to prevent a call to CommitUpdates }
    end;
    CustomerQuery.CommitUpdates;             { on success, clear the internal cache }
end;

```

ApplyUpdates 호출 도중 예외가 발생하면 데이터베이스 트랜잭션이 롤백됩니다. 트랜잭션을 롤백하면 원본으로 사용하는 데이터베이스 테이블이 변경되지 않습니다. *try...except* 블록의 *raise* 문이 예외를 재발생하므로 *CommitUpdates*에 대한 호출이 방지됩니다. *CommitUpdates*가 호출되지 않았으므로 업데이트의 내부 캐시가 지워지지 않아 오류 상황을 처리하고 업데이트를 재시도할 수 있습니다.

OnUpdateRecord 이벤트 핸들러 생성

BDE 활성 데이터셋이 그 캐시된 업데이트를 적용할 때 캐시에 레코딩된 변경을 통해 반복 사용되므로 기존 테이블의 해당 레코드에 적용하고자 시도합니다. 변경, 삭제 또는 새로 삽입된 레코드 각각의 업데이트가 적용되려고 할 때 데이터셋 컴포넌트의 *OnUpdateRecord* 이벤트가 발생합니다.

OnUpdateRecord 이벤트에 핸들러를 제공하면 현재 레코드의 업데이트가 실제로 적용되기 전에 작업을 수행할 수 있습니다. 이러한 작업에는 특수한 데이터 검증이 포함되어 다른 테이블, 특수 매개변수 대체를 업데이트하거나 다중 업데이트 객체를 실행할 수 있습니다. *OnUpdateRecord* 이벤트의 핸들러를 통해 업데이트 프로세스를 제어할 수 있습니다.

다음은 *OnUpdateRecord* 이벤트 핸들러의 뼈대 코드입니다.

```

procedure TForm1.DataSetUpdateRecord(DataSet:TDataSet;
    UpdateKind:TUpdateKind; var UpdateAction:TUpdateAction);
begin
    { perform updates here... }
end;

```

DataSet 매개변수는 업데이트가 있는 캐시된 데이터셋을 지정합니다.

UpdateKind 매개변수는 현재 레코드에 대해 수행되어야 하는 업데이트 타입을 나타냅니다. *UpdateKind*의 값은 *ukModify*, *ukInsert* 및 *ukDelete*입니다. 업데이트 객체를 사용하는 경우에는 업데이트 적용 시 이 매개변수를 업데이트 객체에 전달해야 합니다. 핸들러가 업데이트 종류에 따라 특수 처리를 수행하는 경우 이 매개변수를 조사해야 할 수도 있습니다.

UpdateAction 매개변수는 업데이트를 적용했는지 여부를 나타냅니다. *UpdateAction*의 값은 *uaFail*(기본값), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*입니다. 이벤트 핸들러가 업데이트를 성공적으로 적용한 경우 종료하기 전에 이 매개변수를 *uaApplied*로 변경합니다. 현재 레코드를 업데이트하지 않기로 결정하는 경우 해당 값을 *uaSkip*으로 변경하여 적용하지 않은 변경을 캐시에 저장합니다. *UpdateAction*의 값을 변경하지 않는 경우 데이터셋의 전체 업데이트 작업은 중지되고 예외가 발생합니다. *UpdateAction*을 *uaAbort*로 변경하여 오류 메시지(조용한 예외 발생)를 표시하지 않을 수 있습니다.

이러한 매개변수 외에도 현재 레코드에 연결된 필드 컴포넌트의 *OldValue* 및 *NewValue* 속성을 이용할 수 있습니다. *OldValue*는 데이터베이스에서 페치(fetch)한 원래 필드 값을 제공합니다. 이 속성은 업데이트할 데이터베이스 레코드를 찾는 데 유용합니다. *NewValue*는 적용하려는 업데이트의 편집된 값입니다.

중요 *OnUpdateError* 또는 *OnCalcFields* 이벤트 핸들러와 같은 *OnUpdateRecord* 이벤트 핸들러는 데이터셋의 현재 레코드를 변경하는 메소드를 호출할 수 없습니다.

다음 예제는 이러한 매개변수와 속성을 사용하는 방법을 보여 줍니다. 이 예제에서는 *UpdateTable*이라는 *TTable* 컴포넌트를 사용하여 업데이트를 적용합니다. 실제로는 업데이트 객체를 사용하는 것이 더 쉽지만 테이블을 사용하면 그 사용 가능성을 보다 뚜렷하게 알 수 있습니다.

```

procedure TForm1.EmpAuditUpdateRecord(DataSet:TDataSet;
  UpdateKind:TUpdateKind; var UpdateAction:TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue, DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            UpdateTable.Post;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukInsert:
          begin
            UpdateTable.Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukDelete: UpdateTable.Delete;
      end;
      UpdateAction := uaApplied;
    end;
end;

```

캐시된 업데이트 오류 처리

BDE(Borland Database Engine)는 업데이트를 적용하려고 할 때 특히 사용자 업데이트 충돌 및 기타 조건을 점검한 다음 오류를 보고합니다. 데이터셋 컴포넌트의 *OnUpdateError* 이벤트를 통해 오류를 확인하고 응답할 수 있습니다. 캐시된 업데이트

를 사용하는 경우에는 이 이벤트에 대한 핸들러를 생성해야 합니다. 그렇게 하지 않아서 오류가 발생하면 전체 업데이트 작업이 실패합니다.

다음은 *OnUpdateError* 이벤트 핸들러의 뼈대 코드입니다.

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind:TUpdateKind; var UpdateAction:TUpdateAction);
begin
  { ... perform update error handling here ... }
end;
```

*DataSet*은 업데이트가 적용될 데이터셋을 참조합니다. 이 데이터셋을 사용하여 오류 처리 중 새 값과 이전 값에 액세스할 수 있습니다. 각 레코드에 있는 필드의 원래 값은 *OldValue*라는 읽기 전용 *TField* 속성으로 저장됩니다. 변경된 값은 *NewValue*라는 아날로그 *TField* 속성으로 저장됩니다. 이러한 값들은 이벤트 핸들러에서 업데이트 값을 조사하고 변경하는 유일한 방법을 제공합니다.

경고 *Next* 및 *Prior*와 같은 현재 레코드를 변경하는 데이터셋 메소드를 호출하지 마십시오. 그렇게 하면 이벤트 핸들러에서 무한 순환을 입력합니다.

E 매개변수는 보통 *EDBEngineError* 타입입니다. 이 예외 타입에서 오류 핸들러에서 사용자가 볼 수 있는 오류 메시지를 추출할 수 있습니다. 예를 들어, 다음 코드는 대화 상자의 캡션에서 오류 메시지를 표시하는 데 사용될 수 있습니다.

```
ErrorLabel.Caption := E.Message;
```

이 매개변수는 업데이트 오류의 실제 원인을 결정하는 데 사용될 수도 있습니다. *EDBEngineError*에서 특정 오류 코드를 추출하고 이에 따라 적절한 조치를 취할 수 있습니다.

UpdateKind 매개변수는 오류를 생성하는 업데이트 타입을 나타냅니다. 업데이트 타입의 처리에 따라 오류 핸들러가 특별한 조치를 취하지 않을 경우에는 코드에서 이 매개변수를 이용하지 않습니다.

다음 표에는 *UpdateKind*의 가능한 값이 나열되어 있습니다.

표 20.7 UpdateKind 값

값	의미
<i>ukModify</i>	기존 레코드를 편집하여 오류가 발생했습니다.
<i>ukInsert</i>	새 레코드를 삽입하여 오류가 발생했습니다.
<i>ukDelete</i>	기존 레코드를 삭제하여 오류가 발생했습니다.

*UpdateAction*은 이벤트 핸들러 종료 시 업데이트 프로세스 진행 방법을 BDE에게 지시합니다. 업데이트 오류 핸들러를 처음으로 호출할 때 이 매개변수의 값은 항상 *uaFail*로 설정됩니다. 오류를 발생시킨 레코드의 오류 상황과 이를 해결하기 위해 수행한 작업에 따라 핸들러를 종료하기 전에 보통 *UpdateAction*을 다른 값으로 설정합니다.

- 오류 핸들러에서 핸들러를 호출한 오류 상황을 해결할 수 있을 경우 *UpdateAction*을 적절한 작업으로 설정하여 종료합니다. 해결한 오류 상황에 대해서는 *UpdateAction*을 *uaRetry*로 설정하여 레코드의 업데이트를 다시 적용합니다.

- *uaSkip*으로 설정되어 있을 때 오류를 발생시킨 행의 업데이트는 건너뛰고 레코드의 업데이트는 다른 모두 업데이트가 완료된 후에 캐시에 그대로 남아 있습니다.
- *uaFail*과 *uaAbort*는 전체 업데이트 작업이 종료되도록 합니다. *uaFail*은 예외를 발생시키고 오류 메시지를 표시합니다. *uaAbort*는 예외 숨기기를 발생시키고 오류 메시지는 표시하지 않습니다.

다음 코드는 업데이트 오류가 키 위반과 관계가 있는지 확인하는 *OnUpdateError* 이벤트 핸들러를 보여 주며, 관계가 있으면 *UpdateAction* 매개변수를 *uaSkip*으로 설정합니다.

```
{ Add 'Bde' to your uses clause for this example }
if (E is EDBEngineError) then
  with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip           { key violation, just skip this record }
    else
      UpdateAction := uaAbort;        { don't know what's wrong, abort the update }
    end;
```

참고 캐시된 업데이트의 적용 도중 오류가 발생하면 예외가 발생하고 오류 메시지가 표시됩니다. *ApplyUpdates*가 try...except 생성 내에서 호출되지 않으면 *OnUpdateError* 이벤트 핸들러 내에서 오류 메시지가 표시되므로 애플리케이션에서 동일한 오류 메시지를 두 번 표시하게 됩니다. 오류 메시지 중복을 방지하려면 *UpdateAction*을 *uaAbort*로 설정하여 시스템에서 생성되는 오류 메시지 표시를 해제합니다.

업데이트 객체를 사용하여 데이터셋 업데이트

BDE 활성 데이터셋이 내장 프로시저 또는 "라이브"가 아닌 쿼리를 나타낼 때 데이터셋에서 직접 업데이트를 적용할 수 없습니다. 이러한 데이터셋은 클라이언트 데이터셋을 사용하여 업데이트를 캐시할 경우 문제를 유발할 수도 있습니다. 업데이트를 캐시하기 위해 BDE를 사용하든지 클라이언트 데이터셋을 사용하든지 이러한 문제 데이터셋을 업데이트 객체로 처리할 수 있습니다.

- 1 클라이언트 데이터셋을 사용하는 경우 *TBDEClientDataSet*이 아닌 *TClientDataSet*이 있는 외부 프로바이더 컴포넌트를 사용합니다. 이렇게 하면 BDE 활성 소스 데이터셋의 *UpdateObject* 속성을 설정할 수 있습니다(3 단계).
- 2 *TUpdateSQL* 컴포넌트를 BDE 활성 데이터셋과 동일한 데이터 모듈에 추가합니다.
- 3 데이터 모듈에서 BDE 활성 데이터셋 컴포넌트의 *UpdateObject* 속성을 *TUpdateSQL* 컴포넌트로 설정합니다.
- 4 업데이트 객체의 *ModifySQL*, *InsertSQL* 및 *DeleteSQL* 속성을 사용하여 업데이트 수행에 필요한 SQL 문을 지정합니다. 이러한 문을 만드는 데 도움을 주는 *UpdateSQL* 에디터를 사용할 수 있습니다.
- 5 데이터셋을 닫습니다.
- 6 데이터셋 컴포넌트의 *CachedUpdates* 속성을 *True*로 설정하거나 데이터셋 프로바이더를 사용하여 데이터셋을 클라이언트 데이터셋에 연결합니다.

7 데이터셋을 다시 엽니다.

참고

때로는 다중 업데이트 객체를 사용해야 하는 경우가 있습니다. 예를 들어, 여러 데이터셋의 데이터를 나타내는 다중 테이블 조인 또는 내장 프로시저를 업데이트할 때 업데이트할 각 테이블마다 *TUpdateSQL* 객체를 하나씩 제공해야 합니다. 다중 업데이트 객체를 사용하는 경우에는 *UpdateObject* 속성을 설정하여 업데이트 객체를 데이터셋과 연결할 수 없습니다. 그 대신 *OnUpdateRecord* 이벤트 핸들러(업데이트를 캐시하기 위해 BDE 사용 시) 또는 *BeforeUpdateRecord* 이벤트 핸들러(클라이언트 데이터셋 사용 시)에서 업데이트 객체를 수동으로 호출해야 합니다.

업데이트 객체는 세 개의 *TQuery* 컴포넌트를 실제로 캡슐화합니다. 이러한 쿼리 컴포넌트는 각각 단일 업데이트 작업을 수행합니다. 첫 번째 쿼리 컴포넌트는 기존 레코드를 수정하는 SQL UPDATE 문을 제공하고, 두 번째 쿼리 컴포넌트는 새로운 레코드를 테이블에 추가하는 INSERT 문을 제공하며, 세 번째 컴포넌트는 테이블에서 레코드를 제거하는 DELETE 문을 제공합니다.

데이터 모듈에 업데이트 컴포넌트를 둘 때 캡슐화되는 쿼리 컴포넌트는 보이지 않습니다. 쿼리 컴포넌트는 SQL 문을 제공하는 세 개의 업데이트 속성에 따라 런타임 시 업데이트 컴포넌트에 의해 생성됩니다.

- *ModifySQL*은 UPDATE 문을 지정합니다.
- *InsertSQL*은 INSERT 문을 지정합니다.
- *DeleteSQL*은 DELETE 문을 지정합니다.

런타임 시 업데이트 컴포넌트를 사용하여 업데이트를 적용할 때 다음을 수행합니다.

- 1 현재 레코드를 수정, 삽입 또는 삭제할지 여부에 따라 실행할 SQL 문을 선택합니다.
- 2 SQL 문에 대한 매개변수 값을 제공합니다.
- 3 SQL 문을 준비 및 실행하고 지정된 업데이트를 수행합니다.

업데이트 컴포넌트에 대한 SQL 문 생성

연결된 데이터셋에서 레코드를 업데이트하려면 업데이트 객체는 세 개의 SQL 문 중 하나를 사용합니다. 각 업데이트 객체는 단일 테이블만 업데이트할 수 있으므로 객체의 업데이트 문은 각각 동일한 기준 테이블을 참조해야 합니다.

세 SQL 문은 업데이트에 캐시된 레코드를 삭제, 삽입 또는 수정합니다. 이러한 문을 업데이트 객체의 *DeleteSQL*, *InsertSQL* 및 *ModifySQL* 속성으로 제공해야 합니다. 디자인 타임 또는 런타임 시 이러한 값들을 제공할 수 있습니다. 예를 들어, 다음 코드는 런타임 시 *DeleteSQL* 속성에 대한 값을 지정합니다.

```
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

디자인 타임 시 Update SQL 에디터를 사용하여 업데이트를 적용하는 SQL 문을 만들 수 있습니다.

업데이트 객체는 데이터셋의 원래 및 업데이트된 필드 값을 참조하는 매개변수에 자동 매개변수 바인딩을 제공합니다. 그러므로 보통 SQL 문 작성 시 특수한 형식의 이름을 가지는 매개변수를 삽입합니다. 이러한 매개변수 사용에 대한 내용은 20-43 페이지의 "업데이트 SQL 문의 매개변수 대체에 대한 이해"를 참조하십시오.

Update SQL 에디터 사용

다음과 같은 방법으로 업데이트 컴포넌트에 대한 SQL 문을 생성합니다.

- 1 Object Inspector를 사용하여 데이터셋 *UpdateObject* 속성의 드롭다운 목록에서 업데이트 객체 이름을 선택합니다. 이 단계는 다음 단계에서 호출하는 Update SQL 에디터가 SQL 생성 옵션을 사용하는 데 적합한 기본값을 결정할 수 있도록 합니다.
- 2 업데이트 객체를 마우스 오른쪽 버튼으로 클릭한 다음 컨텍스트 메뉴에서 Update SQL Editor를 선택합니다. 이렇게 하면 Update SQL 에디터가 표시됩니다. 에디터는 원본으로 사용하는 데이터셋 및 이에 제공하는 값에 따라 업데이트 객체의 *ModifySQL*, *InsertSQL* 및 *DeleteSQL* 속성에 대한 SQL 문을 생성합니다.

Update SQL 에디터에는 두 페이지가 있습니다. Options 페이지는 편집기를 처음으로 호출할 때 보입니다. Table Name 콤보 상자를 사용하여 업데이트할 테이블을 선택합니다. 테이블 이름을 지정할 때 Key Fields와 Update Fields 목록 상자는 사용 가능한 열로 구성됩니다.

Update Fields 목록 상자는 업데이트할 열을 표시합니다. 테이블을 처음으로 지정할 때 Update Fields 목록 상자의 모든 열을 포함하는 것으로 선택합니다. 원하는 만큼 많은 필드를 선택할 수 있습니다.

Key Fields 목록 상자는 업데이트하는 동안 키로 사용할 열을 지정하는 데 사용됩니다. Paradox, dBASE 및 FoxPro의 경우 여기에 지정하는 열이 기존 인덱스에 상응해야 하지만 원격 SQL 데이터베이스에서는 그러한 요구 사항이 없습니다. Key Fields를 설정하는 대신 Primary Keys 버튼을 클릭하여 테이블의 기본 인덱스에 기반한 업데이트에 대한 키 필드를 선택할 수 있습니다. Dataset Defaults를 클릭하여 선택 목록을 원래 상태로 반환합니다(모든 필드가 키로 선택되고 모든 필드가 업데이트에 선택됩니다).

사용하고 있는 서버에서 필드 이름을 인용 부호 안에 넣어야 할 경우 Quote Field Names 체크 박스를 선택합니다.

테이블을 지정한 후 키 열을 선택한 다음 업데이트 열을 선택하고, SQL 문을 생성할 Generate SQL을 클릭하여 업데이트 컴포넌트의 *ModifySQL*, *InsertSQL* 및 *DeleteSQL* 속성에 연결합니다. 대부분의 경우 자동 생성된 SQL 문을 미세 조정(fine tune)하게 됩니다.

생성된 SQL 문을 보거나 수정하려면 SQL 페이지를 선택합니다. SQL 문을 생성한 경우에는 이 페이지 선택 시 *ModifySQL* 속성의 문이 SQL Text 메모 상자에 이미 표시되어 있습니다. 원하는 대로 상자에서 문을 편집할 수 있습니다.

중요 생성된 SQL 문은 업데이트 문을 생성하는 출발점입니다. 올바르게 실행하도록 만들려면 SQL 문을 수정해야 합니다. 예를 들어, NULL 값을 포함하는 데이터 사용 시 생성된 필드 변수를 사용하지 않고,

```
WHERE field IS NULL
```


위와 같이 WHERE 절을 수정해야 합니다. 이러한 문을 그대로 받아들이기 전에 각 문을 직접 테스트하십시오.

Statement Type 라디오 버튼을 사용하여 생성된 SQL 문 사이에서 전환하고 원하는 대로 편집합니다.

이 문을 그대로 받아들여 업데이트 컴포넌트의 SQL 속성에 연결하려면 OK를 클릭합니다.

업데이트 SQL 문의 매개변수 대체에 대한 이해

업데이트 SQL 문은 레코드 업데이트의 이전 값 또는 새 값을 대체할 수 있게 해주는 매개변수를 대체하는 특수한 형식을 사용합니다. Update SQL 에디터에서 해당 문을 생성할 때 사용할 필드 값이 결정됩니다. 업데이트 SQL을 쓸 때 사용할 필드 값을 지정합니다.

매개변수 이름이 테이블의 열 이름과 일치할 때 레코드에 대해 캐시된 업데이트의 필드에 있는 새 값이 매개변수의 값으로 자동 사용됩니다. 매개변수 이름이 "OLD_" 문자열이 앞에 오는 열 이름과 일치하면 필드의 이전 값이 사용됩니다. 예를 들어, 아래 업데이트 SQL 문에서 매개변수 :LastName이 삽입된 레코드에 대해 캐시된 업데이트의 새 필드 값에 자동으로 입력됩니다.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

새 필드 값은 보통 *InsertSQL* 및 *ModifySQL* 문에 사용됩니다. 수정된 레코드의 업데이트에서 업데이트 캐시의 새 필드 값이 UPDATE 문에서 사용되어 업데이트된 기준 테이블의 이전 필드 값을 대체합니다.

삭제된 레코드의 경우 새 값이 없으므로 *DeleteSQL* 속성은 ":OLD_FieldName" 구문을 사용합니다. 이전 필드 값도 수정되거나 삭제된 업데이트에 대한 SQL 문의 WHERE 절에서 보통 사용되어 업데이트하거나 삭제할 레코드를 결정합니다.

UPDATE 또는 DELETE 업데이트 SQL 문의 WHERE 절에서 적어도 최소의 매개변수 수를 제공하여 캐시된 데이터로 업데이트되는 기준 테이블의 레코드를 고유하게 식별합니다. 예를 들어, 고객의 이름에서 성만 사용하는 고객 목록은 기준 테이블의 정확한 레코드를 고유하게 식별하기에 충분하지 않습니다. "Smith"라는 성을 가지는 레코드가 매우 많을 것이기 때문입니다. 하지만 성, 이름 및 전화 번호에 대한 매개변수를 사용하면 구분하기에 충분한 조합이 됩니다. 고객 번호와 같은 고유 필드가 있다면 더 좋을 것입니다.

참고 원래 필드 값이나 편집된 필드 값을 참조하지 않는 매개변수를 포함하는 SQL 문을 만들 경우, 업데이트 객체는 해당 값의 바인딩 방법을 모릅니다. 하지만 업데이트 객체의 *Query* 속성을 사용하여 이 작업을 수동으로 수행할 수 있습니다. 자세한 내용은 20-48 페이지의 "업데이트 컴포넌트의 쿼리 속성 사용"을 참조하십시오.

업데이트 SQL 문 작성

디자인 타임 시 Update SQL 에디터를 사용하여 *DeleteSQL*, *InsertSQL* 및 *ModifySQL* 속성에 대한 SQL 문을 쓸 수 있습니다. Update SQL 에디터를 사용하지 않는 경우나 생성된 문을 수정하려는 경우에는 기존 테이블의 레코드를 삭제, 삽입 및 수정할 때 다음 지침에 유의해야 합니다.

DeleteSQL 속성은 DELETE 명령이 있는 SQL 문만 포함할 수 있습니다. 업데이트할 기존 테이블의 이름은 FROM 절에서 지정해야 합니다. SQL 문은 업데이트 캐시에서 삭제된 레코드에 상응하는 기존 테이블의 레코드만 삭제하므로 WHERE 절을 사용합니다. WHERE 절에서 하나 이상의 필드에 매개변수를 사용하여 캐시된 업데이트 레코드에 해당하는 기존 테이블의 레코드를 고유하게 식별합니다. 매개변수 이름이 필드와 동일한 이름으로 지정되었고 "OLD_"가 앞에 오는 경우 캐시된 업데이트 레코드의 해당 필드 값이 매개변수에 자동으로 주어집니다. 매개변수가 다른 방법으로 명명된 경우에는 매개변수 값을 제공해야 합니다.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

일부 테이블 타입에서는 NULL 값을 포함하는 레코드를 식별하는 데 필드가 사용된 경우, 기존 테이블에서 레코드를 찾을 수 없습니다. 이러한 경우 해당 레코드에 대해 삭제 업데이트가 실패합니다. 이를 적용하려면 non-NULL 값의 조건 이외에 IS NULL predicate를 사용하여 NULL을 포함하는 필드에 조건을 추가합니다. 예를 들어, FirstName 필드에 NULL 값이 포함되어 있을 경우를 보면 다음과 같습니다.

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

InsertSQL 문은 INSERT 명령이 있는 SQL 문만 포함해야 합니다. 업데이트할 기존 테이블의 이름은 INTO 절에서 지정해야 합니다. VALUES 절에서 쉼표로 구분된 매개변수 목록을 제공합니다. 매개변수 이름이 필드와 동일한 이름으로 지정된 경우 캐시된 업데이트 레코드의 값이 매개변수에 자동으로 주어집니다. 매개변수가 다른 방법으로 명명된 경우에는 매개변수 값을 제공해야 합니다. 매개변수 목록은 새로 삽입된 레코드의 필드에 값을 제공합니다. 문에 나열된 필드 수만큼 많은 값 매개변수가 있어야 합니다.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

ModifySQL 문은 UPDATE 명령이 있는 SQL 문만 포함해야 합니다. 업데이트할 기존 테이블의 이름은 FROM 절에서 지정해야 합니다. SET 절에서 값 할당을 하나 이상 포함합니다. SET 절 할당의 값이 필드와 동일한 이름으로 명명된 매개변수인 경우 캐시의 업데이트된 레코드에 있는 동일한 이름의 필드 값이 매개변수에 자동으로 주어집니다. 매개변수가 필드와 동일한 이름으로 명명되지 않고 값을 수동으로 제공하면 다른 매개변수를 사용하여 추가 필드 값을 할당할 수 있습니다. *DeleteSQL* 문의 경우처럼 WHERE 절을 제공하여 필드와 동일한 이름으로 명명되고 "OLD_"가 앞에 오는 매개변수를 통해 업데이트될 기존 테이블의 레코드를 고유하게 식별합니다. 아래 업데이트 문에서 매개변수 :ItemNo에 값이 자동으로 주어지지만 :Price에는 값이 자동으로 주어지지 않습니다.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

위의 업데이트 SQL을 고려할 때 애플리케이션 최종 사용자가 기존 레코드를 수정하는 경우를 예로 들어 봅니다. ItemNo 필드의 원래 값은 999입니다. 캐시된 데이터셋에 연결된 그리드에서 최종 사용자는 ItemNo 필드 값을 123으로 변경하고 Amount를 20으로 변경합니다. ApplyUpdates 메소드를 호출하면 이 SQL 문은 매개변수 :OLD_ItemNo의 이전 필드 값을 사용하여 ItemNo 필드가 999인 기준 테이블의 모든 레코드에 영향을 줍니다. 이러한 레코드에서 매개변수 :ItemNo와 그리드의 값을 사용하여 ItemNo 필드 값은 123으로 변경되고 Amount는 20으로 변경됩니다.

다중 업데이트 객체 사용

업데이트 데이터셋에서 참조되는 두 개 이상의 기준 테이블을 업데이트해야 하는 경우에는 다중 업데이트 객체를 사용해야 합니다. 각 기준 테이블마다 하나씩 업데이트됩니다. 데이터셋 컴포넌트의 *UpdateObject*는 업데이트 객체 하나만 데이터셋에 연결되도록 허용하므로 해당 *DataSet* 속성을 데이터셋의 이름에 설정하여 각 업데이트 객체를 각 데이터셋에 연결해야 합니다.

팁 다중 업데이트 객체 사용 시 외부 프로바이더가 있는 *TClientDataSet* 대신 *TBDEClientDataSet*을 사용할 수 있습니다. 그 이유는 소스 데이터셋의 *UpdateObject* 속성을 설정할 필요가 없기 때문입니다.

업데이트 객체의 *DataSet* 속성은 Object Inspector에서 디자인 타임 시 사용할 수 없습니다. 런타임 시에만 이 속성을 설정할 수 있습니다.

```
UpdatesSQL1.DataSet := Query1;
```

업데이트 객체는 이 데이터셋을 사용하여 매개변수 대체의 원래 필드 값과 업데이트된 필드 값을 얻으며, BDE 활성 데이터셋인 경우 이 데이터셋을 사용하여 업데이트 적용 시 사용할 세션과 데이터베이스를 식별합니다. 매개변수 대체는 제대로 작동하므로 업데이트 객체의 *DataSet* 속성은 업데이트된 필드 값을 포함하는 데이터셋이어야 합니다. BDE 활성 데이터셋을 사용하여 업데이트를 캐시하는 경우에는 BDE 활성 데이터셋이어야 합니다. 클라이언트 데이터셋을 사용하는 경우에는 *BeforeUpdateRecord* 이벤트 핸들러에 대한 매개변수로 제공되는 클라이언트 데이터셋입니다.

업데이트 객체가 데이터셋의 *UpdateObject* 속성에 할당되지 않았으면 그 SQL 문은 *ApplyUpdates* 호출 시 자동으로 실행되지 않습니다. 레코드를 업데이트하려면 *OnUpdateRecord* 이벤트 핸들러 (BDE를 사용하여 업데이트를 캐시하는 경우) 또는 *BeforeUpdateRecord* 이벤트 핸들러 (클라이언트 데이터셋을 사용하여 업데이트를 캐시하는 경우)에서 업데이트 객체를 수동으로 호출해야 합니다. 이벤트 핸들러에서 수행해야 할 최소한의 작업은 다음과 같습니다.

- 클라이언트 데이터셋을 사용하여 업데이트를 캐시하는 경우 업데이트 객체의 *DatabaseName*과 *SessionName* 속성은 소스 데이터셋의 *DatabaseName*과 *SessionName* 속성으로 설정됩니다.
- 이벤트 핸들러는 업데이트 객체의 *ExecSQL* 또는 *Apply* 메소드를 호출해야 합니다. 이렇게 하면 업데이트가 필요한 각 레코드에 대해 업데이트 객체가 호출됩니다. 업데이트 문에 대한 자세한 내용은 아래의 "SQL 문 실행"을 참조하십시오.

- 이벤트 핸들러의 *UpdateAction* 매개변수를 *uaApplied (OnUpdateRecord)*로 설정하거나 *Applied* 매개변수를 *True(BeforeUpdateRecord)*로 설정합니다.

데이터 확인, 데이터 수정 또는 각 레코드의 업데이트에 따른 기타 작업을 옵션으로 수행할 수 있습니다.

경고 *OnUpdateRecord* 이벤트 핸들러에서 업데이트 객체의 *ExecSQL* 또는 *Apply* 메소드를 호출하는 경우 데이터셋의 *UpdateObject* 속성을 해당 업데이트 객체에 설정하지 마십시오. 그렇지 않으면 각 레코드의 업데이트를 적용하기 위해 다시 시도해야 합니다.

SQL 문 실행

다중 업데이트 객체를 사용할 때 해당 *UpdateObject* 속성을 설정하면 업데이트 객체와 데이터셋이 연결되지 않습니다. 그 결과로 업데이트를 적용하면 적합한 문이 자동으로 실행되지 않습니다. 그 대신 코드에서 업데이트 객체를 명시적으로 호출해야 합니다.

업데이트 객체를 호출하는 방법에는 두 가지가 있습니다. SQL 문이 필드 값을 나타내는 매개변수를 사용하는지 여부에 따라 적합한 방법을 선택합니다.

- 실행할 SQL 문이 매개변수를 사용하는 경우에는 *Apply* 메소드를 호출합니다.
- 실행할 SQL 문이 매개변수를 사용하지 않는 경우에는 *ExecSQL* 메소드를 호출하는 것이 보다 효율적입니다.

참고 SQL 문이 원래 필드 값과 업데이트된 필드 값에 대한 내장 타입이 아닌 매개변수를 사용하는 경우에는 *Apply* 메소드에 의해 제공되는 매개변수 대체 대신 매개변수 값을 수동으로 제공해야 합니다. 매개변수 값을 수동으로 제공하는 것에 대해서는 20-48 페이지의 "업데이트 컴포넌트의 쿼리 속성 사용"을 참조하십시오.

업데이트 객체의 SQL 문에 있는 매개변수에 대한 기본(default) 매개변수 대체에 대한 내용은 20-43 페이지의 "업데이트 SQL 문의 매개변수 대체에 대한 이해"를 참조하십시오.

Apply 메소드 호출

업데이트 컴포넌트의 *Apply* 메소드는 현재 레코드의 업데이트를 수동으로 적용합니다. 이 프로세스에는 두 단계가 있습니다.

- 레코드의 초기 필드 값과 편집된 필드 값은 해당 SQL 문의 매개변수에 바인딩되어 있습니다.
- SQL 문이 실행됩니다.

Apply 메소드를 호출하여 업데이트 캐시의 현재 레코드에 대한 업데이트를 적용합니다. *Apply* 메소드는 데이터셋의 *OnUpdateRecord* 이벤트 핸들러 또는 프로바이더의 *BeforeUpdateRecord* 이벤트 핸들러에서 가장 빈번하게 호출됩니다.

경고 데이터셋의 *UpdateObject* 속성을 사용하여 데이터셋과 업데이트 객체를 연결하는 경우 *Apply*가 자동으로 호출됩니다. 이 경우 *OnUpdateRecord* 이벤트 핸들러에서 *Apply*를 호출하지 마십시오. 그러면 현재 레코드의 업데이트를 적용하기 위해 다시 시도해야 합니다.

OnUpdateRecord 이벤트 핸들러는 *TUpdateKind* 타입의 *UpdateKind* 매개변수로 적용되어야 하는 업데이트 타입을 표시합니다. 이 매개변수를 *Apply* 메소드로 전달하여 사용할 업데이트 SQL 문을 표시해야 합니다. 다음 코드는 *BeforeUpdateRecord* 이벤트 핸들러를 사용하여 이를 나타냅니다.

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender:TObject;
SourceDS:TDataSet;
DeltaDS:TCustomClientDataSet; UpdateKind:TUpdateKind; var Applied:Boolean);
begin
with UpdateSQL1 do
begin
DataSet := DeltaDS;
DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
SessionName := (SourceDS as TDBDataSet).SessionName;
Apply(UpdateKind);
Applied := True;
end;
end;
```

ExecSQL 메소드 호출

업데이트 컴포넌트의 *ExecSQL* 메소드는 현재 레코드에 대한 업데이트를 수동으로 적용합니다. *ExecSQL*은 *Apply* 메소드와 달리 실행하기 전에 SQL 문에서 매개변수를 바인딩하지 않습니다. *ExecSQL* 메소드는 *OnUpdateRecord* 이벤트 핸들러 (BDE 사용 시) 또는 *BeforeUpdateRecord* 이벤트 핸들러 (클라이언트 데이터셋 사용 시)에서 가장 빈번하게 호출됩니다.

*ExecSQL*은 매개변수 값을 바인딩하지 않으므로 업데이트 객체의 SQL 문에 매개변수가 없을 때 주로 사용됩니다. 매개변수가 없는 경우에도 *Apply*를 대신 사용할 수 있지만 *ExecSQL*은 매개변수를 확인하지 않으므로 보다 효율적입니다.

SQL 문에 매개변수가 있는 경우 명시적으로 매개변수를 바인딩해야 *ExecSQL*을 호출할 수 있습니다. BDE를 사용하여 업데이트를 캐시하는 경우 업데이트 객체의 *DataSet* 속성을 설정한 다음 그 *SetParams* 메소드를 호출하여 명시적으로 매개변수를 바인딩할 수 있습니다. 클라이언트 데이터셋을 사용하여 업데이트를 캐시할 때 *TUpdateSQL*에 의해 유지되는 원본 쿼리 객체에 대한 매개변수를 제공해야 합니다. 이러한 방법에 대한 자세한 내용은 20-48 페이지의 "업데이트 컴포넌트의 쿼리 속성 사용"을 참조하십시오.

경고 데이터셋의 *UpdateObject* 속성을 사용하여 데이터셋과 업데이트 객체를 연결하는 경우 *ExecSQL*이 자동으로 호출됩니다. 이 경우 *OnUpdateRecord* 이벤트 핸들러에서 *ExecSQL*을 호출하지 마십시오. 그러면 현재 레코드의 업데이트를 적용하기 위해 다시 시도해야 합니다.

*OnUpdateRecord*와 *BeforeUpdateRecord* 이벤트 핸들러는 *TUpdateKind* 타입의 *UpdateKind* 매개변수로 적용되어야 하는 업데이트 타입을 표시합니다. 이 매개변수를 *ExecSQL* 메소드로 전달하여 사용할 업데이트 SQL 문을 표시해야 합니다. 다음 코드는 *BeforeUpdateRecord* 이벤트 핸들러를 사용하여 이를 나타냅니다.

```

procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender:TObject; SourceDS:TDataSet;
    DeltaDS:TCustomClientDataSet; UpdateKind:TUpdateKind; var Applied:Boolean);
begin
    with UpdateSQL1 do
        begin
            DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
            SessionName := (SourceDS as TDBDataSet).SessionName;
            ExecSQL(UpdateKind);
            Applied := True;
        end;
    end;

```

업데이트 프로그램 실행 도중에 예외가 발생하는 경우, 예외가 정의되어 있으면 *OnUpdateError* 이벤트에서 계속 실행됩니다.

업데이트 컴포넌트의 쿼리 속성 사용

업데이트 컴포넌트의 *Query* 속성은 *DeleteSQL*, *InsertSQL* 및 *ModifySQL* 문을 구현하는 쿼리 컴포넌트에 대한 액세스를 제공합니다. 대부분의 애플리케이션에서는 이러한 쿼리 컴포넌트를 직접 액세스할 필요가 없습니다. *DeleteSQL*, *InsertSQL* 및 *ModifySQL* 속성을 사용하여 이러한 쿼리들이 실행하는 문을 지정하고 업데이트 객체의 *Apply* 또는 *ExecSQL* 메소드를 호출하여 이들을 실행합니다. 하지만 쿼리 컴포넌트를 직접 처리해야 하는 경우가 종종 있습니다. *Query* 속성이 이전 필드 값과 새 필드 값에 대한 업데이트 객체의 자동적인 매개변수 바인딩에 의존하기보다는 사용자 고유의 값을 SQL 문의 매개변수에 제공하는 경우에 특히 유용합니다.

참고 *Query* 속성은 런타임 시에만 액세스할 수 있습니다.

Query 속성은 *TUpdateKind* 값에 인덱스됩니다.

- *ukModify*의 인덱스를 사용하면 기존 레코드를 업데이트하는 쿼리에 액세스할 수 있습니다.
- *ukInsert*의 인덱스를 사용하면 새 레코드를 삽입하는 쿼리에 액세스할 수 있습니다.
- *ukDelete*의 인덱스를 사용하면 레코드를 삭제하는 쿼리에 액세스할 수 있습니다.

다음 코드는 자동으로 바인딩되지 않는 매개변수 값을 제공하기 위해 *Query* 속성을 사용하는 방법을 보여 줍니다.

```

procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender:TObject; SourceDS:TDataSet;
    DeltaDS:TCustomClientDataSet; UpdateKind:TUpdateKind; var Applied:Boolean);
begin
    UpdateSQL1.DataSet := DeltaDS; { required for the automatic parameter substitution }
    with UpdateSQL1.Query[UpdateKind] do
        begin
            { Make sure the query has the correct DatabaseName and SessionName }
            DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
            SessionName := (SourceDS as TDBDataSet).SessionName;
            ParamByName('TimeOfUpdate').Value = Now;
        end;
    end;

```

```

end;
UpdateSQL1.Apply(UpdateKind); { now perform automatic substitutions and execute }
Applied := True;
end;

```

TBatchMove 사용

*TBatchMove*는 데이터셋을 복제하고, 한 데이터셋에서 다른 데이터셋으로 레코드를 추가하고, 한 데이터셋의 레코드를 다른 데이터셋의 레코드로 업데이트하고, 다른 데이터셋의 레코드와 일치하는 데이터셋의 레코드를 삭제할 수 있게 해주는 BDE(Borland Database Engine) 기능을 캡슐화합니다. *TBatchMove*는 다음 작업을 수행할 때 가장 많이 사용됩니다.

- 분석이나 다른 작업을 위해 서버의 데이터를 로컬 데이터 소스로 다운로드.
- 확장 작업의 일환으로 데스크탑 데이터베이스를 원격 서버의 테이블로 이동.

배치 이동 컴포넌트는 소스 테이블에 상응하는 대상의 테이블을 생성한 다음 적절한 열 이름과 데이터 타입을 자동으로 매핑합니다.

배치 이동 컴포넌트 생성

다음과 같은 방법으로 배치 이동 컴포넌트를 생성합니다.

- 1 레코드를 가져올 데이터셋(*Source* 데이터셋)의 테이블 또는 쿼리 컴포넌트를 폼 또는 데이터 모듈에 둡니다.
- 2 레코드를 이동할 데이터셋(*Destination* 데이터셋)을 폼 또는 데이터 모듈에 둡니다.
- 3 컴포넌트 팔레트의 BDE 페이지에 있는 *TBatchMove* 컴포넌트를 데이터 모듈 또는 폼에 두고 그 *Name* 속성을 애플리케이션에 적합한 고유 값에 설정합니다.
- 4 레코드를 복사, 추가 또는 업데이트할 테이블 이름에 배치 이동 컴포넌트의 *Source* 속성을 설정합니다. 사용 가능한 데이터셋 컴포넌트의 드롭다운 목록에서 테이블을 선택할 수 있습니다.
- 5 *Destination* 속성을 생성, 추가 또는 업데이트할 데이터셋에 설정합니다. 사용 가능한 데이터셋 컴포넌트의 드롭다운 목록에서 대상 테이블을 선택할 수 있습니다.
 - 추가, 업데이트 또는 삭제 중인 경우 *Destination*은 기존 데이터베이스 테이블을 나타내야 합니다.
 - 테이블을 복사 중이고 *Destination*이 기존 테이블을 나타내는 경우 배치 이동을 실행하면 대상 테이블의 모든 현재 데이터를 겹쳐 씁니다.
 - 기존 테이블을 복사하여 완전히 새로 테이블을 생성하는 경우 결과 테이블은 복사하는 테이블 컴포넌트의 *Name* 속성에 지정된 이름을 가집니다. 결과 테이블 타입은 *DatabaseName* 속성에 의해 지정된 서버에 적합한 구조를 가집니다.

- 6 *Mode* 속성을 설정하여 수행할 작업 타입을 나타냅니다. 유효한 작업은 *batAppend* (기본값), *batUpdate*, *batAppendUpdate*, *batCopy*, *batDelete*입니다. 이러한 모드에 대한 내용은 20-50 페이지의 "배치 이동 모드 지정"을 참조하십시오.
- 7 옵션으로 *Transliterate* 속성을 설정합니다. *Transliterate*를 *True*(기본값)로 설정하면 필요에 따라 문자 데이터가 *Source* 데이터셋의 문자 집합에서 *Destination* 데이터셋의 문자 집합으로 번역됩니다.
- 8 *Mappings* 속성을 사용하여 옵션으로 열 매핑을 설정합니다. 배치 이동이 소스 테이블과 대상 테이블의 해당 위치를 기준으로 열과 일치되게 하려면 이 속성을 설정할 필요가 없습니다. 열 매핑에 대한 자세한 내용은 20-51 페이지의 "데이터 타입 매핑"을 참조하십시오.
- 9 옵션으로 *ChangedTableName*, *KeyViolTableName*, *ProblemTableName* 속성을 지정합니다. 배치 이동은 *ProblemTableName*에 의해 지정되는 테이블에서 배치 작업 도중 발생하는 문제 레코드를 저장합니다. 배치 이동을 통해 Paradox 테이블을 업데이트하는 경우, *KeyViolTableName*에 지정된 테이블에 키 위반을 보고할 수 있습니다. *ChangedTableName*은 배치 이동 작업의 결과로 대상 테이블에서 변경된 모든 레코드를 나열합니다. 이 속성들을 지정하지 않는 경우 이러한 오류 테이블이 생성되거나 사용되지 않습니다. 배치 이동 오류 처리에 대한 자세한 내용은 20-53 페이지의 "배치 이동 오류 처리"를 참조하십시오.

배치 이동 모드 지정

Mode 속성은 배치 이동 컴포넌트가 수행하는 작업을 지정합니다.

표 20.8 배치 이동 모드

속성	용도
<i>batAppend</i>	대상 테이블에 레코드를 추가합니다.
<i>batUpdate</i>	소스 테이블과 일치하는 레코드를 갖는 대상 테이블의 레코드를 업데이트합니다. 대상 테이블의 현재 인덱스를 기준으로 업데이트됩니다.
<i>batAppendUpdate</i>	일치하는 레코드가 대상 테이블에 있는 경우 해당 레코드를 업데이트합니다. 그렇지 않을 경우 대상 테이블에 레코드를 추가합니다.
<i>batCopy</i>	소스 테이블의 구조를 기준으로 대상 테이블을 생성합니다. 대상 테이블이 이미 존재하는 경우 해당 테이블이 삭제된 후 재생성됩니다.
<i>batDelete</i>	소스 테이블의 레코드와 일치하는 대상 테이블의 레코드를 삭제합니다.

레코드 추가

데이터를 추가하려면 대상 데이터셋은 기존 테이블을 나타내야 합니다. 추가 작업 중 BDE는 필요한 경우 대상 데이터셋에 적절한 데이터 타입 및 크기로 데이터를 변환합니다. 변환할 수 없는 경우 예외가 발생하고 데이터는 추가되지 않습니다.

레코드 업데이트

데이터를 업데이트하려면 대상 데이터셋이 기존 테이블을 나타내야 하고 인덱스는 레코드가 일치되도록 정의되어 있어야 합니다. 기본 인덱스 필드가 레코드를 일치시키는 데 사용되는 경우 소스 데이터셋의 인덱스 필드 레코드와 일치하는 대상 데이터셋의 인덱스 필드 레코드가 소스 데이터와 겹쳐 쓰여집니다. 업데이트 작업 중 BDE는 필요한 경우 대상 데이터셋에 적합한 데이터 타입 및 크기로 데이터를 변환합니다.

레코드 추가 및 업데이트

데이터를 추가하고 업데이트하려면 대상 데이터셋이 기존 테이블을 나타내야 하고 인덱스는 레코드가 일치되도록 정의되어 있어야 합니다. 기본 인덱스 필드가 레코드를 일치시키는 데 사용되는 경우 소스 데이터셋의 인덱스 필드 레코드와 일치하는 대상 데이터셋의 인덱스 필드 레코드가 소스 데이터와 겹쳐 쓰여집니다. 그렇지 않을 경우 소스 데이터셋의 데이터가 대상 데이터셋에 추가됩니다. 추가 및 업데이트 작업 중 BDE는 필요한 경우 대상 데이터셋에 적절한 데이터 타입 및 크기로 데이터를 변환합니다.

데이터셋 복사

소스 데이터셋을 복사하려면 대상 데이터셋이 기존 테이블을 나타내지 않아야 합니다. 대상 데이터셋이 기존 테이블을 나타내는 경우 배치 이동 작업에 의해 기존 테이블과 소스 데이터셋의 복사본이 겹쳐 쓰여집니다.

Paradox와 InterBase 등 다른 타입의 데이터베이스 엔진에서 소스 데이터셋과 대상 데이터셋을 관리하는 경우, BDE는 소스 데이터셋의 구조에 가장 근접한 구조를 갖는 대상 데이터셋을 생성하고 필요한 경우 데이터 타입 및 크기 변환을 자동으로 수행합니다.

참고 *TBatchMove*는 인덱스, 제약 조건 및 내장 프로시저 등의 메타데이터 구조를 복사하지 않습니다. 데이터베이스 서버에, 또는 SQL 탐색기를 통해 적절하게 이러한 메타데이터 객체를 재생성해야 합니다.

레코드 삭제

대상 데이터셋에서 데이터를 삭제하려면 기존 테이블을 나타내야 하고 인덱스는 레코드가 일치되도록 정의되어 있어야 합니다. 기본 인덱스 필드가 레코드를 일치시키는 데 사용되는 경우 소스 데이터셋의 인덱스 필드 레코드와 일치하는 대상 데이터셋의 인덱스 필드 레코드가 대상 테이블에서 삭제됩니다.

데이터 타입 매핑

batAppend 모드에서 배치 이동 컴포넌트는 소스 테이블의 열 데이터 타입을 기반으로 대상 테이블을 생성합니다. 열과 타입은 소스 테이블과 대상 테이블의 해당 위치에 기반하여 일치됩니다. 즉, 소스 테이블의 첫 번째 열은 대상 테이블의 첫 번째 열과 일치됩니다.

기본 열 매핑을 오버라이드하려면 *Mappings* 속성을 사용합니다. *Mappings*는 출당 하나의 열을 매핑하는 열 매핑 목록입니다. 이 목록은 두 가지 형식 중 하나입니다. 소스 테이블의 열을 대상 테이블의 동일한 이름을 갖는 열에 매핑하려면 일치시킬 열 이름을 지정하는 간단한 목록을 사용합니다. 예를 들어, 다음 매핑은 소스 테이블의 *ColName*이라는 열이 대상 테이블의 동일한 이름을 갖는 열에 매핑되어야 한다는 것을 지정합니다.

```
ColName
```

소스 테이블의 *SourceColName*이라는 열을 대상 테이블의 *DestColName*이라는 열에 매핑하려면 구문이 다음과 같아야 합니다.

```
DestColName = SourceColName
```

소스 열 데이터 타입과 대상 열 데이터 타입이 같지 않으면 배치 이동 작업은 "가장 적합한 값 (best fit)" 찾기를 시도합니다. 필요 시 문자 데이터 타입을 자르고 가능한 경우 제한된 회수의 변환을 수행하려고 시도합니다. 예를 들어, CHAR(10) 열을 CHAR(5) 열로 매핑하면 소스 열의 끝에 있는 5개 문자가 잘립니다.

변환의 예로, 문자 데이터 타입의 소스 열이 정수 타입의 대상으로 매핑되면 배치 이동 작업에 의해 '5'의 문자 값이 해당 정수 값으로 변환됩니다. 변환될 수 없는 값은 오류를 발생시킵니다. 오류에 대한 자세한 내용은 20-53 페이지의 "배치 이동 오류 처리"를 참조하십시오.

다른 테이블 타입 간에 데이터를 이동할 때 배치 이동 컴포넌트는 데이터셋의 서버 타입에 적합한 데이터 타입을 번역합니다. 서버 타입 간의 매핑에 대한 최신 테이블은 BDE 온라인 도움말 파일을 참조하십시오.

참고 SQL 서버 데이터베이스에 데이터를 배치 이동하려면 해당 데이터베이스 서버와 적절한 SQL 링크가 설치되어 있는 Delphi 버전이 있어야 하며, 타사의 해당 ODBC 드라이버가 설치되어 있는 경우에는 ODBC를 사용할 수 있습니다.

배치 이동 실행

Execute 메소드를 사용하여 런타임 시 이전에 준비한 배치 작업을 실행합니다. 예를 들어, *BatchMoveAdd*가 배치 이동 컴포넌트의 이름인 경우 다음 문은 해당 작업을 실행합니다.

```
BatchMoveAdd.Execute;
```

배치 이동 컴포넌트에서 마우스 오른쪽 버튼을 클릭한 다음 컨텍스트 메뉴에서 *Execute*를 선택하여 디자인 타임 시 배치 이동을 실행할 수도 있습니다.

MovedCount 속성은 배치 이동 실행 시 이동하는 레코드 수를 추적합니다.

RecordCount 속성은 이동할 최대 레코드 수를 지정합니다. *RecordCount*가 0이면 소스 데이터셋의 첫 번째 레코드부터 시작해서 모든 레코드가 이동됩니다. *RecordCount*가 양수이면 소스 데이터셋의 현재 레코드부터 시작해서 최대한의 *RecordCount* 레코드가 이동됩니다. *RecordCount*가 소스 데이터셋의 현재 레코드와 그 마지막 레코드 사이에 있는 레코드 수보다 크면 배치 이동은 소스 데이터셋의 끝에 이르렀을 때 종료됩니다. *MoveCount*를 검사하여 실제로 전송할 레코드 수를 결정할 수 있습니다.

배치 이동 오류 처리

배치 이동 작업에서 발생할 수 있는 두 가지 오류 타입은 데이터 타입 변환 오류와 무결성 위반입니다. *TBatchMove*는 오류 처리를 보고 및 제어하는 다양한 속성을 갖고 있습니다.

AbortOnProblem 속성은 데이터 타입 변환 오류가 발생할 때 작업 중지 여부를 지정합니다. *AbortOnProblem*이 *True*이면 오류 발생 시 배치 이동 작업이 취소되고, *False*이면 작업이 계속됩니다. *ProblemTableName*에서 지정하는 테이블을 검사하여 문제를 발생시킨 레코드를 확인할 수 있습니다.

AbortOnKeyViol 속성은 Paradox 키 위반 발생 시 작업을 중지할지 여부를 나타냅니다.

ProblemCount 속성은 데이터를 잃지 않으면 대상 테이블에서 처리할 수 없는 레코드 수를 나타냅니다. *AbortOnProblem*이 *True*인 경우 오류 발생 시 작업이 중지되므로 이 수는 1입니다.

다음 속성들을 사용하면 배치 이동 컴포넌트에서 배치 이동 작업을 문서화하는 추가 테이블을 생성할 수 있습니다.

- *ChangedTableName*을 지정하면 업데이트 또는 삭제 작업의 결과로 변경된 대상 테이블의 모든 레코드가 들어 있는 로컬 Paradox 테이블이 생성됩니다.
- *KeyViolTableName*을 지정하면 Paradox 테이블 사용 시 키 위반을 발생시킨 소스 테이블의 모든 레코드가 들어 있는 로컬 Paradox 테이블이 생성됩니다. *AbortOnKeyViol*이 *True*인 경우 문제 첫 발생 시 작업이 중지되므로 이 테이블에는 항목이 하나만 있습니다.
- *ProblemTableName*이 지정된 경우 데이터 타입 변환 오류에 기인하여 대상 테이블에 게시할 수 없는 모든 레코드가 들어 있는 로컬 Paradox 테이블을 생성합니다. 예를 들어, 테이블은 대상 테이블에 맞게 잘라야 하는 데이터를 갖는 소스 테이블의 레코드를 포함할 수 있습니다. *AbortOnProblem*이 *True*인 경우 문제 첫 발생 시 작업이 중지되므로 이 테이블에는 항목이 하나만 있습니다.

참고 *ProblemTableName*이 지정되지 않은 경우 레코드의 데이터는 잘라져서 대상 테이블에 놓입니다.

데이터 사전

BDE를 사용하여 데이터에 액세스할 때 애플리케이션에서 데이터 사전에 액세스할 수 있습니다. 데이터 사전은 애플리케이션과 무관하게 사용자 지정 저장 영역을 제공하며 이 영역에서 데이터의 내용과 외관을 나타내는 확장 필드 속성 집합을 생성할 수 있습니다.

예를 들어, 재무 관련 애플리케이션을 자주 개발하는 경우에는 통화를 여러 가지 다른 표시 형식으로 나타내는 특수 필드 속성 집합을 생성할 수 있습니다. 디자인 타임 시 애플리케이션에 대한 데이터셋을 생성할 때 Object Inspector를 사용하여 각 데이터셋의 통화 필드를 수동으로 설정하는 대신 이러한 통화 필드와 데이터 사전의 확장 필드 속성

집합을 연결할 수 있습니다. 데이터 사전을 사용하면 생성하는 애플리케이션에서 일관성 있는 데이터 모양을 얻을 수 있습니다.

클라이언트/서버 환경에서 데이터 사전은 추가 정보 공유를 위한 원격 서버에 상주할 수 있습니다.

디자인 타임 시 Fields 편집기의 확장 필드 속성 집합을 생성하는 방법에 대해 알아보려면 19-13 페이지의 "필드 컴포넌트의 속성 집합 만들기"를 참조하십시오. 데이터 사전 생성과 SQL 및 Database Explorers의 확장 필드 속성에 대한 자세한 내용은 해당 관련 온라인 도움말 파일을 참조하십시오.

데이터 사전에 대한 프로그래밍 인터페이스는 lib 디렉토리의 `drntf unit`에 있습니다. 이 인터페이스는 다음과 같은 메소드를 제공합니다.

표 20.9 데이터 사전 인터페이스

루틴	용도
DictionaryActive	데이터 사전의 활성화 여부를 나타냅니다.
DictionaryDeactivate	데이터 사전을 활성화 해제합니다.
IsNullID	지정된 ID가 Null ID인지 여부를 나타냅니다.
FindDatabaseID	별칭이 주어진 데이터베이스에 대한 ID를 반환합니다.
FindTableID	지정된 데이터베이스의 테이블에 대한 ID를 반환합니다.
FindFieldID	지정된 테이블의 필드에 대한 ID를 반환합니다.
FindAttrID	명명된 속성 집합에 대한 ID를 반환합니다.
GetAttrName	ID가 지정된 속성 집합에 대한 이름을 반환합니다.
GetAttrNames	사전의 각 속성 집합에 대한 콜백을 실행합니다.
GetAttrID	지정된 필드의 속성 집합에 대한 ID를 반환합니다.
NewAttr	필드 컴포넌트의 새 속성 집합을 생성합니다.
UpdateAttr	필드의 속성과 일치하도록 속성 집합을 업데이트합니다.
CreateField	내장 속성을 기반으로 필드 컴포넌트를 생성합니다.
UpdateField	지정된 속성 집합과 일치하도록 필드 속성을 변경합니다.
AssociateAttr	속성 집합을 지정된 필드 ID에 연결합니다.
UnassociateAttr	필드 ID의 속성 집합 연결을 제거합니다.
GetControlClass	지정된 속성 ID에 대한 제어 클래스를 반환합니다.
QualifyTableName	사용자 이름별로 한정된 전체 테이블 이름을 반환합니다.
QualifyTableNameByName	사용자 이름별로 한정된 전체 테이블 이름을 반환합니다.
HasConstraints	데이터셋이 사전의 제약 조건을 갖는지 여부를 나타냅니다.
UpdateConstraints	데이터셋의 가져온 제약 조건을 업데이트합니다.
UpdateDataset	사전의 현재 설정과 제약 조건에 맞게 데이터셋을 업데이트합니다.

BDE 사용을 위한 툴

BDE를 데이터 액세스 메커니즘으로 사용하는 이점 중 하나는 Delphi와 함께 제공되는 지원 유틸리티가 풍부하다는 것입니다. 이러한 유틸리티는 다음과 같습니다.

- **SQL 탐색기와 Database Explorer:** Delphi의 구입 버전에 따라 두 애플리케이션 중 하나가 제공됩니다. 두 탐색기는 다음과 같은 작업을 수행합니다.
 - 기존 데이터베이스 테이블과 구조를 검사합니다. SQL 탐색기를 통해 원격 SQL 데이터베이스를 검사하고 쿼리할 수 있습니다.
 - 데이터로 테이블을 구성합니다.
 - 데이터 사전에서 확장 필드 속성 집합을 생성하거나 이러한 확장 필드 속성 집합을 애플리케이션의 필드에 연결합니다.
 - BDE 별칭을 생성하고 관리합니다.

SQL 탐색기를 통해 다음과 같은 작업도 수행할 수 있습니다.

- 원격 데이터베이스 서버의 내장 프로시저와 같은 SQL 객체를 생성합니다.
- 원격 데이터베이스 서버에서 SQL 객체의 재구성된 텍스트를 봅니다.
- SQL 스크립트를 실행합니다.
- **SQL Monitor:** SQL Monitor를 통해 원격 데이터베이스 서버와 BDE 간에 전달되는 모든 통신을 볼 수 있습니다. 보려는 메시지를 관심 있는 부분에만 제한하여 필터링할 수 있습니다. SQL Monitor는 애플리케이션 디버깅 시 특히 유용합니다.
- **BDE Administration 유틸리티:** BDE Administration 유틸리티를 통해 새로운 데이터베이스 드라이버를 추가하고, 기존 드라이버에 대해 기본값을 구성하며, BDE 별칭을 새로 만들 수 있습니다.
- **데이터베이스 데스크탑:** Paradox 또는 dBASE 테이블을 사용하는 경우 데이터베이스 데스크탑을 통해 데이터를 보거나 편집하고, 테이블을 새로 생성하고, 기존 테이블을 재구성할 수 있습니다. 데이터베이스 데스크탑을 사용하면 *TTable* 컴포넌트를 사용하는 것보다 더 잘 제어할 수 있습니다(예를 들면, 유효성 검사와 랭귀지 드라이버를 지정할 수 있습니다). 또한 BDE의 API를 직접 호출하는 것 외에도 Paradox 및 dBASE 테이블을 재구성하는 유일한 메커니즘을 제공합니다.

21

ADO 컴포넌트 사용

ADOExpress 컴포넌트는 ADO 프레임워크를 통한 데이터 액세스를 제공합니다. ADO(Microsoft ActiveX Data Objects)는 OLE DB 프로바이더를 통해 데이터에 액세스하는 일련의 COM 객체입니다. Delphi *ADOExpress* 컴포넌트는 ADO 객체들을 Delphi 데이터베이스 아키텍처에 캡슐화합니다.

ADO 기반 애플리케이션의 ADO 층은 Microsoft ADO 2.1, 데이터 저장소 액세스를 위한 OLE DB 프로바이더 또는 ODBC 드라이버, SQL 데이터베이스의 경우 사용되는 특정 데이터베이스 시스템에 대한 클라이언트 소프트웨어, SQL 데이터베이스 시스템용 애플리케이션에 액세스할 수 있는 데이터베이스 백엔드 시스템, 데이터베이스로 구성되었습니다. 이 모두를 완전히 사용하기 위해서는 ADO 기반 애플리케이션에 액세스할 수 있어야 합니다.

가장 두드러지는 ADO 객체는 Connection, Command, Recordset 객체입니다. 이 ADO 객체들은 *TADOConnection*, *TADOCommand* 및 ADO 데이터셋 컴포넌트로 둘러 싸여 있습니다. ADO 프레임워크에는 Field 및 Properties 객체와 비슷한 다른 "helper" 객체가 있지만 일반적으로 Delphi 애플리케이션에서 직접 사용되지 않으며 전용 컴포넌트로 둘러 싸여 있지 않습니다.

이 장에서는 *ADOExpress* 컴포넌트들에 대해서 알아보며 공통 Delphi 데이터베이스 아키텍처에 추가하는 고유한 기능들을 설명합니다. *ADOExpress* 컴포넌트 특유의 기능에 대해 읽기 전에 17장 "데이터베이스에 연결"과 18장 "데이터셋 이해"에 설명된 데이터베이스 연결 컴포넌트와 데이터셋의 공통 기능을 잘 알아야 합니다.

ADO 컴포넌트 개요

컴포넌트 팔레트의 ADO 페이지에는 *ADOExpress* 컴포넌트가 있습니다. 이 컴포넌트를 사용할 경우 ADO 프레임워크를 사용하여 ADO 데이터 저장소에 연결하고 명령을 실행하고 데이터베이스의 테이블에서 데이터를 검색할 수 있습니다. 그러기 위해서는 ADO 2.1(또는 이상)이 호스트 컴퓨터에 설치되어야 합니다. 또 특정 데이터베이스 시스템에 맞는 OLE DB 드라이버나 ODBC 드라이버뿐만 아니라 Microsoft SQL server와

같은 대상 데이터베이스 시스템에 대한 클라이언트 소프트웨어가 추가적으로 설치되어야 합니다.

대부분의 *ADOExpress* 컴포넌트에는 데이터베이스 연결 컴포넌트(*TADOConnection*)와 다양한 타입의 데이터셋처럼 다른 데이터 액세스 방식에 사용할 수 있는 컴포넌트에 직접적으로 대응되는 것이 있습니다. 또한 *ADOExpress*에는 데이터셋은 아니지만 ADO 데이터 저장소에서 실행될 SQL 명령을 나타내는 간단한 컴포넌트인 *TADOCommand*가 있습니다.

다음 표는 ADO 컴포넌트를 나타낸 것입니다.

표 21.1 ADO 컴포넌트

컴포넌트	용도
<i>TADOConnection</i>	ADO 데이터 저장소와 연결되는 데이터베이스 연결 컴포넌트입니다. 여러 ADO 데이터셋과 명령 컴포넌트는 이 연결을 공유하여 명령을 실행하고 데이터를 검색하고 메타데이터로 작업합니다.
<i>TADODataSet</i>	데이터를 검색하고 작업하기 위한 기본 데이터셋입니다. <i>TADODataSet</i> 은 단일 또는 다중 테이블에서 데이터를 검색할 수 있으며 데이터 저장소에 직접 연결하거나 <i>TADOConnection</i> 컴포넌트를 사용할 수 있습니다.
<i>TADOTable</i>	단일 데이터베이스 테이블로 만든 레코드셋을 검색하고 작업하기 위한 테이블 타입 데이터셋입니다. <i>TADOTable</i> 은 데이터 저장소에 직접 연결하거나 <i>TADOConnection</i> 컴포넌트를 사용할 수 있습니다.
<i>TADOQuery</i>	유효한 SQL 문에서 만든 레코드셋을 검색하고 작업하기 위한 쿼리 타입 데이터셋입니다. <i>TADOQuery</i> 는 DDL(Data Definition Language) SQL 문도 실행할 수 있습니다. 데이터 저장소에 직접 연결하거나 <i>TADOConnection</i> 컴포넌트를 사용할 수 있습니다.
<i>TADOStoredProc</i>	내장 프로시저를 실행하기 위한 내장 프로시저 타입 데이터셋입니다. <i>TADOStoredProc</i> 는 데이터를 검색 가능하거나 불가능한 내장 프로시저를 실행합니다. 데이터 저장소에 직접 연결하거나 <i>TADOConnection</i> 컴포넌트를 사용할 수 있습니다.
<i>TADOCommand</i>	명령, 즉 결과 집합을 반환하지 않는 SQL 문을 실행하기 위한 간단한 컴포넌트입니다. <i>TADOCommand</i> 는 지원 데이터셋 컴포넌트와 함께 사용될 수 있거나 테이블에서 데이터셋을 검색할 수 있습니다. 데이터 저장소에 직접 연결하거나 <i>TADOConnection</i> 컴포넌트를 사용할 수 있습니다.

ADO 데이터 저장소에 연결

Delphi ADO 기반 애플리케이션은 ADO(Microsoft ActiveX Data Objects) 2.1을 사용하여 데이터 저장소에 연결하고 그 데이터에 액세스하는 OLE DB 프로바이더와 상호 작용합니다. 데이터 저장소가 나타낼 수 있는 항목 중 하나는 데이터베이스입니다. ADO 기반 애플리케이션을 사용하려면 ADO 2.1이 클라이언트 컴퓨터에 설치되어 있어야 합니다. ADO와 OLE DB는 Microsoft가 제공하며 Windows와 함께 설치됩니다.

ADO 프로바이더는 원시 OLE DB 드라이버에서 ODBC 드라이버까지 다양한 타입의 액세스 중 하나를 나타냅니다. 이 드라이버는 클라이언트 컴퓨터에 설치되어야 합니다. 다양한 데이터베이스 시스템에 맞는 OLE DB 드라이버는 데이터베이스 공급업체나 협력 업체에서 제공합니다. 애플리케이션이 Microsoft SQL Server 또는 Oracle과 같은 SQL

데이터베이스를 사용할 경우, 해당 데이터베이스 시스템에 맞는 클라이언트 소프트웨어도 클라이언트 컴퓨터에 설치되어 있어야 합니다. 클라이언트 소프트웨어는 데이터베이스 공급업체에서 제공하며 데이터베이스 시스템 CD 또는 디스크로 설치할 수 있습니다.

애플리케이션을 데이터 저장소와 연결하려면 ADO 연결 컴포넌트(*TADOConnection*)를 사용하십시오. 사용할 수 있는 ADO 프로바이더 중 하나를 사용하도록 ADO 연결 컴포넌트를 구성합니다. ADO 명령과 데이터셋 컴포넌트는 해당 *ConnectionString* 속성을 사용하여 연결할 수 있으므로 *TADOConnection*이 반드시 필요한 것은 아니지만 *TADOConnection*을 사용하면 여러 ADO 컴포넌트 간에 단일 연결을 공유할 수 있습니다. 이렇게 하면 리소스를 더 적게 소모하여 여러 데이터셋에 걸친 트랜잭션을 만들 수 있습니다.

다른 데이터베이스 연결 컴포넌트와 마찬가지로 *TADOConnection*은 다음을 지원합니다.

- 연결 제어
- 서버 로그인 제어
- 트랜잭션 관리
- 연결된 데이터셋 작업
- 서버에 명령 전송
- 메타데이터 얻기

*TADOConnection*은 모든 데이터베이스 연결 컴포넌트에 공통적으로 사용되는 이 기능들 외에도 자체적으로 다음을 지원합니다.

- 연결을 미세 조정(fine-tuning)하는 데 사용할 수 있는 광범위한 옵션
- 연결을 사용하는 명령 객체 나열
- 공통 작업 수행 시의 추가 이벤트

TADOConnection을 사용하여 데이터 저장소에 연결

하나 이상의 ADO 데이터셋과 명령 컴포넌트는 *TADOConnection*을 사용하여 데이터 저장소에 대한 단일 연결을 공유할 수 있습니다. 그렇게 하려면 *Connection* 속성을 통해 데이터셋과 명령 컴포넌트를 연결 컴포넌트와 연결하십시오. 디자인 타임 시 Object Inspector에서 *Connection* 속성의 드롭다운 목록에서 원하는 연결 컴포넌트를 선택하십시오. 런타임 시 *Connection* 속성에 참조를 할당하십시오. 예를 들어, 다음 명령줄은 *TADODataSet* 컴포넌트와 *TADOConnection* 컴포넌트를 연결합니다.

```
ADODataset1.Connection := ADOConnection1;
```

연결 컴포넌트는 ADO 연결 객체를 나타냅니다. 연결 객체를 사용하여 연결하기 전에 연결할 데이터 저장소를 확인해야 합니다. 일반적으로 *ConnectionString* 속성을 사용하여 정보를 제공합니다. *ConnectionString*은 하나 이상의 명명된 연결 매개변수를 나열하는 세미콜론으로 구분된 문자열입니다. 이 매개변수들은 연결 정보가 포함된 파일 이름이나 ADO 프로바이더의 이름과 데이터 저장소를 나타내는 참조를 지정하여 데이

터 저장소를 나타냅니다. 이러한 정보를 제공하려면 이미 정의되어 있는 다음 매개변수 이름들을 사용하십시오.

매개변수	설명
<i>Provider</i>	연결에 사용할 로컬 ADO 프로바이더의 이름
<i>Data Source</i>	데이터 저장소의 이름
<i>File name</i>	연결 정보가 들어 있는 파일의 이름
<i>Remote Provider</i>	원격 시스템에 있는 ADO 프로바이더의 이름
<i>Remote Server</i>	원격 프로바이더를 사용할 때 원격 서버의 이름

따라서 일반적인 *ConnectionString*의 값은 다음과 같은 형태를 갖습니다.

```
Provider=MSDASQL.1;Data Source=MQIS
```

참고 *ConnectionString*에서 연결 매개변수는 *Provider* 속성을 사용하여 ADO 프로바이더를 지정할 경우, *Provider* 또는 *Remote Provider* 매개변수를 포함할 필요가 없습니다. 마찬가지로 *DefaultDatabase* 속성을 사용할 경우 *Data Source* 매개변수를 지정할 필요가 없습니다.

위에 나오는 매개변수 외에 *ConnectionString*은 사용하고 있는 특정 ADO 프로바이더에만 해당되는 연결 매개변수를 포함할 수 있습니다. 이 추가 연결 매개변수들은 로그인 정보를 코딩할 경우 사용자 ID와 암호를 포함할 수 있습니다.

디자인 타임 시 Connection String Editor를 사용하면 목록에서 프로바이더나 서버와 같은 연결 요소를 선택하여 연결 문자열을 만들 수 있습니다. Object Inspector에서 *ConnectionString* 속성의 생략 기호 버튼을 클릭하면 ADO의 ActiveX 속성 편집기인 Connection String Editor가 실행됩니다.

ConnectionString 속성(그리고 경우에 따라 *Provider* 속성)을 지정했으면 먼저 다른 속성들을 사용하여 연결을 미세 조정(fine-tuning)할 수 있도록 ADO 연결 컴포넌트를 사용하여 ADO 데이터 저장소에 연결하거나 연결을 끊을 수 있습니다. 데이터 저장소에 연결하거나 연결을 끊을 때 *TADOConnection*을 사용하면 모든 데이터베이스 연결 컴포넌트에 공통적인 이벤트 외에도 몇 가지 추가 이벤트에 응답할 수 있습니다. 이 추가 이벤트는 21-8 페이지의 "연결 시의 이벤트"와 21-8 페이지의 "연결 중지 시의 이벤트"에 설명되어 있습니다.

참고 연결 컴포넌트의 *Connected* 속성을 *True*로 설정하여 연결을 명시적으로 활성화하지 않으면 첫째 데이터셋 컴포넌트가 열려 있거나 처음 ADO 명령 컴포넌트를 사용하여 명령을 실행할 때 자동으로 연결됩니다.

Connection 객체 액세스

*TADOConnection*의 *ConnectionObject* 속성을 사용하여 *TADOConnection*의 기본 ADO 연결 객체에 액세스합니다. 이 참조를 사용하면 기본 ADO Connection 객체의 속성과 호출 메소드에 액세스할 수 있습니다.

기본 ADO Connection 객체를 사용할 경우, 일반적인 ADO 객체 사용 방법을 알아야 하며 특히 ADO Connection 객체에 대해 잘 알아야 합니다. Connection 객체 연산에 대해

서 잘 모를 경우에는 Connection 객체를 사용하지 않는 것이 좋습니다. ADO Connection 객체 사용에 대한 자세한 내용은 Microsoft Data Access SDK 도움말을 참조하십시오.

연결 미세 조정(fine-tuning)

ADO 명령과 데이터셋 컴포넌트에 대한 연결 문자열을 제공하는 대신 데이터 저장소에 연결하기 위해 *TADOConnection*을 사용하면 연결의 상태와 속성을 더 잘 제어할 수 있다는 이점이 있습니다.

강제적인 비동기(asynchronous) 연결

ConnectOptions 속성을 사용하여 연결을 비동기화할 수 있습니다. 비동기 연결을 통해 연결이 완전히 열릴 때까지 기다리지 않고도 애플리케이션은 계속 작업할 수 있습니다.

기본적으로 *ConnectionOptions*는 서버가 최고의 연결 유형을 결정할 수 있는 *coConnectUnspecified*로 설정되어 있습니다. 연결을 명시적으로 비동기화하려면 *ConnectOptions*를 *coAsyncConnect*로 설정하십시오.

아래의 루틴은 지정된 연결 컴포넌트에서 비동기 연결을 활성화한 예제와 활성화하지 않은 예제입니다.

```

procedure TForm1.AsyncConnectButtonClick(Sender:TObject);
begin
    with ADOConnection1 do begin
        Close;
        ConnectOptions := coAsyncConnect;
        Open;
    end;
end;

procedure TForm1.ServerChoiceConnectButtonClick(Sender:TObject);
begin
    with ADOConnection1 do begin
        Close;
        ConnectOptions := coConnectUnspecified;
        Open;
    end;
end;

```

시간 초과 제어

명령과 연결이 실제로 생각되어 *ConnectionTimeout*과 *CommandTimeout* 속성을 사용하여 중단되기 전에 경과하는 시간을 제어할 수 있습니다.

*ConnectionTimeout*은 데이터 저장소에 대한 연결 시도가 시간을 초과하기 전에 시간을 초로 지정합니다. 연결이 *ConnectionTimeout*에 지정된 시간이 끝나기 전에 성공적으로 컴파일되지 않으면 연결 시도가 취소됩니다.

```

with ADOConnection1 do begin
    ConnectionTimeout := 10 {seconds};
    Open;

```

`end;`

`CommandTimeout`은 시도된 명령이 시간을 초과하기 전에 시간을 초로 지정합니다. `Execute` 메소드에 대한 호출로 초기화된 명령이 `CommandTimeout`에 지정된 시간이 만료되기 전에 성공적으로 완료되지 않으면 명령은 취소되고 ADO는 다음과 같은 예외를 생성합니다.

```
with ADOConnection1 do begin
    CommandTimeout := 10 {seconds};
    Execute('DROP TABLE Employee1997', cmdText, []);
end;
```

연결이 지원하는 작업 유형 나타내기

ADO 연결은 파일을 열 때 사용한 모드와 비슷한 특정 모드를 이용하여 설정됩니다. 연결 모드에 따라 연결에 사용할 수 있는 권한, 즉 그 연결을 사용하여 수행될 수 있는 읽기 및 쓰기와 같은 작업 유형이 결정됩니다.

`Mode` 속성을 사용하여 연결 모드를 나타냅니다. 사용 가능한 값이 표 21.2에 나열되어 있습니다.

표 21.2 ADO 연결 모드

연결 모드	의미
<code>cmUnknown</code>	연결에 대한 권한이 아직 설정되지 않았거나 파악할 수 없습니다.
<code>cmRead</code>	연결에 대한 읽기 전용 권한이 있습니다.
<code>cmWrite</code>	연결에 대한 쓰기 전용 권한이 있습니다.
<code>cmReadWrite</code>	연결에 대한 읽기/쓰기 권한이 있습니다.
<code>cmShareDenyRead</code>	읽기 권한을 가진 다른 사용자가 연결을 열 수 없습니다.
<code>cmShareDenyWrite</code>	쓰기 권한을 가진 다른 사용자가 연결을 열 수 없습니다.
<code>cmShareExclusive</code>	다른 사용자가 연결을 열 수 없습니다.
<code>cmShareDenyNone</code>	어느 권한을 가진 다른 사용자도 연결을 열 수 없습니다.

`Mode`에 사용 가능한 값은 기본 ADO 연결 객체에서 `Mode` 속성의 `ConnectModeEnum`에 대응됩니다. 이 값에 대한 자세한 내용은 Microsoft Data Access SDK 도움말을 참조하십시오.

연결의 자동적인 트랜잭션 초기화 여부 지정

`Attributes` 속성을 사용하여 연결 컴포넌트의 커밋 유지와 중단 유지 사용을 제어합니다. 연결 컴포넌트가 커밋 유지를 사용할 경우, 애플리케이션이 트랜잭션을 커밋할 때마다 새로운 트랜잭션이 자동으로 시작됩니다. 연결 컴포넌트가 중단 유지를 사용할 경우, 애플리케이션이 트랜잭션을 롤백할 때마다 새로운 트랜잭션이 자동으로 시작됩니다.

`Attributes`는 상수 `xaCommitRetaining`과 `xaAbortRetaining` 중 한 개나 두 개를 포함하거나 또는 두 개 모두 포함하지 않는 집합입니다. `Attributes`에 `xaCommitRetaining`이 있으면 연결은 커밋 유지를 사용합니다. `Attributes`에 `xaAbortRetaining`이 있으면 중단 유지를 사용합니다.

in 연산자를 사용하여 커밋 유지 또는 중단 유지가 활성화되는지 확인합니다. *attributes* 속성에 해당 값을 추가하여 커밋 또는 중단 유지를 활성화합니다. 값을 빼면 활성화되지 않습니다. 아래의 루틴은 ADO 연결 컴포넌트에서 각각 커밋 유지를 활성화한 예제와 활성화하지 않은 예제입니다.

```

procedure TForm1.RetainingCommitsOnButtonClick(Sender:TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining]);
    Open;
  end;
end;

procedure TForm1.RetainingCommitsOffButtonClick(Sender:TObject);
begin
  with ADOConnection1 do begin
    Close;
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
  end;
end;

```

연결 명령 액세스

다른 데이터베이스 연결 컴포넌트와 마찬가지로 *DataSets*와 *DataSetCount* 속성을 사용하여 연결과 연관된 데이터셋에 액세스할 수 있습니다. 그러나 *ADOExpress*에는 데이터셋은 아니지만 연결 컴포넌트와 비슷한 관계를 유지하는 *TADOCCommand* 객체도 포함되어 있습니다.

*DataSets*와 *DataSetCount* 속성을 사용하여 연관된 데이터셋에 액세스할 때와 마찬가지로 *TADOConnection*의 *Commands*와 *CommandCount* 속성을 사용하여 연관된 ADO 명령 객체에 액세스할 수 있습니다. 활성 데이터셋만 나열하는 *DataSets*와 *DataSetCount* 및 *Commands*와 *CommandCount*는 연결 컴포넌트와 연관된 모든 *TADOCCommand* 컴포넌트에 대한 참조를 제공합니다.

*Commands*는 ADO 명령 컴포넌트에 대해 인덱스가 0부터 시작하는 참조 배열입니다. *CommandCount*는 *Commands*에 나열된 모든 명령의 총 횟수를 제공합니다. 이 속성들을 함께 사용하면 다음 코드에 설명된 대로 연결 컴포넌트를 사용하는 모든 명령에서 반복할 수 있습니다.

```

var
  i:Integer
begin
  for i := 0 to (ADOConnection1.CommandCount - 1) do
    ADOConnection1.Commands[i].Execute;
end;

```

ADO 연결 이벤트

모든 데이터베이스 연결 컴포넌트에 대해 발생하는 일상적인 이벤트 외에 *TADOConnection* 은 보통 사용하는 동안 발생하는 많은 추가 이벤트를 생성합니다.

연결 시의 이벤트

모든 데이터베이스 연결 컴포넌트에 공통적으로 사용되는 *BeforeConnect*와 *AfterConnect* 이벤트 외에 *TADOConnection*은 연결 시 *OnWillConnect*와 *OnConnectComplete* 이벤트도 생성합니다. 이 이벤트들은 *BeforeConnect* 이벤트 다음에 발생합니다.

- *OnWillConnect*는 ADO 프로바이더가 연결하기 전에 발생합니다. 이를 통해 연결 문자열을 최종적으로 변경하고, 로그인 지원을 처리할 경우 사용자 이름과 암호를 입력하고, 비동기 연결을 강제로 적용하고, 연결을 열기 전에 취소할 수도 있습니다.
- *OnConnectComplete*는 연결이 열린 후에 발생합니다. *TADOConnection*이 비동기 연결을 나타낼 수 있으므로 연결이 열리기 전에 연결 컴포넌트가 ADO 프로바이더에게 연결을 열도록 지시한 다음에 발생하는 *AfterConnect* 이벤트 대신 연결이 열리거나 오류 조건 때문에 연결에 실패한 후 발생하는 *OnConnectComplete*를 사용해야 합니다.

연결 중지 시의 이벤트

모든 데이터베이스 연결 컴포넌트에 공통적으로 사용되는 *BeforeDisconnect*와 *AfterDisconnect* 이벤트 외에 *TADOConnection*은 연결을 닫은 다음 *OnDisconnect* 이벤트도 생성합니다. *OnDisconnect*는 연결이 닫힌 후에 발생하지만 연관 데이터셋이 닫히기 전과 *AfterDisconnect* 이벤트 전에 발생합니다.

트랜잭션 관리 시의 이벤트

ADO 연결 컴포넌트는 트랜잭션 관련 프로세스가 완료되었을 때 감지하도록 많은 이벤트를 제공합니다. 이 이벤트들은 *BeginTrans*, *CommitTrans*, *RollbackTrans* 메소드에 의해 초기화된 트랜잭션이 데이터 저장소에서 성공적으로 완료되었음을 나타냅니다.

- *OnBeginTransComplete* 이벤트는 *BeginTrans* 메소드에 대한 호출 다음에 데이터 저장소가 트랜잭션을 성공적으로 시작했을 때 발생합니다.
- *OnCommitTransComplete* 이벤트는 *CommitTrans*에 대한 호출로 트랜잭션이 성공적으로 커밋된 다음에 발생합니다.
- *OnRollbackTransComplete* 이벤트는 *RollbackTrans*에 대한 호출로 트랜잭션이 성공적으로 중단된 다음에 발생합니다.

기타 이벤트

ADO 연결 컴포넌트에는 기본 ADO 연결 객체로부터 공지에 응답하는 데 사용할 수 있는 다음과 같은 두 가지 추가 이벤트도 있습니다.

- *OnExecuteComplete* 이벤트는 연결 컴포넌트가 데이터 저장소에서 명령을 실행한 후(예: *Execute* 메소드 호출 후)에 발생합니다. *OnExecuteComplete*는 실행의 성공 여부를 나타냅니다.
- *OnInfoMessage* 이벤트는 기본 연결 객체가 작업이 완료된 후에 자세한 정보를 제공할 때 발생합니다. *OnInfoMessage* 이벤트 핸들러는 자세한 정보와 작업의 성공 여부를 나타내는 상태 코드를 담은 ADO Error 객체에 대한 인터페이스를 수신합니다.

ADO 데이터셋 사용

ADO 데이터셋 컴포넌트는 ADO Recordset 객체를 캡슐화합니다. 그리고 18장 "데이터셋 이해"에 설명되어 있는 공통적인 데이터셋 기능을 상속 받고 ADO를 사용하여 구현합니다. ADO 데이터셋을 사용하기 위해서는 그 공통적인 기능들을 잘 알아야 합니다.

공통 데이터셋 기능 외에도 모든 ADO 데이터셋은 다음 항목들에 대한 속성, 이벤트, 메소드를 추가합니다.

- ADO 데이터 저장소에 연결
- 기본 Recordset 객체 액세스
- 북마크에 기초한 레코드 필터링
- 비동기식으로 레코드 페치(fetch)
- 배치 업데이트(업데이트 캐시) 수행
- 디스크의 파일을 사용하여 데이터 저장

ADO 데이터셋에는 다음 네 가지가 있습니다.

- *TADOTable*. 단일 데이터베이스 테이블의 모든 행과 열을 나타내는 테이블 타입 데이터셋입니다. *TADOTable* 및 기타 테이블 타입 데이터셋 사용에 대한 자세한 내용은 18-25 페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오.
- *TADOQuery*. SQL 문을 캡슐화하고 애플리케이션이 결과 레코드셋에 액세스할 수 있도록 하는 쿼리 타입 데이터셋입니다. *TADOQuery* 및 기타 테이블 타입 데이터셋 사용에 대한 자세한 내용은 18-42 페이지의 "쿼리 타입 데이터셋 사용"을 참조하십시오.
- *TADOStoredProc*. 데이터베이스 서버에 정의된 내장 프로시저를 실행하는 내장 프로시저 타입 데이터셋입니다. *TADOStoredProc* 및 기타 내장 프로시저 타입 데이터셋 사용에 대한 자세한 내용은 18-50 페이지의 "내장 프로시저 타입 데이터셋 사용"을 참조하십시오.
- *TADODataSet*. 다른 세 가지 타입의 성능을 포함하는 범용 데이터셋입니다. *TADODataSet* 고유의 기능 설명은 21-16 페이지의 "TADODataSet 사용"을 참조하십시오.

참고 ADO를 사용하여 데이터베이스 정보에 액세스할 때에는 *TADOQuery*와 같은 데이터셋을 사용하여 커서를 반환하지 않는 SQL 명령을 나타낼 필요가 없습니다. 그 대신 데이터셋이 아닌 간단한 *TADOCommand* 컴포넌트를 사용할 수 있습니다. *TADOCommand*에 대한 자세한 내용은 21-17 페이지의 "Command 객체 사용"을 참조하십시오.

데이터 저장소에 ADO 데이터셋 연결

ADO 데이터셋은 함께 또는 개별적으로 ADO 데이터 저장소에 연결할 수 있습니다.

데이터셋을 함께 연결할 때 각 데이터셋의 *Connection* 속성을 *TADOConnection* 컴포넌트로 설정하십시오. 그런 다음 각 데이터셋은 ADO 연결 컴포넌트의 연결을 사용합니다.

```
ADODataset1.Connection := ADOConnection1;
ADODataset2.Connection := ADOConnection1;
...
```

데이터셋을 함께 연결하면 다음과 같은 이점이 있습니다.

- 데이터셋들이 연결 객체의 속성을 공유합니다.
- *TADOConnection* 하나에 대한 연결만 설정하면 됩니다.
- 데이터셋은 트랜잭션에 참여할 수 있습니다.

TADOConnection 사용에 대한 자세한 내용은 21-2 페이지의 "ADO 데이터 저장소에 연결"을 참조하십시오.

개별적으로 데이터셋들을 연결할 때 각 데이터셋의 *ConnectionString* 속성을 설정합니다. *ConnectionString*을 사용하는 각 데이터셋은 애플리케이션의 다른 데이터셋 연결과 관계 없이 데이터 저장소에 연결합니다.

ADO 데이터셋의 *ConnectionString* 속성은 *TADOConnection*의 *ConnectionString* 속성과 같은 방식으로 사용됩니다. 이것은 다음과 같은 일련의 세미콜론으로 구분된 연결 매개변수입니다.

```
ADODataset1.ConnectionString := 'Provider=YourProvider;Password=SecretWord;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;' +
  'Initial Catalog=Employee';
```

디자인 타임 시 Connection String Editor를 사용하여 연결 문자열을 바인딩할 수 있습니다. 연결 문자열에 대한 자세한 내용은 21-3 페이지의 "TADOConnection을 사용하여 데이터 저장소에 연결"을 참조하십시오.

레코드셋 사용

Recordset 속성은 데이터셋 컴포넌트에 기초하여 ADO 레코드셋 객체에 직접 액세스할 수 있게 해줍니다. 이 객체를 사용하면 애플리케이션에서 레코드셋 객체의 속성과 호출 메소드에 액세스할 수 있습니다. *Recordset*을 사용하여 직접 기본 ADO 레코드셋 객체에 액세스하려면 일반적인 ADO 객체 사용 방법을 알아야 하며 특히 ADO Connection 객체에 대해 잘 알아야 합니다. 레코드셋 객체 작업에 익숙하지 않으면 직접 레코드셋 객체를 사용하지 않는 것이 좋습니다. ADO 레코드셋 객체 사용에 대한 자세한 내용은 Microsoft Data Access SDK 도움말을 참조하십시오.

RecordsetState 속성은 기본 레코드셋 객체의 현재 상태를 나타냅니다. *RecordsetState* 는 ADO 레코드셋 객체의 *State* 속성에 대응됩니다. *RecordsetState*의 값은 *stOpen*, *stExecuting*, *stFetching* 중 하나입니다(*RecordsetState* 속성 타입인 *TObjectState*는 다른 값을 정의하지만 레코드셋과 관련된 것은 *stOpen*, *stExecuting*, *stFetching* 세 가지뿐입니다.). *stOpen*의 값은 레코드셋이 현재 유효 상태임을 나타냅니다. *stExecuting*의 값은 명령을 실행 중임을 나타냅니다. *stFetching*의 값은 연관 테이블에서 행을 폐치 중임을 나타냅니다.

*RecordsetState*의 값을 사용하여 데이터셋의 현재 상태에 따라 작업을 수행합니다. 예를 들면, 데이터를 업데이트하는 루틴은 *RecordsetState* 속성을 확인하여 데이터셋이 활성화되었는지, 데이터 연결 또는 폐치와 같은 다른 작업들을 처리하는 중이 아닌지 알아봅니다.

북마크에 기반한 레코드 필터링

ADO 데이터셋은 북마크를 사용하여 특정 레코드를 표시하고 특정 레코드로 돌아가는 일반적인 데이터셋 기능을 지원합니다. 다른 데이터셋과 마찬가지로 ADO 데이터셋에서는 필터를 사용하여 데이터셋에서 사용할 수 있는 레코드를 제한할 수 있습니다. ADO 데이터셋은 이러한 두 가지 일반적인 데이터셋 기능을 결합한 추가 기능, 즉 북마크가 나타내는 일련의 레코드에 대한 필터링 기능을 제공합니다.

다음과 같은 방법으로 일련의 북마크를 필터링합니다.

- 1 *Bookmark* 메소드를 사용하여 필터링된 데이터셋에 포함할 레코드를 표시합니다.
- 2 *FilterOnBookmarks* 메소드를 호출하여 데이터셋을 필터링한 후 북마크 표시된 레코드만 나타냅니다.

이 프로세스는 다음과 같습니다.

```
procedure TForm1.Button1Click(Sender:TObject);
var
  BM1, BM2: TBookmarkStr;
begin
  with ADODataset1 do begin
    BM1 := Bookmark;
    BMList.Add(Pointer(BM1));
    MoveBy(3);
    BM2 := Bookmark;
    BMList.Add(Pointer(BM2));
    FilterOnBookmarks([BM1, BM2]);
  end;
end;
```

위의 예제는 북마크를 목록 객체 *BMList*에도 추가합니다. 애플리케이션이 더 이상 필요로 하지 않을 때는 나중에 북마크를 없애도록 하기 위해 필요합니다.

북마크 사용에 대한 자세한 내용은 18-9 페이지의 "레코드 표시 및 반환"을 참조하십시오. 다른 타입의 필터에 대한 자세한 내용은 18-12 페이지의 "필터를 사용한 데이터 서브셋 표시 및 편집"을 참조하십시오.

비동기식으로 레코드 페치

다른 데이터셋과 달리 ADO 데이터셋은 데이터를 비동기식으로 페치할 수 있습니다. 따라서 애플리케이션은 데이터셋이 데이터 저장소의 데이터로 채우고 있는 동안 계속해서 다른 작업을 수행할 수 있습니다.

데이터를 페치할 경우 데이터셋이 비동기식으로 데이터를 페치하도록 제어하려면 *ExecuteOptions* 속성을 사용하십시오. *ExecuteOptions*는 *Open*을 호출하거나 *Active*를 *True*로 설정할 때 데이터셋이 레코드를 페치하는 방법을 제어합니다. 데이터셋이 어떤 레코드도 반환하지 않는 쿼리나 내장 프로시저를 나타낼 경우, *ExecuteOptions*는 *ExecSQL* 또는 *ExecProc*를 호출할 때 쿼리나 내장 프로시저가 실행되는 방법을 제어합니다.

*ExecuteOptions*는 다음 값을 하나도 포함하지 않거나 하나 이상 포함한 집합입니다.

표 21.3 ADO 데이터셋의 실행 옵션

실행 옵션	의미
eoAsyncExecute	명령 또는 데이터 페치 작업이 비동기식으로 실행됩니다.
eoAsyncFetch	데이터셋은 먼저 <i>CacheSize</i> 속성이 지정하는 레코드 수를 동시에 페치한 다음 비동기식으로 나머지 행을 페치합니다.
eoAsyncFetchNonBlocking	비동기화 데이터 또는 명령 실행은 현재 실행 스레드를 차단하지 않습니다.
eoExecuteNoRecords	데이터를 반환하지 않는 명령 또는 내장 프로시저입니다. 행이 검색될 경우 무시되며 반환되지 않습니다.

배치 업데이트 사용

업데이트를 캐시하는 한 가지 방법은 데이터셋 프로바이더를 사용하여 ADO 데이터셋을 클라이언트 데이터셋에 연결하는 것입니다. 이 방법은 23-16 페이지의 "클라이언트 데이터셋을 사용하여 업데이트 캐시"에 설명되어 있습니다.

그러나 ADO 데이터셋 컴포넌트는 배치 업데이트라고 하는 캐시된 업데이트를 지원합니다. 다음 표는 클라이언트 데이터셋을 사용하여 업데이트를 캐시하는 것과 배치 업데이트 기능을 사용하여 업데이트를 캐시하는 것의 대응 관계를 나타냅니다.

표 21.4 ADO 와 클라이언트 데이터셋 캐시된 업데이트 비교

ADO 데이터셋	TClientDataSet	설명
LockType	사용 안함: 클라이언트 데이터셋은 항상 업데이트를 캐시합니다.	데이터셋이 배치 업데이트 모드에서 열려 있는지 지정합니다.
CursorType	사용 안함: 클라이언트 데이터셋은 항상 데이터의 인메모리(in-memory) 스냅샷으로 작업합니다.	서버의 변경으로 인해 ADO 데이터셋이 분리된 방법을 지정합니다.
RecordStatus	UpdateStatus	현재 행에서 업데이트가 발생했다면 어떤 업데이트가 발생했는지 나타냅니다. <i>RecordStatus</i> 는 <i>UpdateStatus</i> 보다 많은 정보를 제공합니다.

표 21.4 ADO 와 클라이언트 데이터셋 캐시된 업데이트 비교 (계속)

ADO 데이터셋	TClientDataSet	설명
FilterGroup	StatusFilter	사용할 수 있는 레코드 타입을 지정합니다. <i>FilterGroup</i> 은 보다 광범위한 정보를 제공합니다.
UpdateBatch	ApplyUpdates	캐시된 업데이트를 다시 데이터베이스 서버에 적용합니다. <i>ApplyUpdates</i> 와 달리 <i>UpdateBatch</i> 를 사용하면 적용할 업데이트 타입을 제한할 수 있습니다.
CancelBatch	CancelUpdates	대기하고 있는 업데이트를 버리고 원래 값으로 되돌립니다. <i>CancelUpdates</i> 와 달리 <i>CancelBatch</i> 를 사용하면 취소할 업데이트 타입을 제한할 수 있습니다.

ADO 데이터셋 컴포넌트의 배치 업데이트 기능은 다음과 같은 경우에 사용됩니다.

- 배치 업데이트 모드에서 데이터셋 열기
- 개별적인 행의 업데이트 상태 검사
- 업데이트 상태에 기반한 여러 행 필터링
- 기준 테이블에 배치 업데이트 적용
- 배치 업데이트 취소

배치 업데이트 모드에서 데이터셋 열기

배치 업데이트 모드에서 ADO 데이터셋을 열려면 다음 기준을 충족해야 합니다.

- 1 컴포넌트의 *CursorType* 속성은 *ctKeySet*(기본 속성 값) 또는 *ctStatic*이어야 합니다.
- 2 *LockType* 속성은 *ltBatchOptimistic*이어야 합니다.
- 3 명령은 SELECT 쿼리여야 합니다.

데이터셋 컴포넌트를 활성화하기 전에 *CursorType*과 *LockType* 속성을 위에 설명된 대로 설정하십시오. SELECT 문을 컴포넌트의 *CommandText* 속성(*TADODataSet*의 경우) 또는 *SQL* 속성(*TADOQuery*의 경우)으로 할당합니다. *TADOStoredProc* 컴포넌트의 경우, *ProcedureName*을 결과 집합을 반환하는 내장 프로시저의 이름으로 설정하십시오. 이 속성들은 Object Inspector를 통해 디자인 타임 시에 또는 프로그램에 따라 런타임 시에 설정할 수 있습니다. 아래 예제는 배치 업데이트 모드에서의 *TADODataSet* 컴포넌트 준비를 보여 줍니다.

```
with ADODataset1 do begin
  CursorLocation := clUseClient;
  CursorType := ctStatic;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

데이터셋이 배치 업데이트 모드에서 열렸으면 모든 데이터 변경 사항은 기준 테이블에 직접 적용되기보다는 캐시됩니다.

개별적인 행의 업데이트 상태 검사

주어진 행을 현재로 만든 다음 ADO 데이터 컴포넌트의 *RecordStatus* 속성을 검사하여 주어진 행의 업데이트 상태를 결정합니다. *RecordStatus*는 현재 행과 그 행만의 업데이트 상태를 반영합니다.

```
case ADOQuery1.RecordStatus of
  rsUnmodified: StatusBar1.Panels[0].Text := 'Unchanged record';
  rsModified:   StatusBar1.Panels[0].Text := 'Changed record';
  rsDeleted:    StatusBar1.Panels[0].Text := 'Deleted record';
  rsNew:        StatusBar1.Panels[0].Text := 'New record';
end;
```

업데이트 상태에 기반한 여러 행 필터링

FilterGroup 속성을 사용하여 같은 업데이트 상태의 행 그룹에 속하는 행들만 표시하도록 레코드셋을 필터링합니다. 표시할 행의 업데이트 상태를 나타내는 *TFilterGroup* 상수로 *FilterGroup*을 설정합니다. *fgNone*(이 속성의 기본값)의 값은 어떤 필터링도 적용되지 않으며 삭제하려고 표시된 행을 제외한 모든 행이 업데이트 상태와 관계 없이 보이도록 지정합니다. 아래 예제는 대기하고 있는 배치 업데이트 행만 보여 줍니다.

```
FilterGroup := fgPendingRecords;
Filtered := True;
```

참고 *FilterGroup* 속성이 효과를 가지기 위해서는 ADO 데이터셋 컴포넌트의 *Filtered* 속성이 *True*로 설정되어야 합니다.

기준 테이블에 배치 업데이트 적용

UpdateBatch 메소드를 호출하여 아직 적용되지 않았거나 취소된 대기하고 있는 데이터의 변경 사항을 적용합니다. 변경되고 적용된 행은 레코드셋의 기반이 되는 기준 테이블에 놓여집니다. 그러면 삭제를 위해 표시된 캐시된 행으로 인해 해당 기준 테이블 행이 삭제됩니다. 캐시에 있지만 기준 테이블에는 없는 레코드 삽입은 기준 테이블에 추가됩니다. 그러면 수정된 행으로 인해 기존 테이블에서 해당 행의 열은 캐시의 새 열 값으로 변경됩니다.

매개변수 없이 단독으로 사용된 *UpdateBatch*는 모든 대기하고 있는 업데이트에 적용됩니다. *TAffectRecords* 값은 경우에 따라 *UpdateBatch*에 대한 매개변수로 전달될 수 있습니다. *arAll*을 제외한 값이 전달되면 대기하고 있는 변경 사항의 일부만 적용됩니다. *arAll*을 전달하는 것은 매개변수를 전달하지 않는 것과 같으므로 모든 대기하고 있는 업데이트에 적용됩니다. 아래 예제는 적용할 현재 활성 행만 적용한 것입니다.

```
ADODataset1.UpdateBatch(arCurrent);
```

배치 업데이트 취소

CancelBatch 메소드를 호출하여 아직 취소되거나 적용되지 않은 대기하고 있는 데이터의 변경 사항을 취소합니다. 대기하고 있는 배치 업데이트를 취소할 경우, 변경된 행의 필드 값은 *CancelBatch*와 *UpdateBatch* 중 하나가 호출되었다면 마지막 호출 이전 값으로, 또는 현재 대기하고 있는 변경된 배치 이전 값으로 되돌아갑니다.

매개변수 없이 단독으로 사용된 *CancelBatch*는 모든 대기하고 있는 업데이트를 취소합니다. *TAffectRecords* 값은 경우에 따라 *CancelBatch*에 대한 매개변수로 전달될 수 있습니다. *arAll*을 제외한 값이 전달되었다면 대기하고 있는 변경 사항의 일부만 취소됩니다. *arAll*을 전달하는 것은 매개변수 없이 전달하는 것과 같으며 모든 대기하고 있는 업데이트가 취소됩니다. 아래 예제는 모든 대기하고 있는 변경 사항을 취소하는 것입니다.

```
ADODataSet1.CancelBatch;
```

파일에서 데이터 로드와 파일에 데이터 저장

ADO 데이터셋 컴포넌트를 통해 검색된 데이터는 나중에 검색할 수 있도록 같은 컴퓨터나 다른 컴퓨터에 저장할 수 있습니다. 데이터는 ADTG 또는 XML 형식 중 하나로 저장됩니다. ADO는 유일하게 이 두 형식만 지원합니다. 그러나 ADO의 모든 버전에서 이 두 형식을 모두 적절하게 지원하는 것은 아닙니다. 어떤 파일 저장 형식이 지원되는지 알아보려면 사용하고 있는 버전의 ADO 설명서를 참조하십시오.

SaveToFile 메소드를 사용하여 파일에 데이터를 저장합니다. *SaveToFile*은 두 개 매개변수를 가집니다. 하나는 데이터가 저장되는 파일의 이름이고 또 하나는 선택적으로 데이터를 저장하는 형식(ADTG 또는 XML)입니다. *Format* 매개변수를 *pfADTG* 또는 *pfXML*로 설정하여 저장된 파일의 형식을 나타냅니다. *FileName* 매개변수에서 지정한 파일이 이미 있을 경우, *SaveToFile*은 *EOleException*을 발생시킵니다.

LoadFromFile 메소드를 사용하여 파일에서 데이터를 검색합니다. *LoadFromFile*은 로드할 파일의 이름인 단일 매개변수를 가집니다. 지정된 파일이 없으면 *LoadFromFile*은 *EOleException* 예외를 발생시킵니다. *LoadFromFile* 메소드를 호출하면 데이터셋 컴포넌트가 자동으로 활성화됩니다.

아래 예제에서 첫째 프로시저는 *TADODataSet* 컴포넌트 *ADODataSet1*이 검색한 데이터셋을 파일에 저장합니다. 대상 파일은 로컬 드라이브에 저장된 *SaveFile*이라는 이름의 ADTG 파일입니다. 둘째 프로시저는 저장된 이 파일을 *TADODataSet* 컴포넌트 *ADODataSet2*에 로드합니다.

```
procedure TForm1.SaveBtnClick(Sender:TObject);
begin
  if (FileExists('c:\SaveFile')) then
  begin
    DeleteFile('c:\SaveFile');
    StatusBar1.Panels[0].Text := 'Save file deleted!';
  end;
  ADODataSet1.SaveToFile('c:\SaveFile', pfADTG);
end;

procedure TForm1.LoadBtnClick(Sender:TObject);
begin
  if (FileExists('c:\SaveFile')) then
    ADODataSet2.LoadFromFile('c:\SaveFile')
  else
    StatusBar1.Panels[0].Text := 'Save file does not exist!';
end;
```

데이터를 저장하고 로드하는 데이터셋은 위와 같은 형태나 같은 애플리케이션 또는 같은 컴퓨터에도 있을 필요가 없습니다. 따라서 데이터를 브리프케이스 방식 (briefcase-

style)으로 마치 서류 가방에 담아 옮기듯이 한 컴퓨터에서 다른 컴퓨터로 쉽게 전송할 수 있습니다.

TADODataset 사용

*TADODataset*은 ADO 데이터 저장소의 데이터로 작업하기 위한 범용 데이터셋입니다. 다른 ADO 데이터셋 컴포넌트와 달리 *TADODataset*은 테이블 타입, 쿼리 타입, 내장 프로시저 타입 데이터셋 중 어느 것도 아닙니다. 그 대신 다음과 같은 타입처럼 사용될 수 있습니다.

- 테이블 타입 데이터셋과 마찬가지로 *TADODataset*을 사용하여 단일 데이터베이스 테이블의 모든 행과 열을 나타낼 수 있습니다. 이렇게 사용하려면 *CommandType* 속성을 *cmdTable*로, *CommandText* 속성을 테이블의 이름으로 설정하십시오. *TADODataset*은 다음과 같은 테이블 타입 작업을 지원합니다.
 - 인덱스를 할당하여 레코드 기반 검색을 기준으로 레코드나 폼을 정렬합니다. 18-26 페이지의 "인덱스로 레코드 정렬"에 설명된 표준 인덱스 속성과 메소드 외에 *TADODataset*을 사용하면 *Sort* 속성을 설정하여 임시 인덱스를 사용하여 정렬할 수 있습니다. *Seek* 메소드를 사용하여 수행된 인덱스 기반 검색은 현재 인덱스를 사용합니다.
 - 데이터셋을 비웁니다. *DeleteRecords* 메소드는 삭제할 레코드를 지정할 수 있으므로 다른 테이블 타입 데이터셋에 관련된 메소드보다 더 많은 제어 기능을 제공합니다.

*TADODataset*이 지원하는 테이블 타입 작업은 *cmdTable*의 *CommandType*을 사용하지 않는 경우에도 사용할 수 있습니다.

- 쿼리 타입 데이터셋과 마찬가지로 *TADODataset*을 사용하면 데이터셋을 열 때 실행되는 단일 SQL 명령을 지정할 수 있습니다. 이렇게 사용하려면 *CommandType* 속성을 *cmdText*로, *CommandText* 속성을 실행할 SQL 명령으로 설정하십시오. 디자인 타임 시 SQL 명령을 구성할 때 도움말이 필요하면 Object Inspector에서 *CommandText* 속성을 더블 클릭하여 Command Text Editor를 사용하십시오. *TADODataset*은 다음과 같은 쿼리 타입 작업을 지원합니다.
 - 쿼리 텍스트에서 매개변수를 사용합니다. 쿼리 매개변수에 대한 자세한 내용은 18-45 페이지의 "쿼리의 매개변수 사용"을 참조하십시오.
 - 매개변수를 사용하여 마스터/디테일 관계를 설정합니다. 이 작업을 수행하는 방법에 대한 자세한 내용은 18-47 페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"을 참조하십시오.
 - *Prepared* 속성을 *True*로 설정하여 성능 개선에 앞서 쿼리를 준비합니다.
- 내장 프로시저 타입 데이터셋과 마찬가지로 *TADODataset*을 사용하면 데이터셋을 열 때 실행되는 내장 프로시저를 지정할 수 있습니다. 이렇게 사용하려면 *CommandType* 속성을 *cmdStoredProc*로, *CommandText* 속성을 내장 프로시저의 이름으로 설정하십시오. *TADODataset*은 다음과 같은 내장 프로시저 타입의 작업을 지원합니다.

- 내장 프로시저 매개변수를 사용합니다. 내장 프로시저 매개변수에 대한 자세한 내용은 18-51 페이지의 "내장 프로시저 매개변수 작업"을 참조하십시오.
- 여러 결과 집합을 폐치합니다. 이 작업을 수행하는 방법에 대한 자세한 내용은 18-54 페이지의 "여러 결과 집합 폐치"를 참조하십시오.
- *Prepared* 속성을 *True*로 설정하여 성능 개선에 앞서 내장 프로시저를 준비합니다.

또한 *TADODataSet*을 사용하면 *CommandType* 속성을 *cmdFile*로, *CommandText* 속성을 파일 이름으로 설정하여 파일에 저장된 데이터를 사용할 수 있습니다.

*CommandText*와 *CommandType* 속성을 설정하기 전에 *Connection* 또는 *Connectionstring* 속성을 설정하여 *TADODataSet*을 데이터 저장소에 연결해야 합니다. 이 프로세스는 21-10 페이지의 "데이터 저장소에 ADO 데이터셋 연결"에서 다룹니다. 또 다른 방법으로는 RDS DataSpace 객체를 사용하여 *TADODataSet*을 ADO 기반 애플리케이션 서버에 연결할 수도 있습니다. RDS DataSpace 객체를 사용하려면 *RDSConnection* 속성을 *TRDSConnection* 객체로 설정하십시오.

Command 객체 사용

ADO 환경에서 명령은 프로바이더 특유의 작업 결과를 텍스트로 나타낸 것입니다. 대개 DDL(Data Definition Language)과 DML(Data Manipulation Language) SQL 문입니다. 명령에 사용되는 랭귀지는 서버 특정적이지만 일반적으로 SQL 랭귀지의 SQL-92 표준과 호환됩니다.

*TADOQuery*를 사용하여 항상 명령을 실행할 수 있어도 데이터셋 컴포넌트 사용을 원하지 않을 수도 있습니다. 특히 명령이 결과 집합을 반환하지 않을 경우에 그렇습니다. 또 한 가지 방법으로 명령을 한 번에 하나씩 실행하도록 만들어진 간단한 객체인 *TADOCommand* 컴포넌트를 사용할 수도 있습니다. *TADOCommand*는 기본적으로 DDL(Data Definition Language) SQL 문과 같은 결과 집합을 반환하지 않는 명령을 실행하도록 만들어졌습니다. 그러나 오버로드된 *Execute* 메소드 버전을 통해 ADO 데이터셋 컴포넌트의 *RecordSet* 속성에 할당될 수 있는 결과 집합을 반환할 수 있습니다.

데이터를 폐치하고 레코드를 탐색하고 데이터를 편집하는 데에 표준 데이터셋 메소드를 사용할 수 없다는 것을 제외하면 *TADOCommand*로 작업하는 것은 일반적으로 *TADODataSet*으로 작업하는 것과 매우 비슷합니다. *TADOCommand* 객체는 ADO 데이터셋과 같은 방식으로 데이터 저장소에 연결합니다. 자세한 내용은 21-10 페이지의 "데이터 저장소에 ADO 데이터셋 연결"을 참조하십시오.

다음 항목에서는 *TADOCommand*를 사용하여 명령을 지정하고 실행하는 방법을 자세히 설명합니다.

명령 지정

CommandText 속성을 사용하여 *TADOCommand* 컴포넌트에 대한 명령을 지정합니다. *TADODataSet*과 마찬가지로 *TADOCommand*를 사용하면 *CommandType* 속성에

따라 다르게 명령을 지정할 수 있습니다. *CommandType*의 가능한 값에는 *cmdText*(명령이 SQL 문일 때 사용됨), *cmdTable*(테이블 이름), *cmdStoredProc*(명령이 내장 프로시저의 이름일 경우)가 있습니다. 디자인 타임에는 Object Inspector의 목록에서 해당 명령 타입을 선택하십시오. 런타임에는 타입 *TCommandType*의 값을 *CommandType* 속성에 할당하십시오.

```
with ADOCommand1 do begin
  CommandText := 'AddEmployee';
  CommandType := cmdStoredProc;
  ...
end;
```

특정 타입이 지정되지 않을 경우, 서버가 *CommandText*의 명령에 따라 가장 최선의 결정을 내립니다.

*CommandText*는 매개변수나 매개변수를 사용하는 내장 프로시저의 이름을 포함하는 SQL 쿼리의 텍스트를 포함할 수 있습니다. 그런 경우 명령을 실행하기 전에 매개변수에 바인딩된 매개변수 값을 제공해야 합니다. 자세한 내용은 21-19 페이지의 "명령 매개변수 처리"를 참조하십시오.

Execute 메소드 사용

*TADOCommand*가 명령을 실행하기 전에 데이터 저장소에 올바르게 연결되어 있어야 합니다. 이것은 ADO 데이터셋의 경우와 마찬가지로 방법으로 설정됩니다. 자세한 내용은 21-10 페이지의 "데이터 저장소에 ADO 데이터셋 연결"을 참조하십시오.

명령을 실행하려면 *Execute* 메소드를 호출하십시오. *Execute*는 오버로드된 메소드로서 이 메소드를 이용하여 명령을 실행하기에 가장 적합한 방법을 선택할 수 있습니다.

매개변수를 필요로 하지 않는 명령과 영향을 받은 레코드 수를 알 필요가 없을 경우, 매개변수 없이 *Execute*를 호출하십시오.

```
with ADOCommand1 do begin
  CommandText := 'UpdateInventory';
  CommandType := cmdStoredProc;
  Execute;
end;
```

*Execute*의 다른 버전에서는 Variant 배열을 사용하여 매개변수 값을 제공하고 명령의 영향을 받은 레코드 수를 알 수 있습니다.

결과 집합을 반환하는 명령을 실행하는 것에 대한 자세한 내용은 21-19 페이지의 "명령으로 결과 집합 검색"을 참조하십시오.

명령 취소

명령을 비동기식으로 실행할 경우, *Execute*를 호출한 다음 *Cancel* 메소드를 호출하여 실행을 중단할 수 있습니다.

```
procedure TDataForm.ExecuteButtonClick(Sender:TObject);
begin
```



```

        ADOCommand1.Execute;
    end;

    procedure TDataForm.CancelButtonClick(Sender:TObject);
    begin
        ADOCommand1.Cancel;
    end;

```

Cancel 메소드는 대기하고 있는 명령이 있고 비동기식으로 실행되었을 때만 효력이 있습니다(*eoAsynchExecute*는 *Execute* 메소드의 *ExecuteOptions* 매개변수에 있습니다). 명령은 *Execute* 메소드가 취소되었지만 명령이 아직 완료되지 않았거나 시간이 초과되었을 때 대기하고 있다고 할 수 있습니다.

명령은 *CommandTimeout* 속성에 지정된 시간(초)이 만료되기 전에 명령이 완료되지 않거나 취소되었을 때 시간 초과됩니다. 기본적으로 명령은 30초 후에 시간 초과됩니다.

명령으로 결과 집합 검색

결과 집합 반환 여부에 따라 다른 메소드를 사용하여 실행하는 *TADOQuery* 컴포넌트와 달리 *TADOCommand*는 항상 결과 집합 반환 여부와 상관 없이 *Execute* 명령을 사용하여 명령을 실행합니다. 이 명령이 결과 집합을 반환하면 *Execute*는 *ADO_RecordSet* 인터페이스에 인터페이스를 반환합니다.

이 인터페이스를 이용하여 가장 편리하게 작업하는 방법은 ADO 데이터셋의 *RecordSet* 속성에 인터페이스를 할당하는 것입니다.

예를 들면, 다음 코드는 *TADOCommand(ADOCommand1)*를 사용하여 결과 집합을 반환하는 SELECT 쿼리를 실행합니다. 그러면 이 결과 집합은 *TADODataSet* 컴포넌트(*ADODataSet1*)의 *RecordSet* 속성에 할당됩니다.

```

with ADOCommand1 do begin
    CommandText := 'SELECT Company, State ' +
        'FROM customer ' +
        'WHERE State = :StateParam';
    CommandType := cmdText;
    Parameters.ParamByName('StateParam').Value := 'HI';
    ADODataSet1.Recordset := Execute;
end;

```

결과 집합이 ADO 데이터셋의 *Recordset* 속성에 할당되는 즉시 데이터셋이 자동으로 활성화되고 그 데이터를 사용할 수 있게 됩니다.

명령 매개변수 처리

TADOCommand 객체는 다음 두 가지 방법으로 매개변수를 사용할 수 있습니다.

- *CommandText* 속성은 매개변수를 포함하는 쿼리를 지정할 수 있습니다. *TADOCommand*에서 매개변수화된 쿼리로 작업하는 것은 ADO 데이터셋에서 매개변수화된 쿼리를 사용하는 것과 같습니다. 매개변수화된 쿼리에 대한 자세한 내용은 18-45 페이지의 "쿼리의 매개변수 사용"을 참조하십시오.

- *CommandText* 속성은 매개변수를 사용하는 내장 프로시저를 지정할 수 있습니다. 내장 프로시저 매개변수는 ADO 데이터셋을 이용할 때와 마찬가지로 *TADOCCommand* 를 사용하여 작동합니다. 내장 프로시저 매개변수에 대한 자세한 내용은 18-51 페이지의 "내장 프로시저 매개변수 작업"을 참조하십시오.

*TADOCCommand*로 작업할 때 매개변수 값을 제공하는 방법에는 다음 두 가지가 있습니다. *Execute* 메소드를 호출할 때 매개변수 값을 제공하거나 *Parameters* 속성을 사용하기 전에 매개변수 값을 지정하는 것입니다.

Execute 메소드는 오버로드되어 Variant 배열처럼 일련의 매개변수 값을 사용하는 버전을 포함합니다. 이것은 *Parameters* 속성을 설정하지 않고 매개변수 값을 빨리 제공하려고 할 때 유용합니다.

```
ADOCCommand1.Execute(VarArrayOf([Edit1.Text, Date]));
```

출력 매개변수를 반환하는 내장 프로시저로 작업할 때는 대신 *Parameters* 속성을 사용해야 합니다. 출력 매개변수를 읽을 필요가 없을 경우에도 *Parameters* 속성을 사용할 수 있습니다. 그러면 디자인 타임 시 매개변수를 제공할 수 있으며 데이터셋에 매개변수로 작업하는 것과 같은 방법으로 *TADOCCommand* 속성을 사용할 수 있습니다.

CommandText 속성을 설정하면 *Parameters* 속성은 자동으로 업데이트되어 쿼리의 매개변수나 내장 프로시저에서 사용하는 매개변수에 반영됩니다. 디자인 타임 시 Object Inspector에서 *Parameters* 속성에 대한 생략 기호 버튼을 클릭하면 Parameter Editor에서 매개변수에 액세스할 수 있습니다. 런타임 시 *TParameter*의 속성과 메소드를 사용하여 각 매개변수의 값을 설정하거나 구할 수 있습니다.

```
with ADOCCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName('NewValueParam').Value := 57;
  Execute
end;
```

22

단방향 데이터셋 사용

*dbExpress*는 SQL 데이터베이스 서버에 빠르게 액세스하도록 하는 경량급 (lightweight) 데이터베이스 드라이버 집합입니다. 각 지원 데이터베이스에 대해 *dbExpress*는 일정한 *dbExpress* 인터페이스 집합에 적합한 서버 특정 소프트웨어를 사용하는 드라이버를 제공합니다. *dbExpress*를 사용하는 데이터베이스 애플리케이션을 배포하는 경우, 구축할 애플리케이션 파일을 가지는 서버 특정 드라이버인 하나의 dll만 포함시키면 됩니다.

*dbExpress*는 단방향 데이터셋을 사용하여 데이터베이스에 액세스할 수 있게 합니다. 단방향 데이터셋은 최소한의 오버헤드를 가지고 데이터베이스 정보에 대한 빠른 경량급 (lightweight) 액세스를 하도록 디자인되었습니다. 다른 데이터셋처럼 데이터베이스 서버에 SQL 명령을 보낼 수 있고, 명령이 레코드 집합을 반환하는 경우 레코드에 액세스하기 위해 커서를 얻을 수 있습니다. 하지만 단방향 데이터셋은 단방향 커서를 검색만 할 수 있습니다. 이것은 메모리에 데이터를 버퍼로 저장하지 않으며 다른 타입의 데이터셋에 비해 실행 속도가 더 빠르고 리소스를 덜 차지합니다. 하지만 레코드가 버퍼로 저장되지 않기 때문에 단방향 데이터셋은 다른 데이터셋보다 유연하지 않습니다. *TDataSet*에 있는 많은 기능들이 단방향 데이터셋에서 구현되지 않거나 또는 예외를 발생시킵니다. 예를 들면, 다음과 같습니다.

- 지원되는 유일한 탐색 메소드는 *First* 와 *Next* 메소드입니다. 대부분의 다른 메소드는 예외를 발생시킵니다. 북마크 지원과 관련된 메소드 같은 일부 메소드는 아무런 기능도 하지 않습니다.
- 편집 기능은 편집 내용을 보유하는 버퍼가 필요하므로 여기에는 편집에 대한 기본 제공 지원이 없습니다. *CanModify* 속성이 항상 *False*를 표시하기 때문에 데이터셋을 편집 모드로 두려고 해도 항상 실패하게 됩니다. 하지만 SQL UPDATE 명령을 사용하여 데이터를 업데이트하기 위해 단방향 데이터셋을 사용하거나, *dbExpress* 활성 클라이언트 데이터셋을 사용하거나 데이터셋을 클라이언트 데이터셋에 연결하여 편집 기능을 지원할 수 있습니다(14-10 페이지의 "다른 데이터셋에 연결" 참조).

- 여러 레코드를 사용하여 작업하는 필터는 버퍼를 필요로 하기 때문에 필터에 대한 지원은 없습니다. 단방향 데이터셋을 필터링하려고 시도하면 예외가 발생합니다. 그 대신 데이터 표시에 대한 모든 제한 사항은 데이터셋의 데이터를 정의하는 SQL 명령을 사용하여 부과되어야 합니다.
- 조회 필드는 조회 값들을 포함해서 여러 레코드를 보유하기 위한 버퍼가 필요하기 때문에 조회 필드에 대한 지원이 없습니다. 단방향 데이터셋에서 조회 필드를 정의하는 경우 아무런 적절한 기능도 하지 않습니다.

이러한 제한 사항에도 불구하고 단방향 데이터셋은 데이터를 액세스하는 확실한 방법입니다. 단방향 데이터셋은 가장 빠른 데이터 액세스 메커니즘으로서 사용과 배포가 매우 간단합니다.

단방향 데이터셋의 타입

컴포넌트 팔레트의 *dbExpress* 페이지에는 *TSQLDataSet*, *TSQLQuery*, *TSQLTable* 및 *TSQLStoredProc*와 같은 네 가지 타입의 단방향 데이터셋이 들어 있습니다.

*TSQLDataSet*은 네 가지 타입 중에서 가장 일반적인 타입입니다. SQL 데이터셋을 사용하면 *dbExpress*를 통해 사용 가능한 데이터를 나타내거나 *dbExpress*를 통해 액세스된 데이터베이스에 명령을 보낼 수 있습니다. 새 데이터베이스 애플리케이션의 데이터베이스 테이블로 작업하는 경우에는 이 컴포넌트를 사용할 것을 권장합니다.

*TSQLQuery*는 SQL 문을 캡슐화하고 애플리케이션이 결과 레코드에 액세스할 수 있게 하는 쿼리 타입 데이터셋입니다. 쿼리 타입 데이터셋의 사용에 대한 내용은 18-42 페이지의 "쿼리 타입 데이터셋 사용"을 참조하십시오.

*TSQLTable*은 단일 데이터베이스 테이블의 모든 행과 열을 나타내는 테이블 타입 데이터셋입니다. 테이블 타입 데이터셋의 사용에 대한 내용은 18-25 페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오.

*TSQLStoredProc*는 데이터베이스 서버에 정의된 내장 프로시저를 실행하는 내장 프로시저 타입 데이터셋입니다. 내장 프로시저 타입 데이터셋의 사용에 대한 내용은 18-50 페이지의 "내장 프로시저 타입 데이터셋 사용"을 참조하십시오.

참고 *dbExpress* 페이지에는 단방향 데이터셋이 아닌 *TSQLClientDataSet*도 들어 있습니다. 오히려 이것은 단방향 데이터셋을 내부적으로 사용하여 데이터에 액세스하는 클라이언트 데이터셋입니다.

데이터베이스 서버에 연결

단방향 데이터셋을 사용하는 첫 번째 단계는 이를 데이터베이스 서버에 연결하는 것입니다. 디자인 타임에 데이터셋이 데이터베이스 서버에 대한 연결이 활성화되면 Object Inspector는 다른 속성에 대한 값의 드롭다운 목록을 제공할 수 있습니다. 예를 들어 내장 프로시저를 나타낼 때는 연결이 활성화되어야 Object Inspector가 서버에서 사용할 수 있는 내장 프로시저의 목록을 나타낼 수 있습니다.

데이터베이스 서버에 대한 연결은 개별 *TSQLConnection* 컴포넌트에 의해 나타납니다. 다른 데이터베이스 연결 컴포넌트처럼 *TSQLConnection*을 사용합니다. 데이터베이스 연결 컴포넌트에 대한 내용은 17장 "데이터베이스에 연결"을 참조하십시오.

*TSQLConnection*을 사용하여 단방향 데이터셋을 데이터베이스 서버에 연결하려면 *SQLConnection* 속성을 설정합니다. 디자인 타임에 Object Inspector의 드롭다운 목록에서 SQL 연결 컴포넌트를 선택할 수도 있습니다. 런타임에 이러한 할당을 지정하는 경우에는 연결이 활성화되었는지 확인하십시오.

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

일반적으로 여러 데이터베이스 서버의 데이터를 사용하지 않는다면 애플리케이션의 모든 단방향 데이터셋은 동일한 연결 컴포넌트를 공유합니다. 하지만 서버가 각각의 연결에 대해 여러 문장을 지원하지 않는 경우 각 데이터셋에 대해 분리된 연결을 사용할 수도 있습니다. *MaxStmtsPerConn* 속성을 읽어서 데이터베이스 서버가 각 데이터셋에 대한 개별 연결이 필요한지 확인하십시오. 기본적으로 서버가 제한하는 경우, *TSQLConnection*은 한 연결에서 실행될 수 있는 문장 수의 필요에 따라 연결을 생성합니다. 사용 중인 연결을 더 엄격하게 추적하려면 *AutoClone* 속성을 *False*로 설정합니다.

SQLConnection 속성을 할당하기 전에 데이터베이스 서버와 모든 필요한 연결 매개변수(서버에서 사용할 데이터베이스, 서버를 실행할 시스템의 호스트 이름, 사용자 이름, 암호 등을 포함)를 식별하기 위해 *TSQLConnection* 컴포넌트를 설정해야 합니다.

TSQLConnection 설정

연결을 여는 *TSQLConnection*에 대한 데이터베이스 연결의 세부 사항을 충분히 설명하기 위해 사용할 드라이버와 이 드라이버에 전달되는 일련의 연결 매개변수를 모두 식별해야 합니다.

드라이버 식별

드라이버는 INTERBASE, MYSQL, ORACLE 또는 DB2와 같은 설치되어 있는 *dbExpress* 드라이버 이름인 *DriverName* 속성에 의해 식별됩니다. 드라이버 이름은 다음 두 파일에 연결됩니다.

- *dbExpress* 드라이버. 이 파일은 dbexpint.dll, dbexpora.dll, dbexpmys.dll 또는 dbexpdb2.dll 등의 이름을 갖는 동적 연결 라이브러리거나 애플리케이션에 정적으로 연결할 수 있는 컴파일된 유닛(dbexptint.dcu, dbexpora.dcu, dbexpmys.dcu 또는 dbexpdb2.dcu) 중 하나가 될 수 있습니다.
- 클라이언트측 지원을 위해 데이터베이스 공급업체에 의해 제공되는 동적 연결 라이브러리.

이러한 두 파일과 데이터베이스 이름의 관계는 dbxdrivers.ini 라는 파일에 저장되는데 dbExpress 드라이버를 설치할 때 업데이트됩니다. 일반적으로 *DriverName* 값이 주어질 때 SQL 연결 컴포넌트가 dbxdrivers.ini에서 파일들을 조회하기 때문에 이러한 파일들에 대해서 염려할 필요는 없습니다. *DriverName* 속성을 설정할 때 *TSQLConnection* 은 자동으로 *LibraryName*과 *VendorLib* 속성을 연결된 dll의 이름으로 설정합니다. 일단 *LibraryName*과 *VendorLib*가 설정되면 애플리케이션은 dbxdrivers.ini에 의존할 필요가 없습니다(즉, 런타임 시 *DriverName* 속성을 설정하지 않으면 애플리케이션과 함께 dbxdrivers.ini를 배포할 필요는 없습니다.).

연결 매개변수 지정

Params 속성은 이름/값 쌍을 나열하는 문자열 목록입니다. 각각의 쌍은 *Name=Value*의 형식을 취하며 여기서 *Name*은 매개변수의 이름이고 *Value*는 할당하려는 값입니다.

필요한 특정 매개변수는 사용 중인 데이터베이스 서버에 따라 다릅니다. 하지만 *Database* 매개변수는 모든 서버에 필요합니다. 매개변수의 값은 사용 중인 서버에 따라 다릅니다. 예를 들어, *Database*는 InterBase에서는 .gdb 파일의 이름이고 ORACLE에서는 TNSNames.ora의 항목인 반면, DB2에서는 클라이언트측 노드 이름입니다.

다른 일반적인 매개변수에는 *User_Name*(로그인할 때 사용하는 이름), *Password*(*User_Name*에 대한 암호), *HostName*(시스템 이름 또는 서버가 위치하는 IP 주소) 및 *TransIsolation*(도입하는 트랜잭션이 다른 트랜잭션에 의한 변경 사항을 인식하는 정도)이 포함됩니다. 드라이버 이름을 지정할 때 *Params* 속성은 해당 드라이버 타입에 필요한 모든 매개변수가 기본값으로 초기화되어 미리 로드됩니다.

*Params*는 문자열 목록이기 때문에 디자인 타임에 Object Inspector의 *Params* 속성을 더블 클릭하여 String List 편집기를 사용해서 매개변수를 편집할 수 있습니다. 런타임 시 *Params.Values* 속성을 사용하여 개별적인 매개변수에 값을 할당합니다.

연결에 대한 설명 이름 지정

*DatabaseName*과 *Params* 속성만 사용하여 언제나 연결을 지정할 수 있지만 특정 조합을 명명하여 이름에 의해 연결을 식별하는 것이 더 편리할 수 있습니다. dbExpress 데이터베이스와 매개변수 조합을 명명할 수 있는데 이는 dbxconnections.ini라는 파일에 저장됩니다. 각 조합의 이름은 연결 이름으로 불러집니다.

연결 이름을 정의하고 나면 *ConnectionName* 속성을 올바른 연결 이름으로 간단히 설정하여 데이터베이스 연결을 식별할 수 있습니다. *ConnectionName*을 설정하면 자동으로 *DriverName* 및 *Params* 속성이 설정됩니다. *ConnectionName*을 설정한 다음 *Params* 속성을 편집하여 저장된 매개변수 값들에 일시적으로 변화를 줄 수는 있지만 *DriverName* 속성을 변경할 경우 *Params* 및 *ConnectionName*이 모두 지워집니다.

연결 이름을 사용하면 하나의 데이터베이스(예: Local InterBase)를 사용하여 애플리케이션을 개발하여 다른 데이터베이스(예: ORACLE)에서 사용할 목적으로 배포하는 경우에 유용합니다. 이런 경우 *DriverName* 및 *Params*는 개발 당시 사용한 값이 애플리케이션을 배포하는 시스템에 따라 다를 수 있습니다. dbxconnections.ini 파일의 두 버전을 사용하여 두 연결 설명 간에 쉽게 전환할 수 있습니다. 디자인 타임에 애플리케이션은 디자인 타임 버전의 dbxconnections.ini에서 *DriverName*과 *Params*를 로드합니다. 그런 다음 애플리케이션을 배포할 때 "실제" 데이터베이스를 사용하는

dbxconnections.ini의 분리 버전에서 이러한 값들을 로드합니다. 하지만 이렇게 하기 위해서는 연결 컴포넌트가 런타임에 *DriverName* 및 *Params* 속성을 다시 로드하도록 지시해야 합니다. 이를 수행하는 방법은 다음 두 가지가 있습니다.

- *LoadParamsOnConnect* 속성을 *True*로 설정합니다. 그러면 연결이 열릴 때 *TSQLConnection*이 *DriverName* 및 *Params*를 dbxconnections.ini 파일의 *ConnectionName*에 연결된 값으로 자동으로 설정합니다.
- *LoadParamsFromIniFile* 메소드를 호출합니다. 이 메소드는 *DriverName*과 *Params*를 dbxconnections.ini(또는 지정한 다른 파일)의 *ConnectionName*에 연결된 값들로 설정합니다. 연결을 열기 전에 특정 매개변수 값을 오버라이드하려는 경우 이 메소드를 사용할 수 있습니다.

Connection Editor 사용

연결 이름과 연결된 드라이버 및 연결 매개변수 간의 관계는 dbxconnections.ini 파일에 저장됩니다. Connection Editor를 사용하면 이들 연결을 만들거나 수정할 수 있습니다.

Connection Editor를 표시하려면 *TSQLConnection* 컴포넌트를 더블 클릭합니다. Connection Editor에는 모든 사용 가능한 드라이버가 들어 있는 드롭 다운 목록, 현재 선택된 드라이버에 대한 연결 이름 목록, 현재 선택된 연결 이름에 대한 연결 매개변수를 나열한 테이블이 나타납니다.

이 대화 상자를 사용하여 드라이버와 연결 이름을 선택하여 사용할 연결을 나타낼 수 있습니다. 원하는 구성을 선택한 다음 Test Connection 버튼을 클릭해서 올바른 구성을 선택했는지 확인합니다.

또한 다음과 같이 이 대화 상자를 사용하여 dbxconnections.ini에서 명명된 연결을 편집할 수 있습니다.

- 매개변수 테이블의 매개변수 값을 편집하여 현재 선택되어 있는 명명된 연결을 변경합니다. OK를 클릭하여 대화 상자를 종료할 때 새 매개변수 값들이 dbxconnections.ini에 저장됩니다.
- Add Connection 버튼을 클릭하여 새로운 명명된 연결을 정의합니다. 대화 상자가 나타나서 사용할 드라이버와 새로운 연결의 이름을 지정할 수 있습니다. 일단 연결이 명명되면 원하는 연결을 지정하기 위해 매개변수를 편집하고 OK 버튼을 클릭하여 dbxconnections.ini에 새 연결을 저장합니다.
- Delete Connection 버튼을 클릭하여 dbxconnections.ini에서 현재 선택되어 있는 명명된 연결을 삭제합니다.
- Rename Connection 버튼을 클릭하여 현재 선택되어 있는 명명된 연결의 이름을 변경합니다. 매개변수에 대한 편집 내용이 OK 버튼을 클릭할 때 새로운 이름으로 저장되는 것에 유의하십시오.

표시할 데이터 지정

단방향 데이터셋이 나타낼 데이터를 지정하는 방법은 여러 가지입니다. 사용하고 있는 단방향 데이터셋의 타입에 따라 사용할 방법이 달라지며 정보를 단일 데이터베이스 테이블, 쿼리의 결과 또는 내장 프로시저 중 어디에서 가져 오는지에 따라 방법이 다릅니다.

TSQLDataSet 컴포넌트를 사용할 때 *CommandType* 속성을 사용하여 데이터셋이 데이터를 얻는 위치를 지정합니다. *CommandType*은 다음 중 어떤 값이라도 취할 수 있습니다.

- *ctQuery*: *CommandType*이 *ctQuery*이면 *TSQLDataSet*은 사용자가 지정하는 쿼리를 실행합니다. 쿼리가 SELECT 명령이면 데이터셋은 레코드의 결과 집합을 포함합니다.
- *ctTable*: *CommandType*이 *ctTable*이면 *TSQLDataSet*은 지정된 테이블로부터의 모든 레코드를 검색합니다.
- *ctStoredProc*: *CommandType*이 *ctStoredProc*이면 *TSQLDataSet*은 내장 프로시저를 실행합니다. 내장 프로시저가 커서를 반환하면 데이터셋은 반환된 레코드를 포함합니다.

참고 서버에서 이용할 수 있는 메타데이터가 단방향 데이터셋에 있을 수도 있습니다. 이러한 방법에 대한 자세한 내용은 22-13 페이지의 "메타데이터를 단방향 데이터셋에 패치"를 참조하십시오.

쿼리의 결과 표시

쿼리 사용은 레코드 집합을 지정하는 가장 일반적인 방법입니다. 쿼리는 단순히 SQL로 작성된 명령입니다. 쿼리 결과를 나타내기 위해 *TSQLDataSet* 또는 *TSQLQuery*를 사용할 수 있습니다.

*TSQLDataSet*을 사용하는 경우, *CommandType* 속성을 *ctQuery*로 설정하고 쿼리 문의 텍스트를 *CommandText* 속성에 할당합니다. *TSQLQuery*를 사용할 경우에는 쿼리를 *SQL* 속성에 할당합니다. 이러한 속성들은 모든 범용 데이터셋이나 쿼리 타입 데이터셋에 대해 동일한 방식으로 작용합니다. 18-43 페이지의 "쿼리 지정"에서 더 자세한 내용을 설명합니다.

쿼리를 지정할 때 디자인 타임이나 런타임에 값이 바뀔 수 있는 매개변수나 변수를 포함시킬 수 있습니다. 매개변수는 SQL 문에 나타나는 데이터 값을 대체할 수 있습니다. 쿼리에서의 매개변수 사용 및 매개변수에 값을 제공하는 것은 18-45 페이지의 "쿼리의 매개변수 사용"에서 설명됩니다.

SQL은 레코드를 반환하지 않고 서버에서 작업을 수행하는 UPDATE 쿼리와 같은 쿼리를 정의합니다. 그러한 쿼리는 22-10 페이지의 "레코드를 반환하지 않는 명령 실행"에서 설명됩니다.

테이블의 레코드 표시

하나의 기본 데이터베이스 테이블에 모든 필드와 모든 레코드를 표시하려는 경우, SQL 문을 사용자가 직접 작성하는 것보다 *TSQLDataSet* 또는 *TSQLTable*을 사용하여 쿼리를 생성할 수 있습니다.

참고 서버 성능이 문제가 될 경우에는 쿼리의 자동 생성되는 쿼리에 의존하기보다 명시적으로 쿼리를 작성할 수도 있습니다. 자동으로 생성된 쿼리는 테이블에 있는 필드를 명시적으로 나열하기보다는 대표 문자(와일드카드)를 사용합니다. 이는 서버 성능에 약간의 저하를 초래합니다. 자동으로 생성되는 쿼리의 대표 문자(*)는 서버의 필드를 변경하는데 좀더 강력합니다.

TSQLDataSet을 사용하여 테이블 표시

*TSQLDataSet*이 단일 데이터베이스 테이블의 모든 필드와 모든 레코드를 페치하는 쿼리를 생성하도록 하려면 *CommandType* 속성을 *ctTable*로 설정합니다.

*CommandType*이 *ctTable*이면 *TSQLDataSet*은 두 가지 속성 값을 기반으로 쿼리를 생성합니다.

- *CommandText*는 *TSQLDataSet* 객체가 나타낼 데이터베이스 테이블의 이름을 지정합니다.
- *SortFieldNames*는 데이터 정렬을 위해 사용하는 모든 필드 이름을 중요도 순으로 나열합니다.

예를 들어, 다음과 같이 지정한다고 가정해 봅니다.

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

*TSQLDataSet*은 Employee 테이블에서 HireDate로 정렬하고 HireDate 내에서 Salary로 정렬하여 모든 레코드를 나열하는 다음과 같은 쿼리를 생성합니다.

```
select * from Employee order by HireDate, Salary
```

TSQLTable을 사용하여 테이블 표시

*TSQLTable*을 사용할 경우, *TableName* 속성을 사용하여 원하는 테이블을 지정합니다.

데이터셋에 있는 필드의 순서를 지정하려면 인덱스를 지정해야 합니다. 이를 수행하는 방법은 다음 두 가지가 있습니다.

- *IndexName* 속성을 원하는 순서를 부여하는 서버에 정의된 인덱스의 이름으로 설정합니다.
- *IndexFieldNames* 속성을 정렬할 필드 이름의 세미콜론으로 구분한 목록으로 설정합니다. *IndexFieldNames*는 쉽표가 아닌 세미콜론을 구분자로 사용한다는 점만 제외하면 *TSQLDataSet*의 *SortFieldNames* 속성과 동일한 기능을 합니다.

내장 프로시저의 결과 표시

내장 프로시저는 SQL 서버에 이름이 지정되고 저장된 SQL 문의 집합입니다. 실행하려는 내장 프로시저를 나타내는 방법은 사용하고 있는 단방향 데이터셋의 타입에 따라 다릅니다.

*TSQLDataSet*을 사용할 경우, 다음과 같은 방법으로 내장 프로시저를 지정합니다.

- *CommandType* 속성을 *ctStoredProc*로 설정합니다.
- *CommandText* 속성의 값으로 내장 프로시저의 이름을 지정합니다.

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

*TSQLStoredProc*를 사용할 때에는 *StoredProcName* 속성 값으로서 내장 프로시저의 이름을 지정하기만 하면 됩니다.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

내장 프로시저를 식별한 후에 애플리케이션에서 내장 프로시저의 입력 매개변수 값을 입력하거나 내장 프로시저를 실행한 후의 출력 매개변수 값을 검색해야 할 수도 있습니다. 내장 프로시저 매개변수 작업에 대한 내용은 18-51 페이지의 "내장 프로시저 매개변수 작업"을 참조하십시오.

데이터 페치

데이터의 소스를 지정한 다음 데이터를 페치해야 애플리케이션에서 액세스할 수 있습니다. 데이터셋이 데이터를 페치하고 나면 데이터 소스를 통해 데이터셋에 연결된 data-aware 컨트롤이 자동으로 데이터 값을 표시하고 프로바이더를 통해 데이터셋에 연결된 클라이언트 데이터셋을 레코드로 채울 수 있습니다.

다른 데이터셋에서와 같이 단방향 데이터셋에 대해 데이터를 페치하는 방법은 두 가지가 있습니다.

- 디자인 타임 시 Object Inspector에서 또는 런타임 시 코드에서 *Active* 속성을 *True*로 설정합니다.

```
CustQuery.Active := True;
```

- 런타임 시 *Open* 메소드를 호출합니다.

```
CustQuery.Open;
```

서버에서 레코드를 얻는 단방향 데이터셋과 함께 *Active* 속성이나 *Open* 메소드를 사용합니다. 이러한 레코드들이 SELECT 쿼리 (*CommandType*이 *ctTable*일 때 자동으로 생성된 쿼리 포함) 또는 내장 프로시저 중 어디에서 비롯된 것인지는 중요하지 않습니다.

데이터셋 준비

서버에서 쿼리 또는 내장 프로시저를 실행하기 전에 먼저 데이터셋을 "준비"해야 합니다. 데이터셋 준비는 *dbExpress* 및 서버가 문장 및 매개변수에 대한 리소스를 할당하는 것을 의미합니다. *CommandType*이 *ctTable*인 경우, 이는 데이터셋이 SELECT 쿼리를 생성할 때입니다. 서버에 의해 바인딩되지 않은 모든 매개변수는 이 시점에서 쿼리에 포함됩니다.

단방향 데이터셋은 *Active*를 *True*로 설정하거나 *Open* 메소드를 호출할 때 자동으로 준비됩니다. 데이터셋을 닫으면 문장 실행을 위해 할당된 리소스가 해제됩니다. 쿼리나 내장 프로시저를 두 번 이상 실행하려는 경우, 데이터셋을 처음 열기 전에 명시적으로 준비하면 성능을 향상시킬 수 있습니다. 데이터셋을 명시적으로 준비하려면 *Prepared* 속성을 *True*로 설정합니다.

```
CustQuery.Prepared := True;
```

데이터셋을 명시적으로 준비하면 *Prepared*를 *False*로 설정하기 전에는 구문 실행을 위해 할당된 리소스가 해제되지 않습니다.

데이터셋이 실행되기 전에 다시 준비되는지 확인하려면 (예를 들어, 매개변수 값이나 *SortFieldNames* 속성을 변경하는 경우) *Prepared* 속성을 *False*로 설정합니다.

여러 데이터셋 페치

일부 내장 프로시저는 레코드의 여러 집합을 반환합니다. 데이터셋은 사용자가 이를 열 때 첫 번째 집합만 페치합니다. 다른 레코드 집합에 액세스하려면 *NextRecordSet* 메소드를 호출합니다.

```
var
  DataSet2: TSQLDataSet;
  nRows: Integer;
begin
  DataSet2 := SQLDataSet1.NextRecordSet(nRows);
  ...
```

*NextRecordSet*은 그 다음 레코드 집합에 대한 액세스를 제공하는 새로 생성된 *TCustomSQLDataSet* 컴포넌트를 반환합니다. 즉, *NextRecordSet*을 처음 호출할 때 두 번째 레코드 집합에 대한 데이터셋을 반환합니다. *NextRecordSet*을 호출하면 세 번째 데이터셋을 반환하며 더 이상 남은 레코드 집합이 없을 때까지 반복됩니다. 남은 데이터셋이 없으면 *NextRecordSet*은 *nil*을 반환합니다.

레코드를 반환하지 않는 명령 실행

단방향 데이터셋이 나타내는 쿼리 또는 내장 프로시저가 어떤 레코드도 반환하지 않는 경우에도 단방향 데이터셋을 사용할 수 있습니다. 이러한 명령은 SELECT 문(예를 들면 INSERT, DELETE, UPDATE, CREATE INDEX 및 ALTER TABLE 명령은 어떠한 레코드도 반환하지 않습니다.)이 아닌 DDL(Data Definition Language) 또는 DML(Data Manipulation Language) 문을 사용하는 문장을 포함합니다. 명령에 사용되는 랭귀지는 서버 특정적이지만 일반적으로 SQL 랭귀지의 SQL-92 표준과 호환됩니다.

실행하는 SQL 명령은 사용 중인 서버에서 수용 가능해야 합니다. 단방향 데이터셋은 SQL을 평가하지 않고 이를 실행하지도 않습니다. 단방향 데이터셋은 실행을 위해 서버에 명령을 전달할 뿐입니다.

참고 명령이 어떠한 레코드도 반환하지 않는 경우에는 레코드 집합에 대한 액세스를 제공하는 데이터셋 메소드가 필요 없으므로 단방향 데이터셋을 사용할 필요가 전혀 없습니다. 데이터베이스 서버에 연결하는 SQL 연결 컴포넌트는 서버에서 명령을 실행하기 위해 직접 사용될 수 있습니다. 자세한 내용은 17-10 페이지의 "서버에 명령 전송"을 참조하십시오.

실행할 명령 지정

단방향 데이터셋에서 실행할 명령을 지정하는 방법은 명령이 데이터셋의 결과이든 아니든 간에 동일합니다. 즉, 다음과 같습니다.

*TSQLDataSet*을 사용할 경우, *CommandType* 및 *CommandText* 속성을 사용하여 명령을 지정합니다.

- *CommandType*이 *ctQuery*이면 *CommandText*는 서버에 전달할 SQL 문입니다
- *CommandType*이 *ctStoredProc*이면 *CommandText*는 실행할 내장 프로시저의 이름입니다.

*TSQLQuery*를 사용할 때 *SQL* 속성을 사용하여 서버에 전달할 SQL 문을 지정합니다.

*TSQLStoredProc*를 사용할 때 *StoredProcName* 속성을 사용하여 실행할 내장 프로시저의 이름을 지정합니다.

레코드를 검색하는 것과 동일한 방법으로 명령을 지정할 때처럼 레코드를 반환하는 쿼리 및 내장 프로시저에서와 같이 쿼리 매개변수 또는 내장 프로시저 매개변수를 사용합니다. 자세한 내용은 18-45 페이지의 "쿼리의 매개변수 사용"과 18-51 페이지의 "내장 프로시저 매개변수 작업"을 참조하십시오.

명령 실행

레코드를 반환하지 않는 쿼리나 내장 프로시저를 실행하려는 경우에는 *Active* 속성이나 *Open* 메소드를 사용하지 않습니다. 그 대신 다음을 사용해야 합니다.

- 데이터셋이 *TSQLDataSet* 또는 *TSQLQuery*의 인스턴스인 경우, *ExecSQL* 메소드를 사용해야 합니다.

```
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
FixTicket.ExecSQL;
```

- 데이터셋이 *TSQLStoredProc*의 인스턴스인 경우, *ExecProc* 메소드를 사용해야 합니다.

```
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';
SQLStoredProc1.ExecProc;
```

팁 쿼리나 내장 프로시저를 여러 번 실행하는 경우에는 *Prepared* 속성을 *True*로 설정하는 것이 좋습니다.

서버 메타데이터 생성 및 수정

데이터를 반환하지 않는 대부분의 명령은 두 가지로 분류할 수 있습니다. 하나는 데이터를 편집하기 위해 사용하는 명령 (예: INSERT, DELETE 및 UPDATE 명령) 과 다른 하나는 테이블, 인덱스 및 내장 프로시저와 같은 서버의 항목을 작성하거나 수정하기 위해 사용하는 명령입니다.

편집을 위해 명시적인 SQL 명령을 사용하는 것을 원하지 않는 경우에는 단방향 데이터셋을 클라이언트 데이터셋에 연결하고 편집과 관련하여 생성된 모든 SQL 명령을 처리하게 할 수 있습니다(14-12 페이지의 "클라이언트 데이터셋을 동일한 애플리케이션의 다른 데이터셋에 연결" 참조). 사실, data-aware 컨트롤은 *TClientDataSet*과 같은 데이터셋을 통해 편집을 수행하도록 디자인되었기 때문에 이 방법을 권장합니다.

애플리케이션이 서버의 메타데이터를 생성 또는 수정할 수 있는 유일한 방법은 사실상 명령을 전송하는 것입니다. 모든 데이터베이스 드라이버가 동일한 SQL 구문을 지원하지는 않습니다. 각 데이터베이스 타입에 의해 지원되는 SQL 구문과 데이터베이스 타입 사이의 차이점에 대해서는 여기에서 설명할 수 없습니다. 특정 데이터베이스 시스템의 SQL 구현에 대한 포괄적인 최신 정보는 해당 시스템에서 제공하는 설명서를 참조하십시오.

일반적으로 데이터베이스에 테이블을 작성하기 위해서 CREATE TABLE 문을 사용하고 이들 테이블에 대한 새로운 인덱스를 작성하기 위해서 CREATE INDEX 문을 사용합니다. 지원이 가능한 경우에는 여러 메타데이터 객체 추가를 위해 CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA 및 CREATE PROCEDURE 등의 다른 CREATE 문을 사용합니다.

각 CREATE 문마다 메타데이터 객체를 삭제하는 데 이용하는 DROP 문이 있습니다. 이러한 문장에는 DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA 및 DROP PROCEDURE가 포함됩니다.

테이블의 구조를 변경하려면 ALTER TABLE 문을 사용합니다. ALTER TABLE은 테이블에 새로운 요소를 작성하고 이를 삭제할 수 있는 ADD 및 DROP 절이 있습니다. 예를 들어, 테이블에 새로운 열을 추가하려면 ADD COLUMN 절을 사용하고 테이블에서 기존의 제약 조건을 삭제하려면 DROP CONSTRAINT 절을 사용합니다.

예를 들면, 다음 문장은 InterBase 데이터베이스에 GET_EMP_PROJ라는 내장 프로시저를 생성합니다.

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

다음 코드는 *TSQLDataSet*를 사용하여 이 내장 프로시저를 생성합니다. 데이터셋이 내장 프로시저 정의 (:EMP_NO 및 :PROJ_ID)의 매개변수를 내장 프로시저를 생성하는 쿼리의 매개변수와 혼동하지 않도록 방지하기 위해 *ParamCheck* 속성을 사용한다는 점에 유의하십시오.

```
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
    'RETURNS (PROJ_ID CHAR(5)) AS ' +
    'BEGIN ' +
    'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
    'WHERE EMP_NO = :EMP_NO ' +
    'INTO :PROJ_ID ' +
    'DO SUSPEND; ' +
    'END';
  ExecSQL;
end;
```

마스터/디테일 연결 커서 설정

연결된 커서를 사용하여 단방향 데이터셋과의 마스터/디테일 관계를 디테일 집합으로 설정하는 방법에는 두 가지가 있습니다. 사용할 방법은 사용하고 있는 단방향 데이터셋의 타입에 의존합니다. 마스터/디테일 관계를 설정하고 나면 단방향 데이터셋 (일대다 관계에서 "다")은 마스터 셋 (일대다 관계에서 "일")의 현재 레코드에 해당하는 레코드에 대해서만 액세스를 제공합니다.

*TSQLDataSet*과 *TSQLQuery*는 매개변수화된 쿼리를 사용하여 마스터/디테일 관계를 설정해야 합니다. 이는 모든 쿼리 타입 데이터셋에 그러한 관계를 생성하는 기술입니다. 쿼리 타입 데이터셋과 마스터/디테일 관계 생성에 대한 내용은 18-47 페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"을 참조하십시오.

디테일 집합이 *TSQLTable*의 인스턴스인 곳에 마스터/디테일 관계를 설정하려면 다른 테이블 타입 데이터셋을 사용하는 경우처럼 *MasterSource*와 *MasterFields* 속성을 사용합니다. 테이블 타입 데이터셋과 마스터/디테일 관계 생성에 대한 자세한 내용은 18-47 페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"을 참조하십시오.

스키마 정보 액세스

서버에서 사용할 수 있는 것에 대한 정보를 얻는 방법에는 두 가지가 있습니다. 스키마 정보 또는 메타데이터라고 하는 이 정보에는 서버에서 어떤 테이블과 내장 프로시저를 이용할 수 있는지와 이러한 테이블 및 내장 프로시저에 대한 정보(예: 테이블에 있는 필드, 정의된 인덱스, 내장 프로시저가 사용하는 매개변수 등)가 포함되어 있습니다.

이 메타데이터를 얻는 가장 간단한 방법은 *TSQLConnection*의 메소드를 사용하는 것입니다. 이러한 메소드들은 기존 문자열 목록이나 목록 객체를 테이블 이름, 내장 프로시저, 필드, 인덱스 또는 매개변수 디스크립터로 채웁니다. 이 기술은 목록을 다른 데이터베이스 연결 컴포넌트에 대한 메타데이터로 채우는 방법과 동일합니다. 이러한 메소드는 17-13 페이지의 "메타데이터 얻기"에서 설명합니다.

더 자세한 스키마 정보가 필요한 경우, 단방향 데이터셋을 메타데이터로 채울 수 있습니다. 간단한 목록 대신 단방향 데이터셋은 스키마 정보로 채워지는데 각 레코드는 단일 테이블, 내장 프로시저, 인덱스, 필드 또는 매개변수를 표시합니다.

메타데이터를 단방향 데이터셋에 페치

단방향 데이터셋을 데이터베이스 서버의 메타데이터로 채우려면 먼저 *SetSchemaInfo* 메소드를 사용하여 보고자 하는 데이터를 지정해야 합니다. *SetSchemaInfo*는 다음 세 가지 매개변수를 취합니다.

- 페치하려는 스키마 정보(메타데이터)의 타입. 테이블 목록(*stTables*), 시스템 테이블 목록(*stSysTables*), 내장 프로시저 목록(*stProcedures*), 테이블의 필드 목록(*stColumns*), 인덱스 목록(*stIndexes*) 또는 내장 프로시저에서 사용되는 매개변수 목록(*stProcedureParams*)들이 여기에 해당됩니다. 각각의 정보 타입은 다른 필드 집합을 사용하여 목록의 항목을 설명합니다. 이들 데이터셋의 구조에 대한 자세한 내용은 22-14 페이지의 "메타데이터 데이터셋의 구조"를 참조하십시오.
- 필드, 인덱스 또는 내장 프로시저 매개변수에 대한 정보를 페치할 경우에 적용할 테이블 또는 내장 프로시저의 이름. 스키마 정보의 다른 타입을 페치하는 경우 이 매개변수는 nil입니다.

- 반환되는 모든 이름에 대해 일치되어야 하는 패턴. 이 패턴은 대표 문자(와일드카드) '%' (모든 길이의 임의 문자의 문자열에 일치) 및 '_' (하나의 임의 문자에 일치)를 사용하는 'Cust%'와 같은 SQL 패턴입니다. 패턴에서 퍼센트나 밑줄을 사용하려면 문자를 두 번 씩습니다(%% 또는 __). 패턴을 사용하지 않으려면 매개변수가 nil이어도 됩니다.

참고 테이블(*stTables*)에 대한 스키마 정보를 페치하려는 경우, 결과 스키마 정보는 SQL 연결의 *TableScope* 속성 값에 따라 순서 테이블, 시스템 테이블, 뷰, 동의어(synonym)를 나타낼 수 있습니다.

다음의 호출은 모든 시스템 테이블(메타데이터를 포함하는 서버 테이블)을 나열하는 테이블을 요청합니다.

```
SQLDataSet1.SetSchemaInfo(stSysTable, '', '');
```

*SetSchemaInfo*에 대한 이러한 호출 다음에 데이터셋을 열면 결과 데이터셋은 테이블 이름, 타입, 스키마 이름 등을 제공하는 열이 있는 각 테이블에 대한 레코드를 갖습니다. 서버가 MySQL과 같은 메타데이터를 저장하기 위한 시스템 테이블을 사용하지 않는 경우에는 데이터셋을 열 때 아무런 레코드도 포함되지 않습니다.

앞의 예제는 첫 번째 매개변수만 사용하였습니다. 그 대신 'MyProc'라는 이름의 내장 프로시저에 대한 입력 매개변수의 목록을 얻는 경우를 가정해 봅니다. 또한 이 내장 프로시저를 생성한 사람이 접두어를 사용하여 모든 매개변수에 이름을 지정해서 입력 매개변수인지 또는 출력 매개변수인지 나타내는 경우를 가정해 봅니다('inName', 'outValue' 등). 다음과 같이 *SetSchemaInfo*를 호출할 수 있습니다.

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, 'MyProc', 'in%');
```

결과 데이터셋은 각 매개변수의 속성을 기술하는 열이 있는 입력 매개변수 테이블입니다.

메타데이터에 대한 데이터셋 사용한 후 데이터 페치

SetSchemaInfo 호출 이후에 데이터셋의 쿼리나 내장 프로시저 실행으로 반환되는 방법은 두 가지입니다.

- 데이터를 페치하려는 쿼리, 테이블 또는 내장 프로시저를 지정하여 *CommandText* 속성을 변경합니다.
- 첫 번째 매개변수를 *stNoSchema*로 설정하여 *SetSchemaInfo*를 호출합니다. 이 경우에 데이터셋은 *CommandText*의 현재값에 의해 지정된 데이터 페치로 되돌아갑니다.

메타데이터 데이터셋의 구조

*TSQLDataSet*을 사용하여 액세스할 수 있는 각각의 메타데이터 타입에는 요청된 타입의 항목에 대한 정보로 채워진 이미 정의된 집합의 열(필드)이 있습니다.

테이블에 대한 정보

테이블(*stTables* 또는 *stSysTables*)에 대한 정보를 요청하면 결과 데이터셋은 각 테이블에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

표 22.1 테이블을 나열하는 메타데이터의 테이블에 있는 열

열 이름	필드 타입	내용
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호.
CATALOG_NAME	ftString	테이블을 포함하는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 매개변수와 동일합니다.
SCHEMA_NAME	ftString	테이블의 소유자를 식별하는 스키마 이름.
TABLE_NAME	ftString	테이블 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.
TABLE_TYPE	ftInteger	테이블의 타입을 식별합니다. 이 값은 다음 값들 중 하나 이상의 합입니다. 1: Table 2: View 4: System table 8: Synonym 16: Temporary table 32: Local table

내장 프로시저에 대한 정보

내장 프로시저(*stProcedures*)에 대한 정보를 요청하면 결과 데이터셋은 각 내장 프로시저에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

표 22.2 내장 프로시저를 나열하는 메타데이터의 테이블에 있는 열

열 이름	필드 타입	내용
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호.
CATALOG_NAME	ftString	내장 프로시저를 포함하는 카탈로그(데이터베이스) 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 매개변수와 동일합니다.
SCHEMA_NAME	ftString	내장 프로시저의 소유자를 식별하는 스키마 이름.
PROC_NAME	ftString	내장 프로시저 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.
PROC_TYPE	ftInteger	내장 프로시저의 타입을 식별합니다. 이 값은 다음 값들 중 하나 이상의 합입니다. 1: Procedure 2: Function 4: Package 8: System procedure
IN_PARAMS	ftSmallint	입력 매개변수의 수.
OUT_PARAMS	ftSmallint	출력 매개변수의 수.

필드에 대한 정보

특정 테이블(*stColumns*)의 필드에 대한 정보를 요청하면 결과 데이터셋은 각 필드에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

표 22.3 필드를 나열하는 메타데이터의 테이블에 있는 열

열 이름	필드 타입	내용
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호.
CATALOG_NAME	ftString	나열하는 필드의 테이블을 포함하는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 매개변수와 동일합니다.
SCHEMA_NAME	ftString	필드의 소유자를 식별하는 스키마의 이름.
TABLE_NAME	ftString	필드를 포함하는 테이블의 이름.
COLUMN_NAME	ftString	필드 이름의 값은 데이터셋의 정렬 순서를 결정합니다.
COLUMN_POSITION	ftSmallint	테이블에서 열의 위치.
COLUMN_TYPE	ftInteger	필드 값의 타입을 식별합니다. 이 값은 다음 중 하나 이상의 합입니다. 1: Row ID 2: Row Version 4: Auto increment field 8: Field with a default value
COLUMN_DATATYPE	ftSmallint	열의 데이터 타입. 이것은 <i>sqlinks.pas</i> 에 정의된 논리 필드 타입 상수들 중의 하나입니다.
COLUMN_TYPPENAME	ftString	데이터 타입을 기술하는 문자열. 이것은 <i>COLUMN_DATATYPE</i> 과 <i>COLUMN_SUBTYPE</i> 에 포함된 일부 DDL 문에 사용된 형식의 정보와 동일한 정보입니다.
COLUMN_SUBTYPE	ftSmallint	열의 데이터 타입에 대한 하위 타입. 이것은 <i>sqlinks.pas</i> 에 정의되어 있는 논리 하위 타입 상수 중 하나입니다.
COLUMN_PRECISION	ftInteger	필드 타입의 크기 (문자열의 문자 수, 바이트 필드의 바이트, BCD 값의 유효 숫자, ADT 필드의 멤버 등).
COLUMN_SCALE	ftSmallint	BCD 값의 소수점 오른쪽의 숫자 수 또는 ADT 및 배열 필드의 자손 수.
COLUMN_LENGTH	ftInteger	필드 값 저장에 필요한 바이트 수
COLUMN_NULLABLE	ftSmallint	필드를 공백으로 둘 수 있는지 여부를 나타내는 부울(0은 필드에 값이 필요함을 의미).

인덱스에 대한 정보

테이블(*stIndexes*)의 인덱스에 대한 정보를 요청하면 결과 데이터셋은 각 레코드별로 각 필드 레코드를 포함합니다. (다중 레코드 인덱스는 다중 레코드를 사용하여 기술됩니다.) 데이터셋은 다음과 같은 열을 가집니다.

표 22.4 인덱스를 나열하는 메타데이터의 테이블에 있는 열

열 이름	필드 타입	내용
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호.
CATALOG_NAME	ftString	인덱스를 포함하는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 매개변수와 동일합니다.
SCHEMA_NAME	ftString	인덱스 소유자를 식별하는 스키마의 이름.
TABLE_NAME	ftString	인덱스가 정의된 테이블의 이름.
INDEX_NAME	ftString	인덱스 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.
PKEY_NAME	ftString	기본 키의 이름을 나타냅니다.
COLUMN_NAME	ftString	인덱스의 필드(열) 이름.
COLUMN_POSITION	ftSmallint	인덱스의 이 필드의 위치.
INDEX_TYPE	ftSmallint	인덱스의 타입을 식별합니다. 이 값은 다음 값들 중 하나 이상의 합입니다. 1: Non-unique 2: Unique 4: Primary key
SORT_ORDER	ftString	인덱스가 오름차순(a)인지 내림차순(d)인지 나타냅니다.
FILTER	ftString	인덱스된 레코드를 제한하는 필터 조건을 기술합니다.

내장 프로시저 매개변수에 대한 정보

내장 프로시저(*stProcedureParams*)의 매개변수에 대한 정보를 요청하면 결과 데이터셋은 각 매개변수에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

표 22.5 매개변수를 나열하는 메타데이터의 테이블에 있는 열

열 이름	필드 타입	내용
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호.
CATALOG_NAME	ftString	내장 프로시저를 포함하는 카탈로그(데이터베이스) 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 매개변수와 동일합니다.
SCHEMA_NAME	ftString	내장 프로시저의 소유자를 식별하는 스키마 이름.
PROC_NAME	ftString	매개변수를 포함하는 내장 프로시저의 이름.
PARAM_NAME	ftString	매개변수 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.
PARAM_TYPE	ftSmallint	매개변수의 타입을 식별합니다. 이것은 <i>TParam</i> 객체의 <i>ParamType</i> 속성과 동일합니다.

표 22.5 매개변수를 나열하는 메타데이터의 테이블에 있는 열

열 이름	필드 타입	내용
PARAM_DATATYPE	ftSmallint	매개변수의 데이터 타입. 이것은 sqllinks.pas에 정의된 논리 필드 타입 상수들 중의 하나입니다.
PARAM_SUBTYPE	ftSmallint	매개변수의 데이터 타입에 대한 하위 타입. 이것은 sqllinks.pas에 정의되어 있는 논리 하위 타입 상수 중 하나입니다.
PARAM_TYPENAME	ftString	데이터 타입을 기술하는 문자열. 이것은 PARAM_DATATYPE 및 PARAM_SUBTYPE에 포함되어 있지만 일부 DDL 문에 사용되는 형식으로 된 동일한 정보입니다.
PARAM_PRECISION	ftInteger	부동 소수점 값의 최대 수 또는 최대 바이트 수(문자열 및 Bytes 필드).
PARAM_SCALE	ftSmallint	부동 소수점 값의 소수점 오른쪽의 숫자 수.
PARAM_LENGTH	ftInteger	매개변수 값 저장에 필요한 바이트 수.
PARAM_NULLABLE	ftSmallint	매개변수를 공백으로 둘 수 있는지 여부를 나타내는 부울(0은 매개변수에 값이 필요함을 의미).

dbExpress 애플리케이션 디버깅

데이터베이스 애플리케이션을 디버깅하는 동안 사용자를 위해 자동으로(예를 들면, 프로바이더 컴포넌트나 *dbExpress* 드라이버에 의해) 생성되는 메시지를 비롯하여 연결 컴포넌트를 통해 데이터베이스 서버와 주고 받는 SQL 메시지를 모니터링하는 것이 좋습니다.

SQL 명령을 모니터링하기 위한 TSQLMonitor 사용

*TSQLConnection*은 짝을 이루는 컴포넌트인 *TSQLMonitor*를 사용하여 이 메시지들을 인터셉트하고 이를 문자열 목록에 저장합니다. *TSQLMonitor*는 *dbExpress*에 의해 관리되는 모든 명령이 아닌 단일 *TSQLConnection* 컴포넌트에 관련된 명령만 모니터링한다는 점을 제외하면 BDE와 함께 사용할 수 있는 SQL 모니터 유틸리티와 매우 유사하게 동작합니다.

다음과 같은 방법으로 *TSQLMonitor*를 사용합니다.

- 1 모니터하려는 SQL 명령이 있는 *TSQLConnection* 컴포넌트를 갖고 있는 폼이나 데이터 모듈에 *TSQLMonitor* 컴포넌트를 추가합니다.
- 2 *SQLConnection* 속성을 *TSQLConnection* 컴포넌트로 설정합니다.
- 3 SQL 모니터의 *Active* 속성을 *True*로 설정합니다.

SQL 명령이 서버에 전송될 때 SQL 모니터의 *TraceList* 속성은 인터셉트되는 모든 SQL 명령을 나열하도록 자동으로 업데이트됩니다.

FileName 속성에 대한 값을 지정한 다음 *AutoSave* 속성을 *True*로 설정하여 이 목록을 파일로 저장할 수 있습니다. *AutoSave*는 SQL 모니터가 새로운 메시지가 기록될 때마다 *TraceList* 속성의 내용을 파일로 저장하게 합니다.

메시지가 기록될 때마다 파일을 저장해야 하는 수고를 하지 않으려면 *OnLogTrace* 이벤트 핸들러를 사용하여 일정 수의 메시지가 기록된 후에만 파일로 저장하도록 할 수 있습니다. 예를 들면, 다음과 같은 이벤트 핸들러는 10번째 메시지마다 *TraceList*의 내용을 저장한 다음 저장한 다음 로그를 지워서 목록이 너무 커지지 않게 합니다.

```
procedure TForm1.SQLMonitor1LogTrace(Sender:TObject; CBInfo:Pointer);
var
  LogFileName:string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
    begin
      LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
      Tag := Tag + 1; {ensure next log file has a different name }
      SaveToFile(LogFileName);
      TraceList.Clear; { clear list }
    end;
  end;
end;
```

참고 앞의 이벤트 핸들러를 사용했다면 애플리케이션이 종료될 때 항목이 10 이하인 부분 목록을 저장할 수도 있습니다.

SQL 명령을 모니터하기 위한 콜백 사용

*TSQLMonitor*를 사용하는 대신 SQL 연결 컴포넌트의 *SetTraceCallbackEvent* 메소드를 사용하여 애플리케이션이 SQL 명령을 추적하는 방법을 사용자 지정할 수 있습니다. *SetTraceCallbackEvent*는 *TSQLCallbackEvent* 타입의 콜백 및 콜백 함수로 전달되는 사용자 정의 값, 두 개의 매개변수를 취합니다.

콜백 함수는 *CallType* 및 *CBInfo*의 두 개의 매개변수를 취합니다.

- *CallType*은 나중에 사용하기 위해 저장해 둡니다.
- *CBInfo*는 범주(*CallType*과 같음), SQL 명령의 텍스트 및 *SetTraceCallbackEvent* 메소드로 전달되는 사용자 정의 값을 포함하는 레코드에 대한 포인터입니다.

콜백은 일반적으로 *cbrUSEDEF*인 *CBRTyp*e 타입의 값을 반환합니다.

dbExpress 드라이버는 SQL 연결 컴포넌트가 서버에 명령을 전달하거나 서버가 오류 메시지를 반환할 때마다 사용자의 콜백을 호출합니다.

경고 *TSQLConnection* 객체에 연결된 *TSQLMonitor* 컴포넌트가 있을 경우에는 *SetTraceCallbackEvent*를 호출하지 마십시오. *TSQLMonitor*는 작업을 위해 콜백 메커니즘을 사용하며 *TSQLConnection*은 한 번에 하나의 콜백만 지원할 수 있습니다.

23

클라이언트 데이터셋 사용

클라이언트 데이터셋은 모든 데이터를 메모리에 저장하는 특수 데이터셋입니다. 메모리에 저장한 데이터 처리는 midaslib.dcu 또는 midas.dll에서 지원됩니다. 데이터를 저장하는 데 사용하는 형식 클라이언트 데이터셋은 독립적이며 쉽게 전송되므로 클라이언트 데이터셋은 다음을 수행할 수 있습니다.

- 파일 기반 데이터셋처럼 디스크의 전용 파일을 읽고 씁니다. 이 메커니즘을 지원하는 속성 및 메소드는 23-33 페이지의 "파일 기반 데이터로 클라이언트 데이터셋 사용"에서 설명합니다.
- 데이터베이스 서버의 데이터 업데이트를 캐시합니다. 캐시된 업데이트를 지원하는 클라이언트 데이터셋 기능에 대해서는 23-16 페이지의 "클라이언트 데이터셋을 사용하여 업데이트 캐시"에서 설명합니다.
- 다계층 애플리케이션의 클라이언트 부분에 있는 데이터를 표시합니다. 이런 방식으로 작동하려면 클라이언트 데이터셋은 23-25 페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"에 설명된 바와 같이 외부 프로바이더를 사용해야 합니다. 다계층 데이터베이스 애플리케이션에 대한 자세한 내용은 25장 "다계층 (multi-tiered) 애플리케이션 생성"을 참조하십시오.
- 데이터셋이 아닌 다른 소스의 데이터를 표시합니다. 클라이언트 데이터셋은 외부 프로바이더의 데이터를 사용할 수 있으므로 전문적인 프로바이더는 다양한 정보 소스를 활용하여 클라이언트 데이터셋을 사용할 수 있습니다. 예를 들어, XML 프로바이더를 사용하여 클라이언트 데이터셋이 XML 문서의 정보를 표시하도록 할 수 있습니다.

파일 기반 데이터용 클라이언트 데이터셋, 업데이트 캐시, 외부 프로바이더의 데이터(예: XML 문서나 다계층 애플리케이션을 사용하는 경우) 또는 "브리프케이스 (briefcase) 모델" 애플리케이션과 같은 이러한 방식의 조합을 사용하는지 여부에 관계 없이 데이터 작업을 지원하는 클라이언트 데이터셋의 광범위한 기능을 사용할 수 있습니다.

클라이언트 데이터셋을 사용하는 데이터 작업

데이터셋과 마찬가지로 클라이언트 데이터셋은 데이터 소스 컴포넌트를 사용하여 data-aware 컨트롤에 데이터를 제공할 수 있습니다. data-aware 컨트롤에 데이터베이스 정보를 표시하는 방법에 대한 내용은 15장 "데이터 컨트롤 사용"을 참조하십시오.

클라이언트 데이터셋은 *TDataSet*에서 상속된 메소드와 모든 속성을 구현합니다. 이러한 일반적인 데이터셋 행동에 대한 전체적인 내용은 18장 "데이터셋 이해"를 참조하십시오.

또한 클라이언트 데이터셋은 다음과 같이 테이블 타입 데이터셋에 공통적인 많은 기능들을 구현합니다.

- 인덱스로 레코드 정렬
- 인덱스를 사용하여 레코드 검색
- 범위로 레코드 제한
- 마스터/디테일 관계 생성
- 읽기/쓰기 액세스 제어
- 원본으로 사용한 데이터셋 작성
- 데이터셋 비우기
- 클라이언트 데이터셋 동기화

이러한 기능에 대한 자세한 내용은 18-25 페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오.

클라이언트 데이터셋은 모든 데이터를 메모리에 보관한다는 점에서 다른 데이터셋과 다릅니다. 이러한 이유에서 공통적인 데이터베이스 기능을 지원하기 위해 추가적인 기능이나 고려 사항이 필요할 수도 있습니다. 이 장에서는 클라이언트 데이터셋이 가진 공통적인 일부 기능과 차이점들을 설명합니다.

클라이언트 데이터셋에서 데이터 탐색

애플리케이션에서 표준 data-aware 컨트롤을 사용한다면 사용자는 이러한 컨트롤의 기본 제공 행동을 사용하여 클라이언트 데이터셋의 레코드를 탐색할 수 있습니다. 또한 프로그램에서 *First*, *Last*, *Next*, *Prior* 같은 표준 데이터셋 메소드를 사용하여 레코드를 탐색할 수 있습니다. 이 메소드에 대한 자세한 내용은 18-5 페이지의 "데이터셋 탐색"을 참조하십시오.

대부분의 데이터셋과 달리 클라이언트 데이터셋은 *RecNo* 속성을 사용하여 데이터셋의 특정 레코드에 커서를 둘 수도 있습니다. 일반적으로 애플리케이션은 *RecNo*를 사용하여 현재 레코드의 레코드 번호를 결정합니다. 그러나 클라이언트 데이터셋은 *RecNo*를 특정 레코드 번호로 설정하여 해당 레코드를 현재 레코드로 만들 수 있습니다.

레코드 표시 제한

사용 가능한 데이터의 서브셋을 사용하도록 하는 임시 제한을 두기 위한 방법으로 범위와 필터를 사용할 수 있습니다. 범위나 필터를 사용하면 클라이언트 데이터셋은 인메모리 캐시에 있는 모든 데이터를 보여 주지 않고 범위나 필터 조건에 맞는 데이터만을 표시합니다. 필터 사용에 대한 자세한 내용은 18-12 페이지의 "필터를 사용한 데이터 서브셋 표시 및 편집"을 참조하십시오. 범위에 대한 자세한 내용은 18-30 페이지의 "범위로 레코드 제한"을 참조하십시오.

대부분의 데이터셋에서 필터 문자열은 데이터베이스 서버에서 구현되는 SQL 명령으로 구문 분석됩니다. 따라서 서버의 SQL 언어는 필터 문자열에서 사용되는 연산을 제한합니다. 클라이언트 데이터셋은 다른 데이터셋보다 더 많은 연산을 포함하는 고유의 필터 지원을 구현합니다. 예를 들어, 클라이언트 데이터셋을 사용하면 필터 표현식에는 부분 문자열을 반환하는 문자열 연산자, 날짜/시간 값을 구문 분석하는 연산자 등이 포함될 수 있습니다. 또한 클라이언트 데이터셋은 BLOB 필드 또는 ADT 필드와 배열 필드 같이 복잡한 필드 타입에 필터를 허용합니다.

필터를 지원하는 다른 데이터셋에 대한 비교와 함께 클라이언트 데이터셋이 필터에서 사용할 수 있는 다양한 연산자 및 함수는 다음과 같습니다.

표 23.1 클라이언트 데이터셋에서의 필터 지원

연산자 또는 함수	예	다른 데이터셋의 지원	설명
비교			
=	State = 'CA'	예	
<>	State <> 'CA'	예	
>=	DateEntered >= '1/1/1998'	예	
<=	Total <= 100,000	예	
>	Percentile > 50	예	
<	Field1 < Field2	예	
BLANK	State <> 'CA' or State = BLANK	예	빈 레코드는 필터에 명시적으로 포함되지 않으면 표시되지 않습니다.
IS NULL	Field1 IS NULL	아니오	
IS NOT NULL	Field1 IS NOT NULL	아니오	
논리 연산자			
and	State = 'CA' and Country = 'US'	예	
or	State = 'CA' or State = 'MA'	예	
not	not (State = 'CA')	예	
산술 연산자			
+	Total + 5 > 100	드라이버에 따라 다름	숫자, 문자열 또는 날짜(시간) + 숫자에 적용됩니다.

표 23.1 클라이언트 데이터셋에서의 필터 지원 (계속)

연산자 또는 함수	예	다른 데이터 셋의 지원	설명
-	Field1 - 7 <> 10	드라이버에 따라 다름	숫자, 날짜 또는 날짜(시간) - 숫자에 적용됩니다.
*	Discount * 100 > 20	드라이버에 따라 다름	숫자에만 적용됩니다.
/	Discount > Total / 5	드라이버에 따라 다름	숫자에만 적용됩니다.
문자열 함수			
Upper	Upper(Field1) = 'ALWAYS'	아니오	
Lower	Lower(Field1 + Field2) = 'josp'	아니오	
Substring	Substring(DateFld,8) = '1998' Substring(DateFld,1,3) = 'JAN'	아니오	값은 두 번째 인수의 위치에서 끝으로 이동하거나 세 번째 인수의 문자 수만큼 이동합니다. 첫 번째 문자의 위치는 1입니다.
Trim	Trim(Field1 + Field2) Trim(Field1, '-')	아니오	앞과 뒤에서 세 번째 인수를 제거합니다. 세 번째 인수가 없으면 공백을 지웁니다.
TrimLeft	TrimLeft(StringField) TrimLeft(Field1, '\$') <> "	아니오	Trim 참조.
TrimRight	TrimRight(StringField) TrimRight(Field1, '.') <> "	아니오	Trim 참조.
DateTime 함수			
Year	Year(DateField) = 2000	아니오	
Month	Month(DateField) <> 12	아니오	
Day	Day(DateField) = 1	아니오	
Hour	Hour(DateField) < 16	아니오	
Minute	Minute(DateField) = 0	아니오	
Second	Second(DateField) = 30	아니오	
GetDate	GetDate - DateField > 7	아니오	현재 날짜와 시간을 나타냅니다.
Date	DateField = Date(GetDate)	아니오	datetime 값의 날짜 부분을 반환합니다.
Time	TimeField > Time(GetDate)	아니오	datetime 값의 시간 부분을 반환합니다.
기타			

표 23.1 클라이언트 데이터셋에서의 필터 지원 (계속)

연산자 또는 함수	예	다른 데이터셋의 지원	설명
Like	Memo LIKE '%filters%'	아니오	ESC 절이 없는 SQL-92 처럼 작동합니다. BLOB 필드에 적용할 경우 FilterOptions는 대소문자가 고려되었는지 확인합니다.
In	Day(DateField) in (1,7)	아니오	SQL-92처럼 작동합니다. 두 번째 인수는 타입이 같은 모든 값의 목록입니다.
*	State = 'M*'	예	부분 비교를 위한 대표 문자(와일드카드)입니다.

범위나 필터를 적용할 때에도 클라이언트 데이터셋은 모든 레코드를 메모리에 저장합니다. 범위나 필터는 클라이언트 데이터셋의 데이터를 탐색하거나 표시하는 컨트롤에 사용할 수 있는 레코드를 결정하기만 합니다.

참고 프로바이더에서 데이터를 페치(fetch)하면 클라이언트 데이터셋이 프로바이더에 매개 변수를 제공하여 저장하는 데이터를 제한할 수도 있습니다. 자세한 내용은 23-29 페이지의 "매개변수로 레코드 제한"을 참조하십시오.

데이터 편집

클라이언트 데이터셋은 자체 데이터를 인메모리 데이터 패킷으로 나타냅니다. 이 패킷은 클라이언트 데이터셋의 *Data* 속성의 값입니다. 그러나 기본적으로 편집 내용은 *Data* 속성에 저장되지 않습니다. 대신 사용자나 프로그램에 의한 추가, 삭제, 수정 내용은 *Delta* 속성으로 나타내는 내부 변경 로그에 저장됩니다. 변경 로그는 다음과 같이 두 가지 용도로 사용됩니다.

- 데이터베이스 서버나 외부 프로바이더 컴포넌트에 업데이트를 적용하려면 변경 로그가 필요합니다.
- 변경 로그는 변경 내용을 실행 취소할 수 있는 방법을 지원합니다.

LogChanges 속성을 사용하면 로그를 사용 불가능하게 만들 수 있습니다. *LogChanges*가 *True*인 경우, 로그에 변경 내용이 기록됩니다. *LogChanges*가 *False*인 경우는 *Data* 속성으로 직접 변경이 이루어집니다. 실행 취소 지원이 필요하지 않은 경우는 파일 기반 애플리케이션에서 변경 로그를 사용할 수 없게 만들 수 있습니다.

변경 로그의 편집 내용은 애플리케이션에서 삭제하기 전까지 계속 변경 로그에 남아 있습니다. 애플리케이션에서는 다음의 경우 편집 내용을 삭제합니다.

- 변경 취소 시
- 변경 저장 시

참고 클라이언트 데이터셋을 파일에 저장하면 편집 내용은 변경 로그에서 삭제되지 않습니다. 데이터셋을 다시 로드할 때 *Data* 속성과 *Delta* 속성은 데이터가 저장되었을 당시와 동일합니다.

변경 취소

레코드의 원래 버전이 *Data*에 바뀌지 않고 남아 있더라도 사용자가 레코드를 편집하거나 다른 데이터로 이동했다가 다시 돌아올 때마다 사용자는 항상 최근에 변경된 버전의 레코드를 보게 됩니다. 사용자나 애플리케이션에서 레코드를 자주 편집하는 경우 레코드의 변경된 각 버전은 개별 항목으로 변경 로그에 저장됩니다.

레코드에 대한 각 변경을 저장하면 레코드의 이전 상태를 복구해야 할 경우 여러 단계의 실행 취소 작업을 지원할 수 있게 됩니다.

- 레코드의 마지막 변경을 삭제하려면 *UndoLastChange*를 호출하십시오. *UndoLastChange*에는 부울 매개변수로 *FollowChange*가 필요하며, 이는 커서의 위치를 복구된 레코드에 다시 돌지 (*True*) 아니면 현재 레코드에 그냥 남겨 돌지 (*False*) 여부를 나타냅니다. 하나의 레코드를 여러 번 변경한 경우 *UndoLastChange*를 호출할 때마다 단계적으로 변경 내용이 삭제됩니다. *UndoLastChange*는 성공이나 실패를 나타내는 부울 값을 반환합니다. 삭제가 발생하면 *UndoLastChange*는 *True*를 반환합니다. *ChangeCount* 속성을 사용하여 실행 취소할 변경이 더 있는지 확인할 수 있습니다. *ChangeCount*는 변경 로그에 저장된 변경 횟수를 의미합니다.
- 하나의 레코드의 각 변경 내용을 삭제하는 대신 한 번에 모두 삭제할 수 있습니다. 레코드의 모든 변경 내용을 삭제하려면 레코드를 선택하고 *RevertRecord*를 호출하십시오. *RevertRecord*는 변경 로그에서 현재 레코드에 대한 모든 변경을 삭제합니다.
- 삭제된 레코드를 복구하려면 먼저 *StatusFilter* 속성을 [*usDeleted*]로 설정하여 삭제된 레코드를 "표시"합니다. 그런 다음 복구할 레코드로 이동하고 *RevertRecord*를 호출합니다. 마지막으로 *StatusFilter* 속성을 [*usModified*, *usInserted*, *usUnmodified*]로 복구하여 복구된 레코드를 포함하는 데이터셋의 편집된 버전이 다시 표시되도록 합니다.
- 편집 도중 언제든지 *SavePoint* 속성을 사용하여 변경 로그의 현재 상태를 저장할 수 있습니다. *SavePoint*를 읽으면 마커를 변경 로그의 현재 위치로 반환합니다. 나중에 저장 위치(save point)를 읽은 후 발생한 모든 변경 내용을 취소하려면 *SavePoint*를 이전에 읽은 값으로 설정하십시오. 애플리케이션에서는 여러 저장 위치에 대한 값을 얻을 수 있습니다. 그러나 일단 변경 로그를 저장 위치로 백업하면 이후에 애플리케이션에서 읽은 모든 저장 위치의 값은 유효하지 않습니다.
- *CancelUpdates*를 호출하면 변경 로그에 기록된 모든 변경을 취소할 수 있습니다. *CancelUpdates*는 모든 레코드에 대한 모든 편집 내용을 버림으로써 변경 로그를 지웁니다. *CancelUpdates*를 호출할 때는 주의를 기울여야 합니다. *CancelUpdates*를 호출하고 나면 로그에 있었던 어떠한 변경 내용도 복구할 수 없습니다.

변경 내용 저장

클라이언트 데이터셋이 데이터를 파일에 저장하는지 또는 프로바이더를 통해 얻은 데이터를 나타내는지 여부에 따라 클라이언트 데이터셋은 변경 로그의 변경 내용을 통합하는 데 다른 메커니즘을 사용합니다. 어떤 메커니즘을 사용하는지 상관 없이 모든 업데이트 내용이 통합되면 변경 로그는 자동으로 비워집니다.

파일 기반 애플리케이션은 *Data* 속성에 의해 나타나는 로컬 캐시로 변경 내용을 쉽게 합병할 수 있습니다. 또한 다른 사용자에 의해 이루어진 변경 내용으로 로컬 편집을 해석하는 데 신경 쓸 필요가 없습니다. 변경 로그를 *Data* 속성으로 합병하려면 *MergeChangeLog* 메소드를 호출하십시오. 23-35 페이지의 "데이터에 변경 내용 병합"에서 이 프로세스에 대해 설명합니다.

클라이언트 데이터셋을 사용하여 업데이트를 캐시하거나 외부 프로바이더 컴포넌트의 데이터를 나타내는 경우에는 *MergeChangeLog*를 사용할 수 없습니다. 변경 로그의 정보는 데이터베이스나 소스 데이터셋에 저장된 데이터로 업데이트된 레코드를 해결하는 데 필요합니다. 그 대신 데이터베이스 서버나 소스 데이터셋에 수정 사항을 쓰려고 하는 *ApplyUpdates*를 호출하고, 수정 사항이 성공적으로 커밋된 경우에만 *Data* 속성을 업데이트합니다. 이 프로세스에 대한 자세한 내용은 23-21 페이지의 "업데이트 적용"을 참조하십시오.

데이터 값 제약 조건

클라이언트 데이터셋은 사용자가 데이터를 편집한 내용에 제약 조건을 적용할 수 있습니다. 이러한 제약 조건은 사용자가 변경 로그에 변경 내용을 포스트하려고 할 때 적용됩니다. 사용자는 항상 사용자 지정 제약 조건을 제공할 수 있습니다. 이렇게 하면 클라이언트 데이터셋에 포스트한 값에 애플리케이션에서 정의된 제한을 사용자 임의로 제공할 수 있습니다.

또한 클라이언트 데이터셋이 BDE를 사용하여 액세스한 서버 데이터를 나타낼 때는 데이터베이스 서버에서 import한 데이터 제약 조건을 적용하기도 합니다. 클라이언트 데이터셋이 외부 프로바이더 컴포넌트를 사용하면 프로바이더는 해당 제약 조건이 클라이언트 데이터셋으로의 전달 여부를 제어할 수 있으며, 클라이언트 데이터셋은 그 제약 조건의 사용 여부를 제어할 수 있습니다. 데이터 패킷에 제약 조건의 포함 여부를 프로바이더가 제어하는 방법에 대한 자세한 내용은 24-13 페이지의 "서버 제약 조건 처리"를 참조하십시오. 클라이언트 데이터셋이 서버 제약 조건 적용을 해제할 수 있는 방법과 이유에 대한 자세한 내용은 23-30 페이지의 "서버에서 제약 조건 처리"를 참조하십시오.

사용자 지정 제약 조건의 지정

클라이언트 데이터셋의 필드 컴포넌트 속성을 사용하여 사용자가 입력할 수 있는 데이터에 사용자가 만든 제약 조건을 부과할 수 있습니다. 각 필드 컴포넌트에는 다음과 같이 제약 조건을 지정하는 데 사용할 수 있는 두 가지 속성이 있습니다.

- *DefaultExpression* 속성은 사용자가 값을 입력하지 않은 경우에 필드에 할당되는 기본값을 정의합니다. 데이터베이스 서버나 소스 데이터셋이 필드에 기본 표현식을 할당하면 데이터베이스 서버나 소스 데이터셋에 다시 업데이트가 적용되기 전에 기본 표현식이 할당되기 때문에 클라이언트 데이터셋의 버전이 우선 적용됩니다.
- *CustomConstraint* 속성을 사용하면 필드 값을 포스트하기 전에 만족해야 하는 제약 조건을 할당할 수 있습니다. 이런 방식으로 정의된 사용자 지정 제약 조건은 서버에서 import한 모든 제약 조건과 함께 적용됩니다. 필드 컴포넌트의 사용자 지정 제약 조건 사용에 대한 자세한 내용은 19-21 페이지의 "사용자 지정 제약 조건 만들기"를 참조하십시오.

또한 클라이언트 데이터셋의 *Constraints* 속성을 사용하여 레코드 수준의 제약 조건을 만들 수 있습니다. *Constraints*는 *TCheckConstraint* 객체의 컬렉션으로 각 객체는 각각의 조건을 나타냅니다. *TCheckConstraint* 객체의 *CustomConstraint* 속성을 사용하여 레코드를 포스트할 때 검사하는 사용자가 만든 제약 조건을 추가하십시오.

정렬과 인덱싱

인덱스를 사용하면 여러 가지 장점이 있습니다.

- 클라이언트 데이터셋에서 데이터를 빨리 찾을 수 있습니다.
- 사용 가능 레코드를 제한하는 범위를 적용할 수 있습니다.
- 사용 중인 애플리케이션이 조회 테이블이나 마스터/디테일 폼과 같은 다른 데이터셋과의 관계를 설정할 수 있도록 합니다.
- 레코드가 나타나는 순서를 지정합니다.

클라이언트 데이터셋이 서버 데이터를 나타내거나 외부 프로바이더를 사용하면 기본 인덱스 및 받은 데이터를 기반으로 하는 정렬 순서를 상속합니다. 기본 인덱스를 `DEFAULT_ORDER`라고 합니다. 이 순서를 사용할 수는 있지만 인덱스를 변경하거나 삭제할 수 없습니다.

기본 인덱스 이외에도 클라이언트 데이터셋은 변경 로그(*Delta* 속성)에 저장된 변경된 레코드에서 `CHANGEINDEX`라는 보조 인덱스를 유지합니다. `CHANGEINDEX`는 클라이언트 데이터셋의 모든 레코드를 *Delta*에 지정된 변경 내용이 적용되었을 경우 나타나는 대로 정렬합니다. `CHANGEINDEX`는 `DEFAULT_ORDER`에서 상속된 순서에 기반합니다. `DEFAULT_ORDER`에서 `CHANGEINDEX` 인덱스를 변경하거나 삭제할 수 없습니다.

기존의 다른 인덱스를 사용할 수 있으며 사용자의 고유한 인덱스를 만들 수도 있습니다. 다음 단원에서 클라이언트 데이터셋에서 인덱스를 만들고 사용하는 방법을 설명합니다.

참고 또한 클라이언트 데이터셋에도 적용되는 테이블 타입 데이터셋의 인덱스에서 자료를 볼 수도 있습니다. 이 자료는 18-26 페이지의 "인덱스로 레코드 정렬" 및 18-30 페이지의 "범위로 레코드 제한"에 있습니다.

새 인덱스 추가

클라이언트 데이터셋에 인덱스를 추가하는 방법에는 세 가지가 있습니다.

- 클라이언트 데이터셋에 레코드를 정렬하는 임시 인덱스를 런타임에 만들기 위해 *IndexFieldNames* 속성을 사용할 수 있습니다. 세미콜론으로 구분하여 필드 이름을 지정합니다. 목록에 있는 필드 이름 순서대로 인덱스에서의 순서가 결정됩니다.

이 메소드는 인덱스를 추가하는 가장 약한 메소드입니다. 내림차순 또는 대소문자 구별 인덱스를 지정할 수 없으며 결과 인덱스에서 그룹화를 지원하지 않습니다. 이 인덱스들은 데이터셋을 닫으면 영구적으로 남지 않으며 클라이언트 데이터셋을 파일에 저장할 때 저장되지 않습니다.

- 런타임에 그룹화를 위해 사용될 수 있는 인덱스를 만들려면 *AddIndex*를 호출하십시오. *AddIndex*를 사용하면 다음을 포함하여 인덱스의 속성을 지정할 수 있습니다.
 - 인덱스 이름. 런타임 시 인덱스 전환을 위해 사용할 수 있습니다.
 - 인덱스를 구성하는 필드. 인덱스는 이 필드를 사용하여 레코드를 정렬하고 이 필드에서 특정 값을 가진 레코드를 찾습니다.
 - 인덱스에서 레코드를 정렬하는 방법. 기본적으로 인덱스는 컴퓨터의 로케일에 따라 오름차순으로 정렬됩니다. 기본 정렬 순서는 대소문자가 구별됩니다. 전체 인덱스를 대소문자로 구별할지, 내림차순으로 정렬할지 옵션을 설정할 수 있습니다. 또는 대소문자를 구별해서 정렬할 필드의 목록과 내림차순으로 정렬할 필드의 목록을 제공할 수 있습니다.
 - 인덱스에 대한 기본 수준의 그룹화 지원.

*AddIndex*로 만든 인덱스는 클라이언트 데이터셋이 닫히면 영구적으로 남지 않습니다. 즉, 클라이언트 데이터셋을 다시 열 때 없어집니다. 데이터셋이 닫히면 *AddIndex*를 호출할 수 없습니다. *AddIndex*를 사용하여 추가하는 인덱스는 클라이언트 데이터셋을 파일에 저장할 때 저장되지 않습니다.

- 인덱스를 생성하는 세 번째 방법은 클라이언트 데이터셋이 만들어질 때 생성된 인덱스를 사용하는 것입니다. 클라이언트 데이터셋을 만들기 전에 *IndexDefs* 속성을 사용하여 원하는 인덱스를 지정하십시오. 그러면 *CreateDataSet*을 호출할 때 원본으로 사용하는 데이터셋과 함께 인덱스가 만들어집니다. 클라이언트 데이터셋 만들기 에 대한 자세한 내용은 18-38 페이지의 "테이블 생성 및 삭제"를 참조하십시오.

*AddIndex*에서 데이터셋 지원 그룹화로 만드는 인덱스는 일부 필드에서는 오름차순으로, 일부 필드에서는 내림차순으로 정렬이 가능합니다. 또 일부 필드에서는 대소문자 구별이 불가능하고, 일부 필드에서는 대소문자 구별이 가능합니다. 이러한 방법으로 생성되는 인덱스는 영구적으로 남으며 클라이언트 데이터셋을 파일에 저장할 때 저장됩니다.

팁 내부적으로 계산된 (calculated) 필드에서 클라이언트 데이터셋으로 인덱스를 만들고 정렬할 수 있습니다.

인덱스 삭제와 전환

클라이언트 데이터셋에 대해 만든 인덱스를 삭제하려면 *DeleteIndex*를 호출하고 삭제할 인덱스의 이름을 지정하십시오. DEFAULT_ORDER 인덱스와 CHANGEINDEX 인덱스는 삭제할 수 없습니다.

하나 이상의 인덱스를 사용할 수 있을 때 다른 인덱스를 사용하려면 *IndexName* 속성을 사용하여 사용할 인덱스를 선택합니다. 디자인 타임에 Object Inspector의 *IndexName* 속성 드롭다운 상자에 있는 사용할 수 있는 인덱스 중에서 선택할 수 있습니다.

인덱스를 사용하여 데이터 그룹화

클라이언트 데이터셋에서 인덱스를 사용할 경우 자동적으로 레코드에 정렬 순서를 부여합니다. 이 순서로 인해 이웃하는 레코드에는 대개 인덱스를 구성하는 필드에 값이 중복적으로 포함됩니다. 예를 들어, SalesRep 필드와 Customer 필드로 인덱스된 주문 테이블에서 다음 부분을 생각해 보십시오.

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

정렬 순서 때문에 SalesRep 열의 이웃하는 값이 중복됩니다. SalesRep 1의 레코드 내에서 Customer 열의 이웃하는 값이 중복됩니다. 즉, 데이터가 SalesRep에 의해 그룹화되며 SalesRep 그룹 내에서는 Customer에 의해 그룹화됩니다. 각 그룹화에는 관련 수준이 있습니다. 이 경우 SalesRep 그룹은 다른 그룹에 의해 중첩되지 않았으므로 수준 1이 되고, Customer 그룹은 수준 1인 그룹에 중첩되었으므로 수준 2가 됩니다. 그룹화 수준은 인덱스에서 필드의 순서에 해당합니다.

클라이언트 데이터셋은 특정 그룹화 수준 내에서 현재 레코드를 어디에 놓을지 결정하게 해줍니다. 이렇게 하면 그룹에서 첫 번째 레코드인지, 그룹의 중간에 있는지 또는 그룹의 마지막 레코드인지에 따라 애플리케이션에서 레코드를 다르게 표시할 수 있습니다. 예를 들어, 필드 값이 그룹의 첫 번째 레코드에 있는 경우 중복 값을 삭제하여 필드 값만을 표시할 수 있습니다. 앞 테이블에 대해 이를 적용하면 다음과 같은 결과를 얻습니다.

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

현재 레코드가 그룹의 어느 위치에 있는지 알아내려면 *GetGroupState* 메소드를 사용하십시오. *GetGroupState*는 그룹의 수준을 부여하는 정수를 사용하며 그룹에서 현재 레코드가 그룹의 어느 곳에 있는지(첫 번째 레코드, 마지막 레코드 또는 사이 레코드) 알려 주는 값을 반환합니다.

인덱스를 생성할 때 인덱스가 지원하는 그룹화 수준을 인덱스에 있는 필드 수만큼 지정할 수 있습니다. *GetGroupState*는 인덱스가 추가 필드의 레코드를 정렬하더라도 해당 수준 이상으로 그룹에 대한 정보를 제공할 수는 없습니다.

계산된 값 표시

모든 데이터셋에서 클라이언트 데이터셋에 계산된(calculated) 필드를 추가할 수 있습니다. 이 필드는 동일한 레코드에 있는 다른 필드의 값에 따라 동적으로 계산할 수 있습니다. 계산된 필드 사용에 대한 자세한 내용은 19-7 페이지의 "계산된 필드 정의"를 참조하십시오.

그러나 클라이언트 데이터셋을 사용하면 내부적으로 계산된 필드를 사용하여 필드가 계산될 때 최적화할 수 있습니다. 내부적으로 계산된(calculated) 필드에 대한 자세한 내용은 다음에 나오는 "클라이언트 데이터셋에서 내부적으로 계산된 필드 사용"을 참조하십시오.

또한 유지 관리되는 집계를 사용하여 클라이언트 데이터셋에서 일부 레코드의 데이터를 요약하는 계산된 값을 생성하도록 할 수 있습니다. 유지 관리되는 집계에 대한 자세한 내용은 23-12 페이지의 "유지 관리되는 집계(Maintained Aggregates) 속성 사용"을 참조하십시오.

클라이언트 데이터셋에서 내부적으로 계산된 필드 사용

다른 데이터셋의 경우 애플리케이션은 레코드가 변경되거나 사용자가 현재 레코드의 필드를 편집할 때마다 계산된 필드의 값을 계산해야 합니다. 애플리케이션은 *OnCalcFields* 이벤트 핸들러에서 이를 수행합니다.

클라이언트 데이터셋을 사용하면 클라이언트 데이터셋의 데이터에 계산된 값을 저장하여 계산된 필드를 다시 계산해야 하는 횟수를 최소화할 수 있습니다. 계산된 값이 클라이언트 데이터셋으로 저장되면 사용자가 현재 레코드를 편집할 때마다 값을 일일이 다시 계산해야 하지만 애플리케이션에서는 현재 레코드가 변경될 때마다 값을 다시 계산하지 않아도 됩니다. 클라이언트 데이터셋의 데이터에 계산된 값을 저장하려면 계산된 필드 대신 내부적으로 계산된 필드를 사용하십시오.

계산된 필드와 마찬가지로 내부적으로 계산된 필드는 *OnCalcFields* 이벤트 핸들러에서 계산됩니다. 하지만 클라이언트 데이터셋의 *State* 속성을 검사하여 이벤트 핸들러를 최적화할 수 있습니다. *State*가 *dsInternalCalc*인 경우는 내부적으로 계산된 필드를 다시 계산해야 합니다. *State*가 *dsCalcFields*인 경우는 정기적으로 계산된 필드를 다시 계산하기만 하면 됩니다.

내부적으로 계산된 필드를 사용하려면 사용자가 클라이언트 데이터셋을 생성하기 전에 필드를 내부적으로 계산된 필드로 정의해야 합니다. 영구적 필드를 사용하는지 필드 정의를 사용하는지 여부에 따라 다음 방법 중 하나로 이 작업을 수행합니다.

- 영구적 필드를 사용하는 경우에는 Fields Editor에서 InternalCalc를 선택하여 필드를 내부적으로 계산된 필드로 정의합니다.
- 필드 정의를 사용하는 경우에는 관련 필드 정의의 *InternalCalcField* 속성을 *True*로 설정합니다.

참고 다른 종류의 데이터셋은 내부적으로 계산된 필드를 사용합니다. 그러나 다른 데이터셋의 경우에는 *OnCalcFields* 이벤트 핸들러에서 이 값들을 계산하지 않아도 됩니다. BDE 또는 원격 데이터베이스 서버에서 자동으로 계산됩니다.

유지 관리되는 집계(Maintained Aggregates) 속성 사용

클라이언트 데이터셋은 여러 레코드 그룹에 대해서 데이터를 요약할 수 있는 기능을 제공합니다. 이 요약들은 데이터셋에서 데이터를 편집할 때 자동적으로 업데이트되기 때문에 이 요약된 데이터를 "유지 관리되는 집계(Maintained Aggregates)"라고 합니다.

가장 단순한 형태의 유지 관리되는 집계를 사용하여 클라이언트 데이터셋의 열에 있는 모든 값의 합계와 같은 정보를 얻을 수 있습니다. 하지만 유지 관리되는 집계에서는 다양한 요약 계산 기능을 제공하며 그룹화를 지원하는 인덱스의 필드에 의해 정의되는 여러 레코드 그룹에 대한 요약을 제공할 수 있을 정도로 충분한 유연성을 갖추고 있습니다.

Aggregates 지정

클라이언트 데이터셋의 레코드에 대한 요약을 계산하려면 *Aggregates* 속성을 사용하십시오. *Aggregates*는 집계 사양 (*TAggregate*)의 컬렉션입니다. 디자인 타임에 Collection Editor를 사용하거나 또는 런타임에 *Aggregates*의 *Add* 메소드를 사용하여 클라이언트 데이터셋에 집계 사양을 추가할 수 있습니다. 집계에 대한 필드 컴포넌트를 생성하려면 Fields Editor에서 집계된 값에 대한 영구적 필드를 만드십시오.

참고 집계된 필드를 생성할 때 적절한 집계 객체가 클라이언트 데이터셋의 *Aggregates* 속성에 자동으로 추가됩니다. 집계된 영구적 필드를 생성할 때는 명시적으로 이를 추가하지 마십시오. 집계되는 영구적 필드 생성에 대한 자세한 내용은 19-10 페이지의 "집계 필드(Aggregates field) 정의"를 참조하십시오.

각 집계에서 *Expression* 속성은 집계가 나타내는 요약 계산을 나타냅니다. *Expression*에는 단순한 요약 표현식을 포함할 수 있습니다.

```
Sum(Field1)
```

여러 필드의 정보를 결합하는 복잡한 표현식을 포함할 수도 있습니다.

```
Sum(Qty * Price) - Sum(AmountPaid)
```

집계 표현식에는 표 23.2에 있는 하나 이상의 요약 연산자를 사용합니다.

표 23.2 유지 관리되는 집계의 요약 연산자

연산자	용도
Sum	숫자 필드 또는 표현식의 값의 합계
Avg	숫자나 날짜 시간 필드 또는 표현식의 평균 값 계산
Count	필드 또는 표현식의 비어 있지 않은 값들의 수
Min	문자열, 숫자, 날짜 시간 필드나 표현식의 최소값
Max	문자열, 숫자, 날짜 시간 필드나 표현식의 최대값

요약 연산자는 필터 작성에 사용하는 연산자와 같은 연산자로 필드 값 또는 필드 값으로부터 만들어진 표현식에 사용됩니다. 요약 연산자는 중첩할 수 없습니다. 요약된 값과 다른 요약된 값에 연산자를 사용하거나 요약된 값과 상수에 연산자를 사용하여 표현식을 만들 수 있습니다. 그러나 요약된 값과 필드 값을 결합할 수 없습니다. 왜냐하면 그러한

표현식은 어떤 레코드가 필드 값을 제공하는지에 대한 지시가 없기 때문에 모호합니다. 이러한 규칙은 다음 표현식에서 설명됩니다.

Sum(Qty * Price)	{적합 -- 필드 상에서 표현식을 요약}
Max(Field1) - Max(Field2)	{적합 -- 요약의 표현식}
Avg(DiscountRate) * 100	{적합 -- 요약과 상수의 표현식}
Min(Sum(Field1))	{부적합 -- 중첩된 요약}
Count(Field1) - Field2	{부적합 -- 요약과 필드의 표현식}

레코드 그룹의 집계

유지 관리되는 집계는 기본적으로 클라이언트 데이터셋의 모든 레코드를 요약하도록 계산됩니다. 하지만 그 대신 그룹 안의 레코드에 대해서만 요약하도록 지정할 수 있습니다. 이렇게 하면 공통 필드 값을 공유하는 레코드 그룹의 합계와 같은 중간 요약을 제공할 수 있습니다.

레코드 그룹의 유지 관리되는 집계를 지정하기 전에 적절한 그룹화를 제공하는 인덱스를 이용해야 합니다. 그룹화 지원에 대한 자세한 내용은 23-10 페이지의 "인덱스를 사용하여 데이터 그룹화"를 참조하십시오.

요약하고자 하는 방식으로 데이터를 그룹화하는 인덱스를 만들었다면 어떤 인덱스를 사용하는지와 해당 인덱스의 어떤 그룹 또는 하위 그룹이 인덱스가 요약하는 레코드를 정의하는지를 나타내는 집계의 *IndexName*과 *GroupingLevel* 속성을 지정합니다.

예를 들어, SalesRep로 그룹화되고, SalesRep에서는 Customer로 그룹화된 주문 테이블의 다음 부분을 봅시다.

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

다음 코드는 각 판매 대리점의 총 수량을 나타내는 유지 관리되는 집계를 설정합니다.

```

Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';

```

특정 판매 대리점 내의 각 고객에 대해 요약하는 집계를 추가하려면 수준 2의 유지 관리되는 집계를 만듭니다.

레코드 그룹을 요약하는 유지 관리되는 집계는 지정된 인덱스에 연결됩니다. *Aggregates* 속성은 다른 인덱스를 사용하는 집계를 포함할 수 있습니다. 그러나 전체 데이터셋을 요약하는 집계와 현재 인덱스를 사용하는 집계만이 유효합니다. 현재 인덱스를 변경하면

유효한 집계도 변경됩니다. 임의의 시간에 어떤 집계가 유효한지 결정하려면 *ActiveAggs* 속성을 사용합니다.

집계 값 얻기

유지 관리되는 집계의 값을 얻으려면 집계를 나타내는 *TAggregate* 객체의 *Value* 메소드를 호출합니다. *Value*는 클라이언트 데이터셋의 현재 레코드를 가지고 있는 그룹의 유지 관리되는 집계를 반환합니다.

전체 클라이언트 데이터셋을 요약하는 경우 유지 관리되는 집계를 얻기 위해 언제라도 *Value*를 호출할 수 있습니다. 그러나 그룹화된 정보를 요약하는 경우 현재 레코드가 요약할 원하는 그룹 내에 있는지 확인해야 합니다. 이 때문에 그룹의 첫 레코드로 이동할 때나 그룹의 마지막 레코드로 이동할 때처럼 명확하게 지정된 시기에 집계 값을 구하는 것이 좋습니다. 그룹 내에서 현재 레코드가 어디에 있는지 알아내기 위해서는 *GetGroupState* 메소드를 사용합니다.

유지 관리되는 집계를 data-aware 컨트롤에 표시하려면 Fields Editor를 사용하여 영구적인 집계 (aggregate) 필드 컴포넌트를 만듭니다. Fields Editor에서 집계 필드를 지정하는 경우 클라이언트 데이터셋의 *Aggregates*는 적절한 집계 사양을 포함하도록 자동으로 업데이트됩니다. *AggFields* 속성은 새로 집계되는 필드 컴포넌트를 포함하고 *FindField* 메소드는 이를 반환합니다.

다른 데이터셋에서 데이터 복사

디자인 타임에 다른 데이터셋으로부터 데이터를 복사하려면 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Assign Local Data를 선택합니다. 프로젝트에서 사용 가능한 모든 데이터셋을 나열하는 대화 상자가 나타납니다. 복사하려는 데이터 및 구조를 선택하고 확인을 선택합니다. 소스 데이터셋을 복사하면 클라이언트 데이터셋은 자동으로 활성화됩니다.

런타임에 다른 데이터셋에서 복사하려면 데이터를 직접 할당하거나 소스가 다른 클라이언트 데이터셋에 있는 경우 커서를 복제할 수 있습니다.

데이터 직접 할당

클라이언트 데이터셋의 *Data* 속성을 사용하여 다른 데이터셋에서 클라이언트 데이터셋으로 데이터를 할당할 수 있습니다. *Data*는 *OleVariant* 형식의 데이터 패킷입니다. 데이터 패킷은 다른 클라이언트 데이터셋이나 프로바이더를 사용하여 임의의 다른 데이터셋에서 올 수 있습니다. 일단 데이터 패킷을 *Data*에 할당하면 데이터 소스 컴포넌트에 의해 클라이언트 데이터셋에 연결된 data-aware 컨트롤에 내용이 자동으로 표시됩니다.

서버 데이터를 나타내거나 외부 프로바이더 컴포넌트를 사용하는 클라이언트 데이터셋을 열면 데이터 패킷이 자동으로 *Data*에 할당됩니다.

클라이언트 데이터셋이 프로바이더를 사용하지 않으면 다음과 같이 다른 클라이언트 데이터셋에서 데이터를 복사할 수 있습니다.

```
ClientDataSet1.Data := ClientDataSet2.Data;
```

참고 다른 클라이언트 데이터셋의 *Data* 속성을 복사하면 변경 로그는 복사되지만 적용된 필터나 범위는 반영하지 않습니다. 필터나 범위를 포함하려면 소스 데이터셋의 커서를 대신 복제해야 합니다.

클라이언트 데이터셋이 아닌 데이터셋에서 복사하면 데이터셋 프로바이더 컴포넌트를 만들고 소스 데이터셋에 연결한 다음 그 데이터를 복사할 수 있습니다.

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

참고 사용자가 직접 *Data* 속성에 할당하면 새 데이터 패키트는 기존의 데이터에 합병되지 않습니다. 그 대신 모든 이전 데이터가 교체됩니다.

데이터를 복사하는 대신 다른 데이터셋의 변경 내용을 병합하려면 프로바이더 컴포넌트를 사용해야 합니다. 앞의 예제에서처럼 데이터셋 프로바이더를 만듭니다. 그러나 대상 데이터셋에 연결한 후 데이터 속성을 복사하는 대신 *ApplyUpdates* 메소드를 사용합니다.

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

클라이언트 데이터셋 커서 복제

클라이언트 데이터셋은 사용자가 런타임에 데이터의 두 번째 뷰로 작업할 수 있도록 *CloneCursor* 메소드를 사용합니다. *CloneCursor*를 사용하면 두 번째 클라이언트 데이터셋은 원래 클라이언트 데이터셋의 데이터를 공유할 수 있습니다. 이 방법은 모든 원래의 데이터를 복사하는 것보다 비용이 적게 들지만 데이터를 공유하기 때문에 두 번째 클라이언트 데이터셋은 원래 클라이언트 데이터셋에 영향을 주지 않고 데이터를 수정할 수 없습니다.

*CloneCursor*는 세 개의 매개변수를 사용하는데 이 중 *Source*는 복제할 클라이언트 데이터셋을 지정합니다. 나머지 두 매개변수(*Reset*과 *KeepSettings*)는 데이터 이외의 정보를 복사할지 여부를 나타냅니다. 모든 필터, 현재 인덱스, 마스터 테이블로의 연결(소스 데이터셋이 디테일 정보를 포함할 때), *ReadOnly* 속성 및 연결 컴포넌트나 프로바이더 인터페이스로의 모든 연결 등이 이 정보에 해당합니다.

*Reset*과 *KeepSettings*가 *False*이면 복제된 클라이언트 데이터셋이 열리고 소스 클라이언트 데이터셋의 설정 값이 대상의 속성 설정에 사용됩니다. *Reset*이 *True*이면 대상 데이터셋의 속성에 기본값(인덱스 없음, 필터 없음, 마스터 테이블 없음, *ReadOnly*는 *False*, 연결 컴포넌트 또는 프로바이더가 지정되지 않음)이 설정됩니다. *KeepSettings*가 *True*이면 대상 데이터셋의 속성은 변경되지 않습니다.

데이터에 애플리케이션 특정 정보 추가

애플리케이션 개발자는 클라이언트 데이터셋의 *Data* 속성에 고객 정보를 추가할 수 있습니다. 이 정보는 데이터 패킷과 함께 제공되기 때문에 데이터를 파일이나 스트림에 저장할 때 포함됩니다. 그리고 이 정보는 데이터를 다른 데이터셋에 복사할 때 복사됩니다. 옵션으로서 클라이언트 데이터셋에서 업데이트를 받을 때 이 정보를 프로바이더가 읽을 수 있도록 *Delta* 속성과 함께 포함될 수 있습니다.

Data 속성과 함께 애플리케이션 특정 정보를 저장하려면 *SetOptionalParam* 메소드를 사용합니다. 이 메소드를 통해 특정 이름의 데이터를 포함하는 *OleVariant*를 저장할 수 있습니다.

이 애플리케이션 특정 정보를 검색하려면 정보를 저장할 때 사용한 이름을 전달하는 *GetOptionalParam* 메소드를 사용합니다.

클라이언트 데이터셋을 사용하여 업데이트 캐시

기본적으로 대부분의 데이터셋에서 데이터를 편집하면 데이터셋은 레코드를 삭제하거나 포스트할 때마다 트랜잭션을 생성하고, 데이터베이스 서버에 해당 레코드를 삭제하거나 쓰고, 트랜잭션을 커밋합니다. 데이터베이스에 변경 내용을 쓰는 데 문제가 있으면 애플리케이션에 즉시 통지됩니다. 레코드를 포스트할 때 데이터셋에서 예외가 발생합니다.

데이터셋이 원격 데이터베이스 서버를 사용할 경우 이러한 방식은 현재 레코드 편집 후 새 레코드로 이동할 때마다 애플리케이션과 서버 사이의 네트워크 소동량으로 인한 성능 저하를 가져올 수 있습니다. 네트워크 소동량을 최소화하기 위해 업데이트를 로컬로 캐시할 수 있습니다. 업데이트를 캐시할 때 애플리케이션은 데이터베이스에서 데이터를 검색하여 로컬로 캐시하고 편집한 다음 단일 트랜잭션으로 캐시된 업데이트를 데이터베이스에 적용합니다. 업데이트를 캐시하면 변경 내용 포스트나 레코드 삭제와 같은 데이터셋 변경은 데이터셋의 해당 테이블에 직접 작성되지 않고 로컬로 저장됩니다. 변경이 완료되면 애플리케이션은 캐시된 변경 내용을 데이터베이스에 쓰고 캐시를 지우는 메소드를 호출합니다.

업데이트 캐시로 트랜잭션 시간을 최소화하고 네트워크 소동량을 줄일 수 있습니다. 그러나 캐시된 데이터는 애플리케이션에 로컬이며 트랜잭션에서 제어되지 않습니다. 즉, 메모리 내 로컬 데이터 복사본에서 작업하는 동안 다른 애플리케이션이 원본으로 사용한 데이터베이스 테이블에서 데이터를 변경할 수 있다는 의미입니다. 또한 변경 내용은 캐시된 업데이트를 적용하기 전에는 볼 수 없습니다. 따라서 변경 내용을 데이터베이스로 병합할 때 너무 많은 충돌이 발생할 수 있으므로 캐시된 업데이트는 불안정한 데이터와 작업하는 애플리케이션에 적합하지 않을 수 있습니다.

BDE와 ADO가 업데이트 캐시에 대체할 메커니즘을 제공하기는 하지만 업데이트 캐시에 클라이언트 데이터셋을 사용하면 다음과 같은 몇 가지 장점이 있습니다.

- 데이터셋이 마스터/디테일 관계에 연결될 때 업데이트를 적용하는 것이 모두 처리됩니다. 이렇게 하여 연결된 여러 데이터셋에 대한 업데이트가 올바른 순서로 적용되는지 확인합니다.

- 클라이언트 데이터셋은 업데이트 프로세스를 최대한 제어합니다. 레코드를 업데이트 하도록 생성된 SQL에 영향을 주는 속성을 설정하거나, 다중 테이블 조인에서 레코드를 업데이트할 때 사용할 테이블을 지정하거나, *BeforeUpdateRecord* 이벤트 핸들러에서 직접 업데이트를 적용할 수 있습니다.
- 데이터베이스 서버에 캐시된 업데이트를 적용할 때 오류가 발생하면 데이터베이스 서버의 현재 레코드 값, 데이터셋의 편집되지 않은 원래 값과 실패한 업데이트의 편집된 새 값에 대한 정보는 클라이언트 데이터셋과 데이터셋 프로바이더에서만 제공됩니다.
- 클라이언트 데이터셋을 사용하면 전체 업데이트가 롤백되기 전에 허용할 업데이트 오류 수를 지정할 수 있습니다.

캐시된 업데이트 사용 개요

캐시된 업데이트를 사용하려면 애플리케이션에서 다음과 같은 순서로 프로세스를 진행해야 합니다.

- 1 **편집할 데이터를 지정합니다.** 데이터를 지정하는 방법은 사용 중인 클라이언트 데이터셋의 타입에 따라 달라집니다.
 - *TClientDataSet*을 사용하고 있으면 편집하고자 하는 데이터를 나타내는 프로바이더 컴포넌트를 지정합니다. 이 내용은 23-25 페이지의 "프로바이더 지정"에 설명되어 있습니다.
 - 특정 데이터 액세스 메커니즘과 관련된 클라이언트 데이터셋을 사용하고 있을 경우
 - *DBConnection* 속성을 적절한 연결 컴포넌트로 설정하여 데이터베이스 서버를 식별해야 합니다.
 - *CommandText* 및 *CommandType* 속성을 지정하여 보려는 데이터를 지정합니다. *CommandType*은 *CommandText*가 실행할 SQL 문인지, 내장 프로시저 이름인지, 테이블 이름인지를 나타냅니다. *CommandText*가 쿼리 또는 내장 프로시저이면 *Params* 속성을 사용하여 입력 매개변수를 제공합니다.
 - 옵션으로 *Options* 속성을 사용하여 중첩 디테일 집합 및 BLOB 데이터가 데이터 패킷에 포함되는지 또는 개별적으로 폐치되는지, 특정 유형의 편집 (삽입, 수정 또는 삭제)을 사용할 수 없게 되는지, 한 번의 업데이트로 여러 서버 레코드에 영향을 줄 수 있는지 및 업데이트를 적용할 때 클라이언트 데이터셋의 레코드를 새로 고치는지 여부를 나타냅니다. *Options*는 프로바이더의 *Options* 속성과 일치합니다. 결과적으로 관련되지 않거나 적절하지 않은 옵션을 설정할 수 있게 합니다. 예를 들어, 클라이언트 데이터셋이 영구적 필드로 데이터셋의 데이터를 폐치하지 않기 때문에 *poIncFieldProps*를 포함할 이유가 없습니다. 반대로 클라이언트 데이터셋이 원하는 데이터를 지정할 때 사용하는 *CommandText* 속성을 사용할 수 없게 하므로 기본적으로 포함되는 *poAllowCommandText*는 제외하지 않으려고 합니다. 프로바이더의 *Options* 속성에 대한 내용은 24-5 페이지의 "데이터 패킷에 영향을 미치는 옵션 설정"을 참조하십시오.

- 2 **데이터를 표시하고 편집하며**, 새 레코드 삽입을 허용하고, 기존 레코드 삭제를 지원 합니다. 각 레코드의 원래 복사본과 모든 편집 내용은 모두 메모리에 저장됩니다. 이 프로세스는 23-5 페이지의 "데이터 편집"에 설명되어 있습니다.
- 3 **필요에 따라 추가 레코드를 폐치합니다**. 기본적으로 클라이언트 데이터셋은 모든 레코드를 폐치하고 메모리에 저장합니다. 데이터셋에 많은 레코드 또는 광범위한 BLOB 필드가 있는 레코드가 포함되면 필요에 따라 표시하고 다시 폐치하기에 적당한 정도의 레코드만 폐치하도록 클라이언트 데이터셋을 변경할 수 있습니다. 레코드 폐치 프로세스를 제어하는 방법에 대한 자세한 내용은 23-26 페이지의 "소스 데이터셋이나 문서에서 데이터 요청"을 참조하십시오.
- 4 **옵션으로 레코드를 새로 고칩니다**. 시간이 경과하면 다른 사용자가 데이터베이스 서버의 데이터를 수정할 수 있습니다. 이렇게 하면 클라이언트 데이터셋의 데이터가 점점 더 많이 데이터에서 벗어날 수 있으며 업데이트를 적용할 때 오류가 발생할 수 있는 가능성이 증가합니다. 이 문제를 해결하기 위해 이미 편집된 레코드를 새로 고칠 수 있습니다. 자세한 내용은 23-31 페이지의 "레코드 새로 고침"을 참조하십시오.
- 5 **로컬로 캐시된 레코드를 데이터베이스에 적용하거나** 업데이트를 취소합니다. 데이터베이스에 쓴 레코드의 경우에는 *BeforeUpdateRecord* 이벤트가 트리거됩니다. 데이터베이스에 개별 레코드를 쓸 때 오류가 발생하면 애플리케이션은 *OnUpdateError* 이벤트를 사용하여 오류를 수정하고 업데이트를 계속할 수 있습니다. 업데이트가 완료되면 성공적으로 적용된 모든 업데이트가 로컬 캐시에서 지워집니다. 데이터베이스에 업데이트를 적용하는 것에 대한 자세한 내용은 23-20 페이지의 "레코드 업데이트"를 참조하십시오.

애플리케이션은 업데이트를 적용하는 대신 업데이트를 취소할 수 있으며 변경 내용을 데이터베이스에 쓰지 않고 변경 로그를 비웁니다. *CancelUpdates* 메소드를 호출하여 업데이트를 취소할 수 있습니다. 캐시에서 삭제된 모든 레코드의 삭제가 취소되고, 수정된 레코드가 원래 값으로 되돌아가며, 새로 삽입된 레코드는 사라집니다.

업데이트 캐시에 필요한 데이터셋 타입 선택

Delphi에는 업데이트 캐시에 필요한 특수 클라이언트 데이터셋 컴포넌트가 포함되어 있습니다. 각 클라이언트 데이터셋은 특정 데이터 액세스 메커니즘에 연결됩니다. 다음 표 23.3에 나열되어 있습니다.

표 23.3 업데이트 캐시에 필요한 특수 클라이언트 데이터셋

클라이언트 데이터셋	데이터 액세스 메커니즘
TBDEClientDataSet	Borland Database Engine
TSQLClientDataSet	dbExpress
TIBClientDataSet	InterBase Express

또한 외부 프로바이더 및 소스 데이터셋과 함께 일반 클라이언트 데이터셋(*TClientDataSet*)을 사용하여 업데이트를 캐시할 수 있습니다. 외부 프로바이더와 함께 *TClientDataSet*을 사용하는 것에 대한 자세한 내용은 23-25 페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

참고 각 데이터 액세스 메커니즘에 연결된 특수 클라이언트 데이터셋은 실제로 프로바이더 및 소스 데이터셋을 사용합니다. 그러나 프로바이더와 소스 데이터셋 모두 클라이언트 데이터셋에 대해 내부입니다.

업데이트를 캐시할 때는 특수 클라이언트 데이터셋 중 하나를 사용하는 것이 가장 간단합니다. 그러나 외부 프로바이더와 함께 *TClientDataSet*을 사용하는 것이 더 좋은 경우가 있습니다.

- 특수 클라이언트 데이터셋이 없는 데이터 액세스 메커니즘을 사용하고 있으면 외부 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용해야 합니다. 예를 들어, 데이터가 XML 문서나 사용자 지정 데이터셋에서 오는 경우가 있습니다.
- 마스터/디테일 관계로 관계가 설정된 테이블 작업을 하고 있는 경우에는 *TClientDataSet*을 사용해야 하며 프로바이더를 사용하여 마스터/디테일 관계로 연결된 두 소스 데이터셋의 마스터 테이블에 연결해야 합니다. 클라이언트 데이터셋은 디테일 데이터셋을 중첩 데이터셋 필드로 간주합니다. 이러한 방식은 마스터 및 디테일 테이블을 올바른 순서로 업데이트할 수 있게 할 때 필요합니다.
- 예를 들어, 클라이언트 데이터셋이 프로바이더의 데이터를 폐치하기 전과 후처럼 클라이언트 데이터셋과 프로바이더 사이의 통신에 응답하는 이벤트 핸들러를 코딩하려면 외부 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용해야 합니다. 특수 클라이언트 데이터셋은 업데이트를 적용하는 데 가장 중요한 이벤트 (*OnReconcileError*, *BeforeUpdateRecord* 및 *OnGetTableName*)를 게시하지만 클라이언트 데이터셋과 프로바이더 사이의 통신 주위에서 발생하는 이벤트는 주로 다계층 애플리케이션에 사용하도록 되어 있으므로 게시하지 않습니다.
- BDE를 사용할 때에는 업데이트 개체를 사용해야 하는 경우 외부 프로바이더와 소스 데이터셋을 사용할 수 있습니다. *TBDEClientDataSet*의 *BeforeUpdateRecord* 이벤트 핸들러에서 업데이트 개체를 코딩할 수 있지만 소스 데이터셋의 *UpdateObject* 속성을 할당하는 것이 더 간단할 수 있습니다. 업데이트 개체 사용에 대한 자세한 내용은 20-40 페이지의 "업데이트 객체를 사용하여 데이터셋 업데이트"를 참조하십시오.

수정된 레코드 표시

사용자가 클라이언트 데이터셋을 편집하는 동안 편집된 내용에 대한 피드백을 제공하는 것이 유용할 수 있습니다. 예를 들어, 편집 내용으로 이동하여 "실행 취소" 버튼을 클릭하는 경우처럼 사용자가 특정 편집을 실행 취소할 수 있게 하는 데 특히 유용합니다.

UpdateStatus 메소드 및 *StatusFilter* 속성은 업데이트가 발생한 부분에 대한 피드백을 제공할 때 유용합니다.

- *UpdateStatus*는 현재 레코드에 발생한 업데이트 유형을 나타냅니다. 다음 값 중 하나가 될 수 있습니다.
 - *usUnmodified*는 현재 레코드가 변경되지 않았음을 나타냅니다.
 - *usModified*는 현재 레코드가 편집되었음을 나타냅니다.
 - *usInserted*는 사용자가 삽입한 레코드를 나타냅니다.
 - *usDeleted*는 사용자가 삭제한 레코드를 나타냅니다.

- *StatusFilter*는 변경 로그에서 표시되는 업데이트 종류를 제어합니다. *StatusFilter*는 일반적인 데이터에서 필터가 작동하는 것과 같은 방식으로 캐시된 레코드에서 작동합니다. *StatusFilter*는 집합이므로 다음과 같은 값의 조합이 포함될 수 있습니다.
 - *usUnmodified*는 수정되지 않은 레코드를 나타냅니다.
 - *usModified*는 수정된 레코드를 나타냅니다.
 - *usInserted*는 삽입된 레코드를 나타냅니다.
 - *usDeleted*는 삭제된 레코드를 나타냅니다.

기본적으로 *StatusFilter*는 집합 [*usModified*, *usInserted*, *usUnmodified*]입니다. 이 집합에 *usDeleted*를 추가하여 삭제된 레코드에 대한 피드백을 제공할 수 있습니다.

참고 *UpdateStatus* 및 *StatusFilter*는 *BeforeUpdateRecord* 및 *OnReconcileError* 이벤트 핸들러에도 유용합니다. *BeforeUpdateRecord*에 대한 자세한 내용은 23-22 페이지의 "업데이트가 적용될 때 조정"을 참조하십시오. *OnReconcileError*에 대한 자세한 내용은 23-23 페이지의 "업데이트 오류 해결"을 참조하십시오.

다음 예제에서는 *UpdateStatus* 메소드를 사용하여 레코드의 업데이트 상태에 대한 피드백을 제공하는 방법을 보여 줍니다. *usDeleted*를 포함하도록 *StatusFilter* 속성을 변경하여 삭제된 레코드가 데이터셋에 표시되도록 한 것으로 가정합니다. 더 나아가 계산된 필드를 "Status"라는 데이터셋에 추가한 것으로 가정합니다.

```
procedure TForm1.ClientDataSet1CalcFields(DataSet:TDataSet);
begin
  with ClientDataSet1 do begin
    case UpdateStatus of
      usUnmodified: FieldByName('Status').AsString := '';
      usModified: FieldByName('Status').AsString := 'M';
      usInserted: FieldByName('Status').AsString := 'I';
      usDeleted: FieldByName('Status').AsString := 'D';
    end;
  end;
end;
```

레코드 업데이트

변경 로그의 내용은 클라이언트 데이터셋의 *Delta* 속성에 데이터 패킷으로 저장됩니다. *Delta*의 변경 내용이 영구적이게 하려면 클라이언트 데이터셋은 이 변경 내용을 데이터베이스 또는 소스 데이터셋이나 XML 문서에 적용해야 합니다.

클라이언트는 다음과 같은 단계를 따라 서버에 업데이트를 적용합니다.

- 1 클라이언트 애플리케이션은 클라이언트 데이터셋 객체의 *ApplyUpdates* 메소드를 호출합니다. 이 메소드는 클라이언트 데이터셋의 *Delta* 속성 내용을 내부 또는 외부 프로바이더에 전달합니다. *Delta*는 클라이언트 데이터셋의 업데이트, 삽입 및 삭제된 레코드를 포함하는 데이터 패킷입니다.
- 2 프로바이더는 자체적으로 해결할 수 없는 모든 문제 레코드를 캐시하여 업데이트를 적용합니다. 프로바이더가 업데이트를 적용하는 방법에 대한 자세한 내용은 24-8 페이지의 "클라이언트 업데이트 요청에 대한 응답"을 참조하십시오.

- 3 프로바이더는 해결하지 못한 모든 레코드를 *Result* 데이터 패킷의 클라이언트 데이터셋에 반환합니다. *Result* 데이터 패킷에는 업데이트되지 않은 모든 레코드가 있습니다. 또한 오류 메시지와 오류 코드와 같은 오류 정보도 포함합니다.
- 4 클라이언트 데이터셋은 레코드별로 *Result* 데이터 패킷에 반환된 업데이트 오류의 해결을 시도합니다.

업데이트 적용

클라이언트 데이터셋의 로컬 데이터 복사본에서 변경한 내용은 클라이언트 애플리케이션에서 *ApplyUpdates* 메소드를 호출한 후에야 데이터베이스 서버 또는 XML 문서로 전달됩니다. *ApplyUpdates*는 변경 로그를 사용하여 프로바이더에 *Delta*라는 데이터 패킷으로 변경 내용을 보냅니다. 대부분의 클라이언트 데이터셋을 사용할 때 프로바이더는 클라이언트 데이터셋에 대해 내부입니다.

*ApplyUpdates*는 *MaxErrors*라는 매개변수를 하나 사용하는데 이는 업데이트 프로세스를 중단할 때까지 허용할 수 있는 최대 오류 수를 나타냅니다. *MaxErrors*가 0이면 업데이트 오류가 발생하는 즉시 전체 업데이트 프로세스가 종료됩니다. 데이터베이스에 대해 아무 변경이 없으면 클라이언트 데이터셋의 변경 로그는 바뀌지 않은 상태로 남습니다. *MaxErrors*가 -1인 경우에는 수에 상관 없이 오류가 허용되며 변경 로그는 성공적으로 적용되지 않은 모든 레코드를 포함합니다. *MaxErrors*가 양수이고 *MaxErrors*에서 허용하는 수보다 오류가 많이 발생한 경우 모든 업데이트는 중지됩니다. *MaxErrors*에서 지정된 수보다 오류가 작게 발생한 경우 성공적으로 적용된 모든 레코드는 클라이언트 데이터셋의 변경 로그에서 자동으로 지워집니다.

*ApplyUpdates*는 실제로 발생한 오류의 수를 반환하는데 이 수는 항상 *MaxErrors* + 1보다 작거나 같습니다. 이 반환 값은 데이터베이스에 기록될 수 없는 레코드의 수를 나타냅니다.

클라이언트 데이터셋의 *ApplyUpdates* 메소드는 다음을 수행합니다.

- 1 프로바이더의 *ApplyUpdates* 메소드를 간접적으로 호출합니다. 프로바이더의 *ApplyUpdates* 메소드는 데이터베이스, 소스 데이터셋 또는 XML 문서에 업데이트를 기록하고 발생한 오류를 수정하려고 합니다. 오류 조건으로 인해 적용될 수 없는 레코드는 클라이언트 데이터셋으로 다시 보내집니다.
- 2 그런 다음 클라이언트 데이터셋의 *ApplyUpdates* 메소드는 *Reconcile* 메소드를 호출하여 이런 문제 레코드를 해결하려 합니다. *Reconcile*은 *OnReconcileError* 이벤트 핸들러를 호출하는 오류 처리 루틴입니다. 오류를 수정하려면 *OnReconcileError* 이벤트 핸들러를 직접 코딩해야 합니다. *OnReconcileError* 사용에 대한 자세한 내용은 23-23 페이지의 "업데이트 오류 해결"을 참조하십시오.
- 3 마지막으로 *Reconcile*은 성공적으로 적용된 변경 내용을 변경 로그에서 제거하고 새로 업데이트된 레코드를 반영하도록 *Data*를 업데이트합니다. *Reconcile*을 완료하면 *ApplyUpdates*는 발생한 오류 수를 보고합니다.

중요 경우에 따라 프로바이더는 업데이트를 적용하는 방법을 결정할 수 없습니다. 예를 들면, 내장 프로시저나 다중 테이블 조인에서 업데이트를 적용하는 경우입니다. 클라이언트 데이터셋과 프로바이더 컴포넌트는 이러한 상황을 처리할 수 있는 이벤트를 생성합니다. 자세한 내용은 다음의 "업데이트가 적용될 때 조정"을 참조하십시오.

팁 프로바이더가 스테이트리스(stateless) 애플리케이션 서버에 있는 경우 업데이트를 적용하기 전후에 영구적인 상태 정보에 대해 프로바이더와 통신하기를 원할 수도 있습니다. *TClientDataSet*은 업데이트를 보내기 전에 영구적인 상태 정보를 서버로 보낼 수 있는 *BeforeApplyUpdates* 이벤트를 받습니다. 업데이트를 적용한 후 해결 프로세스 전에 *TClientDataSet*은 애플리케이션 서버가 반환하는 영구적인 상태 정보에 응답할 수 있는 *AfterApplyUpdates* 이벤트를 받습니다.

업데이트가 적용될 때 조정

클라이언트 데이터셋이 업데이트를 적용할 때 프로바이더는 데이터베이스 서버나 소스 데이터셋에 삽입, 삭제 및 수정 작성을 처리하는 방법을 결정합니다. 외부 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용하면 해당 프로바이더의 속성 및 이벤트를 사용하여 업데이트가 적용되는 방법에 영향을 줄 수 있습니다. 이 내용은 24-8 페이지의 "클라이언트 업데이트 요청에 대한 응답"에서 설명합니다.

그러나 프로바이더가 데이터 액세스 메커니즘과 연결된 클라이언트 데이터셋을 위한 것으로 내부적이면 속성을 설정하거나 이벤트 핸들러를 제공할 수 없습니다. 따라서 클라이언트 데이터셋은 내부 프로바이더가 업데이트를 적용하는 방법에 영향을 줄 수 있는 속성 하나와 이벤트 두 개를 게시합니다.

- *UpdateMode*는 프로바이더가 업데이트를 적용하기 위해 생성한 SQL 문에서 레코드를 찾는 데 사용하는 필드를 제어합니다. *UpdateMode*는 프로바이더의 *UpdateMode* 속성과 동일합니다. 프로바이더의 *UpdateMode* 속성에 대한 내용은 24-10 페이지의 "업데이트 적용 방법 변경"을 참조하십시오.
- *OnGetTableName*을 사용하면 업데이트를 적용해야 하는 데이터베이스 테이블의 이름으로 프로바이더를 제공할 수 있습니다. 이렇게 하면 프로바이더는 *CommandText*에서 지정한 쿼리나 내장 프로시저에서 데이터베이스 테이블을 식별할 수 없을 때 업데이트를 위한 SQL 문을 생성할 수 있습니다. 예를 들면, 쿼리가 하나의 테이블에만 업데이트가 필요한 다중 테이블 조인을 실행하는 경우 *OnGetTableName* 이벤트 핸들러를 제공하면 내부 프로바이더가 정확하게 업데이트를 적용할 수 있습니다.

OnGetTableName 이벤트 핸들러에는 내부 프로바이더 컴포넌트, 서버에서 데이터를 폐지하는 내부 데이터셋 및 생성된 SQL에 사용하도록 테이블 이름을 반환하는 매개변수 등 세 가지 매개변수가 있습니다.

- *BeforeUpdateRecord*는 델타 패킷의 모든 레코드에 대해 발생합니다. 이 이벤트를 통해 레코드를 삽입, 삭제 또는 수정하기 전에 마지막 변경이 가능하게 합니다. 또한 프로바이더가 올바른 SQL을 생성할 수 없는 경우(예를 들어, 여러 테이블이 업데이트되어야 하는 다중 테이블 조인)에 업데이트를 적용할 사용자가 만든 SQL 문을 실행하는 방법을 제공합니다.

BeforeUpdateRecord 이벤트 핸들러에는 내부 프로바이더 컴포넌트, 서버에서 데이터를 폐지하는 내부 데이터셋, 업데이트하려는 레코드에 있는 델타 패킷, 업데이트가 삽입, 삭제 또는 수정인지를 나타내는 표시 및 이벤트 핸들러가 업데이트를 수행했는지 여부를 반환하는 매개변수 등 다섯 가지 매개변수가 있습니다. 이러한 매개변수의 사용에 대해서는 다음과 같은 이벤트 핸들러에서 설명합니다. 예제에서는 간단하게 하기 위해 필드 값만 필요로 하는 전역 변수로 SQL 문을 사용할 수 있는 것으로 가정합니다.

```

procedure TForm1.SQLClientDataSet1BeforeUpdateRecord(Sender:TObject;
  SourceDS: TDataSet; DeltaDS:TCustomClientDataSet; UpdateKind: TUpdateKind;
  var Applied Boolean);
var
  SQL:string;
  Connection:TSQLConnection;
begin
  Connection := (SourceDS as TCustomSQLDataSet).SQLConnection;
  case UpdateKind of
    ukModify:
      begin
        { 1st dataset: update Fields[1], use Fields[0] in where clause }
        SQL := Format(UpdateStmt1, [DeltaDS.Fields[1].NewValue, DeltaDS.Fields[0].OldValue]);
        Connection.Execute(SQL, nil, nil);
        { 2nd dataset: update Fields[2], use Fields[3] in where clause }
        SQL := Format(UpdateStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].OldValue]);
        Connection.Execute(SQL, nil, nil);
      end;
    ukDelete:
      begin
        { 1st dataset: use Fields[0] in where clause }
        SQL := Format(DeleteStmt1, [DeltaDS.Fields[0].OldValue]);
        Connection.Execute(SQL, nil, nil);
        { 2nd dataset: use Fields[3] in where clause }
        SQL := Format(DeleteStmt2, [DeltaDS.Fields[3].OldValue]);
        Connection.Execute(SQL, nil, nil);
      end;
    ukInsert:
      begin
        { 1st dataset: values in Fields[0] and Fields[1] }
        SQL := Format(InsertStmt1, [DeltaDS.Fields[0].NewValue, DeltaDS.Fields[1].NewValue]);
        Connection.Execute(SQL, nil, nil);
        { 2nd dataset: values in Fields[2] and Fields[3] }
        SQL := Format(InsertStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].NewValue]);
        Connection.Execute(SQL, nil, nil);
      end;
    end;
    Applied := True;
  end;

```

업데이트 오류 해결

업데이트 과정 중에 발생하는 오류를 처리할 수 있는 이벤트에는 두 가지가 있습니다.

- 내부 프로바이더는 업데이트 과정 중에 처리할 수 없는 업데이트가 발생할 때마다 *OnUpdateError* 이벤트를 생성합니다. *OnUpdateError* 이벤트 핸들러에서 문제를 수정하면 해당 오류는 *ApplyUpdates* 메소드로 전달되는 최대 오류 수에 포함되지 않습니다. 이 이벤트는 내부 프로바이더를 사용하는 클라이언트 데이터셋에 대해서만 발생합니다. *TClientDataSet*을 사용하는 경우에는 프로바이더 컴포넌트의 *OnUpdateError* 이벤트를 사용할 수 있습니다.

- 전체 업데이트 작업이 완료되면 클라이언트 데이터셋은 프로바이더가 데이터베이스 서버에 적용할 수 없는 모든 레코드에 대해 *OnReconcileError* 이벤트를 생성합니다. 반환된 레코드 중에서 적용할 수 없는 레코드를 제거하는 경우에도 *OnReconcileError* 또는 *OnUpdateError* 이벤트 핸들러를 항상 코딩해야 합니다. 이 두 가지 이벤트에 대한 이벤트 핸들러는 동일하게 작동합니다. 다음과 같은 매개변수가 포함되어 있습니다.
- *DataSet*: 적용될 수 없는 업데이트된 레코드를 포함하는 클라이언트 데이터셋. 클라이언트 데이터셋 메소드를 사용하여 문제 레코드에 대한 정보를 얻고 문제를 해결하기 위해 레코드를 편집할 수 있습니다. 특히 업데이트 문제의 원인을 알아내기 위해 현재 레코드에 있는 필드의 *CurValue*, *OldValue* 및 *NewValue* 속성 사용을 원할 수도 있습니다. 그러나 이벤트 핸들러에서 현재 레코드를 변경하는 어떤 클라이언트 데이터셋 메소드도 호출해서는 안 됩니다.
- *E*: 발생한 문제를 나타내는 객체. 이 예외를 사용하여 오류 메시지를 추출하거나 업데이트 오류의 원인을 알아낼 수 있습니다.
- *UpdateKind*: 오류를 생성한 업데이트의 종류. *UpdateKind*는 *ukModify*(수정된 기존 레코드를 업데이트할 때 발생하는 문제), *ukInsert*(새 레코드를 삽입할 때 발생하는 문제) 또는 *ukDelete*(기본 레코드를 삭제할 때 발생하는 문제) 중의 하나입니다.
- *Action*: 이벤트 핸들러가 존재할 때 처리되는 액션을 나타내는 **var** 매개변수. 이벤트 핸들러에서 이 매개변수를 설정하여 다음을 수행합니다.
 - 변경 로그에 남겨둔 채로 이 레코드를 건너뛸지 않습니다(*rrSkip* 또는 *raSkip*).
 - 모든 해결 작업을 중지합니다(*rrAbort* 또는 *raAbort*).
 - 서버에서 해당 레코드에 실패한 수정을 합병합니다(*rrMerge* 또는 *raMerge*). 이 작업은 서버 레코드에 클라이언트 데이터셋의 레코드에서 수정된 필드에 변경 내용이 포함되지 않는 경우에만 가능합니다.
 - 변경 로그의 현재 업데이트를 이미 수정되었을 이벤트 핸들러의 레코드 값으로 교체합니다(*rrApply* 또는 *raCorrect*).
 - 오류를 완전히 무시합니다(*rrIgnore*). 이 작업은 *OnUpdateError* 이벤트 핸들러에서만 가능하며 이벤트 핸들러가 업데이트를 다시 데이터베이스 서버에 적용하는 경우에 사용됩니다. 업데이트된 레코드는 변경 로그에서 제거되고 프로바이더가 업데이트를 적용한 것처럼 *Data*로 병합됩니다.
 - 클라이언트 데이터셋에서 이 레코드의 변경 내용을 원래 제공된 값으로 복귀하여 되돌립니다(*raCancel*). 이 작업은 *OnReconcileError* 이벤트 핸들러에서만 가능합니다.
 - 현재 레코드 값을 업데이트하여 서버의 레코드와 일치하도록 합니다(*raRefresh*). 이 작업은 *OnReconcileError* 이벤트 핸들러에서만 가능합니다.

다음 코드는 객체 레포지토리 디렉토리에 있는 RecError 유닛에서 오류 해결 대화 상자를 사용하는 *OnReconcileError* 이벤트 핸들러를 보여 줍니다. 이 대화 상자를 사용하기 위해 uses 절에 RecError를 추가합니다.

```

procedure TForm1.ClientDataSetReconcileError(DataSet:TCustomClientDataSet;
E:EReconcileError; UpdateKind:TUpdateKind, var Action TReconcileAction);
begin
    Action := HandleReconcileError(DataSet, UpdateKind, E);
end;

```

프로바이더와 함께 클라이언트 데이터셋 사용

클라이언트 데이터셋은 다음과 같은 경우에 프로바이더를 사용하여 데이터를 제공하고 업데이트를 적용합니다.

- 데이터베이스 서버나 다른 데이터셋에서 업데이트를 캐시합니다.
- XML 문서에 데이터를 나타냅니다.
- 다계층 애플리케이션의 클라이언트 부분에 데이터를 저장합니다.

*TClientDataSet*이 아닌 모든 클라이언트 데이터셋에 대해 이 프로바이더는 내부 프로바이더가 되므로 애플리케이션에서 직접 액세스할 수 없습니다. *TClientDataSet*을 사용하면 프로바이더는 데이터의 외부 소스에 클라이언트 데이터셋을 연결하는 외부 컴포넌트가 됩니다.

외부 프로바이더 컴포넌트는 클라이언트 데이터셋과 동일한 애플리케이션에 있거나 다른 시스템에서 실행 중인 별도 애플리케이션의 일부일 수 있습니다. 프로바이더 컴포넌트에 대한 자세한 내용은 24장 "프로바이더 컴포넌트 사용"을 참조하십시오. 프로바이더가 다른 시스템의 별도의 애플리케이션에 있는 애플리케이션에 대한 자세한 내용은 25장 "다계층(multi-tiered) 애플리케이션 생성"을 참조하십시오.

내부 또는 외부 프로바이더를 사용할 때 클라이언트 데이터셋은 항상 업데이트를 캐시합니다. 이 작업 방법에 대한 자세한 내용은 23-16 페이지의 "클라이언트 데이터셋을 사용하여 업데이트 캐시"를 참조하십시오.

다음 항목에서는 클라이언트 데이터셋이 프로바이더와 작업할 때 사용할 수 있는 추가 속성 및 메소드에 대해 설명합니다.

프로바이더 지정

데이터 액세스 메커니즘과 연결된 클라이언트 데이터셋과 달리 *TClientDataSet*에는 업데이트를 적용하거나 데이터를 패키지로 만드는 내부 프로바이더 컴포넌트가 없습니다. 소스 데이터셋이나 XML 문서의 데이터를 나타내려면 클라이언트 데이터셋을 외부 프로바이더 컴포넌트와 연결해야 합니다.

프로바이더를 *TClientDataSet*에 연결하는 방법은 프로바이더가 클라이언트 데이터셋과 동일한 애플리케이션에 있는지 또는 다른 시스템에서 실행 중인 원격 애플리케이션 서버에 있는지에 따라 다릅니다.

- 프로바이더가 클라이언트 데이터셋과 동일한 애플리케이션에 있는 경우, Object Inspector의 *ProviderName* 속성의 드롭다운 목록에서 프로바이더를 선택하여 프로바이더에 연결할 수 있습니다. 이는 프로바이더가 클라이언트 데이터셋과 동일한 소유자 (*Owner*)를 가지는 동안 작동합니다. 클라이언트 데이터셋과 프로바이더는 동일한 폼이나 데이터 모듈에 있는 경우 동일한 소유자를 가집니다. 다른 소유자를 가지는 지역 프로바이더를 사용하려면 클라이언트 데이터셋의 *SetProvider* 메소드를 사용하여 런타임에 연결을 만들어야 합니다.

결국 원격 프로바이더로 확장하거나 *IAppServer* 인터페이스를 직접 호출하려는 경우에는 *RemoteServer* 속성을 *TLocalConnection* 컴포넌트로 설정할 수도 있습니다. *TLocalConnection*을 사용하면 *TLocalConnection* 인스턴스는 애플리케이션에 로컬인 모든 프로바이더 목록을 관리하고 클라이언트 데이터셋의 *IAppServer* 호출을 처리합니다. *TLocalConnection*을 사용하지 않으면 Delphi는 클라이언트 데이터셋의 *IAppServer* 호출을 처리하는 숨겨진 객체를 만듭니다.

- 프로바이더가 원격 애플리케이션 서버에 있으면 *ProviderName* 속성뿐만 아니라 클라이언트 데이터셋을 애플리케이션 서버에 연결하는 컴포넌트를 지정해야 합니다. 이 작업을 처리할 수 있는 속성으로는 두 가지가 있습니다. 프로바이더 목록을 얻을 수 있는 연결 컴포넌트 이름을 지정하는 *RemoteServer*와 클라이언트 데이터셋과 연결 컴포넌트 사이의 간접 참조 수준을 추가할 중앙 브로커를 지정하는 *ConnectionBroker*입니다. 연결 컴포넌트 및 연결 브로커(사용될 경우)는 클라이언트 데이터셋과 같은 데이터 모듈에 상주합니다. 연결 컴포넌트는 애플리케이션 서버에 연결을 만들고 유지 관리하므로 "데이터 브로커"라고도 합니다. 자세한 내용은 25-4 페이지의 "클라이언트 애플리케이션의 구조"를 참조하십시오.

디자인 타임에 *RemoteServer* 또는 *ConnectionBroker*를 지정한 다음 Object Inspector의 *ProviderName* 속성의 드롭다운 목록에서 프로바이더를 선택할 수 있습니다. 이 목록에는 동일한 폼 또는 데이터 모듈의 지역 프로바이더와 연결 컴포넌트를 통해 액세스할 수 있는 원격 프로바이더가 있습니다.

참고 연결 컴포넌트가 *TDCOMConnection*의 인스턴스이면 애플리케이션 서버는 클라이언트 컴퓨터에 등록해야 합니다.

런타임 시 코드에 *ProviderName*을 설정하여 사용 가능한 지역 프로바이더와 원격 프로바이더 사이를 전환할 수 있습니다.

소스 데이터셋이나 문서에서 데이터 요청

클라이언트 데이터셋은 프로바이더에서 데이터 패킷을 페치하는 방법을 제어할 수 있습니다. 기본적으로 클라이언트 데이터셋은 소스 데이터셋에서 모든 레코드를 검색합니다. *TBDEClientDataSet*, *TSQLClientDataSet* 및 *TIBClientDataSet*에서처럼 소스 데이터셋과 프로바이더가 내부 컴포넌트인지 *TClientDataSet*에 데이터를 제공하는 개별 컴포넌트인지 여부는 분명합니다.

클라이언트 데이터셋이 *PacketRecords* 및 *FetchOnDemand* 속성을 사용하여 레코드를 페치하는 방법을 변경할 수 있습니다.

점진적 페치(Incremental fetching)

PacketRecords 속성을 변경하면 클라이언트 데이터셋이 더 작은 청크로 데이터를 페치하도록 지정할 수 있습니다. *PacketRecords*는 한 번에 페치하는 레코드 수 또는 반환하는 레코드의 타입을 지정합니다. 기본적으로 *PacketRecords*는 -1 로 설정되는데 이는 클라이언트 데이터셋이 처음 열린 때나 애플리케이션이 명시적으로 *GetNextPacket*을 호출할 때 사용 가능한 모든 레코드를 한꺼번에 페치함을 의미합니다. *PacketRecords*가 -1 인 경우 클라이언트 데이터셋이 데이터를 처음 페치하고 나면 이미 사용 가능한 모든 레코드가 있기 때문에 더 이상 데이터를 페치할 필요가 없습니다.

작은 배치(batch) 단위로 레코드를 페치하려면 *PacketRecords*를 페치할 레코드의 수로 설정합니다. 예를 들어, 다음 문장은 각 데이터 패킷의 크기를 10개 레코드로 설정합니다.

```
ClientDataSet1.PacketRecords := 10;
```

배치로 레코드를 페치하는 이러한 프로세스를 "점진적 페치(incremental fetch)"라고 합니다. 클라이언트 데이터셋은 *PacketRecords*가 0보다 크면 점진적 페치를 사용합니다.

각 레코드를 배치 단위로 페치하기 위해 클라이언트 데이터셋은 *GetNextPacket* 속성을 호출합니다. 새로 페치된 패킷은 클라이언트 데이터셋에 이미 있는 데이터의 끝에 추가됩니다. *GetNextPacket*은 페치한 레코드 수를 반환합니다. 반환 값이 *PacketRecords*와 동일한 경우 사용 가능한 레코드의 끝에 도달하지 않습니다. 반환 값이 0보다 크고 *PacketRecords*보다 작은 경우 마지막 레코드는 페치 작업 중에 도달됩니다. *GetNextPacket*이 0을 반환하면 더 이상 페치할 레코드가 없습니다.

경고 점진적 페치는 스테이트리스(stateless) 애플리케이션 서버의 원격 프로바이더로부터 데이터를 페치하는 경우 작동하지 않습니다. 스테이트리스(stateless) 원격 데이터 모듈로 점진적 페치를 사용하는 방법에 대한 자세한 내용은 25-20 페이지의 "원격 데이터 모듈의 상태 정보 지원"을 참조하십시오.

참고 또한 *PacketRecords*를 사용하여 소스 데이터셋에 대한 메타데이터 정보를 페치할 수 있습니다. 메타데이터 정보를 검색하려면 *PacketRecords*를 0으로 설정합니다.

요구 즉시 페치(Fetch-on-demand)

레코드의 자동 페치는 *FetchOnDemand* 속성에서 제어합니다. *FetchOnDemand*가 *True*(기본값)이면 클라이언트 데이터셋은 필요에 따라 자동으로 레코드를 페치합니다. 레코드의 자동 페치를 방지하려면 *FetchOnDemand*를 *False*로 설정합니다. *FetchOnDemand*가 *False*인 경우 애플리케이션은 레코드를 페치하기 위해서 *GetNextPacket*을 명시적으로 호출해야 합니다.

예를 들면 대량의 읽기 전용 데이터셋을 나타내야 하는 애플리케이션은 *FetchOnDemand*를 해제하여 클라이언트 데이터셋이 메모리를 초과하는 데이터를 로드하지 않게 할 수 있습니다. 페치 사이에서 클라이언트 데이터셋은 *EmptyDataSet* 메소드를 사용하여 캐시를 해제합니다. 그러나 이 방법은 클라이언트가 서버에 업데이트를 포스트해야 하는 경우에는 잘 작동하지 않습니다.

프로바이더는 데이터 패킷의 레코드에 BLOB 데이터와 중복되는 디테일 데이터셋이 포함되어 있는지를 제어합니다. 프로바이더가 레코드에서 이러한 정보를 제외하면 *FetchOnDemand*

속성은 클라이언트 데이터셋이 BLOB 데이터와 디테일 데이터셋을 필요에 따라 자동으로 페치할 수 있게 합니다. *FetchOnDemand*가 *False*이고 프로바이더가 레코드에 BLOB 데이터와 디테일 데이터셋을 포함하지 않으면 명시적으로 *FetchBlobs*나 *FetchDetails* 메소드를 호출하여 이 정보를 검색해야 합니다.

소스 데이터셋에서 매개변수 얻기

클라이언트 데이터셋이 매개변수 값을 페치해야 하는 상황에는 두 가지가 있습니다.

- 애플리케이션이 내장 프로시저에 출력 매개변수 값을 필요로 합니다.
- 애플리케이션은 쿼리나 내장 프로시저의 입력 매개변수를 소스 데이터셋의 현재 값으로 초기화하려고 합니다.

클라이언트 데이터셋은 매개변수 값을 *Params* 속성에 저장합니다. 클라이언트 데이터셋이 소스 데이터셋에서 데이터를 페치(fetch)할 때마다 이 값은 출력 매개변수로 새로고쳐집니다. 그러나 클라이언트 애플리케이션의 *TClientDataSet* 컴포넌트가 데이터를 페치하지 않는 경우에도 출력 매개변수가 필요할 때가 있습니다.

레코드를 페치하지 않는 경우에도 출력 매개변수를 페치하거나 입력 매개변수를 초기화하기 위해서 클라이언트 데이터셋이 *FetchParams* 메소드를 호출하여 소스 데이터셋에서 매개변수 값을 요청할 수 있습니다. 매개변수는 프로바이더로부터 데이터 패킷에 반환되고 클라이언트 데이터셋의 *Params* 속성에 할당됩니다.

디자인 타임에 *Params* 속성은 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Fetch Params를 선택하여 초기화할 수 있습니다.

참고 *Params* 속성은 항상 내부 소스 데이터셋의 매개변수를 반영하므로 클라이언트 데이터셋이 내부 프로바이더와 소스 데이터셋을 사용할 때는 *FetchParams*를 호출할 필요가 전혀 없습니다. *TClientDataSet*과 함께 *FetchParams* 메소드는 연결된 데이터셋이 매개변수를 제공할 수 있는 프로바이더에 클라이언트 데이터셋이 연결된 경우에만 작동합니다. 예를 들어, 소스 데이터셋이 테이블 타입 데이터셋인 경우에는 페치할 매개변수가 없습니다.

프로바이더가 스테이트리스(stateless) 애플리케이션 서버의 일부로서 별도의 시스템에 있는 경우 *FetchParams*를 사용하여 출력 매개변수를 검색할 수 없습니다. 스테이트리스 애플리케이션 서버에서 다른 클라이언트는 *FetchParams*로의 호출 전에 출력 매개변수를 변경하여 쿼리나 내장 프로시저를 변경하고 재실행할 수 있습니다. 스테이트리스 애플리케이션 서버에서 출력 매개변수를 검색하려면 *Execute* 메소드를 사용합니다. 프로바이더가 쿼리나 내장 프로시저에 연결되어 있다면 *Execute*는 프로바이더가 쿼리나 내장 프로시저를 실행하고 출력 매개변수를 반환하게 합니다. 이렇게 반환되는 매개변수는 자동으로 *Params* 속성을 업데이트하는 데 사용됩니다.

소스 데이터셋에 매개변수 전달

클라이언트 데이터셋은 데이터 패킷으로 받기 원하는 데이터를 지정하기 위해서 소스 데이터셋에 매개변수를 전달할 수 있습니다. 이런 매개변수는 다음을 지정할 수 있습니다.

- 애플리케이션 서버에서 실행되는 내장 프로시저 또는 쿼리의 입력 매개변수 값

- 데이터 패킷으로 보내는 레코드를 제한하는 필드 값

디자인 타임이나 런타임에 클라이언트 데이터셋이 소스 데이터셋에 보내는 매개변수 값을 지정할 수 있습니다. 디자인 타임에 클라이언트 데이터셋을 선택한 다음 Object Inspector의 *Params* 속성을 더블 클릭합니다. 이렇게 하면 매개변수를 추가, 삭제 또는 재배치할 수 있는 Collection Editor가 나타납니다. Collection Editor에서 매개변수를 선택하여 Object Inspector를 이용해서 해당 매개변수의 속성을 편집할 수 있습니다.

런타임에 *Params* 속성의 *CreateParam* 메소드를 사용하여 클라이언트 데이터셋에 매개변수를 추가합니다. *CreateParam*은 매개변수 객체, 지정된 이름, 매개변수 타입 및 데이터 타입을 반환합니다. 그런 다음 해당 매개변수 객체의 속성을 사용하여 매개변수에 값을 할당합니다.

예를 들어, 다음 코드는 값이 605인 CustNo라는 입력 매개변수를 추가합니다.

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
    AsInteger := 605;
```

클라이언트 데이터셋이 활성화되지 않은 경우, 매개변수를 애플리케이션 서버에 보낼 수 있고 단순히 *Active* 속성을 *True*로 설정하여 매개변수 값을 반영하는 데이터 패킷을 검색할 수 있습니다.

쿼리 또는 내장 프로시저 매개변수 보내기

클라이언트 데이터셋의 *CommandType* 속성이 *ctQuery*나 *ctStoredProc*이거나 클라이언트 데이터셋이 *TClientDataSet* 인스턴스인 경우, 연결된 프로바이더가 쿼리나 내장 프로시저의 결과를 나타내면 *Params* 속성을 사용하여 매개변수 값을 지정할 수 있습니다. 클라이언트 데이터셋이 소스 데이터셋으로부터 데이터를 요청하거나 *Execute* 메소드를 사용하여 데이터셋을 반환하지 않는 쿼리나 내장 프로시저를 실행하는 경우 이 매개변수 값을 데이터에 대한 요청이나 실행 명령과 함께 전달합니다. 프로바이더는 이 매개변수 값을 받으면 연결된 데이터셋에 할당합니다. 그런 다음 매개변수 값을 사용하여 쿼리나 내장 프로시저를 실행하도록 데이터셋에 지시하고 클라이언트 데이터셋이 데이터를 요청한 경우 결과 셋의 첫 번째 레코드로 시작하는 데이터를 제공하기 시작합니다.

참고 매개변수 이름은 소스 데이터셋의 해당 매개변수 이름과 일치해야 합니다.

매개변수로 레코드 제한

클라이언트 데이터셋이 다음 두 인스턴스 중 하나인 경우

- 연결된 프로바이더가 *TTable*이나 *TSQLTable* 컴포넌트를 나타내는 *TClientDataSet* 인스턴스
- *CommandType* 속성이 *ctTable*인 *TSQLClientDataSet*이나 *TBDEClientDataSet* 인스턴스

Params 속성을 사용하여 메모리에 캐시되는 레코드를 제한할 수 있습니다. 각 매개변수는 클라이언트 데이터셋의 데이터에 레코드를 포함시키기 전에 일치해야 하는 필드 값을 나타냅니다. 이렇게 하면 필터처럼 작동하고 필터가 있는 경우를 제외하고는 레코드는 여전히 메모리에 캐시되지만 사용할 수는 없습니다.

각 매개변수 이름은 필드 이름과 일치해야 합니다. *TClientDataSet*을 사용할 때는 프로바이더와 연결된 *TTable* 또는 *TSQLTable* 컴포넌트의 필드 이름이 됩니다. *TSQLClientDataSet*이나 *TBDEClientDataSet*을 사용할 때는 데이터베이스 서버의 테이블에 있는 필드 이름이 됩니다. 따라서 클라이언트 데이터셋의 데이터는 해당 필드의 값이 매개변수에 할당된 값과 일치하는 레코드만 포함합니다.

예를 들어, 고객 한 명의 주문 내용을 보여 주는 애플리케이션을 생각해 봅시다. 사용자가 고객 한 명을 식별할 때 클라이언트 데이터셋은 그 고객의 주문 정보들만 보이도록 식별하는 값인 *CustID*라는 단일 매개변수를 포함하도록 *Params* 속성을 설정합니다. 클라이언트 데이터셋이 소스 데이터셋에서 데이터를 요청하면 이 매개변수 값을 전달합니다. 그러면 프로바이더는 이 고객에 대한 레코드만 보냅니다. 이것은 프로바이더가 모든 주문 레코드를 클라이언트 애플리케이션에 보내고 클라이언트 데이터셋을 사용하여 레코드를 필터링하는 것보다 훨씬 효율적입니다.

서버에서 제약 조건 처리

데이터베이스 서버가 유효한 데이터에 제약 조건을 정의할 때 클라이언트 데이터셋이 제약 조건을 인식하면 유용합니다. 클라이언트 데이터셋은 이런 방법으로 사용자 편집 내용이 해당 서버 제약 조건을 위반하는지 확인할 수 있습니다. 따라서 이러한 위반은 위반 사항을 거부하는 데이터베이스 서버로는 전달되지 않습니다. 이는 업데이트 프로세스 동안 업데이트가 오류 조건을 덜 발생시킨다는 의미입니다.

데이터의 소스에 관계 없이 클라이언트 데이터셋에 제약 조건을 명시적으로 추가하여 이러한 서버 제약 조건을 중복할 수 있습니다. 이 프로세스는 23-7 페이지의 "사용자 지정 제약 조건의 지정"에서 다룹니다.

그러나 서버 제약 조건이 데이터 패킷에 자동으로 포함되면 더 편리합니다. 그러면 기본 표현식을 명시적으로 지정할 필요가 없으며 클라이언트 데이터셋은 서버 제약 조건이 변경될 때 적용되는 값을 변경합니다. 기본적으로 다음과 같은 방식이 정확히 수행됩니다. 소스 데이터셋이 서버 제약 조건을 인식하면 프로바이더가 자동으로 데이터 패킷에 포함시키고 클라이언트 데이터셋은 사용자가 편집 내용을 변경 로그에 포스트할 때 제약 조건을 적용합니다.

참고 BDE를 사용하는 데이터셋만 서버에서 제약 조건을 import할 수 있습니다. 즉, BDE 기반 데이터셋을 나타내는 프로바이더와 함께 *TBDEClientDataSet*이나 *TClientDataSet*을 사용할 때만 서버 제약 조건이 데이터 패킷에 포함된다는 의미입니다. 서버 제약 조건을 import하는 방법과 프로바이더가 제약 조건을 데이터 패킷에 포함할 수 없게 하는 방법에 대한 자세한 내용은 24-13 페이지의 "서버 제약 조건 처리"를 참조하십시오.

참고 import된 제약 조건으로 작업하는 것에 대한 자세한 내용은 19-22 페이지의 "서버 제약 조건 사용"을 참조하십시오.

서버 제약 조건 및 표현식을 import하는 작업은 애플리케이션이 데이터 무결성을 유지할 수 있게 하는 매우 가치있는 기능이지만 일시적으로 제약 조건을 사용할 수 없게 해야 하는 경우가 있을 수 있습니다. 예를 들어, 서버 제약 조건이 필드의 현재 최대값에 기반을 둔 것이지만 클라이언트 데이터셋이 증가적으로 폐지하는 경우, 클라이언트 데이터셋에 있는 필드의 현재 최대값이 데이터베이스 서버의 최대값과 다를 수 있으며 제약 조건이 다르게 실행될 수 있습니다. 다른 경우를 예로 들면, 제약 조건이 사용 가능할

때 클라이언트 데이터셋에서 필터를 레코드에 적용하면 필터는 의도하지 않게 제약 조건에 방해가 될 수 있습니다. 이러한 경우 애플리케이션에서 제약 조건 검사를 사용 불가능으로 만들 수 있습니다.

일시적으로 제약 조건을 사용할 수 없도록 하려면 *DisableConstraints* 메소드를 호출합니다. *DisableConstraints*가 호출될 때마다 참조 카운트가 늘어납니다. 참조 카운트가 0보다 크면 클라이언트 데이터셋에서 제약 조건이 실행되지 않습니다.

클라이언트 데이터셋의 제약 조건을 다시 사용 가능하도록 만들려면 데이터셋의 *EnableConstraints* 메소드를 호출하십시오. *EnableConstraints*를 호출할 때마다 참조 카운트가 줄어듭니다. 참조 카운트가 0이면 제약 조건이 다시 사용 가능하게 됩니다.

팁 의도한 대로 제약 조건이 사용 가능하게 되었는지 확인하려면 항상 *DisableConstraints*와 *EnableConstraints*를 쌍으로 호출하십시오.

레코드 새로 고침

클라이언트 데이터셋은 소스 데이터셋의 메모리 내 데이터 스냅샷으로 작업합니다. 소스 데이터셋이 서버 데이터를 나타내는 경우 시간이 경과하면 다른 사용자가 해당 데이터를 수정할 수 있습니다. 그러면 클라이언트 데이터셋의 데이터는 원본 데이터의 정확한 값과는 점점 달라집니다.

다른 데이터셋과 같이 클라이언트 데이터셋도 서버의 현재 값과 일치하도록 레코드를 업데이트하는 *Refresh* 메소드를 가지고 있습니다. 그러나 *Refresh* 호출은 변경 로그에 편집 내용이 없는 경우에만 작동합니다. 적용되지 않은 편집 내용이 있을 때 *Refresh*를 호출하면 예외가 발생합니다.

클라이언트 데이터셋은 변경 로그는 그대로 남겨 두고 데이터를 업데이트할 수도 있습니다. 이렇게 하려면 *RefreshRecord* 메소드를 호출합니다. *Refresh* 메소드와 달리 *RefreshRecord*는 클라이언트 데이터셋의 현재 레코드만 업데이트합니다. *RefreshRecord*는 프로바이더에서 원래 얻은 레코드 값만 변경하고 변경 로그의 모든 변경 내용은 그냥 둡니다.

경고 *RefreshRecord* 호출이 적절하지 않은 경우도 있습니다. 사용자의 편집이 다른 사용자가 만든 원본으로 사용하는 데이터셋에 대한 변경 내용과 충돌을 일으키는 경우 *RefreshRecord* 호출은 이 충돌을 마스크합니다. 클라이언트 데이터셋이 업데이트를 적용할 때 오류 해결이 발생하지 않으며 애플리케이션은 충돌을 해결할 수 없습니다.

업데이트 오류의 마스킹을 막으려면 *RefreshRecord*를 호출하기 전에 보류 중인 업데이트가 없는지 확인해야 합니다. 예를 들어, 다음의 *AfterScroll*은 사용자가 확인한 새 레코드로 이동할 때마다 가장 최신 값으로 레코드를 새로 고치지만 안전할 때만 고칩니다.

```
procedure TForm1.ClientDataSet1AfterScroll(DataSet:TDataSet);
begin
    if ClientDataSet1.UpdateStatus = usUnModified then
        ClientDataSet1.RefreshRecord;
end;
```

사용자 지정 이벤트를 사용하여 프로바이더와 통신

클라이언트 데이터셋은 *IAppServer*라는 특수한 인터페이스를 통해 프로바이더 컴포넌트와 통신합니다. 지역 프로바이더의 경우 *IAppServer*는 클라이언트 데이터셋과 프로바이더 사이의 모든 통신을 처리하는 자동으로 생성된 객체에 대한 인터페이스입니다. 원격 프로바이더의 경우 *IAppServer*는 애플리케이션 서버의 원격 데이터 모듈에 대한 인터페이스입니다.

*TClientDataSet*은 *IAppServer* 인터페이스를 사용하는 통신을 사용자 지정할 수 있는 기회를 많이 제공합니다. 클라이언트 데이터셋의 프로바이더에서 직접 전달되는 모든 *IAppServer* 메소드 호출 전후에 *TClientDataSet*은 프로바이더와 정보를 교환할 수 있게 하는 특별한 이벤트를 받습니다. 이러한 이벤트는 프로바이더의 비슷한 이벤트들과 일치합니다. 예를 들어, 클라이언트 데이터셋이 *ApplyUpdates* 메소드를 호출하면 다음 이벤트가 발생합니다.

- 1 클라이언트 데이터셋은 *OleVariant* 타입의 *OwnerData*라는 속성에 임의의 사용자 지정 정보를 지정하는 *BeforeApplyUpdates* 이벤트를 받습니다.
- 2 프로바이더는 클라이언트 데이터셋의 *OwnerData*에 응답하고 *OwnerData*의 값을 새 정보로 업데이트할 수 있는 *BeforeApplyUpdates* 이벤트를 받습니다.
- 3 프로바이더는 데이터 패킷을 조합하기 위한 일반적인 프로세스를 수행합니다(여기에 따른 이벤트들이 발생합니다.).
- 4 프로바이더가 *OwnerData*의 현재 값에 응답하고 클라이언트 데이터셋의 값으로 업데이트할 수 있는 *AfterApplyUpdates* 이벤트를 받습니다.
- 5 클라이언트 데이터셋은 *OwnerData*의 반환된 값에 응답할 수 있는 *AfterApplyUpdates* 이벤트를 받습니다.

다른 모든 *IAppServer* 메소드 호출도 클라이언트 데이터셋과 프로바이더 사이의 통신을 사용자 지정할 수 있게 하는 이와 비슷한 일련의 *BeforeXXX* 및 *AfterXXX* 이벤트들이 발생합니다.

또한 클라이언트 데이터셋은 *DataRequest*라는 특별한 메소드를 가지며 이 메소드의 유일한 용도는 프로바이더와 애플리케이션 특정 통신을 가능하게 하는 것입니다. 클라이언트 데이터셋이 *DataRequest*를 호출하는 경우 원하는 모든 정보를 포함할 수 있는 *OleVariant*를 매개변수로 전달합니다. 그런 다음 사용자가 임의의 애플리케이션 정의 방식으로 응답할 수 있고 클라이언트 데이터셋에 값을 반환할 수 있는 *OnDataRequest* 이벤트를 프로바이더에 생성합니다.

소스 데이터셋 오버라이드

특정 데이터 액세스 메커니즘과 연결된 클라이언트 데이터셋은 *CommandText* 및 *CommandType* 속성을 사용하여 표시하는 데이터를 지정할 수 있습니다. 그러나 *TClientDataSet*을 사용할 때 데이터는 클라이언트 데이터셋이 아닌 소스 데이터셋에서 지정합니다. 일반적으로 이 소스 데이터에는 SQL 문을 지정하여 데이터나 데이터베이스 테이블 또는 내장 프로시저의 이름을 생성하는 속성이 있습니다.

프로바이더가 허용하면 *TClientDataSet*은 표시되는 데이터를 나타내는, 소스 데이터셋의 속성을 오버라이드할 수 있습니다. 즉, 프로바이더가 허용하면 클라이언트 데이터셋의 *CommandText* 속성은 표시되는 데이터를 지정하는 프로바이더의 데이터셋 속성을 교체합니다. 따라서 *TClientDataSet*은 보려는 데이터를 동적으로 지정할 수 있습니다.

기본적으로 외부 프로바이더 컴포넌트는 클라이언트 데이터셋이 이런 방법으로 *CommandText* 값을 사용할 수 없게 합니다. *TClientDataSet*이 *CommandText* 속성을 사용하도록 하려면 *poAllowCommandText*를 프로바이더의 *Options* 속성에 추가해야 합니다. 그렇지 않으면 *CommandText*의 값이 무시됩니다.

참고 *TSQLClientDataSet*, *TBDEClientDataSet* 또는 *TIBClientDataSet*의 *Options* 속성에서 *poAllowCommandText*를 제거하지 마십시오. 클라이언트 데이터셋의 *Options* 속성은 내부 프로바이더로 전달되므로 *poAllowCommandText*를 제거하면 클라이언트 데이터셋이 액세스할 데이터를 지정할 수 없게 합니다.

클라이언트 데이터셋은 다음 두 경우에 *CommandText* 문자열을 프로바이더로 보냅니다.

- 클라이언트 데이터셋을 처음 열 때. 프로바이더에서 첫 번째 데이터 패킷을 검색한 후 다음의 연속적인 데이터 패킷을 폐치할 때 클라이언트 데이터셋은 *CommandText*를 보내지 않습니다.
- 클라이언트 데이터셋이 *Execute* 명령을 프로바이더로 보낼 때.

나중에 언제든지 SQL 명령을 보내거나 테이블 또는 내장 프로시저 이름을 변경하려면 *AppServer* 속성으로 사용 가능한 *IAppServer* 인터페이스를 명시적으로 사용해야 합니다. 이 속성은 클라이언트 데이터셋이 프로바이더와 통신하는 인터페이스를 나타냅니다.

파일 기반 데이터로 클라이언트 데이터셋 사용

클라이언트 데이터셋은 서버 데이터뿐만 아니라 디스크의 전용 파일로 작업할 수 있습니다. 이렇게 하면 클라이언트 데이터셋을 파일 기반 데이터베이스 애플리케이션과 "브리프케이스(briefcase) 모델" 애플리케이션에서 사용할 수 있습니다. 데이터에 대해 클라이언트 데이터셋이 사용하는 특수 파일을 MyBase라고 합니다.

팁 모든 클라이언트 데이터셋은 브리프케이스 모델 애플리케이션에 적당하지만 프로바이더를 사용하지 않는 순수 MyBase 애플리케이션의 경우에는 오버헤드가 더 적으므로 *TClientDataSet*을 사용하는 것이 좋습니다.

순수 MyBase 애플리케이션에서 클라이언트 애플리케이션은 서버에서 테이블 정의 및 데이터를 가져올 수 없으며 업데이트를 적용할 수 있는 서버가 없습니다. 그 대신 클라이언트 데이터셋은 다음을 독립적으로 수행해야 합니다.

- 테이블 정의와 생성
- 저장된 데이터 로드
- 데이터에 편집 내용 합병
- 데이터 저장

새로운 데이터셋 생성

다음과 같은 세 가지 방법으로 서버 데이터를 나타내지 않는 클라이언트 데이터셋을 정의하고 생성합니다.

- 영구적 필드 및 인덱스 정의를 사용하여 클라이언트 데이터셋을 새로 만들고 정의할 수 있습니다. 그리고 나서 테이블 타입 데이터셋을 만드는 것과 같은 스키마를 수행합니다. 자세한 내용은 18-38 페이지의 "테이블 생성 및 삭제"를 참조하십시오.
- 디자인 타임이나 런타임 시에 기존 데이터셋을 복사할 수 있습니다. 기존 데이터셋 복사에 대한 자세한 내용은 23-14 페이지의 "다른 데이터셋에서 데이터 복사"를 참조하십시오.
- 임의의 XML 문서에서 클라이언트 데이터셋을 만들 수 있습니다. 자세한 내용은 26-6 페이지의 "XML 문서를 데이터 패킷으로 변환"을 참조하십시오.

일단 데이터셋을 만들면 파일에 저장할 수 있습니다. 그 때부터 테이블을 다시 만들 필요가 없으며 저장한 파일에서 로드하면 됩니다. 파일 기반 데이터베이스 애플리케이션을 시작할 때 애플리케이션 자체를 쓰기 전에 먼저 데이터베이스에 빈 파일을 만들고 저장할 수 있습니다. 이미 정의된 클라이언트 데이터셋의 메타데이터로 시작하는 이런 방식으로 사용자 인터페이스를 설정하는 것이 더 쉽습니다.

파일이나 스트림에서 데이터 로드

파일에서 데이터를 로드하려면 클라이언트 데이터셋의 *LoadFromFile* 메소드를 호출합니다. *LoadFromFile*은 데이터를 읽어 올 파일을 지정하는 문자열 매개변수가 하나 있습니다. 파일 이름은 전체 경로 이름이 될 수도 있습니다. 같은 파일에서 클라이언트 데이터셋의 데이터를 항상 로드하면 *FileName* 속성을 대신 사용할 수 있습니다. *FileName*이 기존 파일을 명명하면 데이터는 클라이언트 데이터셋이 열릴 때 자동으로 로드됩니다.

스트림에서 데이터를 로드하려면 클라이언트 데이터셋의 *LoadFromStream* 메소드를 호출합니다. *LoadFromStream*은 데이터를 제공하는 스트림 객체인 하나의 매개변수를 가집니다.

LoadFromFile(*LoadFromStream*)이 로드하는 데이터는 *LoadFromFile*이 클라이언트 데이터셋의 데이터 서식으로 이전에 저장했거나 *SaveToFile* (*SaveToStream*) 메소드를 사용하여 다른 클라이언트 데이터셋에 저장했어야 합니다. 데이터를 파일이나 스트림에 저장하는 것에 대한 내용은 23-35 페이지의 "파일이나 스트림에 데이터 저장"을 참조하십시오. XML 문서에서 클라이언트 데이터셋 데이터를 만드는 것에 대한 자세한 내용은 26장 "데이터베이스 애플리케이션에서 XML 사용"을 참조하십시오.

LoadFromFile 또는 *LoadFromStream*을 호출할 때 파일의 모든 데이터를 *Data* 속성으로 읽습니다. 데이터가 저장될 때 변경 로그에 있는 모든 편집 내용을 *Delta* 속성으로 읽습니다. 그러나 파일에서 읽어 오는 인덱스는 데이터셋으로 작성된 인덱스뿐입니다.

데이터에 변경 내용 병합

클라이언트 데이터셋의 데이터를 편집하면 데이터의 모든 편집 내용은 메모리 내 변경 로그에만 존재합니다. 이 로그는 클라이언트 데이터셋을 사용하는 객체로 완전히 전송되지만 데이터 자체와 별도로 유지할 수 있습니다. 즉, 클라이언트 데이터셋을 탐색하거나 그 데이터를 표시하는 컨트롤은 변경 내용을 포함하는 데이터를 볼 수 있습니다. 그러나 변경 내용을 취소하지 않고자 하는 경우 *MergeChangeLog* 메소드를 호출하여 변경 로그를 클라이언트 데이터셋의 데이터로 병합해야 합니다. *MergeChangeLog*는 변경 로그의 변경된 필드 값으로 *Data*의 레코드를 덮어씁니다.

*MergeChangeLog*를 실행한 후 *Data*는 기존 데이터와 변경 로그에 있는 변경 내용을 혼합하여 포함합니다. 이 혼합은 향후 변경이 수행될 때 새로운 *Data* 기준이 됩니다. *MergeChangeLog*는 모든 레코드의 변경 로그를 지우고 *ChangeCount* 속성을 0으로 재설정합니다.

경고 프로바이더를 사용하는 클라이언트 데이터셋에 대해 *MergeChangeLog*를 호출하지 마십시오. 이런 경우에는 *ApplyUpdates*를 호출하여 데이터베이스에 변경 내용을 씁니다. 자세한 내용은 23-21 페이지의 "업데이트 적용"을 참조하십시오.

참고 또한 해당 데이터셋이 원래 *Data* 속성에 제공되어진 경우 각각의 클라이언트 데이터셋의 데이터로 변경 내용을 병합할 수 있습니다. 이 작업을 하려면 데이터셋 프로바이더를 사용해야 합니다. 이 작업을 하는 방법의 예는 23-14 페이지의 "데이터 직접 할당"을 참조하십시오.

변경 로그의 확장된 실행 취소 기능을 사용하지 않으려면 클라이언트 데이터셋의 *LogChanges* 속성을 *False*로 설정할 수 있습니다. *LogChanges*가 *False*이면 편집 내용은 레코드를 포스트할 때 자동으로 병합되므로 *MergeChangeLog*를 호출하지 않아도 됩니다.

파일이나 스트림에 데이터 저장

변경 내용을 클라이언트 데이터셋의 데이터로 병합한 경우에도 이 데이터는 여전히 메모리 내에만 존재합니다. 클라이언트 데이터셋을 닫고 애플리케이션에서 다시 연 경우 데이터는 영구적이지만 애플리케이션을 종료한 경우에는 사라집니다. 데이터를 영구적으로 만들려면 디스크에 써야 합니다. *SaveToFile* 메소드를 사용하여 디스크에 변경 내용을 쓰십시오.

*SaveToFile*은 데이터를 기록할 파일을 지정하는 문자열인 하나의 매개변수를 가집니다. 파일 이름은 전체 경로 이름이 될 수도 있습니다. 파일이 이미 존재하는 경우 현재 내용을 완전히 덮어씁니다.

참고 *SaveToFile*은 런타임 시 클라이언트 데이터셋에 추가한 인덱스는 저장하지 않고 클라이언트 데이터셋을 만들 때 추가한 인덱스만 저장합니다.

데이터를 항상 같은 파일에 저장하는 경우 *FileName* 속성을 대신 사용할 수 있습니다. *FileName*을 설정하면 클라이언트 데이터셋이 닫힐 때 데이터를 자동으로 명명된 파일에 저장합니다.

또한 *SaveToStream* 메소드를 사용하여 데이터를 스트림에 저장할 수도 있습니다. *SaveToStream*은 데이터를 받는 스트림 객체인 하나의 매개변수를 가집니다.

참고 편집 내용이 여전히 변경 로그에 있는 경우 클라이언트 데이터셋을 저장하면 데이터에 병합하지 않습니다. *LoadFromFile* 또는 *LoadFromStream* 메소드를 사용하여 데이터를 다시 로드하면 변경 로그에는 여전히 병합되지 않은 편집 내용이 있습니다. 이것은 해당 변경 내용을 애플리케이션 서버의 프로바이더 컴포넌트에 적용해야 하는 브리프 케이스 모델을 지원하는 애플리케이션에서 중요합니다.

24

프로바이더 컴포넌트 사용

프로바이더 컴포넌트 (*TDataSetProvider*와 *TXMLTransformProvider*)는 클라이언트 데이터셋이 데이터를 얻는 가장 일반적인 메커니즘을 제공합니다. 프로바이더는 다음을 수행합니다.

- 클라이언트 데이터셋 또는 XML 브로커에서 데이터 요청을 받아 요청된 데이터를 페치하고, 데이터를 수송 데이터 패키지로 패키징하며, 클라이언트 데이터셋 또는 XML 브로커에 데이터를 반환합니다. 이러한 활동을 "공급(providing)"이라고 합니다.
- 데이터셋 또는 XML 브로커에서 업데이트된 데이터를 받아 업데이트 사항을 데이터베이스 서버, 소스 데이터셋 또는 XML 문서에 적용하고, 적용될 수 없는 업데이트 사항을 로그하여 향후 조정을 위해 클라이언트 데이터셋에 미해결된 업데이트 사항을 반환합니다. 이러한 활동을 "해결(resolving)"이라고 합니다.

프로바이더 컴포넌트의 대부분의 작업은 자동으로 이루어집니다. 업데이트 사항을 적용할 데이터셋이나 XML 문서의 데이터로부터 데이터 패킷을 생성하기 위해 프로바이더에 코드를 작성해야 합니다. 그러나 프로바이더 컴포넌트는 애플리케이션이 클라이언트에 대해 패키징된 정보와 클라이언트 요청에 대해 애플리케이션이 응답하는 방법을 보다 직접적으로 제어할 수 있도록 하는 다수의 이벤트와 속성을 포함합니다.

TBDEClientDataSet, *TSQLClientDataSet* 또는 *TIBClientDataSet*을 사용하면 프로바이더는 클라이언트 데이터셋에 내부적이고, 애플리케이션은 클라이언트 데이터셋에 직접 액세스할 수 없습니다. 그러나 *TClientDataSet* 또는 *TXMLBroker*를 사용할 경우에 프로바이더는 클라이언트를 위해, 그리고 공급 및 해결 프로세스에서 발생하는 이벤트에 응답하기 위해 패키징된 정보를 제어하는 데 사용할 수 있는 별도의 컴포넌트입니다. 내부 프로바이더가 있는 클라이언트 데이터셋은 일부 내부 프로바이더의 속성과 이벤트를 고유한 속성과 이벤트로 나타내지만 보다 많은 제어를 위해서 사용자는 분리 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용하고자 할 수도 있습니다.

별도의 프로바이더 컴포넌트 사용 시 클라이언트 데이터셋 또는 XML 브로커로서 동일한 애플리케이션에 상주하거나 다계층 애플리케이션의 일부로서 애플리케이션 서버에 상주할 수 있습니다.

이 장에서는 프로바이더 컴포넌트를 사용하여 클라이언트 데이터셋 또는 XML 브로커와의 상호 작용을 제어하는 방법에 대해 설명합니다.

데이터의 소스 정하기

프로바이더 컴포넌트를 사용할 때 데이터 패킷으로 조합되는 데이터를 얻는 데 사용할 소스를 지정해야 합니다. Delphi의 버전에 따라 소스를 다음 중 하나로 지정할 수 있습니다.

- 데이터셋으로부터 데이터를 제공하려면 *TDataSetProvider*를 사용합니다.
- XML 문서로부터 데이터를 제공하려면 *TXMLTransformProvider*를 사용합니다.

데이터 소스로 데이터셋 사용

프로바이더가 데이터셋 프로바이더 (*TDataSetProvider*) 인 경우, 프로바이더의 *DataSet* 속성을 설정하여 소스 데이터셋을 나타냅니다. 디자인 타임 시 Object Inspector의 *DataSet* 속성 드롭다운 목록에 있는 사용 가능한 데이터셋 중에서 선택합니다.

*TDataSetProvider*는 *IProviderSupport* 인터페이스를 사용하여 소스 데이터셋과 상호 작용합니다. 이 인터페이스는 *TDataSet*에 포함되어 있기 때문에 모든 데이터셋에서 사용할 수 있습니다. 그러나 *TDataSet*에서 구현된 *IProviderSupport* 메소드는 대개 아무 작업도 하지 않거나 예외를 일으키는 스텝(stub)입니다.

Delphi와 함께 출시된 데이터셋 클래스(BDE 사용 데이터셋, ADO 사용 데이터셋, *dbExpress* 데이터셋 및 *InterBase Express* 데이터셋)는 이러한 메소드를 오버라이드하여 *IProviderSupport* 인터페이스를 보다 유용하게 구현합니다. 클라이언트 데이터셋은 상속된 *IProviderSupport* 구현에 어떤 것도 추가하지 않지만 프로바이더의 *ResolveToDataSet* 속성이 *True*인 한, 여전히 소스 데이터셋으로 사용될 수 있습니다.

*TDataSet*으로부터 고유한 사용자 지정 자손을 만드는 컴포넌트 작성자는 자신의 데이터셋이 프로바이더에 데이터를 공급할 경우 모든 해당 *IProviderSupport* 메소드를 오버라이드해야 합니다. 프로바이더가 단지 읽기 전용으로 데이터 패킷을 공급해야 할 경우, 즉 업데이트를 적용할 필요가 없을 경우 *TDataSet*에서 구현되는 *IProviderSupport* 메소드로 충분할 것입니다.

데이터 소스로 XML 문서 사용

프로바이더가 XML 프로바이더인 경우, 프로바이더의 *XMLDataFile* 속성을 설정하여 소스 문서를 나타냅니다.

XML 프로바이더는 소스 문서를 데이터 패킷으로 변환하고 소스 문서를 나타내는 것과 더불어 사용자는 또한 소스 문서를 데이터 패킷으로 변환하는 방법을 지정해야 합니다. 이 변환은 프로바이더의 *TransformRead* 속성에 의해 처리됩니다. *TransformRead*는 *TXMLTransform* 객체를 나타냅니다. 사용자는 프로바이더의 속성을 설정하여 사용할 변환을 지정하고, 프로바이더의 이벤트를 사용하여 변환에 고유한 입력을 제공할 수

있습니다. XML 프로바이더 사용에 대한 내용은 26-8 페이지의 "XML 문서를 프로바이더의 소스로 사용"을 참조하십시오.

클라이언트 데이터셋과의 통신

프로바이더와 클라이언트 데이터셋이나 XML 브로커 간의 모든 통신은 *IAppServer* 인터페이스를 통해 이루어집니다. 프로바이더가 클라이언트와 동일한 애플리케이션에 있는 경우, 이 인터페이스는 사용자를 위해 자동으로 생성된 숨겨진 객체나 *TLocalConnection* 컴포넌트에 의해 구현됩니다. 프로바이더가 다계층 애플리케이션의 일부인 경우에는 애플리케이션 서버의 원격 데이터 모듈을 위한 인터페이스가 됩니다.

대부분의 애플리케이션은 *IAppServer*를 직접 사용하지 않지만 클라이언트 데이터셋의 속성 및 메소드를 통해 간접적으로 호출합니다. 그러나 필요한 경우 클라이언트 데이터셋의 *AppServer* 속성을 사용하여 *IAppServer* 인터페이스를 직접 호출할 수 있습니다.

표 24.1에는 *IAppServer* 인터페이스의 메소드와 함께 프로바이더 컴포넌트와 클라이언트 데이터셋의 해당 메소드 및 이벤트가 나열되어 있습니다. 이러한 *IAppServer* 메소드에는 *Provider* 매개변수가 포함됩니다. 다계층 애플리케이션에서 이 매개변수는 클라이언트 데이터셋과 서로 통신하는 애플리케이션 서버 상의 프로바이더를 나타냅니다. 또한 대부분의 메소드에는 *OwnerData*라고 하는 *OleVariant* 매개변수가 포함됩니다. 이 매개변수는 클라이언트 애플리케이션과 애플리케이션 서버가 사용자 지정 정보를 앞뒤로 전달할 수 있게 해줍니다. *OwnerData*는 기본으로 사용되지 않지만 모든 이벤트 핸들러에 전달됩니다. 따라서 사용자는 프로바이더가 클라이언트 데이터셋에서 각 호출을 받기 전후에 애플리케이션 정의 정보를 조정할 수 있는 코드를 작성할 수 있습니다.

표 24.1 AppServer 인터페이스 멤버

IAppServer	프로바이더 컴포넌트	TClientDataSet
AS_ApplyUpdates 메소드	ApplyUpdates 메소드, BeforeApplyUpdates 이벤트, AfterApplyUpdates 이벤트	ApplyUpdates 메소드, BeforeApplyUpdates 이벤트, AfterApplyUpdates 이벤트
AS_DataRequest 메소드	DataRequest 메소드, OnDataRequest 이벤트	DataRequest 메소드
AS_Execute 메소드	Execute 메소드, BeforeExecute 이벤트, AfterExecute 이벤트	Execute 메소드, BeforeExecute 이벤트, AfterExecute 이벤트
AS_GetParams 메소드	GetParams 메소드, BeforeGetParams 이벤트, AfterGetParams 이벤트	FetchParams 메소드, BeforeGetParams 이벤트, AfterGetParams 이벤트
AS_GetProviderNames 메소드	사용 가능한 모든 프로바이더를 식별하는 데 사용	ProviderName 속성에 대한 디자인 타임 목록을 만드는 데 사용

표 24.1 AppServer 인터페이스 멤버 (계속)

IAppServer	프로바이더 컴포넌트	TClientDataSet
AS_GetRecords 메소드	GetRecords 메소드, BeforeGetRecords 이벤트, AfterGetRecords 이벤트	GetNextPacket 메소드, Data 속성, BeforeGetRecords 이벤트, AfterGetRecords 이벤트
AS_RowRequest 메소드	RowRequest 메소드, BeforeRowRequest 이벤트, AfterRowRequest 이벤트	FetchBlobs 메소드, FetchDetails 메소드, RefreshRecord 메소드, BeforeRowRequest 이벤트, AfterRowRequest 이벤트

데이터셋 프로바이더를 사용하여 업데이트 사항의 적용 방법 선택

TXMLTransformProvider 컴포넌트는 항상 업데이트 사항을 연결된 XML 문서에 적용합니다. 그러나 *TDataSetProvider*를 사용하면 업데이트 사항을 적용하는 방법을 선택할 수 있습니다. 기본적으로 *TDataSetProvider* 컴포넌트는 업데이트를 적용하고 업데이트 오류를 해결할 때 동적으로 생성되는 SQL 문을 사용하여 데이터베이스 서버와 직접 통신합니다. 이러한 접근 방법은 서버 애플리케이션에서 업데이트 사항을 두 번 통합(먼저 데이터셋에 통합하고 나서 원격 서버에 통합)할 필요가 없도록 하는 이점이 있습니다.

그러나 이 접근 방법을 사용하지 않을 수도 있습니다. 예를 들어, 데이터셋 컴포넌트에 일부 이벤트를 사용하고자 할 수도 있습니다. 또 다른 방법은 사용할 데이터셋이 SQL 문의 사용을 지원하지 않는 것입니다(예를 들어, *TClientDataSet* 컴포넌트로부터 제공 하는 경우).

*TDataSetProvider*는 SQL을 사용하여 데이터베이스 서버에 업데이트를 적용할 것인지, *ResolveToDataSet* 속성을 설정하여 소스 데이터셋에 업데이트를 적용할 것인지 결정할 수 있게 해줍니다. 이 속성이 *True*일 경우 업데이트 내용이 데이터셋에 적용되고 이 속성이 *False*일 경우 업데이트 사항이 원본으로 사용한 데이터베이스 서버에 직접 적용됩니다.

데이터 패킷에 포함될 정보 제어

데이터셋 프로바이더를 사용할 때 클라이언트로 보내고 받는 데이터 패킷에 들어 있는 정보를 제어하는 방법에는 여러 가지가 있습니다. 다음과 같은 방법이 있습니다.

- 데이터 패킷에 표시되는 필드 지정
- 데이터 패킷에 영향을 주는 옵션 설정
- 데이터 패킷에 사용자 지정 정보 추가

참고 데이터 패킷의 내용을 제어하는 기술은 데이터셋 프로바이더에서만 사용 가능합니다. *TXMLTransformProvider*를 사용할 때 프로바이더가 사용하는 변환 파일을 제어함으로써 단지 데이터 패킷의 내용만 제어할 수 있습니다.

데이터 패킷에 표시되는 필드 지정

데이터셋 프로바이더를 사용하는 경우, 프로바이더가 데이터를 구축하는 데 사용하는 데이터셋에 영구적 필드를 생성하여 데이터 패킷에 포함된 필드를 제어할 수 있습니다. 그러면 프로바이더는 이러한 영구적 필드만 포함합니다. 계산된 필드나 조회 필드와 같은 소스 데이터셋에 의해 그 값이 동적으로 생성되는 영구적 필드는 데이터 패킷에 포함될 수 있지만 맨 마지막으로 받는 클라이언트 데이터셋에 대해 정적 읽기 전용 필드로 표시됩니다. 영구적 필드에 대한 자세한 내용은 19-3 페이지의 "영구적(Persistent) 필드 컴포넌트"를 참조하십시오.

클라이언트 데이터셋이 데이터를 편집하고 업데이트를 적용하는 경우에는 데이터 패킷에 중복된 레코드가 존재하지 않도록 충분한 필드를 포함해야 합니다. 그렇지 않으면 업데이트가 적용될 때 업데이트할 레코드를 결정할 수 없습니다. 단지 고유성을 보장하기 위해 제공되는 추가 필드를 클라이언트 데이터셋이 확인 및 사용할 수 없도록 하려면 이러한 필드의 *ProviderFlags* 속성을 설정하여 *pfHidden*을 포함시킵니다.

참고 중복 레코드를 방지하기 위해 충분한 필드를 포함하는 것은 프로바이더의 소스 데이터셋이 쿼리를 나타낼 때에도 고려해야 할 사항입니다. 애플리케이션이 필드 모두를 사용하지 않더라도 모든 레코드가 고유하다는 것을 보장하기에 충분한 필드를 포함하기 위해서 쿼리를 지정해야 합니다.

데이터 패킷에 영향을 미치는 옵션 설정

데이터셋 프로바이더의 *Options* 속성을 사용하면 BLOB 또는 중첩 디테일 테이블을 전달하는 시기, 필드 표시 속성의 포함 여부, 허용할 업데이트 유형 등을 지정할 수 있습니다. 다음 표는 *Options*에 포함할 수 있는 값을 나열한 것입니다.

표 24.2 프로바이더 옵션

값	의미
poAutoRefresh	프로바이더는 업데이트를 적용할 때마다 클라이언트 데이터셋을 현재 레코드 값으로 새로 고칩니다.
poReadOnly	클라이언트 데이터셋은 프로바이더에 업데이트를 적용할 수 없습니다.
poDisableEdits	클라이언트 데이터셋은 기존 데이터 값을 수정할 수 없습니다. 사용자가 필드를 편집하려고 하면 클라이언트 데이터셋은 예외를 일으킵니다. 그러나 레코드를 삽입하거나 삭제하는 클라이언트 데이터셋의 기능에는 영향을 주지 않습니다.
poDisableInserts	클라이언트 데이터셋은 새 레코드를 삽입할 수 없습니다. 사용자가 새 레코드를 삽입하려고 하면 클라이언트 데이터셋은 예외를 일으킵니다. 그러나 레코드를 삭제하거나 기존 데이터를 수정하는 클라이언트 데이터셋의 기능에는 영향을 주지 않습니다.
poDisableDeletes	클라이언트 데이터셋은 레코드를 삭제할 수 없습니다. 사용자가 레코드를 삭제하려고 하면 클라이언트 데이터셋은 예외를 일으킵니다. 그러나 레코드를 삽입하거나 수정하는 클라이언트 데이터셋의 기능에는 영향을 주지 않습니다.

표 24.2 프로바이더 옵션 (계속)

값	의미
poFetchBlobsOnDemand	BLOB 필드 값이 데이터 패킷에 포함되지 않습니다. 그 대신 클라이언트 데이터셋은 이러한 값을 필요에 따라 요청해야 합니다. 클라이언트 데이터셋의 <i>FetchOnDemand</i> 속성이 <i>True</i> 이면 이러한 값을 자동으로 요청합니다. 그렇지 않으면 애플리케이션이 클라이언트 데이터셋의 <i>FetchBlobs</i> 메소드를 호출하여 BLOB 데이터를 검색해야 합니다.
poFetchDetailsOnDemand	프로바이더의 데이터셋이 마스터/디테일 관계의 마스터를 나타 내면 중첩 디테일 값이 데이터 패킷에 포함되지 않습니다. 그 대신 클라이언트 애플리케이션은 이러한 값을 필요에 따라 요청합니다. 클라이언트 데이터셋의 <i>FetchOnDemand</i> 속성이 <i>True</i> 이면 이러한 값을 자동으로 요청합니다. 그렇지 않으면 애플리케이션이 클라이언트 데이터셋의 <i>FetchDetails</i> 메소드를 호출하여 중첩 디테일을 검색해야 합니다.
poIncFieldProps	데이터 패킷은 해당 사항이 있을 경우 다음과 같은 필드 속성을 포함합니다. <i>Alignment, DisplayLabel, DisplayWidth, Visible, DisplayFormat, EditFormat, MaxValue, MinValue, Currency, EditMask, DisplayValues</i>
poCascadeDeletes	프로바이더의 데이터셋이 마스터/디테일 관계의 마스터를 나타 내면 서버는 마스터 레코드가 삭제될 때 디테일 레코드를 삭제합니다. 이 옵션을 사용하려면 참조 무결성의 일부로 연쇄 삭제를 수행하도록 데이터베이스 서버를 설정해야 합니다.
poCascadeUpdates	프로바이더의 데이터셋이 마스터/디테일 관계의 마스터를 나타 내면 마스터 레코드에서 해당 값이 바뀔 때 디테일 테이블의 키 값이 자동으로 업데이트됩니다. 이 옵션을 사용하려면 참조 무결성의 일부로 연쇄 업데이트를 수행하도록 데이터베이스 서버를 설정해야 합니다.
poAllowMultiRecordUpdates	단일 업데이트로 인해 원본으로 사용하는 데이터베이스 테이블에서 둘 이상의 레코드가 변경될 수 있습니다. 이는 트리거, 참조 무결성, 소스 데이터셋의 SQL 문 등의 결과일 수 있습니다. 오류가 발생할 경우 이벤트 핸들러는 결과적으로 변경되는 다른 레코드가 아니라 업데이트된 레코드에 대한 액세스를 제공한다는 점에 유의하십시오.
poNoReset	클라이언트 데이터셋은 데이터를 제공하기 전에 프로바이더가 첫 번째 레코드에 커서를 다시 위치시키도록 지정할 수 없습니다.
poPropogateChanges	서버가 업데이트 프로세스의 일부로 업데이트된 레코드를 변경한 내용은 클라이언트로 다시 보내져 클라이언트 데이터셋에 병합됩니다.
poAllowCommandText	클라이언트는 연결된 데이터셋의 SQL 텍스트 또는 자신이 나타 내는 테이블 또는 내장 프로시저의 이름을 오버라이드할 수 있습니다.
poRetainServerOrder	클라이언트 데이터셋은 기본 순서를 강제로 실행하기 위해서 데이터셋의 레코드를 재분류하지 않아야 합니다.

데이터 패킷에 사용자 지정 정보 추가

데이터셋 프로바이더는 *OnGetDataSetProperties* 이벤트를 사용하여 애플리케이션 정의 정보를 데이터 패킷에 추가할 수 있습니다. 이 정보는 OleVariant로 인코딩되고 사용자가 지정한 이름으로 저장됩니다. 그러면 클라이언트 데이터셋은 *GetOptionalParam* 메소드를 사용하여 정보를 검색할 수 있습니다. 또한 사용자는 레코드를 업데이트할 때 클라이언트 데이터셋이 보내는 델타 패킷에 정보가 포함되도록 지정할 수 있습니다. 이 경우 클라이언트 데이터셋은 정보를 인식하지 못하지만 프로바이더는 자신에게 왕복 메시지를 보낼 수 있습니다.

OnGetDataSetProperties 이벤트에서 사용자 지정 정보를 추가할 때 세 개 요소를 포함한다면 배열을 사용하여 "옵션 매개변수"라고도 하는 각각의 개별 속성을 지정합니다. 여기서 세 개 요소는 이름(문자열), 값(가변) 그리고 클라이언트가 업데이트를 적용할 때 델타 패킷에 정보가 포함되는지 여부를 나타내는 부울 플래그를 말합니다. 가변 배열에 대한 가변 배열을 만들어 여러 속성을 추가합니다. 예를 들어, 다음 *OnGetDataSetProperties* 이벤트 핸들러는 두 개의 값, 즉 데이터가 제공된 시간과 소스 데이터 셋의 총 레코드 수를 보냅니다. 클라이언트 데이터셋이 업데이트를 적용할 경우 데이터가 제공된 시간만 반환됩니다.

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender:TObject;
DataSet:TDataSet; out Properties:OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

클라이언트 데이터셋이 업데이트를 적용할 때 프로바이더의 *OnUpdateData* 이벤트에서 원래 레코드가 제공된 시간을 읽을 수 있습니다.

```
procedure TMyDataModule1.Provider1UpdateData(Sender:TObject;
DataSet:TCustomClientDataSet);
var
  WhenProvided:TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

클라이언트 데이터 요청에 대한 응답

일반적으로 데이터에 대한 클라이언트 요청은 자동으로 처리됩니다. 클라이언트 데이터셋이나 XML 브로커는 *GetRecords*를 호출하여 간접적으로는 *IAppServer* 인터페이스를 통해 데이터 패킷을 요청합니다. 프로바이더는 연결된 데이터셋이나 XML 문서에서 데이터를 폐치하여 데이터 패킷을 만든 다음 패킷을 클라이언트로 보내어 자동으로 응답합니다.

프로바이더는 데이터 패킷을 만든 후 아직 클라이언트에 패킷이 전달되지 않은 상태에서 데이터를 편집할 수 있는 옵션을 갖고 있습니다. 예를 들어, 사용자의 액세스 레벨과

같은 몇 가지 기준에 기초하여 패킷에서 레코드를 제거하거나 다계층 애플리케이션의 경우 클라이언트에 보내기 전에 중요한 데이터를 암호화할 수 있습니다.

클라이언트 데이터셋에 보내기 전에 데이터 패킷을 편집하려면 *OnGetData* 이벤트 핸들러를 작성합니다. *OnGetData* 이벤트 핸들러는 클라이언트 데이터셋의 폼에 매개변수로서 데이터 패킷을 공급합니다. 이 클라이언트 데이터셋의 메소드를 사용하면 클라이언트에 보내기 전에 데이터를 편집할 수 있습니다.

IAppServer 인터페이스를 통한 모든 메소드 호출에 의해 프로바이더는 *GetRecords*에 대한 호출 전후에 영구적 상태 정보를 클라이언트 데이터셋과 통신할 수 있습니다. 이 통신은 *BeforeGetRecords* 및 *AfterGetRecords* 이벤트 핸들러를 통해 이루어집니다. 애플리케이션 서버의 영구적 상태 정보에 대한 설명은 25-20 페이지의 "원격 데이터 모듈의 상태 정보 지원"을 참조하십시오.

클라이언트 업데이트 요청에 대한 응답

프로바이더는 클라이언트 데이터셋이나 XML 브로커에서 받은 *델타* 데이터 패킷에 기초하여 데이터베이스 레코드에 업데이트를 적용합니다. 클라이언트는 *IAppServer* 인터페이스를 통해 간접적으로 *ApplyUpdates* 메소드를 호출하여 업데이트를 요청합니다.

IAppServer 인터페이스를 통한 모든 메소드 요청에 의해 프로바이더는 *ApplyUpdates*에 대한 요청 전후에 영구적 상태 정보를 클라이언트 데이터셋과 통신할 수 있습니다. 이 통신은 *BeforeApplyUpdates* 및 *AfterApplyUpdates* 이벤트 핸들러를 통해 이루어집니다. 애플리케이션 서버의 영구적 상태 정보에 대한 내용은 25-20 페이지의 "원격 데이터 모듈의 상태 정보 지원"을 참조하십시오.

데이터셋 프로바이더를 사용하는 경우, 수많은 추가 이벤트를 통해 다음과 같이 보다 많은 제어를 할 수 있습니다.

업데이트 요청을 받으면 데이터셋 프로바이더는 *OnUpdateData* 이벤트를 생성합니다. 여기서 사용자는 데이터셋에 기록되거나 업데이트 적용 방법에 영향을 주기 전에 *델타* 패킷을 편집할 수 있습니다. *OnUpdateData* 이벤트 이후 프로바이더는 변경된 내용을 데이터베이스 또는 소스 데이터셋에 기록합니다.

프로바이더는 레코드 단위로 업데이트를 수행합니다. 데이터셋 프로바이더는 각 레코드를 적용하기 전에 *BeforeUpdateRecord* 이벤트를 생성하고 사용자는 이 이벤트를 사용하여 업데이트가 적용되기 전에 스크린할 수 있습니다. 레코드를 업데이트할 때 오류가 발생하면 프로바이더는 오류를 해결할 수 있는 *OnUpdateError* 이벤트를 받습니다. 변경 내용이 서버 제약 조건을 위반하거나 프로바이더가 검색한 이후 클라이언트 데이터셋이 업데이트 적용을 요청하기 전에 데이터베이스 레코드가 다른 애플리케이션에 의해 변경된 경우에 대체로 오류가 발생합니다.

업데이트 오류는 데이터셋 프로바이더 또는 클라이언트 데이터셋에 의해 처리될 수 있습니다. 다계층 애플리케이션의 일부인 경우 프로바이더는 해결을 위해 사용자 상호 작용이 필요하지 않은 모든 업데이트 오류를 처리해야 합니다. 오류 조건을 해결할 수 없으면 프로바이더는 오류를 유발한 레코드의 복사본을 임시로 저장합니다. 레코드 처리가 완료되면 프로바이더는 발생한 오류 개수를 클라이언트 데이터셋에 반환하고 해결

되지 않은 레코드를 결과 데이터 패킷에 복사하여 나중에 조정할 수 있도록 클라이언트 데이터셋에 반환합니다.

모든 프로바이더 이벤트에 대한 이벤트 핸들러는 클라이언트 데이터셋으로서 업데이트 집합이 전달됩니다. 이벤트 핸들러가 특정 유형의 업데이트만 처리할 경우 레코드의 업데이트 상태에 기초하여 데이터셋을 필터링할 수 있습니다. 레코드를 필터링하면 이벤트 핸들러는 사용하지 않는 레코드를 정렬할 필요가 없어집니다. 레코드의 업데이트 상태에 따라 클라이언트 데이터셋을 필터링하려면 클라이언트 데이터셋의 *StatusFilter* 속성을 설정합니다.

참고 단일 테이블을 나타내지 않는 데이터셋에 업데이트를 적용할 경우 애플리케이션은 추가 지원을 제공해야 합니다. 이에 대한 자세한 내용은 24-12 페이지의 "단일 테이블을 나타내지 않는 데이터셋에 업데이트 적용"을 참조하십시오.

데이터베이스 업데이트 이전의 델타 패킷 편집

데이터셋 프로바이더는 데이터베이스에 업데이트를 적용하기 전에 *OnUpdateData* 이벤트를 생성합니다. *OnUpdateData* 이벤트 핸들러는 매개변수로 *델타* 패킷의 복사본을 받습니다. 이것은 클라이언트 데이터셋입니다.

OnUpdateData 이벤트 핸들러에서는 클라이언트 데이터셋의 모든 속성 및 메소드를 사용하여 데이터셋에 기록되기 전에 *델타* 패킷을 편집할 수 있습니다. 여기서 특히 유용한 속성 하나가 *UpdateStatus* 속성입니다. *UpdateStatus*는 델타 패킷의 현재 레코드가 나타내는 수정 유형을 표시하며 표 24.3에 있는 값들을 지정할 수 있습니다.

표 24.3 UpdateStatus 값

값	설명
usUnmodified	레코드 내용이 변경되지 않았습니다.
usModified	레코드 내용이 변경되었습니다.
usInserted	레코드가 삽입되었습니다.
usDeleted	레코드가 삭제되었습니다.

예를 들어, 다음과 같은 *OnUpdateData* 이벤트 핸들러는 데이터베이스에 삽입되는 모든 새 레코드에 현재 데이터를 삽입합니다.

```

procedure TMyDataModule1.Provider1UpdateData(Sender:TObject;
DataSet:TCustomClientDataSet);
begin
  with DataSet do
    begin
      First;
      while not Eof do
        begin
          if UpdateStatus = usInserted then
            begin
              Edit;
              FieldByName('DateCreated').AsDateTime := Date;
              Post;
            end;
          end;
        end;
    end;

```

```

Next;
end;
end;

```

업데이트 적용 방법 변경

OnUpdateData 이벤트는 또한 데이터셋 프로바이더가 데이터베이스에 델타 패킷 레코드가 적용되는 방법을 나타내게 해줍니다.

기본적으로 델타 패킷의 변경 내용은 다음과 같이 자동으로 생성된 SQL UPDATE, INSERT 또는 DELETE 문을 통해 데이터베이스에 기록됩니다.

```

UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47

```

따로 지정하지 않으면 델타 패킷 레코드의 모든 필드가 UPDATE 절과 WHERE 절에 포함됩니다. 그러나 이러한 필드의 일부를 제외할 수 있습니다. 일부 필드를 제외하는 방법 중 하나는 프로바이더의 *UpdateMode* 속성을 설정하는 것입니다. *UpdateMode* 에는 다음 값이 지정될 수 있습니다.

표 24.4 UpdateMode 값

값	의미
upWhereAll	모든 필드를 사용하여 레코드를 찾습니다(WHERE 절).
upWhereChanged	키 필드와 변경된 필드만 사용하여 레코드를 찾습니다.
upWhereKeyOnly	키 필드만 사용하여 레코드를 찾습니다.

단지 일부 필드를 제외하는 것 외에 더 세부적으로 제어할 수도 있습니다. 예를 들어, 앞의 문장에서 EMPNO 필드를 UPDATE 절에서 제외하여 수정을 방지하고 TITLE 및 DEPT 필드를 WHERE 절에서 제외하여 다른 애플리케이션이 데이터를 수정했을 때 업데이트 충돌을 피할 수 있습니다. 특정 필드가 표시되는 절을 지정하려면 *ProviderFlags* 속성을 사용합니다. *ProviderFlags*는 표 24.5에 있는 값을 포함할 수 있는 집합입니다.

표 24.5 ProviderFlags 값

값	설명
pfInWhere	<i>UpdateMode</i> 가 <i>upWhereAll</i> 또는 <i>upWhereChanged</i> 일 경우 생성된 INSERT, DELETE 및 UPDATE 문의 WHERE 절에 필드가 표시됩니다.
pfInUpdate	생성된 UPDATE 문의 UPDATE 절에 필드가 표시될 수 있습니다.
pfInKey	<i>UpdateMode</i> 가 <i>upWhereKeyOnly</i> 일 경우 생성된 문의 WHERE 절에 필드가 사용됩니다.
pfHidden	고유성을 보장하기 위해 레코드에 필드가 포함되지만 클라이언트측에서는 필드를 보거나 사용할 수 없습니다.

따라서 다음과 같은 *OnUpdateData* 이벤트 핸들러는 TITLE 필드의 업데이트를 허용하고 EMPNO 및 DEPT 필드를 사용하여 원하는 레코드를 찾습니다. 오류가 발생하고 키

필드에만 기초하여 레코드를 찾는 두 번째 시도가 이루어질 경우 생성된 SQL은 EMPNO 필드만 검색합니다.

```
procedure TMyDataModule1.Provider1UpdateData(Sender:TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('TITLE').ProviderFlags := [pfInUpdate];
    FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
    FieldByName('DEPT').ProviderFlags := [pfInWhere];
  end;
end;
```

참고 동적으로 생성된 SQL을 사용하지 않고 데이터셋에 직접 업데이트를 시도하는 경우에도 *UpdateFlags* 속성을 사용하여 업데이트가 적용되는 방법에 영향을 줄 수 있습니다. 이러한 플래그는 레코드를 찾기 위해 사용되는 필드와 업데이트할 필드를 결정합니다.

개별적인 업데이트(Individual updates) 스크린

프로바이더는 각 업데이트가 적용되기 바로 전에 *BeforeUpdateRecord* 이벤트를 받습니다. 이 이벤트를 사용하면 *OnUpdateData* 이벤트를 사용하여 전체 델타 패킷을 편집하는 것과 비슷한 방법으로 레코드를 적용하기 전에 편집할 수 있습니다. 예를 들어, 프로바이더는 업데이트 충돌을 검사할 때 메모와 같은 BLOB 필드를 비교하지 않습니다. BLOB 필드를 포함한 업데이트 오류를 검사하기 위해 *BeforeUpdateRecord* 이벤트를 사용할 수 있습니다.

또한 이 이벤트를 사용하면 업데이트를 적용하거나 스크린하거나 거부할 수 있습니다. *BeforeUpdateRecord* 이벤트 핸들러는 업데이트가 이미 처리되었기 때문에 적용되어서는 안 된다는 사실을 알려 줍니다. 이 경우 프로바이더는 해당 레코드를 건너뛰지만 업데이트 오류로 간주하지는 않습니다. 예를 들어, 이 이벤트는 이벤트 핸들러에서 레코드가 업데이트된 경우 프로바이더가 자동 처리를 건너뛰도록 하면서 자동으로 업데이트할 수 없는 내장 프로시저에 업데이트를 적용할 수 있는 메커니즘을 제공합니다.

프로바이더에서 업데이트 오류 해결

프로바이더가 델타 패킷에서 레코드를 포스트할 때 오류 조건이 발생하면 *OnUpdateError* 이벤트가 발생합니다. 업데이트 오류를 해결할 수 없는 경우 프로바이더는 오류를 유발한 레코드의 복사본을 임시로 저장합니다. 레코드 처리가 완료되면 프로바이더는 발생한 오류 개수를 반환하고 해결되지 않는 레코드를 결과 데이터 패킷에 복사하여 나중에 조정할 수 있도록 클라이언트에 다시 전달합니다.

다계층 애플리케이션에서 이 메커니즘은 오류 조건 수정을 위한 클라이언트 애플리케이션의 사용자 상호 작용을 계속 허용하면서 애플리케이션 서버에서 메커니즘적으로 해결할 수 있는 모든 업데이트 오류를 처리할 수 있게 해 줍니다.

OnUpdateError 핸들러는 변경할 수 없는 레코드 복사본과 데이터베이스의 오류 코드를 가지며 해결 프로그램이 레코드를 삽입, 삭제 또는 업데이트하려고 했는지 여부를 표시합니다. 문제 레코드는 클라이언트 데이터셋에 다시 전달됩니다. 클라이언트 데이터

셋에서는 데이터 탐색 메소드를 전혀 사용하지 않지만 이 데이터셋의 각 필드에 대해서는 *NewValue*, *OldValue* 및 *CurValue* 속성을 사용하여 문제 원인을 확인하고 필요한 내용을 수정하여 업데이트 오류를 해결할 수 있습니다. *OnUpdateError* 이벤트 핸들러는 문제를 수정할 수 있는 경우 *Response* 매개변수를 설정하여 수정된 레코드를 적용합니다.

단일 테이블을 나타내지 않는 데이터셋에 업데이트 적용

데이터셋 프로바이더가 데이터베이스 서버에 직접 업데이트를 적용하는 SQL 문을 생성할 때 레코드를 포함하는 데이터베이스 테이블의 이름이 필요합니다. 테이블 다입 데이터셋과 같은 많은 데이터셋이나 "라이브" *TQuery* 컴포넌트에 대해 자동으로 처리할 수 있습니다. 그러나 자동 업데이트는 프로바이더가 결과 집합이나 다중 테이블 쿼리를 가지는 내장 프로시저의 밑에 있는 데이터에 업데이트를 적용해야 하는 경우에 문제가 됩니다. 즉, 이 경우 업데이트를 적용해야 할 테이블의 이름을 쉽게 확인할 수 없습니다.

쿼리나 내장 프로시저가 BDE 사용 데이터셋 (*TQuery* 또는 *TStoredProc*)이고 연결된 업데이트 객체가 있는 경우, 프로바이더는 업데이트 객체를 사용합니다. 그러나 업데이트 객체가 없는 경우, *OnGetTableName* 이벤트 핸들러의 프로그램으로 테이블 이름을 제공할 수 있습니다. 이벤트 핸들러가 테이블 이름을 제공하면 프로바이더는 적절한 SQL 문을 생성하여 업데이트를 적용할 수 있습니다.

업데이트 대상이 단일 데이터베이스 테이블인 경우에만, 즉 단일 테이블에 있는 레코드만 업데이트해야 할 경우에 테이블 이름이 제공됩니다. 원본으로 사용하는 여러 데이터베이스 테이블에서 변경이 필요한 업데이트의 경우에는 프로바이더의 *BeforeUpdateRecord* 이벤트를 사용하여 코드에서 업데이트를 명시적으로 적용해야 합니다. 이 이벤트 핸들러가 업데이트를 적용한 후에는 프로바이더가 오류를 생성하지 않도록 이벤트 핸들러의 *Applied* 매개변수를 *True*로 설정할 수 있습니다.

참고 프로바이더가 BDE 사용 데이터셋에 연결된 경우, *BeforeUpdateRecord* 이벤트 핸들러의 업데이트 객체를 사용하여 사용자 지정 SQL 문을 사용하는 업데이트를 적용할 수 있습니다. 자세한 내용은 20-40 페이지의 "업데이트 객체를 사용하여 데이터셋 업데이트"를 참조하십시오.

클라이언트 생성 이벤트에 응답

프로바이더 컴포넌트는 일반적인 용도의 이벤트, 즉 *OnDataRequest*를 구현하므로 사용자는 이 이벤트로 클라이언트 데이터셋에서 직접 프로바이더를 호출할 수 있습니다.

*OnDataRequest*는 프로바이더가 제공하는 일반적인 기능의 일부가 아니며 클라이언트 데이터셋이 프로바이더와 직접 통신할 수 있도록 해줍니다. 이벤트 핸들러는 *OleVariant*를 입력 매개변수로 가져와 *OleVariant*를 반환합니다. *OleVariant*를 사용하기 때문에 인터페이스는 매우 일반적이어서 사용자가 프로바이더와 주고 받고자 하는 거의 모든 정보를 수용할 수 있습니다.

OnDataRequest 이벤트를 생성하려면 클라이언트 애플리케이션이 클라이언트 데이터셋의 *DataRequest* 메소드를 요청합니다.

서버 제약 조건 처리

대부분의 관계 데이터베이스 관리 시스템들은 데이터 무결성을 강요하기 위해 테이블에 제약 조건을 구현합니다. 제약 조건은 테이블과 열의 데이터 값을 지배하거나 다양한 테이블에 있는 열 간의 데이터 관계를 지배하는 규칙입니다. 예를 들어, 대부분의 SQL-92 호환 관계 데이터베이스는 다음과 같은 제약 조건을 지원합니다.

- NOT NULL은 열에 있는 값이 값을 가지도록 보장합니다.
- NOT NULL UNIQUE는 열 값이 값을 가지고 또 다른 레코드의 열에 있는 다른 값은 중복하지 않도록 보장합니다.
- CHECK는 열에 있는 값이 어떤 범주에 속하거나 제한된 수의 가능한 값들 중 하나가 되도록 보장합니다.
- CONSTRAINT는 여러 열에 적용되는 테이블 범위의 check 제약 조건입니다.
- PRIMARY KEY는 인덱스 용도를 위해 사용되는 테이블의 기본 키로서 하나 이상의 열을 지정합니다.
- FOREIGN KEY는 또 다른 테이블을 참조하는 하나 이상의 테이블 열을 지정합니다.

참고 이 목록은 배타적이지 않습니다. 데이터베이스 서버는 부분적 또는 전체적으로 이러한 제약 조건들의 일부 또는 전체를 지원하고, 추가적인 제약 조건을 지원할 수도 있습니다. 지원되는 제약 조건에 대한 자세한 내용은 서버 설명서를 참조하십시오.

데이터베이스 서버 제약 조건은 전형적인 데스크탑 데이터베이스 애플리케이션이 관리하는 많은 데이터 확인 기능들을 확실하게 복제합니다. 애플리케이션이나 클라이언트 애플리케이션 코드에서 제약 조건을 복제할 필요 없이 다계층 데이터베이스 애플리케이션에서 서버 제약 조건을 이용할 수 있습니다.

프로바이더가 BDE 사용 데이터셋을 사용하는 경우, 사용자는 *Constraints* 속성을 이용하여 클라이언트 데이터셋에 송수신되는 데이터에 서버 제약 조건을 복제하고 적용할 수 있습니다. *Constraints*가 *True*(기본값)일 경우, 소스 데이터셋에 저장된 서버 제약 조건은 데이터 패킷에 포함되고 클라이언트의 데이터 업데이트 시도에 영향을 미칩니다.

중요 프로바이더가 제약 조건 정보를 클라이언트 데이터셋에 전달하기 전에 데이터베이스 서버에서 제약 조건을 복구해야 합니다. 서버에서 데이터베이스 제약 조건을 가져오려면 SQL Explorer를 사용하여 데이터베이스 서버의 제약 조건과 기본 표현식을 데이터 사전으로 가져옵니다. 데이터 사건의 제약 조건과 기본 표현식은 BDE 사용 데이터셋에서 자동적으로 사용할 수 있게 됩니다.

서버 제약 조건이 클라이언트 데이터셋에 전달된 데이터에 적용되기를 원하지 않는 경우가 있을 수도 있습니다. 예를 들어, 패킷 데이터를 받아 보다 많은 레코드를 폐치하기 전에 레코드의 지역적 업데이트를 허용하는 클라이언트 데이터셋은 일시적으로 불완전한 데이터 집합이기 때문에 트리거될지도 모르는 일부 서버 제약 조건을 사용 불가능하게 해야 할 수도 있습니다. 프로바이더로부터 클라이언트 데이터셋으로 제약 조건이 복제되는 것을 방지하려면 *Constraints*를 *False*로 설정합니다. 클라이언트 데이터셋이 *DisableConstraints* 및 *EnableConstraints* 메소드를 사용하여 제약 조건을 사용 가능 또는 사용 불가능하게 할 수 있다는 점에 유의하십시오. 클라이언트 데이터셋의 제약 조건

서버 제약 조건 처리

사용 가능 및 불가능에 대한 자세한 내용은 23-30 페이지의 "서버에서 제약 조건 처리"를 참조하십시오.

25

다계층(multi-tiered) 애플리케이션 생성

이 장에서는 다계층 클라이언트/서버 데이터베이스 애플리케이션을 만드는 방법에 대해 설명합니다. 다계층 클라이언트/서버 애플리케이션은 계층이라는 논리 단위로 분할되어 개별 시스템 상에서 결합하여 실행됩니다. 다계층 애플리케이션은 LAN 또는 인터넷을 통해 서로 데이터를 공유하고 통신합니다. 다계층 애플리케이션은 중앙 집중화된 비즈니스 논리 및 썬 클라이언트 애플리케이션과 같은 많은 이점을 제공합니다.

경우에 따라서 간단히 "3계층 모델(three-tiered model)"이라고 불리는 다계층 애플리케이션은 다음 세 가지로 분할됩니다.

- **클라이언트 애플리케이션.** 사용자의 컴퓨터에서 사용자 인터페이스를 제공합니다.
- **애플리케이션 서버.** 모든 클라이언트에 액세스할 수 있는 중앙 네트워킹 위치에 상주하며 일반적인 데이터 서비스를 제공합니다.
- **원격 데이터베이스 서버.** 관계형 데이터베이스 관리 시스템(RDBMS)을 제공합니다.

이러한 3계층 모델에서 애플리케이션 서버는 클라이언트와 원격 데이터베이스 서버 간의 데이터의 흐름을 관리하므로 종종 "데이터 브로커"라고도 합니다. 사용자 스스로 자체 데이터베이스 백엔드(back end)를 생성할 수도 있지만 Delphi에서는 일반적으로 애플리케이션 서버와 그 클라이언트만 생성합니다.

복잡한 다계층 애플리케이션에서는 추가 서비스가 클라이언트와 원격 데이터베이스 서버 간에 상주합니다. 예를 들어, 안전한 인터넷 트랜잭션을 처리하는 보안 서비스 브로커나 다른 플랫폼의 데이터베이스와의 데이터 공유를 처리하는 브리지 서비스가 있을 수도 있습니다.

다계층 애플리케이션 개발에 대한 Delphi의 지원은 클라이언트 데이터셋이 전송 가능한 데이터 패킷을 사용하여 프로바이더 컴포넌트와 통신하는 방법의 확장입니다. 이 장에서는 3계층 데이터베이스 애플리케이션 생성에 초점을 맞춥니다. 3계층 애플리케이션 생성 및 관리 방법을 이해하면 사용자 필요에 맞게 추가 서비스 레이어를 생성하고 추가할 수 있습니다.

다계층 데이터베이스 모델의 이점

다계층 데이터베이스 모델은 데이터베이스 애플리케이션을 논리 단위로 분할합니다. 클라이언트 애플리케이션은 데이터 표시와 사용자 상호 작용에 초점을 맞출 수 있습니다. 이론적으로 클라이언트 애플리케이션에서는 데이터가 저장되거나 유지되는 방법을 모릅니다. 애플리케이션 서버(중간 계층)는 여러 클라이언트로부터의 요청과 업데이트를 조정하고 처리합니다. 또한 데이터셋 정의와 데이터베이스 서버와의 상호 작용에 대한 모든 세부 사항을 처리합니다.

이러한 다계층 모델의 이점은 다음과 같습니다.

- **공유된 중간 계층의 비즈니스 논리 캡슐화.** 서로 다른 클라이언트 애플리케이션들은 모두 동일한 중간 계층에 액세스합니다. 이를 통해 별도의 각 클라이언트 애플리케이션에 대해 비즈니스 룰을 복제해야 하는 중복성(및 유지 비용)을 피할 수 있습니다.
- **썬 클라이언트 애플리케이션.** 처리 작업을 중간 계층에 대해 더 많이 위임함으로써 클라이언트 애플리케이션을 가볍게 작성할 수 있습니다. 클라이언트 애플리케이션은 크기가 작을 뿐만 아니라 데이터베이스 연결 소프트웨어(예를 들어, Borland Database Engine 및 데이터베이스 서버의 클라이언트측 소프트웨어)를 설치, 구성 및 유지 관리할 필요가 없으므로 쉽게 배포할 수 있습니다. 썬 클라이언트 애플리케이션은 인터넷을 통해 배포할 수 있으므로 추가적인 유연성도 가지고 있습니다.
- **분산 데이터 처리.** 여러 컴퓨터에 애플리케이션의 작업을 분산시키면 로드 밸런싱으로 인해 성능이 향상되고, 서버 중단 시 여분의 시스템에서 작업을 대신할 수 있습니다.
- **보안 강화.** 민감하고 중요한 기능을 다른 액세스 제한을 갖는 계층으로 분리할 수 있습니다. 이는 유연하고 구성 가능한 보안 레벨을 제공합니다. 중간 계층은 민감한 자원에 대한 엔트리 포인트를 제한할 수 있으므로 액세스를 쉽게 제어할 수 있습니다. HTTP, CORBA 또는 COM+를 사용 중인 경우 지원되는 보안 모델을 이용할 수 있습니다.

프로바이더 기반 다계층 애플리케이션에 대한 이해

다계층 애플리케이션에 대한 Delphi의 지원은 컴포넌트 팔레트의 DataSnap 페이지와 Data Access 페이지에 있는 컴포넌트, New Items 대화 상자의 Multitier 페이지에 있는 마법사에 의해 생성된 원격 데이터 모듈을 사용합니다. 이들은 데이터를 전송 가능한 데이터 패킷으로 패키지화하는 프로바이더 컴포넌트의 기능에 기반하며 전송 가능한 델타 패킷으로서 받은 업데이트를 처리합니다.

다계층 애플리케이션에 필요한 컴포넌트는 표 25.1에서 설명합니다.

표 25.1 다계층 애플리케이션에 사용되는 컴포넌트

컴포넌트	설명
원격 데이터 모듈	클라이언트 애플리케이션에 들어 있는 모든 프로바이더에 액세스를 제공하기 위해 COM Automation 서버, SOAP 서버 또는 CORBA 서버로서 작동할 수 있는 특수한 데이터 모듈입니다. 애플리케이션 서버에서 사용됨.
프로바이더 컴포넌트	데이터 패킷을 생성하여 데이터를 제공하고 클라이언트 업데이트를 해결하는 데이터 브로커입니다. 애플리케이션 서버에서 사용됨.
클라이언트 데이터셋 컴포넌트	midas.dll 또는 midaslib.dcu를 사용하여 데이터 패킷에 저장된 데이터를 관리하는 특화된 데이터셋입니다. 클라이언트 데이터셋은 클라이언트 애플리케이션에 사용됩니다. 클라이언트 데이터셋은 로컬로 업데이트를 캐시하며 델타 패킷의 업데이트를 애플리케이션 서버에 적용합니다.
연결 컴포넌트	서버를 찾고, 연결을 만들고, <i>IAppServer</i> 인터페이스를 클라이언트 데이터셋에 사용 가능하게 해주는 컴포넌트 패밀리입니다. 각 연결 컴포넌트는 특정 통신 프로토콜을 사용하도록 특수화되었습니다.

프로바이더와 클라이언트 데이터셋 컴포넌트는 데이터 패킷으로 저장된 데이터셋을 관리하는 midas.dll 또는 midaslib.dcu를 필요로 합니다.(프로바이더는 애플리케이션 서버에 사용되고 클라이언트 데이터셋은 클라이언트 애플리케이션에 사용되므로 midas.dll을 사용하는 경우 애플리케이션 서버와 클라이언트 애플리케이션에 모두 배포해야 합니다.)

BDE 활성 데이터셋을 사용하는 경우 애플리케이션 서버는 데이터베이스 관리에 도움을 주고 다계층 애플리케이션의 어떤 레벨에서도 검사할 수 있도록 데이터 사전에 서버 제약 조건을 import하는 SQL 탐색기를 필요로 할 수도 있습니다.

참고 애플리케이션 서버를 배포하기 위해서는 서버 라이선스를 구입해야 합니다.

이러한 컴포넌트가 적용될 아키텍처에 대한 개요는 14-13 페이지의 "다계층 아키텍처 사용"에서 설명합니다.

3계층 애플리케이션 개요

다음의 같이 번호가 매겨진 단계는 프로바이더 기반 3계층 애플리케이션에 대한 이벤트의 일반적인 순서를 나타낸 것입니다.

- 1 사용자는 클라이언트 애플리케이션을 시작합니다. 클라이언트는 디자인 타임이나 런타임 시 지정할 수 있는 애플리케이션 서버에 연결합니다. 애플리케이션 서버가 아직 실행되고 있지 않은 상태라면 실행을 시작합니다. 클라이언트는 애플리케이션 서버에서 *IAppServer* 인터페이스를 받습니다.
- 2 클라이언트는 애플리케이션 서버의 데이터를 요청합니다. 클라이언트는 한 번에 모든 데이터를 요청하거나 세션에 걸쳐 데이터 일부를 요청(요구 즉시 폐치)할 수도 있습니다.

- 3 애플리케이션 서버는 (필요한 경우 먼저 데이터베이스에 연결하여) 데이터를 가져오고, 클라이언트를 위해 데이터를 패키징하고, 데이터 패킷을 클라이언트에 반환합니다. 필드 표시 특징과 같은 추가 정보는 데이터 패킷의 메타데이터에 포함시킬 수 있습니다. 데이터 패킷에 데이터를 패키징하는 프로세스를 "공급"이라고 합니다.
- 4 클라이언트는 데이터 패킷을 디코딩하고 데이터를 사용자에게 표시합니다.
- 5 사용자가 클라이언트 애플리케이션과 상호 작용하면서 데이터가 업데이트(레코드의 추가, 삭제 또는 수정)됩니다. 이러한 수정은 클라이언트의 변경 로그에 저장됩니다.
- 6 결국 클라이언트는 일반적으로 사용자 동작에 응답하여 해당 업데이트를 애플리케이션 서버에 적용합니다. 업데이트를 적용하기 위해서 클라이언트는 변경 로그를 패키징하고 그 변경 로그를 서버에 데이터 패킷으로 보냅니다.
- 7 애플리케이션 서버는 패키지를 디코딩하고 적절한 경우 트랜잭션 안에 업데이트 내용을 포스트합니다. 예를 들어, 클라이언트가 레코드를 요청하고 나서 해당 업데이트를 적용하기 전에 다른 애플리케이션에서 그 레코드를 변경했기 때문에 레코드를 포스트할 수 없는 경우, 애플리케이션 서버는 클라이언트의 변경 내용을 현재 데이터로 조정 또는 포스트할 수 없는 레코드를 저장합니다. 레코드를 포스트하고 문제가 되는 레코드를 캐시하는 프로세스를 "해결"이라고 합니다.
- 8 애플리케이션 서버가 해결 프로세스를 완료하면 추가 해결을 위해 포스트되지 않은 레코드를 클라이언트에 반환합니다.
- 9 클라이언트는 해결되지 않은 레코드를 조정합니다. 클라이언트가 해결되지 않은 레코드를 조정할 수 있는 여러 가지 방법이 있습니다. 대개 클라이언트는 레코드 포스트를 막거나 변경 사항을 무시하는 상황을 고치려고 합니다. 오류 상황이 정정되면 클라이언트는 업데이트를 다시 적용합니다.
- 10 클라이언트는 서버의 데이터를 새로 고칩니다.

클라이언트 애플리케이션의 구조

최종 사용자에게 다계층 애플리케이션의 클라이언트 애플리케이션은 캐시된 업데이트를 사용하는 기존의 2계층 애플리케이션과 비교하여 외관 및 동작에 그다지 차이가 없는 것처럼 보입니다. 사용자 상호 작용은 *TClientDataSet* 컴포넌트의 데이터를 표시하는 표준 data-aware 컨트롤을 통해 발생합니다. 클라이언트 데이터셋의 속성, 이벤트 및 메소드 사용에 대한 자세한 내용은 23장 "클라이언트 데이터셋 사용"을 참조하십시오.

*TClientDataSet*은 외부 프로바이더가 있는 클라이언트 데이터셋을 사용하는 2계층 애플리케이션에서처럼 프로바이더 컴포넌트에서 데이터를 페치하고 업데이트를 프로바이더 컴포넌트에 적용합니다. 프로바이더에 대한 자세한 내용은 24장 "프로바이더 컴포넌트 사용"을 참조하십시오. 프로바이더와 쉽게 통신할 수 있도록 해주는 클라이언트 데이터셋 기능에 대한 자세한 내용은 23-25 페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

클라이언트 데이터셋은 *IAppServer* 인터페이스를 통해 프로바이더와 통신합니다. 클라이언트 데이터셋은 연결 컴포넌트에서 이 인터페이스를 얻습니다. 연결 컴포넌트는 애플리케이션 서버에 연결을 만듭니다. 다른 연결 컴포넌트는 다른 통신 프로토콜을 사용할 수 있습니다. 다음 표는 이러한 연결 컴포넌트에 대해 요약 설명한 것입니다.

표 25.2 연결 컴포넌트

컴포넌트	프로토콜
TDCOMConnection	DCOM
TSocketConnection	Windows 소켓(TCP/IP)
TWebConnection	HTTP
TSOAPConnection	SOAP(HTTP 및 XML)
TCorbaConnection	CORBA (IIOP)

참고 또한 Delphi는 애플리케이션 서버에 전혀 연결되지 않는 연결 컴포넌트를 포함하지만 대신 클라이언트 데이터셋이 동일한 애플리케이션에서 프로바이더와 통신할 때 사용할 수 있는 *IAppServer* 인터페이스를 제공합니다. 이 *TLocalConnection* 컴포넌트는 꼭 필요한 것은 아니지만 나중에 다계층 애플리케이션으로 쉽게 확장할 수 있게 해줍니다. 연결 컴포넌트 사용에 대한 자세한 내용은 25-24 페이지의 "애플리케이션 서버에 연결"을 참조하십시오.

애플리케이션 서버의 구조

애플리케이션 서버를 설정하고 실행할 때는 클라이언트 애플리케이션과 연결되지 않습니다. 그 대신 클라이언트 애플리케이션에 의해 연결이 초기화되고 유지됩니다. 클라이언트 애플리케이션은 자신의 연결 컴포넌트를 사용하여 선택한 프로바이더와 통신하는데 사용할 애플리케이션 서버에 연결합니다. 이 모든 작업은 수신 요청을 관리하고 인터페이스를 제공하는 코드를 작성할 필요 없이 자동으로 일어납니다.

애플리케이션 서버의 기반은 *IAppServer* 인터페이스를 지원하는 특수한 데이터 모듈인 원격 데이터 모듈입니다. 클라이언트 애플리케이션은 *IAppServer* 인터페이스를 사용하여 애플리케이션 서버에서 프로바이더와 통신합니다.

원격 데이터 모듈 타입에는 다음 네 가지가 있습니다.

- **TRemoteDataModule:** 이것은 이중 인터페이스 Automation 서버입니다. 클라이언트가 애플리케이션 서버에 연결하는 데 DCOM, HTTP, 소켓 또는 OLE를 사용하는 경우 애플리케이션 서버를 MTS와 함께 설치하지 않으려면 이런 유형의 원격 데이터 모듈을 사용합니다.
- **TMTSDataModule:** 이것은 이중 인터페이스 Automation 서버입니다. 애플리케이션 서버를 MTS 또는 COM+와 함께 설치되는 Active Library(.DLL)로 만드는 경우 이 타입의 원격 데이터 모듈을 사용합니다. DCOM, HTTP, 소켓 또는 OLE와 함께 MTS 원격 데이터 모듈을 사용할 수 있습니다.
- **TCorbaDataModule:** 이것은 CORBA 서버입니다. 이런 유형의 원격 데이터 모듈을 사용하여 CORBA 클라이언트에 데이터를 제공합니다.

- **TSoapDataModule**: 이것은 웹 서비스 애플리케이션에서 *IAppServer* 자손을 호출 가능한 인터페이스로 구현하는 데이터 모듈입니다. 이런 유형의 원격 데이터 모듈을 사용하여 데이터를 웹 서비스로 액세스하는 클라이언트에 데이터를 제공합니다.

애플리케이션 서버가 MTS 또는 COM+에서 배포되어야 한다면 원격 데이터 모듈은 애플리케이션 서버가 활성화 또는 비활성화될 경우의 이벤트를 포함합니다. 애플리케이션 서버가 활성화될 때 데이터베이스가 연결되고 비활성화될 때 데이터베이스 연결이 해제됩니다.

원격 데이터 모듈의 콘텐츠

다른 데이터 모듈과 마찬가지로 원격 데이터 모듈에 다투어 컴포넌트를 포함시킬 수 있습니다. 반드시 포함시켜야 하는 컴포넌트는 다음과 같습니다.

- 원격 데이터 모듈이 데이터베이스 서버의 정보를 노출시키는 경우, 해당 데이터베이스 서버의 레코드를 나타내는 데이터셋 컴포넌트를 포함시켜야 합니다. 데이터셋이 데이터베이스 서버와 상호 작용할 수 있도록 데이터베이스 연결 컴포넌트와 같은 일부 타입의 다른 컴포넌트가 필요할 수도 있습니다. 데이터셋에 대한 내용은 18장 "데이터셋 이해"를 참조하고, 데이터베이스 연결 컴포넌트에 대한 내용은 17장 "데이터베이스에 연결"을 참조하십시오.

원격 데이터 모듈이 클라이언트에 노출시키는 모든 데이터셋은 데이터셋 프로바이더를 포함해야 합니다. 데이터셋 프로바이더는 데이터를 클라이언트 데이터셋에 보내는 데이터 팩킷으로 패키징하고 클라이언트 데이터셋에서 받은 업데이트를 소스 데이터셋 또는 데이터베이스 서버에 다시 적용합니다. 데이터셋 프로바이더에 대한 자세한 내용은 24장 "프로바이더 컴포넌트 사용"을 참조하십시오.

- 원격 데이터 모듈이 클라이언트에 노출시키는 모든 XML 문서는 XML 프로바이더를 포함해야 합니다. XML 프로바이더는 데이터를 폐치하고 업데이트를 데이터베이스 서버가 아닌 XML 문서에 적용한다는 점을 제외하고는 데이터셋 프로바이더처럼 동작합니다. XML 프로바이더에 대한 자세한 내용은 26-8 페이지의 "XML 문서를 프로바이더의 소스로 사용"을 참조하십시오.

참고 데이터셋을 데이터베이스 서버에 연결하는 데이터베이스 연결 컴포넌트와 다계층 애플리케이션의 클라이언트 애플리케이션에서 사용하는 연결 컴포넌트 간에 혼동하지 마십시오. 다계층 애플리케이션의 연결 컴포넌트는 컴포넌트 팔레트의 DataSnap 페이지에 있습니다.

트랜잭션 데이터 모듈(Transaction Data Module) 사용

MTS (Windows 2000 이전) 또는 COM+ (Windows 2000 버전 이상)가 제공하는 분산 애플리케이션을 위한 특별한 서비스를 이용하는 애플리케이션 서버를 작성할 수 있습니다. 그렇게 하려면 일반적인 원격 데이터 모듈 대신 트랜잭션 데이터 모듈을 만듭니다.

트랜잭션 데이터 모듈을 사용하는 경우 사용자의 애플리케이션에서 다음과 같은 특별한 서비스를 이용할 수 있습니다.

- **보안.** MTS와 COM+는 애플리케이션 서버에 역할 기반 보안을 제공합니다. 클라이언트는 할당된 역할이며 MTS 데이터 모듈의 인터페이스에 액세스할 수 있는 방법을 결정합니다. MTS 데이터 모듈은 *IsCallerInRole* 메소드를 구현하며 이를 통해 현재 연결된 클라이언트의 역할을 확인하고 그 역할에 기반한 특정 기능을 조건적으로 허용합니다. MTS와 COM+ 보안에 대한 자세한 내용은 39-14 페이지의 "역할 기반 보안(Role-based security)"을 참조하십시오.
- **데이터베이스 핸들 풀링.** 트랜잭션 데이터 모듈은 ADO 또는 BDE(MTS를 사용하고 MTS POOLING을 적용하는 경우)를 통해 만들어진 데이터베이스 연결을 자동으로 풀링합니다. 한 클라이언트가 데이터베이스 연결을 끝내면 다른 클라이언트가 그 연결을 재사용할 수 있습니다. 이를 통해 사용자의 중간 계층은 원격 데이터베이스 서버에서 로그아웃한 다음 다시 로그인할 필요가 없으므로 네트워크 트래픽이 감소됩니다. 데이터베이스 핸들을 풀링할 때 사용자의 데이터베이스 연결 컴포넌트가 *KeepConnection* 속성을 *False*로 설정하면 애플리케이션에서 연결 공유가 최대화됩니다. 데이터베이스 핸들 풀링에 대한 자세한 내용은 39-5 페이지의 "데이터베이스 리소스 디스펜서"를 참조하십시오.
- **트랜잭션.** 트랜잭션 데이터 모듈 사용 시 단일 데이터베이스 연결에서의 지원 이상의 확장된 트랜잭션 지원을 제공할 수 있습니다. 트랜잭션 데이터 모듈은 다중 데이터베이스에 걸친 트랜잭션에 관여할 수 있고 전혀 데이터베이스와 관련되지 않은 기능을 포함시킬 수 있습니다. 트랜잭션 데이터 모듈과 같은 트랜잭션 객체에 의해 제공되는 트랜잭션 지원에 대한 자세한 내용은 25-19 페이지의 "다계층 애플리케이션의 트랜잭션 관리"를 참조하십시오.
- **Just-in-time 활성화 및 as-soon-as-possible 비활성화.** MTS 서버를 작성해서 원격 데이터 모듈 인스턴스가 필요에 따라 활성화 또는 비활성화되게 할 수 있습니다. Just-in-time 활성화 및 as-soon-as-possible 비활성화 사용 시 사용자의 원격 데이터 모듈은 클라이언트 요청을 처리해야 하는 경우에만 인스턴스화됩니다. 이를 통해 데이터베이스 핸들과 같은 리소스를 사용하지 않을 때 묶어 두지 않도록 합니다.

Just-in-time 활성화와 as-soon-as-possible 비활성화를 사용하면 단일 원격 데이터 모듈 인스턴스를 통해 모든 클라이언트를 라우팅하는 것과 모든 클라이언트 연결마다 별도의 인스턴스를 생성하는 것 사이에서 중립을 이룹니다. 단일 원격 데이터 모듈 인스턴스에서 애플리케이션 서버는 단일 데이터베이스 연결을 통해 모든 데이터베이스 호출을 처리해야 합니다. 이는 클라이언트가 많을 때 병목 상태로 작용하여 성능을 저하시킬 수 있습니다. 원격 데이터 모듈의 여러 인스턴스에서 각 인스턴스는 개별 데이터베이스 연결을 유지하므로 데이터베이스 액세스를 일련화시킬 필요가 없습니다. 하지만 이는 데이터베이스가 한 클라이언트의 원격 데이터 모듈에 연결되어 있는 동안 다른 클라이언트가 데이터베이스 연결을 사용할 수 없으므로 리소스를 독점하게 됩니다.

트랜잭션, just-in-time 활성화, as-soon-as-possible 비활성화를 이용하려면 원격 데이터 모듈 인스턴스가 상태 없음(stateless)이어야 합니다. 즉, 클라이언트가 상태 정보에 의존하는 경우 추가적인 지원을 제공해야 합니다. 예를 들어, 클라이언트는 점진적 폐지 수행 시 현재 레코드에 대한 정보를 전달해야 합니다. 다계층 애플리케이션의 상태 정보 및 원격 데이터 모듈에 대한 자세한 내용은 25-20 페이지의 "원격 데이터 모듈의 상태 정보 지원"을 참조하십시오.

기본적으로 트랜잭션 데이터 모듈에 대한 모든 자동 생성된 호출은 트랜잭션적입니다. 즉, 이러한 호출은 호출 종료 시 데이터 모듈이 비활성화되고 현재 트랜잭션이 커밋되거나 롤백될 수 있다고 가정합니다. *AutoComplete* 속성을 *False*로 설정하여 영구적인 상태 정보에 의존하는 트랜잭션 데이터 모듈을 작성할 수 있지만 사용자 지정 인터페이스를 사용하지 않는 경우에는 트랜잭션, just-in-time 활성화 또는 as-soon-as-possible 비활성화를 지원하지 않습니다.

경고 트랜잭션 데이터 모듈을 포함하는 애플리케이션 서버는 데이터 모듈이 활성화될 때까지 데이터베이스 연결을 열지 않아야 합니다. 애플리케이션을 개발하는 동안 애플리케이션을 실행하기 전까지는 모든 데이터셋이 활성화되어 있지 않고 데이터베이스가 연결되지 않게 해야 합니다. 애플리케이션 자체에서 데이터 모듈이 활성화되었을 때 데이터베이스 연결을 여는 코드를 추가하고 데이터 모듈이 비활성화되었을 때 데이터베이스 연결을 닫는 코드를 추가합니다.

원격 데이터 모듈 풀링

객체 풀링을 통해 클라이언트들이 공유하는 원격 데이터 모듈의 캐시를 생성할 수 있으므로 리소스가 보존됩니다. 이러한 작업의 방식은 원격 데이터 모듈의 타입과 연결 프로토콜에 따라 다릅니다.

COM+에 설치될 트랜잭션 데이터 모듈을 생성하는 경우, COM+ Component Manager를 사용하여 애플리케이션 서버를 공용 객체로 설치할 수 있습니다. 자세한 내용은 39-8 페이지의 "객체 풀링"을 참조하십시오.

트랜잭션 데이터 모듈을 사용하지 않는 경우에도 HTTP(*TWebConnection*)를 사용하여 연결을 구성하면 객체 풀링을 이용할 수 있습니다. 이러한 두 번째 타입의 객체 풀링에서는 생성된 원격 데이터 모듈의 인스턴스 수를 제한합니다. 이는 원격 데이터 모듈에 의해 사용되는 다른 리소스뿐만 아니라 보유해야 하는 데이터베이스 연결의 수를 제한합니다.

사용자의 원격 데이터 모듈에 호출을 전달하는 웹 서버 애플리케이션이 클라이언트 요청을 받으면 풀에 있는 첫 번째 사용 가능한 원격 데이터 모듈에 호출이 전달됩니다. 사용 가능한 원격 데이터 모듈이 없는 경우 원격 데이터 모듈을 지정한 최대 수까지 새로 생성합니다. 이렇게 함으로써 병목 현상으로 작용할 수 있는 단일 원격 데이터 모듈 인스턴스를 통해 모든 클라이언트를 라우팅하는 것과 많은 리소스를 소모할 수 있는 각 클라이언트에 대한 개별 인스턴스를 생성하는 것 사이에 중립이 제공됩니다.

풀에 있는 원격 데이터 모듈 인스턴스가 일정 시간 동안 클라이언트 요청을 받지 않은 경우 자동으로 해제됩니다. 리소스가 사용되지 않는 한, 풀에서 리소스를 독점하는 것이 방지됩니다.

웹 연결(HTTP) 사용 시 객체 풀링을 설정하려면 원격 데이터 모듈은 *UpdateRegistry* 메소드를 오버라이드해야 합니다. 오버라이드된 메소드에서 원격 데이터 모듈 등록 시 *RegisterPooled*를 호출하고 원격 데이터 모듈 등록 해제 시 *UnregisterPooled*를 호출합니다. 객체 풀링의 방법 사용 시 원격 데이터 모듈은 상태 없음(stateless)이어야 합니다. 왜냐하면 단일 인스턴스가 여러 클라이언트의 요청을 잠재적으로 처리하기 때문입니다. 영구적인 상태 정보에 의존하는 경우 클라이언트들이 서로 방해됩니다. 원격 데이터 모듈이 상태 없음(stateless)인지 확인하는 방법에

대한 자세한 내용은 25-20 페이지의 "원격 데이터 모듈의 상태 정보 지원"을 참조하십시오.

연결 프로토콜 선택

클라이언트 애플리케이션을 애플리케이션 서버에 연결하는 데 사용할 수 있는 각각의 통신 프로토콜은 고유의 이점을 제공합니다. 프로토콜을 선택하기 전에 예상하는 클라이언트의 수, 애플리케이션 배포 방법 및 향후 개발 계획을 고려하십시오.

DCOM 연결 사용

DCOM은 서버에 추가적인 런타임 애플리케이션을 필요로 하지 않는 가장 직접적인 통신 방법을 제공합니다. 하지만 DCOM이 Windows 95에는 포함되어 있지 않으므로 Windows 95 클라이언트 컴퓨터에 DCOM이 설치되어 있지 않을 수도 있습니다.

DCOM은 트랜잭션 데이터 모듈 작성 시 보안 서비스를 사용할 수 있게 해주는 유일한 방법을 제공합니다. 이러한 보안 서비스는 트랜잭션 객체의 호출자에 대한 역할 할당을 기반으로 합니다. DCOM 사용 시 DCOM은 애플리케이션 서버 (MTS 또는 COM+)를 호출하는 시스템에 대해 호출자를 식별합니다. 그러므로 호출자의 역할을 정확하게 결정할 수 있습니다. 하지만 다른 프로토콜 사용 시에는 클라이언트 호출을 받는 애플리케이션 서버와 별도로 런타임 실행 파일이 있습니다. 이 런타임 실행 파일은 클라이언트 대신 애플리케이션 서버에 대한 COM 호출을 만듭니다. 이로 인해 개별 클라이언트에 역할을 할당할 수 없습니다. 런타임 실행 파일은 실제로 유일한 클라이언트입니다. 보안과 트랜잭션 객체에 대한 자세한 내용은 39-14 페이지의 "역할 기반 보안(Role-based security)"을 참조하십시오.

소켓 연결 사용

TCP/IP 소켓을 통해 가벼운 클라이언트를 생성할 수 있습니다. 예를 들어, 웹 기반 클라이언트 애플리케이션을 작성하는 경우 클라이언트 시스템이 DCOM을 지원하는지 확인할 수 없습니다. 소켓은 애플리케이션 서버 연결에 사용할 수 있는 최저의 공통 분모를 제공합니다. 소켓에 대한 자세한 내용은 32장 "소켓 작업"을 참조하십시오.

DCOM처럼 클라이언트로부터 직접 원격 데이터 모듈을 인스턴스화하는 대신 소켓은 클라이언트 요청을 받고 COM을 사용하여 원격 데이터 모듈을 인스턴스화하는 서버의 별도 애플리케이션 (ScktSrvr.exe)을 사용합니다. 클라이언트의 연결 컴포넌트와 서버의 ScktSrvr.exe는 *IAppServer* 호출 마샬링 (marshaling)을 담당합니다.

참고 ScktSrvr.exe는 NT 서비스 애플리케이션으로서 실행할 수 있습니다. -install 명령줄 옵션으로 시작하여 서비스 관리자에 등록합니다. -uninstall 명령줄 옵션을 사용하면 등록 해제할 수 있습니다.

소켓 연결을 사용하기 전에 애플리케이션 서버는 소켓 연결을 통해 클라이언트에 대한 가용성을 등록해야 합니다. 기본적으로 새로운 모든 원격 데이터 모듈은 *UpdateRegistry* 메소드에서 *EnableSocketTransport*에 대한 호출을 추가하여 자동으로 등록합니다. 이 호출을 제거하면 애플리케이션 서버에 대한 소켓 연결을 막을 수 있습니다.

참고 예전의 서버에서는 이러한 등록을 추가하지 않았으므로 ScktSrvr.exe의 Connections\Registered Objects Only 메뉴 항목을 선택 해제하여 애플리케이션 서버의 등록 여부에 대한 검사를 해제할 수 있습니다.

소켓을 사용할 때는 애플리케이션 서버에서 인터페이스에 대한 참조를 해제하기 전에는 클라이언트 시스템 실패에 대한 서버측의 보호책이 없습니다. 소켓을 사용하면 주기적 연결 유지 메시지를 보내는 DCOM 사용 시보다 메시지 트래픽은 감소되는 반면 클라이언트가 연결을 끊었는지 알 수 없으므로 애플리케이션 서버에서 해당 리소스를 해제할 수 없습니다.

웹 연결 사용

HTTP는 방화벽에 의해 보호되는 애플리케이션 서버와 통신할 수 있는 클라이언트를 생성할 수 있게 합니다. HTTP 메시지는 클라이언트 애플리케이션을 널리 안전하게 배포할 수 있도록 내부 애플리케이션에 대해 제어 가능한 액세스를 제공합니다. 소켓 연결과 마찬가지로 HTTP 메시지는 애플리케이션 서버에 연결하는 데 사용할 수 있는 최저의 공통 분모를 제공합니다. HTTP 메시지에 대한 자세한 내용은 27장 "인터넷 애플리케이션 생성"을 참조하십시오.

DCOM처럼 클라이언트에서 직접 원격 데이터 모듈을 인스턴스화하는 대신 HTTP 기반 연결은 COM을 사용하여 클라이언트 요청을 받아들이고 원격 데이터 모듈을 인스턴스화하는 서버의 웹 서버 애플리케이션(httpsrvr.dll)을 사용합니다. 이러한 점 때문에 웹 연결이라고도 합니다. 클라이언트의 연결 컴포넌트와 서버의 httpsrvr.dll은 *IAppServer* 호출 마샬링(marshaling)을 담당합니다.

웹 연결은 wininet.dll(클라이언트 시스템에서 실행되는 인터넷 유틸리티의 라이브러리)에 의해 제공되는 SSL 보안을 이용할 수 있습니다. 서버 시스템에서 인증이 필요하다면 웹 서버를 구성했으면 웹 연결 컴포넌트의 속성을 사용하여 사용자 이름과 암호를 지정할 수 있습니다.

추가적인 보안 방법으로서 애플리케이션 서버는 웹 연결을 사용하여 클라이언트에 대한 가용성을 등록해야 합니다. 기본적으로 새로운 모든 원격 데이터 모듈은 *UpdateRegistry* 메소드에서 *EnableWebTransport*에 대한 호출을 추가함으로써 자동으로 등록합니다. 이 호출을 제거하면 애플리케이션 서버에 대한 웹 연결을 막을 수 있습니다.

웹 연결은 객체 풀링을 이용할 수 있습니다. 이를 통해 서버는 클라이언트 요청에 사용할 수 있는 원격 데이터 모듈 인스턴스의 제한된 풀을 생성할 수 있습니다. 원격 데이터 모듈을 풀링하면 필요한 경우를 제외하고는 서버에서 데이터 모듈과 해당 데이터 베이스 연결에 대한 리소스를 소모하지 않습니다. 객체 풀링에 대한 자세한 내용은 25-8 페이지의 "원격 데이터 모듈 풀링"을 참조하십시오.

대부분의 다른 연결 컴포넌트와 달리 HTTP를 통해 연결된 경우에는 콜백을 사용할 수 없습니다.

SOAP 연결 사용

SOAP는 웹 서비스 애플리케이션에 대해 Delphi에서 지원하는 프로토콜입니다. SOAP Marshals 메소드는 XML 인코딩을 사용하여 호출합니다. SOAP 연결은 HTTP를 전송 프로토콜로 사용합니다.

SOAP 연결은 Windows와 Linux에서 모두 지원되므로 크로스 플랫폼 애플리케이션에서 작동하는 이점을 갖습니다. SOAP 연결은 HTTP를 사용하므로 웹 연결과 동일한 이점을 갖습니다. HTTP는 모든 클라이언트가 사용할 수 있는 최저의 공통 분모를 제공하며 클라이언트는 "방화벽"에 의해 보호되는 애플리케이션 서버와 통신할 수 있습니다. SOAP를 통한 Delphi에서의 애플리케이션 배포에 대한 자세한 내용은 31장 "Web Services 사용"을 참조하십시오.

HTTP 연결과 마찬가지로 SOAP를 통해 연결이 구성된 경우 콜백을 사용할 수 없습니다. SOAP 연결은 또한 애플리케이션 서버에서 단일 원격 데이터 모듈로 제한합니다.

CORBA 연결 사용

CORBA는 사용자가 다계층 데이터베이스 애플리케이션을 CORBA에 표준화된 환경으로 통합할 수 있게 합니다. 예를 들어, Java로 작성된 클라이언트 애플리케이션을 사용하는 경우 CORBA 연결만 사용할 수 있습니다. CORBA(및 Java)는 여러 플랫폼에서 사용할 수 있으므로 사용자가 크로스 플랫폼 다계층 애플리케이션을 작성할 수 있게 합니다.

CORBA를 사용하면 애플리케이션은 로드 밸런싱, 위치 투명성 및 ORB 런타임 소프트웨어의 오류 복구(fail-over)에 대한 이점을 자동으로 얻습니다. 또한 다른 CORBA 서비스의 이점을 추가로 얻을 수 있습니다.

다계층 애플리케이션 구축

다계층 데이터베이스 애플리케이션을 생성하기 위한 일반 단계는 다음과 같습니다.

- 1 애플리케이션 서버를 생성합니다.
- 2 애플리케이션 서버를 등록하거나 설치합니다.
- 3 클라이언트 애플리케이션을 생성합니다.

생성 순서가 중요합니다. 클라이언트를 생성하기 전에 애플리케이션 서버를 생성하여 실행해야 합니다. 그런 다음 디자인 타임 시 애플리케이션 서버에 연결하여 클라이언트를 테스트할 수 있습니다. 물론 디자인 타임 시 애플리케이션 서버를 지정하지 않고 클라이언트를 생성한 후 런타임 시 서버 이름만 제공할 수 있습니다. 하지만 그렇게 하면 디자인 타임 시 코드를 작성할 때 애플리케이션이 예상대로 작동하는지 알 수 없고 Object Inspector에서 서버 및 프로바이더를 선택할 수 없게 됩니다.

참고 서버와 동일한 시스템에 클라이언트 애플리케이션을 생성하지 않고 DCOM 연결을 사용하는 경우 클라이언트 시스템에 애플리케이션 서버를 등록할 수 있습니다. 이를 통해 디자인 타임 시 연결 컴포넌트가 애플리케이션 서버를 인식하여 Object Inspector의 드롭다운 목록에서 서버 이름과 프로바이더 이름을 선택할 수 있습니다. 웹 연결, SOAP 연결 또는 소켓 연결을 사용하는 경우 연결 컴포넌트는 서버 시스템으로부터 등록된 서버 이름을 폐치합니다.

애플리케이션 서버 생성

대부분의 데이터베이스 애플리케이션을 생성하는 것과 거의 동일한 방식으로 애플리케이션 서버를 생성합니다. 중요한 차이점은 애플리케이션 서버는 원격 데이터 모듈을 사용한다는 점입니다.

다음의 단계에 따라 애플리케이션 서버를 생성합니다.

1 새 프로젝트를 시작합니다.

- SOAP를 전송 프로토콜로 사용하는 경우에는 새 웹 서비스 애플리케이션이어야 합니다. File|New|Other를 선택하고 새 항목 대화 상자의 Web Services 페이지에서 Web Service 애플리케이션을 선택합니다.
- 다른 전송 프로토콜의 경우에는 File|New|Application만 선택합니다.

새 프로젝트를 저장합니다.

2 새 원격 데이터 모듈을 프로젝트에 추가합니다. 메인 메뉴에서 File|New |Other를 선택하고 새 항목 대화 상자의 Multitier 페이지에서 다음 중 선택합니다.

- 클라이언트가 DCOM, HTTP 또는 소켓을 사용하여 액세스하는 COM Automation 서버를 생성하는 경우 **Remote Data Module**을 선택합니다.
- MTS 또는 COM+에서 실행되는 원격 데이터 모듈을 생성하는 경우 **Transactional Data Module**을 선택합니다. DCOM, HTTP 또는 소켓을 사용하여 연결을 구성할 수 있습니다. 하지만 DCOM에서만 보안 서비스를 지원합니다.
- CORBA 서버를 생성하는 경우 **CORBA Data Module**을 선택합니다.
- 웹 서비스 애플리케이션에서 SOAP 서버를 생성하는 경우 **SOAP Data Module**을 선택합니다.

원격 데이터 모듈 설정에 대한 자세한 내용은 25-13 페이지의 "원격 데이터 모듈 설정"을 참조하십시오.

참고

원격 데이터 모듈은 간단한 데이터 모듈 이상의 기능을 갖습니다. CORBA 데이터 모듈은 CORBA 서버로서 작동합니다. SOAP 데이터 모듈은 웹 서비스 애플리케이션에서 호출 가능한 인터페이스를 구현합니다. 다른 데이터 모듈은 COM Automation 객체입니다.

3 데이터 모듈에 적절한 데이터셋 컴포넌트를 두고 데이터베이스 서버에 액세스하도록 설정합니다.

4 각 데이터셋의 데이터 모듈에 *TDataSetProvider* 컴포넌트를 놓습니다. 이 프로바이더는 클라이언트 요청을 브로커하고 데이터를 패키징하는 데 필요합니다. 각 프로바이더 컴포넌트의 *DataSet* 속성을 액세스할 데이터셋의 이름으로 설정합니다. 프로바이더에 대한 추가 속성을 설정할 수 있습니다. 프로바이더 설정에 대한 자세한 내용은 24장 "프로바이더 컴포넌트 사용"을 참조하십시오.

XML 문서의 데이터를 사용하는 경우 데이터셋과 *TDataSetProvider* 컴포넌트를 사용하는 대신 *TXMLTransformProvider* 컴포넌트를 사용할 수 있습니다.

*TXMLTransformProvider*를 사용하는 경우 *XMLDataFile* 속성을 데이터를 제공하고 업데이트를 적용하는 XML 문서로 지정합니다.

- 5 이벤트, 공유 비즈니스 룰, 공유 데이터 검증 및 공유 보안을 구현하도록 애플리케이션 서버 코드를 작성합니다. 이러한 코드를 작성할 때 다음과 같은 작업을 합니다.
 - 애플리케이션 서버의 인터페이스를 확장하여 클라이언트 애플리케이션이 서버를 호출하는 방법을 추가합니다. 애플리케이션 서버의 인터페이스 확장에 대해서는 25-17 페이지의 "애플리케이션 서버의 인터페이스 확장"에서 설명합니다.
 - 업데이트 적용 시 자동으로 생성되는 트랜잭션의 범위를 넘는 트랜잭션 지원을 제공합니다. 다계층 데이터베이스 애플리케이션의 트랜잭션 지원에 대해서는 25-19 페이지의 "다계층 애플리케이션의 트랜잭션 관리"에서 설명합니다.
 - 애플리케이션 서버의 데이터셋 간에 마스터/디테일 관계를 생성합니다. 마스터/디테일 관계에 대해서는 25-19 페이지의 "마스터/디테일 관계 지원"에서 설명합니다.
 - 애플리케이션 서버가 상태 없음인지 확인합니다. 상태 정보 처리에 대해서는 25-20 페이지의 "원격 데이터 모듈의 상태 정보 지원"에서 설명합니다.
 - 애플리케이션 서버를 다중 원격 데이터 모듈로 나눕니다. 다중 원격 데이터 모듈 사용에 대해서는 25-22 페이지의 "다중 원격 데이터 모듈 사용"에서 설명합니다.
- 6 애플리케이션 서버를 저장, 컴파일, 등록 또는 설치합니다. 애플리케이션 서버 등록에 대해서는 25-23 페이지의 "애플리케이션 서버 등록"에서 설명합니다.
- 7 서버 애플리케이션에서 DCOM 또는 SOAP를 사용하지 않는 경우에는 클라이언트 메시지를 받고, 원격 데이터 모듈을 인스턴스화하고, 인터페이스 호출을 마샬링하는 런타임 소프트웨어를 설치해야 합니다.
 - TCP/IP 소켓의 경우 런타임 소프트웨어는 소켓 디스패처 애플리케이션, Scktsrvr.exe입니다.
 - HTTP 연결의 경우 런타임 소프트웨어는 웹 서버에 설치되는 httpsrvr.dll, ISAPI/NSAPI DLL입니다.
 - CORBA의 경우 런타임 소프트웨어는 VisiBroker ORB입니다.

원격 데이터 모듈 설정

원격 데이터 모듈을 생성할 때 클라이언트 요청에 응답하는 방법을 나타내는 특정 정보를 제공해야 합니다. 이 정보는 원격 데이터 모듈의 타입에 따라 다릅니다. 필요한 원격 데이터 모듈 타입에 대한 내용은 25-5 페이지의 "애플리케이션 서버의 구조"를 참조하십시오.

TRemoteDataModule 구성

TRemoteDataModule 컴포넌트를 애플리케이션에 추가하려면 File|New|Other를 선택하고 새 항목 대화 상자의 Multitier 페이지에서 Remote Data Module을 선택합니다. Remote Data Module 마법사가 나타납니다.

원격 데이터 모듈에 클래스 이름을 제공해야 합니다. 이 이름은 애플리케이션에서 생성하는 *TRemoteDataModule* 자손의 기본 이름입니다. 이 이름은 해당 클래스에 대한 인터페이스의 기본 이름이기도 합니다. 예를 들어, 클래스 이름 *MyDataServer*를 지정하면 마법사는 *TMyDataServer*를 선언하는 새 유닛, *IMyDataServer*를 구현하는 *TRemoteDataModule*의 자손, *IAppServer*의 자손을 생성합니다.

참고 새 인터페이스에 사용자 고유의 속성 및 메소드를 추가할 수 있습니다. 자세한 내용은 25-17 페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

Remote Data Module 마법사에서 스레드 모델을 지정해야 합니다. Single-threaded, Apartment-threaded, Free-threaded 또는 Both를 선택할 수 있습니다.

- Single-threaded를 선택하면 COM은 한 번에 하나의 클라이언트 요청에만 서비스 하도록 합니다. 클라이언트 요청이 서로 간섭하는지 우려할 필요가 없습니다.
- Apartment-threaded를 선택하면 COM은 원격 데이터 모듈의 인스턴스가 한 번에 한 요청에만 서비스 하도록 합니다. Apartment-threaded 라이브러리에서 코드 작성 시 전역 변수 또는 원격 데이터 모듈에 포함되지 않는 객체를 사용하는 경우 스레드 사이의 충돌을 막을 수 있는 방안을 마련해야 합니다. 이 모델은 BDE 활성 데이터셋 사용 시 권장됩니다. (BDE 활성 데이터셋에서 발생하는 스레드 문제를 처리하려면 *AutoSessionName* 속성이 *True*로 설정된 세션 컴포넌트가 필요합니다.)
- Free-threaded를 선택하면 애플리케이션은 여러 스레드에서 동시에 클라이언트 요청을 받을 수 있습니다. 이 경우 사용자는 애플리케이션이 thread-safe인지 확인해야 합니다. 여러 클라이언트가 사용자의 원격 데이터 모듈에 동시에 액세스할 수 있으므로 인스턴스 데이터(속성, 포함된 객체 등)와 전역 변수를 보호해야 합니다. 이 모델은 ADO 데이터셋 사용 시 권장됩니다.
- Both를 선택하면 라이브러리는 한 가지만 제외하고 Free-threaded를 선택했을 때와 똑같이 작동합니다. 모든 콜백(클라이언트 인터페이스에 대한 호출)이 일련화됩니다.
- Neutral을 선택하면 원격 데이터 모듈은 Free-threaded 모델에서와 같이 여러 스레드의 동시 호출을 받으며 COM은 두 개의 스레드가 동시에 동일한 메소드에 액세스하지 않도록 확인합니다.

EXE를 생성하는 경우 사용할 인스턴스 타입을 지정해야 합니다. Single instance 또는 Multiple instance를 선택할 수 있습니다(내부 인스턴스는 클라이언트 코드가 동일한 프로세스 공간에 속할 때만 적용됩니다.).

- Single instance를 선택하면 각 클라이언트 연결은 실행 파일의 고유 인스턴스를 실행합니다. 이 프로세스는 클라이언트 연결 전용인 원격 데이터 모듈의 단일 인스턴스를 인스턴스화합니다.
- Multiple instance를 선택하면 애플리케이션의 단일 인스턴스는 클라이언트에 생성된 모든 원격 데이터 모듈을 인스턴스화합니다. 각 원격 데이터 모듈은 단일 클라이언트 연결 전용이지만 동일한 프로세스 공간을 모두 공유합니다.

TMTSDataModule 구성

TMTSDataModule 컴포넌트를 애플리케이션에 추가하려면 File|New|Other를 선택하고 새 항목 대화 상자의 Multitier 페이지에서 Transactional Data Module을 선택합니다. Transactional Data Module 마법사가 나타납니다.

원격 데이터 모듈에 클래스 이름을 제공해야 합니다. 이 이름은 애플리케이션에서 생성하는 *TMTSDataModule* 자손의 기본 이름입니다. 이 이름은 해당 클래스에 대한 인터페이스의 기본 이름이기도 합니다. 예를 들어, 클래스 이름 *MyDataServer*를 지정하면 마법사는 *TMyDataServer*를 선언하는 새 유닛, *IMyDataServer*를 구현하는 *TMTSDataModule*의 자손, *IAppServer*의 자손을 생성합니다.

참고 새 인터페이스에 사용자 고유의 속성 및 메소드를 추가할 수 있습니다. 자세한 내용은 25-17 페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

Transactional Data Module 마법사에서 스레드 모델을 지정해야 합니다. Single, Apartment 또는 Both를 선택합니다.

- Single을 선택하면 애플리케이션에서 한 번에 하나의 클라이언트 요청에만 서비스를 제공하도록 클라이언트 요청이 일련화됩니다. 클라이언트 요청이 서로 방해가 되는지 우려할 필요가 없습니다.
- Apartment를 선택하면 시스템에서 원격 데이터 모듈의 인스턴스가 한 번에 한 요청에만 서비스를 제공하고 호출에 항상 동일한 스레드를 사용하는지 확인합니다. 전역 변수 또는 원격 데이터 모듈에 포함되지 않은 객체를 사용하는 경우 스레드 충돌에 유의해야 합니다. 전역 변수를 사용하는 대신 공유 속성 관리자를 사용할 수 있습니다. 공유 속성 관리자에 대한 자세한 내용은 39-6 페이지의 "Shared Property Manager"를 참조하십시오.
- Both를 선택하면 MTS는 Apartment를 선택했을 때와 마찬가지로 원격 데이터 모듈의 인터페이스에 호출합니다. 하지만 클라이언트 애플리케이션에 대한 콜백이 일련화되므로 서로 방해가 되는지 우려할 필요가 없습니다.

참고 MTS 또는 COM+에서의 Apartment 모델은 DCOM에서의 상응하는 모델과 다릅니다. 원격 데이터 모듈의 트랜잭션 속성도 지정해야 합니다. 다음 옵션 중에서 선택할 수 있습니다.

- 트랜잭션을 요청합니다. 이 옵션을 선택하면 클라이언트가 원격 데이터 모듈의 인터페이스를 사용할 때마다 트랜잭션의 컨텍스트에서 해당 호출이 실행됩니다. 호출자가 트랜잭션을 제공하면 새 트랜잭션이 생성되어야 합니다.
- 새 트랜잭션을 요청합니다. 이 옵션을 선택하면 클라이언트가 원격 데이터 모듈의 인터페이스를 사용할 때마다 해당 호출에 대한 새 트랜잭션이 자동으로 생성됩니다.
- 트랜잭션을 지원합니다. 이 옵션을 선택하면 트랜잭션의 컨텍스트에서 원격 데이터 모듈을 사용할 수 있지만 호출자는 인터페이스 호출 시 트랜잭션을 제공해야 합니다.
- 트랜잭션을 지원하지 않습니다. 이 옵션을 선택하면 트랜잭션의 컨텍스트에서 원격 데이터 모듈을 사용할 수 있습니다.

TSoapDataModule 구성

TSoapDataModule 컴포넌트를 애플리케이션에 추가하려면 File|New|Other를 선택하고 새 항목 대화 상자의 Multitier 페이지에서 SOAP Data Module을 선택합니다. SOAP 데이터 모듈 마법사가 나타납니다.

SOAP 데이터 모듈에 클래스 이름을 제공해야 합니다. 이 이름은 애플리케이션에서 생성하는 *TSoapDataModule* 자손의 기본 이름입니다. 이 이름은 해당 클래스에 대한 인터페이스의 기본 이름이기도 합니다. 예를 들어, 클래스 이름 *MyDataServer*를 지정하면 마법사는 *TMyDataServer*를 선언하는 새 유닛, *IMyDataServer*를 구현하는 *TSoapDataModule*의 자손, *IAppServer* 자손을 생성합니다.

사용자 고유의 속성과 메소드를 추가하여 생성된 인터페이스와 *TSoapDataModule* 자손의 정의를 편집할 수 있습니다. 이러한 속성과 메소드는 자동으로 호출되지는 않지만 이름별 새 인터페이스를 요청하는 클라이언트 애플리케이션에서 추가하는 속성과 메소드를 사용할 수 있습니다.

참고 *TSoapDataModule*을 사용하려면 새로운 데이터 모듈이 웹 서비스 애플리케이션에 추가되어야 합니다. *IAppServer* 인터페이스는 새 유닛의 초기화 섹션에 등록되는 호출 가능 인터페이스입니다. 이를 통해 메인 웹 모듈의 호출자 컴포넌트는 들어오는 모든 호출을 데이터 모듈에 전송할 수 있습니다.

TCorbaDataModule 구성

TCorbaDataModule 컴포넌트를 애플리케이션에 추가하려면 File|New를 선택하고 새 항목 대화 상자의 Multitier 페이지에서 CORBA Data Module을 선택합니다. CORBA Data Module 마법사가 나타납니다.

원격 데이터 모듈에 클래스 이름을 제공해야 합니다. 이 이름은 애플리케이션에서 생성하는 *TCorbaDataModule* 자손의 기본 이름입니다. 이 이름은 해당 클래스에 대한 인터페이스의 기본 이름이기도 합니다. 예를 들어, 클래스 이름 *MyDataServer*를 지정하면 마법사는 *TMyDataServer*를 선언하는 새 유닛, *IMyDataServer*를 구현하는 *TCorbaDataModule*의 자손, *IAppServer* 자손을 생성합니다.

참고 새 인터페이스에 사용자 고유의 속성 및 메소드를 추가할 수 있습니다. 데이터 모듈의 인터페이스에의 추가에 대한 자세한 내용은 25-17 페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

CORBA Data Module 마법사를 통해 서버 애플리케이션에서 원격 데이터 모듈의 인스턴스를 생성하는 방법을 지정할 수 있습니다. 공유 또는 클라이언트별 인스턴스를 선택할 수 있습니다.

- 공유를 선택한 경우 애플리케이션에서 모든 클라이언트 요청을 처리하는 원격 데이터 모듈의 단일 인스턴스를 생성합니다. 이는 종래의 CORBA 개발에 사용된 모델입니다.

- 클라이언트별 인스턴스를 선택한 경우 각 클라이언트 연결에 새 원격 데이터 모듈 인스턴스가 생성됩니다. 이 인스턴스는 클라이언트로부터 메시지가 없는 시간 초과 기간이 경과할 때까지 지속됩니다. 이를 통해 클라이언트가 인스턴스를 더 이상 사용하지 않을 때 서버에서 해당 인스턴스가 해제되지만 클라이언트가 서버의 인터페이스를 장시간 사용하지 않는 경우 너무 일찍 해제될 위험이 있습니다.

참고 모델이 실행하는 프로세스의 인스턴스 수를 결정하는 COM 서버의 인스턴스와 달리 CORBA에서는 인스턴스가 사용자 객체의 생성된 인스턴스 수를 결정합니다. 인스턴스는 모두 서버 실행 파일의 단일 인스턴스 내에서 생성됩니다.

인스턴스 모델 이외에 CORBA Data Module 마법사에서 스레드 모델을 지정해야 합니다. 단일 스레드 또는 다중 스레드를 선택할 수 있습니다.

- 단일 스레드를 선택하면 각 원격 데이터 모듈 인스턴스는 한 번에 하나의 클라이언트 요청만 받습니다. 원격 데이터 모듈에 포함된 객체에 안전하게 액세스할 수 있습니다. 하지만 전역 변수 또는 원격 데이터 모듈에 포함된 객체를 사용하는 경우 스레드 충돌에 유의해야 합니다.
- 다중 스레드를 선택하면 각 클라이언트 연결은 그 고유의 전용 스레드를 갖습니다. 하지만 애플리케이션은 여러 클라이언트에 의해 각각의 개별 스레드가 동시에 호출될 수 있습니다. 전역 메모리뿐만 아니라 인스턴스 데이터의 동시 액세스에 유의해야 합니다. 원격 데이터 모듈에 포함된 모든 객체의 사용을 보호해야 하므로 다중 스레드 서버 작성은 공유 원격 데이터 모듈 인스턴스 사용 시 문제가 될 수 있습니다.

애플리케이션 서버의 인터페이스 확장

클라이언트 애플리케이션은 원격 데이터 모듈의 인스턴스를 생성하거나 연결하여 애플리케이션 서버와 상호 작용합니다. 클라이언트 애플리케이션은 애플리케이션 서버와의 모든 통신에 대한 기반으로 인터페이스를 사용합니다.

원격 데이터 모듈의 인터페이스에 추가하여 클라이언트 애플리케이션에 대한 추가 지원을 제공할 수 있습니다. 이 인터페이스는 *IAppServer* 자손이며 원격 데이터 모듈을 생성할 때 마법사에 의해 자동으로 생성됩니다.

원격 데이터 모듈의 인터페이스에 추가하려면 다음을 수행합니다.

- IDE의 Edit 메뉴에서 Add to Interface 명령을 선택합니다. 프로시저, 함수 또는 속성 추가 여부를 표시하고 그 구문을 입력합니다. OK를 클릭하면 코드 에디터의 새로운 인터페이스 멤버 구현 부분으로 이동합니다.
- 타입 라이브러리 에디터를 사용합니다. 타입 라이브러리 에디터에서 애플리케이션 서버의 인터페이스를 선택하고 추가하는 인터페이스 멤버 (메소드 또는 속성)의 톨 버튼을 클릭합니다. Attributes 페이지에서 인터페이스 멤버에 이름을 제공하고 Parameters 페이지에서 매개변수와 타입을 지정한 다음 타입 라이브러리를 새로 고칩니다. 타입 라이브러리 에디터 사용에 대한 자세한 내용은 34 장 "Type Library 작업"을 참조하십시오. 타입 라이브러리 에디터에서 지정한 대부분의 기능(예: 도움말 컨텍스트, 버전 등)은 CORBA 인터페이스에 적용되지 않습니다. 타입 라이브러리 에디터에서 지정한 이러한 값은 무시됩니다.

참고 *TSoapDataModule* 구현 시 이러한 방법은 적용되지 않습니다. *TSoapDataModule* 자손에서는 서버 인터페이스를 직접 편집해야 합니다.

타입 라이브러리 에디터 또는 Add To Interface 명령을 사용하여 인터페이스에 새 항목을 추가할 때 Delphi에서 수행하는 작업은 COM 기반 (*TRemoteDataModule* 또는 *TMTSDataModule*) 또는 CORBA (*TCorbaDataModule*) 서버 생성 여부에 따라 다릅니다.

- COM 인터페이스에 추가하면 유닛 소스 코드와 타입 라이브러리 파일 (.TLB)에 변경 내용이 추가됩니다.
- CORBA 인터페이스에 추가하면 유닛 소스 코드와 자동 생성된 _TLB 유닛에 변경 내용이 반영됩니다. _TLB 유닛은 유닛의 **uses** 절에 추가됩니다. 우선 바인딩하려면 클라이언트 애플리케이션의 **uses** 절에 이 유닛을 추가해야 합니다. 또한 Export to IDL 버튼을 사용하여 타입 라이브러리 에디터의 .IDL 파일을 저장할 수 있습니다. Interface Repository와 Object Activation Daemon에서 인터페이스를 등록하는데 .IDL 파일이 필요합니다.

참고 타입 라이브러리 에디터에서 Refresh를 선택하고 IDE의 변경 내용을 저장한 다음 TLB 파일을 명시적으로 저장해야 합니다.

원격 데이터 모듈의 인터페이스에 추가한 후 원격 데이터 모듈의 구현에 추가된 속성과 메소드를 찾습니다. 새 메소드의 몸체를 채워 구현을 완성하는 코드를 추가합니다.

클라이언트 애플리케이션은 연결 컴포넌트의 *AppServer* 속성을 사용하여 인터페이스 확장자를 호출합니다. 이러한 방법에 대한 자세한 내용은 25-29 페이지의 "서버 인터페이스 호출"을 참조하십시오.

애플리케이션 서버의 인터페이스에 콜백 추가

콜백을 도입하여 애플리케이션 서버에서 클라이언트 애플리케이션을 호출할 수 있습니다. 이 작업을 수행하려면 클라이언트 애플리케이션은 애플리케이션 서버 메소드 중 하나에 인터페이스를 전달하고 이후 필요 시 애플리케이션 서버에서 이 메소드를 호출합니다. 하지만 원격 데이터 모듈의 인터페이스에 대한 확장자가 콜백을 포함하는 경우에는 HTTP 또는 SOAP 기반 연결을 사용할 수 없습니다. *TWebConnection*은 콜백을 지원하지 않습니다. 소켓 기반 연결을 사용하는 경우 클라이언트 애플리케이션은 *SupportCallbacks* 속성을 설정하여 콜백을 사용하는지 여부를 표시해야 합니다. 다른 모든 연결 타입은 콜백을 자동으로 지원합니다.

트랜잭션 애플리케이션 서버의 인터페이스 확장

트랜잭션 또는 just-in-time 활성화를 사용하는 경우 새로운 모든 메소드는 *SetComplete*를 호출하여 완료 시기가 표시되도록 해야 합니다. 이렇게 하면 트랜잭션이 완료되고 원격 데이터 모듈이 비활성화됩니다.

또한 safe reference를 제공하지 않은 경우 클라이언트가 애플리케이션 서버에서 객체 또는 인터페이스와 직접 통신할 수 있게 해주는 새 메소드의 값을 반환할 수 없습니다. 상태 없는 (stateless) MTS 데이터 모듈을 사용하는 경우 원격 데이터 모듈이 활성화되어 있는지 확인할 수 없으므로 safe reference 사용을 소홀히 하면 작동이 중지될 수 있습니다. safe reference에 대한 자세한 내용은 39-20 페이지의 "객체 참조 전달"을 참조하십시오.

다계층 애플리케이션의 트랜잭션 관리

클라이언트 애플리케이션에서 애플리케이션 서버에 업데이트를 적용하면 프로바이더 컴포넌트는 업데이트 적용 및 트랜잭션의 오류 해결 프로세스를 자동으로 래핑합니다. 이 트랜잭션은 문제 레코드의 수가 *ApplyUpdates* 메소드에 대한 인수로 지정된 *MaxErrors* 값을 초과하지 않는 경우에 커밋됩니다. 그렇지 않을 경우에는 롤백됩니다.

또한 데이터베이스 연결 컴포넌트를 추가하거나 데이터베이스 서버에 SQL을 보냄으로써 트랜잭션을 직접 관리하여 서버 애플리케이션에 트랜잭션 지원을 추가할 수 있습니다. 이것은 2계층 애플리케이션에서 트랜잭션을 관리하는 것과 동일한 방식으로 작동합니다. 이러한 종류의 트랜잭션 제어에 대한 자세한 내용은 17-6 페이지의 "트랜잭션 관리"를 참조하십시오.

트랜잭션 데이터 모듈이 있는 경우 MTS 또는 COM+ 트랜잭션을 사용하여 트랜잭션 지원을 넓힐 수 있습니다. 이러한 트랜잭션은 데이터베이스 액세스뿐 아니라 애플리케이션 서버에 비즈니스 논리를 포함할 수 있습니다. 또한 2단계 커밋을 지원하므로 다중 데이터베이스에까지 확장될 수 있습니다.

BDE와 ADO 기반 데이터 액세스 컴포넌트만 2단계 커밋을 지원합니다. 다중 데이터베이스에 확장되는 트랜잭션을 가지려면 InterbaseExpress 또는 dbExpress 컴포넌트를 사용하지 마십시오.

경고 BDE 사용 시 2단계 커밋은 Oracle7 및 MS-SQL 데이터베이스에서만 완전히 구현됩니다. 트랜잭션에 다중 데이터베이스가 관련되어 있고 데이터베이스 중 일부가 Oracle7 또는 MS-SQL이 아닌 원격 서버인 경우 트랜잭션이 부분적으로만 성공할 수 있는 위험이 약간 있습니다. 하지만 어떤 데이터베이스든지 항상 트랜잭션을 지원합니다.

기본적으로 트랜잭션 데이터 모듈의 모든 *IAppServer* 호출은 트랜잭션적입니다. 트랜잭션에 참여해야 함을 표시하려면 데이터 모듈의 트랜잭션 속성을 설정하면 됩니다. 또한 애플리케이션 서버의 인터페이스를 확장하여 정의하는 트랜잭션을 캡슐화하는 메소드 호출을 포함할 수 있습니다.

트랜잭션 속성이 원격 데이터 모듈에 트랜잭션이 필요함을 표시하면 클라이언트가 자신의 인터페이스에서 메소드를 호출할 때마다 트랜잭션에 자동으로 래핑됩니다. 그런 다음 애플리케이션 서버에 대한 모든 클라이언트 호출은 트랜잭션이 완료되었음을 표시할 때까지 해당 트랜잭션에 참여합니다. 이러한 호출은 전체적으로 성공하거나 또는 롤백됩니다.

참고 MTS 또는 COM+ 트랜잭션을 데이터베이스 연결 컴포넌트 또는 명시적 SQL 명령을 통해 생성된 명시적 트랜잭션과 조합하지 마십시오. 트랜잭션 데이터 모듈이 트랜잭션에 참여하면 트랜잭션의 데이터베이스 호출도 모두 자동으로 참여됩니다.

MTS 및 COM+ 트랜잭션 사용에 대한 자세한 내용은 39-8 페이지의 "MTS 및 COM+ 트랜잭션 지원"을 참조하십시오.

마스터/디테일 관계 지원

테이블 타입 데이터셋을 사용하여 설정하는 것과 동일한 방식으로 클라이언트 애플리케이션의 클라이언트 데이터셋 간에 마스터/디테일 관계를 생성할 수 있습니다. 이러한

마스터/디테일 관계 설정에 대한 자세한 내용은 18-35 페이지의 "마스터/디테일 관계 생성"을 참조하십시오.

하지만 이 방법은 다음과 같은 두 가지 큰 단점이 있습니다.

- 디테일 테이블은 한 번에 하나의 디테일 집합만 사용할지라도 애플리케이션 서버의 레코드 모두를 폐치하고 저장합니다. 이 문제는 매개변수를 사용하면 해결됩니다. 자세한 내용은 23-29 페이지의 "매개변수로 레코드 제한"을 참조하십시오.
- 클라이언트 데이터셋은 데이터셋 레벨에서 업데이트를 적용하는데 마스터/디테일 업데이트는 다중 데이터셋에 확장되므로 업데이트를 적용하기가 매우 어렵습니다. 데이터베이스 연결 컴포넌트를 사용하여 단일 트랜잭션의 다중 테이블에 대한 업데이트를 적용하는 2계층 환경에서도 마스터/디테일 폼에 업데이트를 적용하는 것은 까다롭습니다.

다계층 애플리케이션에서 중첩 테이블을 사용하여 마스터/디테일 관계를 나타내면 이러한 문제를 피할 수 있습니다. 데이터셋에서 제공하는 경우 이 작업을 수행하려면 애플리케이션 서버의 데이터셋 간에 마스터/디테일 관계를 설정합니다. 그런 다음 프로바이더 컴포넌트의 *DataSet* 속성을 마스터 테이블로 설정합니다. XML 문서에서 제공하는 경우 중첩 테이블을 사용하여 마스터/디테일 관계를 나타내려면 중첩 디테일 집합을 정의하는 변환 파일을 사용합니다.

클라이언트가 프로바이더의 *GetRecords* 메소드를 호출하면 자동적으로 데이터 패킷의 레코드 안에 *DataSet* 필드로서 디테일 데이터셋을 포함시킵니다. 클라이언트가 프로바이더의 *ApplyUpdates* 메소드를 호출하면 자동적으로 업데이트 적용이 적절한 순서로 처리됩니다.

원격 데이터 모듈의 상태 정보 지원

애플리케이션 서버에서 클라이언트 데이터셋과 프로바이더 간에 모든 통신을 제어하는 *IAppServer* 인터페이스는 대부분 상태 없음 (stateless)입니다. 애플리케이션이 상태 없음이면 클라이언트의 이전 호출에서 발생한 것을 "기억"하지 않습니다. 이러한 상태 없음 특성은 애플리케이션 서버에서 레코드 Currency와 같은 영구적 정보에 대한 데이터베이스 연결 간에 구별할 필요가 없으므로 트랜잭션 데이터 모듈에서 데이터베이스 연결을 풀링할 때 유용합니다. 마찬가지로 이러한 상태 없음 특성은 just-in-time 활성화, 객체 풀링 또는 일반적인 CORBA 서버에서 발생하므로 여러 클라이언트 간에 원격 데이터 모듈 인스턴스를 공유할 때 중요합니다. SOAP 데이터 모듈은 상태 없음이어야 합니다.

하지만 애플리케이션 서버에 대한 호출 간의 상태 정보를 유지하려는 경우가 있습니다. 예를 들어, 잠긴적 폐칭을 사용하는 데이터 요청 시 애플리케이션 서버의 프로바이더는 이전 호출의 정보(현재 레코드)를 "기억"해야 합니다.

클라이언트 데이터셋이 만드는 *IAppServer* 인터페이스 (*AS_ApplyUpdates*, *AS_Execute*, *AS_GetParams*, *AS_GetRecords* 또는 *AS_RowRequest*) 를 호출하기 전후에 고객 상태 정보를 가져오거나 보낼 수 있는 이벤트를 받습니다. 마찬가지로 프로바이더가 이러한 클라이언트가 생성한 호출에 응답하기 전후에 고객 상태 정보를 가져오거나 보낼 수 있는 이벤트를 받습니다. 이 메커니즘을 사용하면 애플리케이션 서버가 상태 없음인 경우에도 클라이언트 애플리케이션과 애플리케이션 서버 간에 영구적인 상태 정보를 통신할 수 있습니다.

예를 들어, 다음과 같은 매개변수화된 쿼리를 나타내는 데이터셋을 고려해 봅시다.

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

상태 없음 애플리케이션 서버에서 점진적 페칭 (incremental fetching) 을 활성화하려면 다음을 수행합니다.

- 프로바이더가 데이터 패킷의 레코드 집합을 패키지화하면 패킷의 최종 레코드에 CUST_NO의 값이 적습니다.

```
TRemoteDataModule1.DataSetProvider1GetData(Sender:TObject;DataSet:TCustomClientDataSet);
begin
  DataSet.Last; { move to the last record }
  with Sender as TDataSetProvider do
    Tag := DataSet.FieldValues['CUST_NO']; {save the value of CUST_NO }
  end;
```

- 프로바이더는 데이터 패킷을 보낸 후 CUST_NO 최종 값을 클라이언트에게 보냅니다.

```
TRemoteDataModule1.DataSetProvider1AfterGetRecords(Sender:TObject;
  var OwnerData:OleVariant);
begin
  with Sender as TDataSetProvider do
    OwnerData := Tag; {send the last value of CUST_NO }
  end;
```

- 클라이언트에서 클라이언트 데이터셋은 CUST_NO 최종 값을 저장합니다.

```
TDataModule1.ClientDataSet1AfterGetRecords(Sender:TObject; var OwnerData:OleVariant);
begin
  with Sender as TClientDataSet do
    Tag := OwnerData; {save the last value of CUST_NO }
  end;
```

- 데이터 패킷을 페칭하기 전에 클라이언트는 받은 CUST_NO의 최종 값을 보냅니다.

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender:TObject; var OwnerData:OleVariant);
begin
  with Sender as TClientDataSet do
    begin
      if not Active then Exit;
      OwnerData := Tag; { Send last value of CUST_NO to application server }
    end;
  end;
```

- 마지막으로 서버에서 프로바이더는 쿼리에서 최소 값으로 보낸 CUST_NO 최종 값을 사용합니다.

```

TRemoteDataModule1.DataSetProvider1.BeforeGetRecords(Sender:TObject;
                var OwnerData:OleVariant);
begin
  if not VarIsEmpty(OwnerData) then
    with Sender as TDataSetProvider do
      with DataSet as TSQLDataSet do
        begin
          Params.ParamValues['MinVal'] := OwnerData;
          Refresh; { force the query to reexecute }
        end;
      end;
    end;
  end;
end;

```

다중 원격 데이터 모듈 사용

다중 원격 데이터 모듈을 사용할 수 있도록 애플리케이션 서버를 구성할 수 있습니다. 다중 원격 데이터 모듈을 사용하면 코드를 분할할 수 있으므로 대형 애플리케이션 서버를 각 유닛이 상대적으로 자체 내장된 여러 유닛으로 구성할 수 있습니다.

독립적으로 기능하는 애플리케이션 서버에서 언제나 다중 원격 데이터 모듈을 생성할 수 있지만 Delphi는 클라이언트로부터 다른 "자식" 원격 데이터 모듈로의 연결을 디스패치하는 하나의 메인 "부모" 원격 데이터 모듈을 갖는 모델을 지원합니다.

부모 원격 데이터 모듈을 생성하려면 해당 *IAppServer* 인터페이스를 확장하여 자식 원격 데이터 모듈의 인터페이스를 노출시키는 속성을 추가해야 합니다. 즉, 각각의 자식 원격 데이터 모듈에 대해 값이 자식 데이터 모듈의 *IAppServer* 인터페이스인 부모 데이터 모듈의 인터페이스에 속성을 추가합니다. getter 속성은 다음과 같습니다.

```

function ParentRDM.Get_ChildRDM: IChildRDM;
begin
  {note the parent RDM uses a factory component defined in the child RDM's unit.
  This is more efficient if it must create several children for different clients }
  Result := ChildRDMFactory.CreateCOMObject(nil) as IChildRDM;
  Result.ParentRDM := Self;
end;

```

부모 원격 데이터 모듈의 인터페이스 확장에 대한 내용은 25-17 페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

팁 각 자식 데이터 모듈에서 인터페이스를 확장하여 부모 데이터 모듈의 인터페이스나 다른 자식 데이터 모듈의 인터페이스를 노출할 수도 있습니다. 이를 통해 애플리케이션 서버의 다양한 데이터 모듈 간에 서로 좀더 자유롭게 통신할 수 있습니다.

자식 원격 데이터 모듈을 나타내는 속성을 메인 원격 데이터 모듈에 추가했으면 클라이언트 애플리케이션에서 애플리케이션 서버의 각 원격 데이터 모듈에 별도로 연결할 필요가 없습니다. 그 대신 클라이언트 애플리케이션에서 "자식" 데이터 모듈에 메시지를 디스패치하는 부모 원격 데이터 모듈에 대한 단일 연결을 공유합니다. 각 클라이언트 애플리케이션에서 모든 원격 데이터 모듈에 동일한 연결을 사용하므로 원격 데이터 모듈은 단일 데이터베이스 연결을 공유하여 리소스를 보존할 수 있습니다. 자식 애플리케이션에서 단일 연결을 공유하는 방법에 대한 내용은 25-31 페이지의 "다중 데이터 모듈을 사용하는 애플리케이션 서버에 대한 연결"을 참조하십시오.

애플리케이션 서버 등록

클라이언트 애플리케이션에서 애플리케이션 서버를 찾고 사용하려면 먼저 등록 또는 설치해야 합니다. (등록이 권장되기는 하나 CORBA 애플리케이션 서버에서는 반드시 등록하지 않아도 됩니다.)

- 애플리케이션 서버에서 DCOM, HTTP 또는 소켓을 통신 프로토콜로 사용하는 경우 Automation 서버로 동작하며 다른 COM 서버처럼 등록되어야 합니다. COM 서버 등록에 대한 내용은 36-17 페이지의 "COM 객체 등록"을 참조하십시오.
- 트랜잭션 데이터 모듈을 사용하는 경우 애플리케이션 서버를 등록하지 않습니다. 그 대신 MTS 또는 COM+와 함께 설치합니다. 트랜잭션 객체 설치에 대한 내용은 39-22 페이지의 "트랜잭션 객체 설치"를 참조하십시오.
- 애플리케이션 서버에서 SOAP를 사용할 때 애플리케이션은 웹 서비스 애플리케이션이어야 합니다. 그러한 경우에는 수신 HTTP 메시지를 받을 수 있도록 웹 서버에 등록해야 합니다. 또한 Delphi를 사용하여 작성되지 않은 클라이언트가 애플리케이션의 모든 인터페이스에 액세스할 수 있도록 하기 위해서 애플리케이션의 호출 가능한 인터페이스를 나타내는 WSDL 문서를 게시할 수 있습니다. 웹 서비스 애플리케이션의 WSDL 문서 export에 대한 내용은 31-7 페이지의 "Web Services 애플리케이션에 대한 WSDL 문서 생성"을 참조하십시오.
- 애플리케이션 서버에서 CORBA를 사용할 때 등록은 옵션입니다. 클라이언트 애플리케이션에서 인터페이스에 대해 동적 바인딩을 사용하게 하려면 Interface Repository에 서버의 인터페이스를 설치해야 합니다. 또한 클라이언트 애플리케이션에서 아직 실행 중이지 않은 애플리케이션 서버를 실행하게 하려면 OAD(Object Activation Daemon)로 등록해야 합니다.

클라이언트 애플리케이션 생성

대부분의 경우 다계층 클라이언트 애플리케이션 생성은 업데이트를 캐시하는 데 클라이언트 데이터셋을 사용하는 2계층 클라이언트 생성과 유사합니다. 주요 차이점은 다계층 클라이언트는 연결 컴포넌트를 사용하여 애플리케이션 서버에 연결한다는 것입니다.

다계층 클라이언트 애플리케이션을 생성하려면 새 프로젝트를 시작하고 다음 단계를 수행합니다.

- 1 프로젝트에 새 데이터 모듈을 추가합니다.
- 2 데이터 모듈에 연결 컴포넌트를 놓습니다. 추가할 연결 컴포넌트의 타입은 사용할 통신 프로토콜에 따라 다릅니다. 자세한 내용은 25-4 페이지의 "클라이언트 애플리케이션의 구조"를 참조하십시오.
- 3 연결 컴포넌트에 대한 속성을 설정하여 연결을 설정할 애플리케이션 서버를 지정합니다. 연결 컴포넌트 설정에 대한 자세한 내용은 25-24 페이지의 "애플리케이션 서버에 연결"을 참조하십시오.

- 4 애플리케이션에 필요한 경우 다른 연결 컴포넌트 속성을 설정합니다. 예를 들어, 연결 컴포넌트가 여러 서버로부터 동적으로 선택하도록 *ObjectBroker* 속성을 설정할 수도 있습니다. 연결 컴포넌트 사용에 대한 자세한 내용은 25-28 페이지의 "서버 연결 관리"를 참조하십시오.
- 5 데이터 모듈에 필요한 만큼 *TClientDataSet* 컴포넌트를 놓고 각 컴포넌트의 *RemoteServer* 속성을 2단계에서 가져다 놓은 연결 컴포넌트의 이름으로 설정합니다. 클라이언트 데이터셋에 대한 전체적인 소개는 23장 "클라이언트 데이터셋 사용"을 참조하십시오.
- 6 각 *TClientDataSet* 컴포넌트의 *ProviderName* 속성을 설정합니다. 디자인 타임 시 연결 컴포넌트가 애플리케이션 서버에 연결된 경우 *ProviderName* 속성의 드롭다운 목록에서 사용 가능한 애플리케이션 서버 프로바이더를 선택할 수 있습니다.
- 7 다른 데이터베이스 애플리케이션을 생성하는 것과 동일한 방식으로 계속 진행합니다. 다계층 애플리케이션의 클라이언트가 사용할 수 있는 몇 가지 추가 기능은 다음과 같습니다.
 - 애플리케이션에서 애플리케이션 서버에 직접 호출할 수 있습니다. 25-29 페이지의 "서버 인터페이스 호출"에서 이를 수행하는 방법에 대해 설명합니다.
 - 프로바이더 컴포넌트와의 상호 작용을 지원하는 클라이언트 데이터셋의 특별한 기능을 사용할 수 있습니다. 이 내용은 23-25 페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"에서 설명합니다.

애플리케이션 서버에 연결

애플리케이션 서버에 대한 연결을 설정 및 유지하기 위해 클라이언트 애플리케이션은 하나 이상의 연결 컴포넌트를 사용합니다. 컴포넌트 팔레트의 DataSnap 페이지에 이러한 컴포넌트가 있습니다.

연결 컴포넌트를 사용하여 다음을 수행합니다.

- 애플리케이션 서버와 통신하기 위해 프로토콜을 식별합니다. 각각의 연결 컴포넌트 타입은 다른 통신 프로토콜을 나타냅니다. 사용 가능한 프로토콜의 이점 및 제한 사항에 대한 자세한 내용은 25-9 페이지의 "연결 프로토콜 선택"을 참조하십시오.
- 서버 컴퓨터를 찾는 방법을 표시합니다. 서버를 식별하는 방법은 프로토콜에 따라 다릅니다.
- 서버 컴퓨터의 애플리케이션 서버를 확인합니다.

CORBA를 사용하지 않는 경우 *ServerName* 또는 *ServerGUID* 속성을 사용하여 서버를 식별합니다. *ServerName*은 애플리케이션 서버에서 원격 데이터 모듈 생성 시 지정하는 클래스의 기본 이름을 식별합니다. 서버에서 이 값을 지정하는 방법에 대한 자세한 내용은 25-13 페이지의 "원격 데이터 모듈 설정"을 참조하십시오. 서버가 클라이언트 컴퓨터에 등록 또는 설치되어 있는 경우나 연결 컴포넌트가 서버 컴퓨터에 연결되어 있는 경우, Object Inspector의 드롭다운 목록에서 선택하여 디자인 타임 시 *ServerName* 속성을 설정할 수 있습니다. *ServerGUID*는 원격 데이터 모듈 인터페이스의 GUID를 지정합니다. 타입 라이브러리 에디터에서 이 값을 조회할 수 있습니다.

CORBA를 사용하는 경우 *RepositoryID* 속성으로 서버를 식별합니다. *RepositoryID*는 애플리케이션 서버 팩토리 인터페이스의 *Repository ID*를 지정하며 CORBA 서버 구현 유닛의 초기화 섹션에 자동으로 추가되는 *TCorbaVCLComponentFactory.Create* 호출의 세 번째 인수로 나타납니다. 이 속성을 CORBA 데이터 모듈 인터페이스의 기본 이름으로 설정하면(다른 연결 컴포넌트의 *ServerName* 속성과 동일한 문자열) 적절한 *Repository ID*로 자동 변환됩니다.

- 서버 연결을 관리합니다. 연결 컴포넌트를 사용하여 연결을 생성 또는 삭제하고 애플리케이션 서버 인터페이스를 호출할 수 있습니다.

일반적으로 애플리케이션 서버는 클라이언트 애플리케이션과 다른 컴퓨터에 있지만 서버가 클라이언트 애플리케이션과 동일한 컴퓨터에 상주하는 경우에도(예: 전체 다계층 애플리케이션 빌드 및 테스트) 연결 컴포넌트를 사용하여 이름별로 애플리케이션을 식별하고, 서버 컴퓨터를 지정하며, 애플리케이션 서버 인터페이스를 사용할 수 있습니다.

DCOM을 사용하여 연결 지정

애플리케이션 서버와 통신하기 위해 DCOM을 사용하는 경우 클라이언트 애플리케이션은 애플리케이션 서버에 연결하기 위한 *TDCOMConnection* 컴포넌트를 포함시킵니다. *TDCOMConnection*은 *ComputerName* 속성을 사용하여 서버가 상주하는 컴퓨터를 식별합니다.

*ComputerName*이 공백일 때 DCOM 연결 컴포넌트는 애플리케이션 서버가 클라이언트 컴퓨터에 상주하거나 애플리케이션 서버에 시스템 레지스트리 항목이 있는 것으로 가정합니다. DCOM 사용 시 클라이언트의 애플리케이션 서버에 대한 시스템 레지스트리 항목을 제공하지 않고 서버가 클라이언트와 다른 컴퓨터에 상주하는 경우 *ComputerName*을 제공해야 합니다.

참고 애플리케이션 서버에 시스템 레지스트리 항목이 있는 경우에도 *ComputerName*을 지정하여 이 항목을 오버라이드할 수 있습니다. 이 방법은 특히 개발, 테스트 및 디버깅 중에 유용하게 사용됩니다.

클라이언트 애플리케이션에서 선택할 수 있는 다중 서버가 있는 경우 *ComputerName*의 값을 지정하는 대신 *ObjectBroker* 속성을 사용할 수 있습니다. 자세한 내용은 25-28 페이지의 "연결 브로커링(Brokering connections)"을 참조하십시오.

제공한 호스트 컴퓨터 또는 서버의 이름이 없는 경우 DCOM 연결 컴포넌트는 연결을 열려고 시도할 때 예외를 발생시킵니다.

소켓을 사용하여 연결 지정

TCP/IP 주소를 갖는 컴퓨터의 소켓을 사용하여 애플리케이션 서버에 연결할 수 있습니다. 이 메소드는 보다 많은 컴퓨터에 적용되는 이점이 있지만 보안 프로토콜 사용에는 제공되지 않습니다. 소켓 사용 시 애플리케이션 서버에 연결하기 위한 *TSocketConnection* 컴포넌트를 포함시킵니다.

*TSocketConnection*은 서버 시스템의 IP 주소 또는 호스트 이름, 서버 컴퓨터에서 실행 중인 소켓 디스패처 프로그램(Scktsrvr.exe)의 포트 번호를 통해 서버 컴퓨터를 식별합니다. IP 주소와 포트 값에 대한 자세한 내용은 32-3 페이지의 "소켓 설명"을 참조하십시오.

*TSocketConnection*의 세 가지 속성은 다음과 같은 정보를 지정합니다.

- *Address*는 서버의 IP 주소를 지정합니다.
- *Host*는 서버의 호스트 이름을 지정합니다.
- *Port*는 애플리케이션 서버의 소켓 디스패처 프로그램의 포트 번호를 지정합니다.

*Address*와 *Host*는 상호 배타적입니다. 하나의 값을 설정하면 다른 하나의 값이 설정 해제됩니다. 둘 중 어느 것을 사용할지에 대해서는 32-4 페이지의 "호스트 설명"을 참조하십시오.

클라이언트 애플리케이션에서 선택할 수 있는 다중 서버가 있는 경우 *Address* 또는 *Host*의 값을 지정하는 대신 *ObjectBroker* 속성을 사용할 수 있습니다. 자세한 내용은 25-28 페이지의 "연결 브로커링(Brokering connections)"을 참조하십시오.

기본적으로 *Port*의 값은 211이고 이 값은 Delphi에서 제공되는 소켓 디스패처 프로그램의 기본 포트 번호입니다. 소켓 디스패처가 다른 포트를 사용하도록 구성된 경우 해당 값과 일치하도록 *Port* 속성을 설정합니다.

참고 Borland Socket Server 트레이 아이콘을 마우스 오른쪽 버튼으로 클릭하고 Properties를 선택하여 실행 중일 동안 소켓 디스패처의 포트를 구성할 수 있습니다.

소켓 연결이 보안 프로토콜 사용에 제공되지 않아도 소켓 연결을 사용자 지정하여 사용자 고유의 암호화를 추가할 수 있습니다. 다음과 같은 방법으로 이 작업을 수행합니다.

- 1 *IDataIntercept* 인터페이스를 지원하는 COM 객체를 생성합니다. 이것은 데이터를 암호화하고 암호 해독하기 위한 인터페이스입니다.
- 2 *TPacketInterceptFactory*를 이 객체의 클래스 팩토리로 사용합니다. 1 단계에서 마법사를 사용하여 COM 객체를 생성하는 경우 *TComponentFactory.Create(...)*가 나타나 있는 초기화 섹션의 줄을 *TPacketInterceptFactory.Create(...)*으로 대체합니다.
- 3 클라이언트 컴퓨터에 새로운 COM 서버를 등록합니다.
- 4 소켓 연결 컴포넌트의 *InterceptName* 또는 *InterceptGUID* 속성을 설정하여 이 COM 객체를 지정합니다. 2 단계에서 *TPacketInterceptFactory*를 사용한 경우 *InterceptName* 속성에 대한 Object Inspector의 드롭다운 목록에 COM 서버가 나타납니다.
- 5 마지막으로 Borland Socket Server 트레이 아이콘을 마우스 오른쪽 버튼으로 클릭하고 Properties를 선택한 다음 Properties 탭에서 인터셉터의 Intercept Name 또는 Intercept GUID를 ProgId 또는 GUID로 설정합니다.

이 메커니즘은 데이터 압축과 압축 해제에 사용할 수도 있습니다.

HTTP를 사용하여 연결 지정

TCP/IP 주소를 갖는 컴퓨터의 HTTP를 사용하여 애플리케이션 서버에 연결할 수 있습니다. 그러나 소켓과 달리 HTTP를 사용하면 SSL 보안을 이용할 수 있고 방화벽에 의해 보호되는 서버와 통신할 수 있습니다. HTTP 사용 시 애플리케이션 서버에 연결하기 위한 *TWebConnection* 컴포넌트를 포함시킵니다.

웹 연결 컴포넌트는 웹 서버 애플리케이션(httpsrvr.dll)에 연결하여 교대로 애플리케이션 서버와 통신합니다. *TWebConnection*은 URL(Uniform Resource Locator)을 사용하여 httpsrvr.dll을 찾습니다. URL은 프로토콜(http, 또는 SSL 보안을 사용하는 경우 https), 웹 서버와 httpsrvr.dll, 웹 서버 애플리케이션에 대한 경로(httpsrvr.dll)를 실행하는 컴퓨터의 호스트 이름을 지정합니다. *URL* 속성을 사용하여 이 값을 지정합니다.

참고 *TWebConnection* 사용 시 wininet.dll이 클라이언트 컴퓨터에 설치되어 있어야 합니다. IE3 이상이 설치되어 있는 경우, wininet.dll은 Windows 시스템 디렉토리에 있습니다.

웹 서버에 인증이 필요한 경우 또는 인증이 필요한 프록시 서버를 사용하고 있을 경우, 연결 컴포넌트가 로그인할 수 있도록 *UserName* 속성 값과 *Password* 속성 값을 설정해야 합니다.

클라이언트 애플리케이션이 선택할 수 있는 다중 서버가 있는 경우 *URL*의 값을 지정하는 대신 *ObjectBroker* 속성을 사용할 수 있습니다. 자세한 내용은 25-28 페이지의 "연결 브로커링(Brokering connections)"을 참조하십시오.

SOAP를 사용하여 연결 지정

TSoapConnection 컴포넌트를 사용하여 SOAP 애플리케이션 서버에 연결할 수 있습니다. *TSoapConnection*도 HTTP를 전송 프로토콜로 사용한다는 점에서 *TWebConnection*과 매우 유사합니다. 그러므로 TCP/IP 주소를 갖는 컴퓨터의 *TSoapConnection*을 사용하여 방화벽에 의해 보호되는 서버와 통신할 수 있는 SSL 보안을 이용할 수 있습니다.

SOAP 연결 컴포넌트는 웹 서비스로서 *IAppServer* 인터페이스를 구현하는 웹 서버 애플리케이션에 연결합니다. *TSoapConnection*은 URL(Uniform Resource Locator)을 사용하여 웹 서버 애플리케이션을 찾습니다. URL은 프로토콜(http, 또는 SSL 보안을 사용하는 경우 https), 웹 서버를 실행하는 컴퓨터의 호스트 이름, 웹 서비스 애플리케이션의 이름, 애플리케이션 서버에서 *THTTPSoapDispatcher*의 경로 이름과 일치하는 경로를 지정합니다. *URL* 속성을 사용하여 이 값을 지정합니다.

참고 *TSoapConnection* 사용 시 wininet.dll이 클라이언트 컴퓨터에 설치되어 있어야 합니다. IE3 이상이 설치되어 있는 경우, wininet.dll은 Windows 시스템 디렉토리에 있습니다.

웹 서버에 인증이 필요한 경우 또는 인증이 필요한 프록시 서버를 사용하고 있을 경우, 연결 컴포넌트가 로그인할 수 있도록 *UserName* 속성 값과 *Password* 속성 값을 설정해야 합니다.

CORBA를 사용하여 연결 지정

단지 *RepositoryID* 속성만 지정하면 CORBA 연결을 할 수 있습니다. 왜냐하면 로컬 네트워크의 Smart Agent가 CORBA 클라이언트에 사용 가능한 서버를 자동으로 찾기 때문입니다.

하지만 CORBA 연결 컴포넌트의 다른 속성들을 이용해서 클라이언트 애플리케이션이 연결할 수 있는 서버를 제한할 수 있습니다. CORBA Smart Agent를 통해 사용 가능한 서버를 찾지 않고 특정 서버 컴퓨터를 지정하려면 *HostName* 속성을 사용합니다. 서버

인스턴스를 구현하는 객체 인스턴스가 두 개 이상 있는 경우 *ObjectName* 속성을 설정하여 사용할 객체를 지정할 수 있습니다.

TCorbaConnection 컴포넌트는 다음의 두 가지 방법 중 하나로 애플리케이션 서버의 CORBA 데이터 모듈에 대한 인터페이스를 가집니다.

- 우선 바인딩 (정적 바인딩) 을 사용하는 경우 타입 라이브러리 에디터에서 생성한 *_TLB.pas* 파일을 클라이언트 애플리케이션에 추가해야 합니다. 컴파일 시 타입을 확인할 수 있고 지연 바인딩 (동적 바인딩) 보다 훨씬 빠르므로 우선 바인딩을 사용하는 것이 좋습니다.
- 지연 바인딩 (동적 바인딩) 을 사용하는 경우 Interface Repository에 인터페이스를 등록해야 합니다.

우선 바인딩과 지연 바인딩 비교에 대한 자세한 내용은 25-29 페이지의 "서버 인터페이스 호출"을 참조하십시오.

연결 브로커링(Brokering connections)

클라이언트 애플리케이션에서 선택할 수 있는 다중 서버가 있는 경우 Object Broker를 통해 사용 가능한 서버 시스템을 찾습니다. Object Broker는 연결 컴포넌트가 선택할 수 있는 서버의 목록을 유지합니다. 연결 컴포넌트는 애플리케이션 서버에 연결해야 할 때 Object Broker에게 컴퓨터 이름(또는 IP 주소, 호스트 이름 또는 URL)을 묻습니다. 브로커에서 이름을 제공하면 연결 컴포넌트는 연결을 구성합니다. 제공된 이름을 사용할 수 없는 경우(예를 들어, 서버 다운 시) 연결이 구성될 때까지 브로커가 다른 이름을 제공합니다.

연결 컴포넌트가 브로커에서 제공한 이름의 연결을 구성했으면 그 이름을 적절한 속성의 값으로 저장합니다(*ComputerName*, *Address*, *Host*, *RemoteHost* 또는 *URL*). 연결 컴포넌트가 연결을 닫은 이후 연결을 다시 열어야 하는 경우 이 속성 값을 사용하여 연결 실패 시에만 브로커로부터 새 이름을 요청합니다.

연결 컴포넌트의 *ObjectBroker* 속성을 지정하여 Object Broker를 사용합니다. *ObjectBroker* 속성이 설정되어 있으면 연결 컴포넌트는 *ComputerName*, *Address*, *Host*, *RemoteHost* 또는 *URL*의 값을 디스크에 저장하지 않습니다.

참고 CORBA 연결을 위해 *ObjectBroker* 속성을 사용하지 마십시오. CORBA는 자신의 브로커링 메커니즘을 갖습니다.

서버 연결 관리

애플리케이션 서버를 찾고 연결하는 것이 연결 컴포넌트의 주 목적입니다. 연결 컴포넌트는 서버 연결을 관리하므로 애플리케이션 서버 인터페이스의 메소드를 호출하는 데에도 연결 컴포넌트를 사용할 수 있습니다.

서버에 연결

애플리케이션 서버를 찾고 연결하려면 먼저 연결 컴포넌트의 속성을 설정하여 애플리케이션 서버를 식별해야 합니다. 이 프로세스는 25-24 페이지의 "애플리케이션 서버에 연결"에서 다룹니다. 연결을 열기 전에 애플리케이션 서버와 통신하는 데 연결 컴포넌트를 사용하는 클라이언트 데이터셋은 연결 컴포넌트를 지정하는 *RemoteServer* 속성을 설정하여 나타내야 합니다.

클라이언트 데이터셋이 애플리케이션 서버에 대한 액세스를 시도하면 연결은 자동으로 열립니다. 예를 들어, 클라이언트 데이터셋의 *Active* 속성을 *True*로 설정하면 *RemoteServer* 속성이 설정되어 있는 한 연결이 열립니다.

클라이언트 데이터셋을 연결 컴포넌트와 연결하지 않은 경우 연결 컴포넌트의 *Connected* 속성을 *True*로 설정하여 연결을 열 수 있습니다.

연결 컴포넌트에서 애플리케이션 서버에 연결하기 전에 *BeforeConnect* 이벤트를 생성합니다. 코딩하는 *BeforeConnect* 핸들러로 연결하기 전에 사용자가 특별한 동작을 수행할 수 있습니다. 연결 컴포넌트는 연결 후 특별한 동작에 대한 *AfterConnect* 이벤트를 생성합니다.

서버 연결 끊기 및 바꾸기

연결 컴포넌트는 다음 작업을 수행할 때 애플리케이션 서버에 대한 연결을 끊습니다.

- *Connected* 속성을 *False*로 설정합니다.
- 연결 컴포넌트를 해제합니다. 연결 객체는 사용자가 클라이언트 애플리케이션을 닫으면 자동으로 해제됩니다.
- 애플리케이션 서버를 식별하는 속성을 변경합니다(*ServerName*, *ServerGUID*, *ComputerName* 및 기타 등등). 이러한 속성을 변경하면 런타임 시 사용 가능한 애플리케이션 서버 간에 전환할 수 있습니다. 연결 컴포넌트는 현재 연결을 끊고 새 연결을 설정합니다.

참고 단일 연결 컴포넌트를 사용하여 사용 가능한 애플리케이션 서버 간에 전환하는 대신 클라이언트 애플리케이션은 연결 컴포넌트를 두 개 이상 가질 수 있으며 각 연결 컴포넌트는 다른 애플리케이션 서버에 연결됩니다.

연결 컴포넌트는 연결을 끊기 전에 *BeforeDisconnect* 이벤트 핸들러를 자동으로 호출합니다. 연결 해체에 앞서 특별한 동작을 수행하려면 *BeforeDisconnect* 핸들러를 작성합니다. 마찬가지로 연결을 끊은 후 *AfterDisconnect* 이벤트 핸들러가 호출됩니다. 연결 해제 후 특별한 동작을 수행하려면 *AfterDisconnect* 핸들러를 작성합니다.

서버 인터페이스 호출

클라이언트 데이터셋의 속성과 메소드 사용 시 적절한 호출이 자동적으로 이루어지므로 애플리케이션에서 *IAppServer* 인터페이스를 직접 호출할 필요가 없습니다. *IAppServer* 인터페이스를 직접 사용할 필요는 없지만 사용자 고유의 확장자를 원격 데이터 모듈 인터페이스에 추가할 수 있습니다. 애플리케이션 서버의 인터페이스를 확장하는 경우 연결 컴포넌트에 의해 생성된 연결을 사용하여 해당 확장자를 호출하기 위한

방법이 필요합니다. SOAP를 사용하지 않는 경우 연결 컴포넌트의 *AppServer* 속성을 사용하여 이 작업을 수행할 수 있습니다. 애플리케이션 서버의 인터페이스 확장에 대한 내용은 25-17 페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

*AppServer*는 애플리케이션 서버의 인터페이스를 나타내는 가변입니다. 다음과 같은 문장을 작성하여 *AppServer*를 사용하는 인터페이스 메소드를 호출할 수 있습니다.

```
MyConnection.AppServer.SpecialMethod(x,y);
```

하지만 이 방법은 인터페이스 호출의 지연 바인딩(동적 바인딩)을 제공합니다. 즉, *SpecialMethod* 프로시저 호출은 호출이 실행되는 런타임 시까지 바인딩되지 않습니다. 지연 바인딩은 매우 유연하지만 사용할 경우 코드 통찰력과 타입 확인성 등의 여러 가지 이점을 잃게 됩니다. 또한 컴파일러에서 호출하기 전에 인터페이스 호출을 설정하기 위해 서버에 대한 추가 호출을 생성하므로 지연 바인딩은 우선 바인딩보다 속도가 느립니다.

DCOM 또는 CORBA를 통신 프로토콜로 사용하는 경우 *AppServer* 호출의 우선 바인딩을 사용할 수 있습니다. **as** 연산자를 사용하여 원격 데이터 모듈 생성 시 만든 *IAppServer* 자손으로 *AppServer* 변수를 타입 변환합니다. 예를 들면, 다음과 같습니다.

```
with MyConnection.AppServer as IMyAppServer do
  SpecialMethod(x,y);
```

DCOM에서 우선 바인딩을 사용하려면 서버의 타입 라이브러리를 클라이언트 컴퓨터에 등록해야 합니다. Delphi와 함께 제공되는 TRegsvr.exe를 사용하여 타입 라이브러리를 등록할 수 있습니다.

참고 프로그램에서 타입 라이브러리를 등록하는 방법의 예는 TRegSvr demo(TRegsvr.exe의 소스 제공)를 참조하십시오.

CORBA에서 우선 바인딩을 사용하려면 타입 라이브러리 에디터에 의해 생성된 _TLB 유닛을 사용자의 프로젝트에 추가해야 합니다. 이 작업을 수행하려면 이 유닛을 사용자 유닛의 **uses** 절에 추가합니다.

TCP/IP 또는 HTTP 사용 시 진정한 의미의 우선 바인딩을 사용할 수는 없지만 원격 데이터 모듈이 이중 인터페이스를 사용하므로 애플리케이션 서버의 dispinterface를 사용하여 간단한 지연 바인딩에 대한 성능을 향상시킬 수 있습니다. dispinterface는 문자열 'Disp'가 붙은 원격 데이터 모듈의 인터페이스 이름과 동일한 이름을 갖습니다. *AppServer* 속성을 이 타입의 변수에 할당하여 dispinterface를 가질 수 있습니다.

```
var
  TempInterface: IMyAppServerDisp;
begin
  TempInterface :=IMyAppServerDisp(IDispatch(MyConnection.AppServer));
  ...
  TempInterface.SpecialMethod(x,y);
  ...
end;
```

참고 dispinterface를 사용하려면 타입 라이브러리 저장 시 생성되는 _TLB 유닛을 클라이언트 모듈의 **uses** 절에 추가해야 합니다.

SOAP를 사용하는 경우 *AppServer* 속성을 사용할 수 없습니다. 그 대신 원격 인터페이스 객체 (*THHTTPRio*)를 사용하여 우선 바운드 호출을 만들 수 있습니다. 모든 우선 바운드 호출에서와 마찬가지로 클라이언트 애플리케이션에서 컴파일 타임 시 애플리케이션

이션 서버의 인터페이스 선언을 알고 있어야 합니다. 서버에서 인터페이스를 선언 및 등록하는 데 사용하는 것과 동일한 유닛을 클라이언트의 `uses` 절에 추가하여 클라이언트 애플리케이션에 이것을 추가하거나 인터페이스를 설명하는 WSDL 문서를 참조할 수 있습니다. 서버 인터페이스를 설명하는 WSDL 문서 `import`에 대한 내용은 31-8 페이지의 "WSDL 문서 import하기"를 참조하십시오.

참고 서버 인터페이스를 선언하는 유닛은 호출 레지스트리에도 등록해야 합니다. 호출 가능한 인터페이스를 등록하는 방법에 대한 자세한 내용은 31-3 페이지의 "호출 가능한 인터페이스 정의"를 참조하십시오.

일단 애플리케이션에서 인터페이스를 선언 및 등록하는 서버 유닛을 사용하거나 WSDL 문서를 `import`해서 해당 유닛을 생성했다면 원하는 인터페이스에 대한 `THttpRio`의 인스턴스를 생성합니다.

```
X := THttpRio.Create(nil);
```

그런 다음 연결 컴포넌트에서 사용하는 URL을 원격 인터페이스 객체에 할당합니다.

```
X.URL := SoapConnection1.URL;
```

그러면 `as` 연산자를 사용하여 `THttpRio`의 인스턴스를 애플리케이션 서버의 인터페이스에 타입 변환할 수 있습니다.

```
InterfaceVariable := X as IMyAppServer;
InterfaceVariable.SpecialMethod(x,y);
```

다중 데이터 모듈을 사용하는 애플리케이션 서버에 대한 연결

25-22 페이지의 "다중 원격 데이터 모듈 사용"에서 설명한 대로 애플리케이션 서버에서 메인 "부모" 원격 데이터 모듈과 여러 자식 데이터 모듈을 사용하는 경우 애플리케이션 서버의 모든 원격 데이터 모듈에 대한 개별 연결 컴포넌트가 필요합니다. 각 연결 컴포넌트는 단일 원격 데이터 모듈에 대한 연결을 나타냅니다.

클라이언트 애플리케이션에서 애플리케이션 서버의 각 원격 데이터 모듈에 대한 독립적인 연결을 구성할 수 있지만 모든 연결 컴포넌트가 공유할 수 있는 애플리케이션 서버에 단일 연결을 사용하는 것이 더 효율적입니다. 즉, 연결하는 단일 연결 컴포넌트를 애플리케이션 서버의 "메인" 원격 데이터 모듈에 추가하면 각 "자식" 원격 데이터 모듈에서 연결을 공유하는 추가 컴포넌트를 메인 원격 데이터 모듈에 추가합니다.

- 1 메인 원격 데이터 모듈에 대한 연결에서 25-24 페이지의 "애플리케이션 서버에 연결"에서 설명한 대로 연결 컴포넌트를 추가하고 설정합니다. 유일한 제한 사항은 CORBA 또는 SOAP 연결을 사용할 수 없다는 것입니다.
- 2 각각의 자식 원격 데이터 모듈은 `TSharedConnection` 컴포넌트를 사용합니다.
 - `ParentConnection` 속성을 1 단계에서 추가한 연결 컴포넌트로 설정합니다. `TSharedConnection` 컴포넌트는 이러한 메인 연결이 설정하는 연결을 공유합니다.
 - `ChildName` 속성을 원하는 자식 원격 데이터 모듈의 인터페이스를 노출하는 메인 원격 데이터 모듈의 인터페이스의 속성 이름으로 설정합니다.

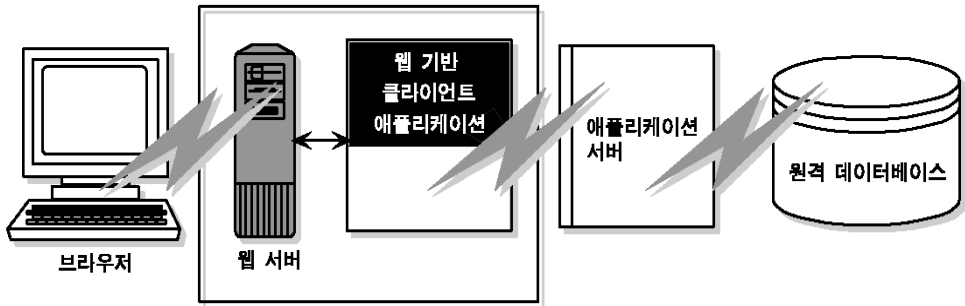
2 단계에서 놓은 `TSharedConnection` 컴포넌트를 클라이언트 데이터셋의 `RemoteServer` 속성 값으로서 할당하면 자식 원격 데이터 모듈에 완전히 독립적인 연

결을 사용하는 것처럼 작동합니다. 하지만 *TSharedConnection* 컴포넌트는 1 단계에서 가져다 놓은 컴포넌트에 의해 설정된 연결을 사용합니다.

웹 기반 클라이언트 애플리케이션 작성

다계층 데이터베이스 애플리케이션에 웹 기반 클라이언트를 생성하려면 클라이언트 계층을 애플리케이션 서버에 클라이언트로서 동시에 작동하고 동일한 컴퓨터에서 웹 서버와 함께 설치된 웹 서버 애플리케이션으로서 작동하는 특별한 웹 애플리케이션으로 대체해야 합니다. 이 아키텍처는 그림 25.1에서 보여 줍니다.

그림 25.1 웹 기반 다계층 데이터베이스 애플리케이션



웹 애플리케이션을 빌드하는 두 가지 방법은 다음과 같습니다.

- 다계층 데이터베이스 아키텍처를 Delphi의 ActiveX 지원과 조합하여 클라이언트 애플리케이션을 ActiveX control로 배포합니다. 이 방법을 사용하면 ActiveX를 지원하는 브라우저에서 클라이언트 애플리케이션을 in-process 서버로 실행할 수 있습니다.
- XML 데이터 패킷을 사용하여 InternetExpress 애플리케이션을 빌드할 수 있습니다. 이 방법을 사용하면 자바스크립트를 지원하는 브라우저에서 html 페이지를 통해 클라이언트 애플리케이션과 상호 작용할 수 있습니다.

이 두 방법은 큰 차이점이 있습니다. 따라서 다음 사항을 고려하여 사용할 방법을 결정하십시오.

- 두 방법은 각기 다른 기술에 의존합니다(ActiveX 및 자바스크립트와 XML). 최종 사용자가 사용할 시스템을 고려합니다. 첫 번째 방법에는 ActiveX를 지원하는 브라우저가 필요합니다(클라이언트를 Windows 플랫폼으로 제한). 두 번째 방법에는 자바스크립트와 Netscape 4 및 Internet Explorer 4에 의해 소개된 DHTML 기능이 필요합니다.
- ActiveX 컨트롤은 in-process 서버로 사용하려면 브라우저에 다운로드해야 합니다. 결과적으로 ActiveX를 이용한 방법을 사용하는 클라이언트는 HTML 기반 애플리케이션의 클라이언트보다 더 많은 메모리가 필요합니다.
- InternetExpress를 이용한 방법은 다른 HTML 페이지와 통합될 수 있습니다. ActiveX 클라이언트는 별도의 창에서 실행해야 합니다.

- InternetExpress를 이용한 방법은 표준 HTTP를 사용하므로 ActiveX 애플리케이션에서 직면하는 방화벽 문제를 피할 수 있습니다.
- ActiveX를 이용한 방법은 애플리케이션 프로그램 방식에 유연성을 제공합니다. 자바스크립트 라이브러리의 기능에 의해 제한되지 않습니다. ActiveX를 이용한 방법에 사용된 클라이언트 데이터셋이 InternetExpress 방법에 사용되는 XML 브로커보다 많은 기능(예: 필터, 범위, 집계, 선택적 매개변수, BLOB의 지연 페칭 또는 중첩 디테일 등)을 제공합니다.

주의 웹 클라이언트 애플리케이션이 다른 브라우저에서는 다르게 보이거나 다르게 작동할 수 있습니다. 최종 사용자가 사용할 것으로 예상되는 브라우저에서 애플리케이션을 테스트하십시오.

클라이언트 애플리케이션을 ActiveX 컨트롤로 배포

다계층 데이터베이스 아키텍처를 Delphi의 ActiveX 기능과 조합하여 클라이언트 애플리케이션을 ActiveX control로 배포할 수 있습니다.

클라이언트 애플리케이션을 ActiveX control로 배포할 때 다른 다계층 애플리케이션에서와 동일하게 애플리케이션 서버를 생성합니다. 유일한 제한 사항은 CORBA 런타임 소프트웨어를 설치했던 클라이언트 컴퓨터에 의존할 수 없기 때문에 DCOM, HTTP, SOAP 또는 소켓을 통신 프로토콜로 사용한다는 것입니다. 애플리케이션 서버 생성에 대한 자세한 내용은 25-12 페이지의 "애플리케이션 서버 생성"을 참조하십시오.

클라이언트 애플리케이션 생성 시 일반 폼 대신 Active Form을 기반으로 사용해야 합니다. 자세한 내용은 "클라이언트 애플리케이션에 Active Form 생성"을 참조하십시오.

클라이언트 애플리케이션을 빌드 및 배포하고 나면 다른 컴퓨터의 ActiveX 기반 웹 브라우저에서 액세스할 수 있습니다. 웹 브라우저에서 클라이언트 애플리케이션을 실행하려면 웹 서버가 클라이언트 애플리케이션이 있는 컴퓨터에서 실행 중이어야 합니다.

클라이언트 애플리케이션에서 DCOM을 사용하여 클라이언트 애플리케이션과 애플리케이션 서버 간에 통신하는 경우 웹 브라우저가 있는 컴퓨터에서 DCOM을 사용할 수 있어야 합니다. 웹 브라우저가 있는 컴퓨터가 Windows 95 컴퓨터이면 Microsoft에서 제공하는 DCOM95가 설치되어 있습니다.

클라이언트 애플리케이션에 Active Form 생성

- 1 클라이언트 애플리케이션은 ActiveX 컨트롤로 배포되기 때문에 클라이언트 애플리케이션과 동일한 시스템에서 실행하는 웹 서버가 있어야 합니다. Microsoft의 퍼스널 웹 서버와 같은 미리 만들어진 서버를 사용하거나 32장 "소켓 작업"에서 설명된 소켓 컴포넌트를 사용하는 사용자 고유의 서버를 작성할 수 있습니다.
- 2 클라이언트 프로젝트를 일반적인 Delphi 프로젝트로 시작하지 않고 File|New|Active Form을 선택하여 시작한다는 점을 제외하고 25-23 페이지의 "클라이언트 애플리케이션 생성"에서 설명한 단계에 따라 클라이언트 애플리케이션을 생성합니다.
- 3 클라이언트 애플리케이션에서 데이터 모듈을 사용하는 경우 호출을 추가하여 활성 폼 초기화의 데이터 모듈을 명시적으로 생성합니다.

- 4 클라이언트 애플리케이션이 작성이 완료되면 프로젝트를 컴파일하고 Project | Web Deployment Options를 선택합니다. Web Deployment Options 대화 상자에서 다음을 수행해야 합니다.
 - 1 Project 페이지에서 Target 디렉토리, 대상 디렉토리의 URL, HTML 디렉토리를 지정합니다. 일반적으로 Target 디렉토리와 HTML 디렉토리는 웹 서버의 프로젝트 디렉토리와 동일합니다. 대상 URL은 보통 Windows Network\DNS 설정에 지정되는 서버 컴퓨터의 이름입니다.
 - 2 Additional Files 페이지에서 클라이언트 애플리케이션에 midas.dll을 포함시킵니다.
- 5 마지막으로 Project|WebDeploy를 선택하여 클라이언트 애플리케이션을 활성 폼으로 배포합니다.

활성 폼을 실행할 수 있는 웹 브라우저는 클라이언트 애플리케이션 배포 시 생성된 .HTM 파일을 지정하여 클라이언트 애플리케이션을 실행할 수 있습니다. 이 .HTM 파일은 클라이언트 애플리케이션 프로젝트와 동일한 이름을 가지며 Target 디렉토리로 지정된 디렉토리에 나타납니다.

InternetExpress를 사용하여 웹 애플리케이션 구축

클라이언트 애플리케이션은 애플리케이션 서버에서 OleVariants 대신 XML로 코드화된 데이터 패킷을 제공하도록 요청할 수 있습니다. XML 코드화된 데이터 패킷, 데이터 베이스 함수의 특별한 자바스크립트 라이브러리 및 Delphi의 웹 서버 애플리케이션 지원을 결합하여 자바스크립트를 지원하는 웹 브라우저를 통해 액세스할 수 있는 웹 클라이언트 애플리케이션을 생성할 수 있습니다. 이러한 애플리케이션들을 통해 Delphi의 InternetExpress 지원이 가능해 집니다.

InternetExpress 애플리케이션을 구축하기 전에 Delphi의 웹 서버 애플리케이션 구조를 이해해야 합니다. 이 내용은 27장 "인터넷 애플리케이션 생성"에서 다룹니다.

InternetExpress 애플리케이션은 애플리케이션 서버의 클라이언트로 작동하도록 기본 웹 서버 애플리케이션 아키텍처를 확장합니다. InternetExpress 애플리케이션은 HTML, XML 및 자바스크립트가 혼합되어 있는 HTML 페이지를 생성합니다. HTML은 최종 사용자의 브라우저에 나타나는 페이지의 레이아웃과 외관을 결정합니다. XML은 데이터베이스 정보를 나타내는 데이터 패킷과 델타 패킷을 인코딩합니다. 자바스크립트를 통해 HTML controls에서 클라이언트 컴퓨터의 XML 데이터 패킷의 데이터를 해석하고 처리할 수 있습니다.

InternetExpress 애플리케이션이 DCOM을 사용하여 애플리케이션 서버에 연결하는 경우에는 애플리케이션 서버에서 클라이언트에게 액세스 및 실행 권한을 부여하도록 하는 추가적인 단계를 거쳐야 합니다. 자세한 내용은 25-37 페이지의 "애플리케이션 서버의 액세스 및 실행 권한 부여"를 참조하십시오.

- 팁** InternetExpress 애플리케이션을 생성하면 애플리케이션 서버가 없는 경우에도 웹 브라우저에 데이터를 제공할 수 있습니다. 프로바이더와 그 데이터셋을 웹 모듈에 추가하면 됩니다.

InternetExpress 애플리케이션 구축

다음 단계는 InternetExpress를 사용하여 웹 애플리케이션을 구축하는 방법을 설명합니다. 결과적으로 애플리케이션에서 사용자가 자바스크립트 기반 웹 브라우저를 통해 애플리케이션 서버의 데이터와 상호 작용할 수 있는 HTML 페이지를 생성할 수 있습니다. 또한 InternetExpress 페이지 프로듀서 (*TInetXPageProducer*)를 사용하여 Site Express 아키텍처를 사용하는 InternetExpress 애플리케이션을 빌드할 수 있습니다.

- 1 File|New를 선택하여 New Items 대화 상자를 표시한 다음, New 페이지에서 Web Server application을 선택합니다. 이 프로세스는 28-1 페이지의 "Web Broker로 웹 서버 애플리케이션 생성"에서 다룹니다.
- 2 컴포넌트 팔레트의 DataSnap 페이지에서 새 웹 서버 애플리케이션 생성 시 나타나는 Web Module에 연결 컴포넌트를 추가합니다. 추가할 연결 컴포넌트의 타입은 사용할 통신 프로토콜에 따라 다릅니다. 자세한 내용은 25-9 페이지의 "연결 프로토콜 선택"을 참조하십시오.
- 3 연결 컴포넌트에 대한 속성을 설정하여 연결을 설정할 애플리케이션 서버를 지정합니다. 연결 컴포넌트 설정에 대한 자세한 내용은 25-24 페이지의 "애플리케이션 서버에 연결"을 참조하십시오.
- 4 클라이언트 데이터셋 대신 컴포넌트 팔레트의 InternetExpress 페이지에 있는 XML broker를 웹 모듈에 추가합니다. *TClientDataSet*과 마찬가지로 *TXMLBroker*는 애플리케이션 서버에 있는 프로바이더의 데이터를 나타내고 그 *IAppServer* 인터페이스를 통해 애플리케이션 서버와 상호 작용합니다. 하지만 클라이언트 데이터셋과 달리 XML 브로커는 데이터 패킷을 OleVariants 대신 XML로 요청하고 데이터 컨트롤 대신 InternetExpress 컴포넌트와 상호 작용합니다.
- 5 XML 브로커의 *RemoteServer* 속성을 2 단계에서 추가한 연결 컴포넌트에 대한 포인트로 설정합니다. *ProviderName* 속성을 설정하여 데이터를 제공하고 업데이트를 적용하는 애플리케이션 서버에서 프로바이더를 나타냅니다. XML 브로커 설정에 대한 자세한 내용은 25-37 페이지의 "XML 브로커 사용"을 참조하십시오.
- 6 InternetExpress 페이지 프로듀서 (*TInetXPageProducer*)를 사용자의 브라우저에 나타나는 각각의 개별 페이지의 웹 모듈에 추가합니다. 각 페이지 프로듀서에서 *IncludePathURL* 속성을 설정하여 그 생성된 HTML 컨트롤을 데이터 관리 기능으로 향상시키는 자바스크립트 라이브러리의 위치를 나타냅니다.
- 7 웹 페이지를 마우스 오른쪽 버튼으로 클릭한 다음 Action Editor를 선택하여 액션 에디터를 표시합니다. 브라우저에서 처리하려는 모든 메시지의 동작 항목을 추가합니다. *Producer* 속성을 설정하거나 *OnAction* 이벤트 핸들러에서 코드를 작성하여 6 단계에서 추가했던 페이지 프로듀서를 이러한 동작과 연결합니다. 액션 에디터를 통한 액션 항목 추가에 대한 자세한 내용은 28-5 페이지의 "디스패처에 액션 추가"를 참조하십시오.

- 8 각 웹 페이지를 더블 클릭하여 Web Page 에디터를 표시합니다. (*WebPageItems* 속성 옆에 있는 Object Inspector에서 생략 부호를 클릭하여 이 에디터를 표시할 수도 있습니다.) 이 에디터에서 Web Items를 추가하여 사용자의 브라우저에 나타날 페이지를 디자인할 수 있습니다. InternetExpress 애플리케이션의 웹 페이지 디자인에 대한 자세한 내용은 25-39 페이지의 "InternetExpress 페이지 프로듀서로 웹 페이지 생성"을 참조하십시오.
- 9 웹 애플리케이션을 구축합니다. 웹 서버에서 이 애플리케이션을 설치하면 브라우저에서 애플리케이션 이름을 URL의 스크립트명 일부로 지정하고 웹 페이지 컴포넌트 이름을 경로명 일부로 지정하여 호출할 수 있습니다.

자바스크립트 라이브러리 사용

InternetExpress 컴포넌트에 의해 생성된 HTML 페이지와 이에 포함된 웹 항목은 Delphi와 함께 제공되는 여러 자바스크립트 라이브러리를 이용합니다.

표 25.3 자바스크립트 라이브러리

라이브러리	설명
xmldom.js	이 라이브러리는 자바스크립트로 쓰여진 DOM 호환 XML 파서입니다. 이를 통해 XML을 지원하지 않는 파서가 데이터 패킷을 사용할 수 있습니다. 이 라이브러리는 IE5 이상에서 지원되는 XML Islands에 대한 지원을 포함하지 않습니다.
xmldb.js	이 라이브러리는 XML 데이터 패킷과 XML 델타 패킷을 관리하는 데이터 액세스 클래스를 정의합니다.
xmldisp.js	이 라이브러리는 xmldb의 데이터 액세스 클래스와 HTML 페이지의 HTML 컨트롤을 연결하는 클래스를 정의합니다.
xmlerrdisp.js	이 라이브러리는 업데이트 오류 조정 시 사용할 수 있는 클래스를 정의합니다. 이러한 클래스는 기본 제공 InternetExpress 컴포넌트에 의해 사용되지 않지만 Reconcile 프로듀서 작성 시 유용합니다.
xmlshow.js	이 라이브러리는 포맷된 XML 데이터 패킷과 XML 델타 패킷을 표시하는 기능을 포함합니다. 이 라이브러리는 InternetExpress 컴포넌트에 의해 사용되지 않지만 디버깅 시 유용합니다.

이러한 라이브러리는 Source/Webmidas 디렉토리에 있습니다. 이러한 라이브러리를 설치한 후에는 모든 InternetExpress 페이지 프로듀서의 *IncludePathURL* 속성을 설정하여 그 위치를 표시해야 합니다.

웹 항목을 사용하여 웹 페이지를 생성하는 대신 이러한 라이브러리에서 제공된 자바스크립트 클래스를 사용하여 사용자 고유의 HTML 페이지를 작성할 수 있습니다. 하지만 이러한 클래스는 생성된 웹 페이지의 크기를 최소화하기 위해 최소의 오류 확인 기능을 포함하므로 코드가 모두 올바른지 확인해야 합니다.

자바스크립트 라이브러리의 클래스는 최신 표준 도구로서 정기적으로 업데이트됩니다. 웹 항목에 의존하지 않고 직접 이러한 클래스를 사용하여 자바스크립트 코드를 생성하려면 community.borland.com의 CodeCentral에서 최신 버전과 사용 방법에 대한 정보를 얻을 수 있습니다.

애플리케이션 서버의 액세스 및 실행 권한 부여

InternetExpress 애플리케이션의 요청은 IUSR_computername 이름의 게스트 계정에서 시작하여 애플리케이션 서버에 나타나며 여기서 computername은 웹 애플리케이션을 실행하는 시스템의 이름입니다. 기본적으로 이 계정에는 애플리케이션 서버에 대한 액세스 또는 실행 권한이 없습니다. 이러한 권한을 부여하지 않고 웹 애플리케이션을 사용하려고 시도할 경우 웹 브라우저에서 요청된 페이지 로드 중 EOLE_ACCESS_ERROR 라는 메시지와 함께 시간 초과되었다고 표시됩니다.

참고 애플리케이션 서버는 이 게스트 계정 하에서 실행하므로 다른 계정에 의해 종료될 수 없습니다.

웹 애플리케이션 액세스 및 실행 권한을 부여하려면 애플리케이션 서버를 실행하는 컴퓨터의 System32 디렉토리에 있는 DCOMCnfg.exe를 실행합니다. 다음 단계는 애플리케이션 서버를 구성하는 방법에 대해 설명합니다.

- 1 DCOMCnfg를 실행할 때 Applications 페이지의 애플리케이션 목록에서 해당 애플리케이션 서버를 선택합니다.
- 2 Properties 버튼을 클릭합니다. 대화 상자 내용이 변경되면 Security 페이지를 선택합니다.
- 3 Use Custom Access Permissions를 선택하고 Edit 버튼을 클릭합니다. IUSR_computername 이름을 액세스 권한이 있는 계정 목록에 추가합니다. 여기서 computername은 웹 애플리케이션을 실행하는 컴퓨터의 이름입니다.
- 4 Use Custom Launch Permissions를 선택하고 Edit 버튼을 클릭합니다. IUSR_computername도 이 목록에 추가합니다.
- 5 Apply 버튼을 클릭합니다.

XML 브로커 사용

XML 브로커에는 다음과 같은 두 가지 주요 기능이 있습니다.

- XML 브로커는 애플리케이션 서버에서 XML 데이터 패킷을 페치하고 InternetExpress 애플리케이션에 HTML을 생성하는 웹 항목에서 이 XML 데이터 패킷을 사용할 수 있도록 합니다.
- 또한 브라우저에서 XML 델타 패킷의 형태로 업데이트를 받아서 애플리케이션 서버에 이 업데이트를 적용합니다.

XML 데이터 패킷 페치

XML 브로커가 HTML 페이지를 생성하는 컴포넌트에 XML 데이터 패킷을 제공하기 전에 애플리케이션 서버에서 XML 데이터 패킷을 페치해야 합니다. 이 작업을 수행하기 위해 XML 브로커는 연결 컴포넌트를 통해 얻은 애플리케이션 서버의 *IAppServer* 인터페이스를 사용합니다. 다음 속성들을 설정해야 XML 프로듀서가 애플리케이션 서버의 *IAppServer* 인터페이스를 사용할 수 있습니다.

- *RemoteServer* 속성을 애플리케이션 서버에 연결하는 연결 컴포넌트로 설정하고 그 *IAppServer* 인터페이스를 얻습니다. 디자인 타임 시 Object Inspector의 드롭다운 목록에서 이 값을 선택할 수 있습니다.
- *ProviderName* 속성을 XML 데이터 패킷이 필요한 데이터셋을 나타내는 애플리케이션 서버의 프로바이더 컴포넌트 이름으로 설정합니다. 이 프로바이더는 모두 XML 데이터 패킷을 제공하고 XML 델타 패킷의 업데이트를 적용합니다. 디자인 타임 시 *RemoteServer* 속성이 설정되어 있고 연결 컴포넌트에 활성 연결이 있으면 Object Inspector는 사용 가능한 프로바이더 목록을 표시합니다 (DCOM 연결을 사용하는 경우 애플리케이션 서버를 클라이언트 컴퓨터에도 등록해야 합니다.).

두 속성을 통해 데이터 패킷에 포함시킬 내용을 표시할 수 있습니다.

- *MaxRecords* 속성을 설정하여 데이터 패킷에 추가되는 레코드 수를 제한할 수 있습니다. InternetExpress 애플리케이션에서 전체 데이터 패킷을 클라이언트 웹 브라우저에 보내기 때문에 이 속성은 대형 데이터셋에서 특히 중요합니다. 데이터 패킷 크기가 너무 크면 다운로드 시간이 너무 길어질 수 있습니다.
- 애플리케이션 서버의 프로바이더가 쿼리 또는 내장 프로시저를 나타내는 경우 XML 데이터 패킷을 얻기 전에 매개변수 값을 제공할 수 있습니다. *Params* 속성을 사용하여 이러한 매개변수 값을 제공할 수 있습니다.

InternetExpress 애플리케이션에 HTML과 자바스크립트를 생성하는 컴포넌트는 그 *XMLBroker* 속성을 설정하면 XML 브로커의 XML 데이터 패킷을 자동으로 사용합니다. 코드에서 직접 XML 데이터 패킷을 얻으려면 *RequestRecords* 메소드를 사용합니다.

참고 XML 브로커는 데이터 패킷을 다른 컴포넌트에 제공하거나 *RequestRecords*를 호출할 때 *OnRequestRecords* 이벤트를 받습니다. 이 이벤트를 사용하여 애플리케이션 서버의 데이터 패킷 대신 사용자 고유의 XML 문자열을 제공할 수 있습니다. 예를 들어, *GetXMLRecords*를 사용하여 애플리케이션 서버에서 XML 데이터 패킷을 패치하여 편집한 다음 나타나는 웹 페이지에 제공할 수 있습니다.

XML 델타 패킷(delta packets)의 업데이트 적용

XML 브로커를 웹 모듈 또는 *TWebDispatcher*를 포함하는 데이터 모듈에 추가하면 XML 브로커는 웹 디스패처와 함께 자동 디스패칭 객체로 등록됩니다. 즉, 다른 컴포넌트와 달리 웹 브라우저의 업데이트 메시지에 응답하기 위해 XML 브로커에 대한 동작 항목을 생성할 필요가 없습니다. 이러한 메시지는 애플리케이션 서버에 적용되어야 하는 XML 델타 패킷을 포함합니다. 일반적으로 이러한 메시지는 웹 클라이언트 애플리케이션에서 생성한 HTML 페이지 중 하나에 만들어진 버튼으로부터 생겨납니다.

디스패처가 XML 브로커의 메시지를 인식할 수 있도록 *WebDispatch* 속성을 사용하여 메시지를 나타내야 합니다. *PathInfo* 속성을 URL의 경로 부분으로 설정하여 어떤 메시지가 보내질지 알려 줍니다. *MethodType*을 해당 URL에 보내지는 업데이트 메시지의 메소드 헤더 값으로 설정합니다(일반적으로 *mtPost*). 지정된 경로로 모든 메시지에 응답하려면 *MethodType*을 *mtAny*로 설정합니다. XML 브로커에서 업데이트 메시지에 직접 응답하지 않도록 하려면(예를 들어, 동작 항목을 사용하여 업데이트 메시지를 명시적으로 처리할 경우) *Enabled* 속성을 *False*로 설정합니다. 웹 디스패처에서 웹 브라

우저의 메시지를 처리하는 컴포넌트를 결정하는 방법에 대한 자세한 내용은 28-5 페이지의 "요청 메시지 디스패칭"을 참조하십시오.

디스패처가 업데이트 메시지를 XML 브로커에 전달하면 XML 브로커는 업데이트를 애플리케이션 서버에 전달하고 업데이트 오류가 있으면 모든 업데이트 오류를 나타내는 XML 델타 패킷을 받습니다. 마지막으로 브라우저에 응답 메시지가 보내지면 여기서 XML 델타 패킷을 생성한 동일한 페이지로 브라우저를 리디렉션하거나 새로운 일부 콘텐츠를 보냅니다.

많은 이벤트를 통해 이 업데이트 프로세스의 모든 단계에 사용자 지정 처리를 삽입할 수 있습니다.

- 1 디스패처가 먼저 업데이트 메시지를 XML 브로커에 전달하면 XML 브로커는 *BeforeDispatch* 이벤트를 받으며 여기서 요청을 미리 처리하거나 완전히 처리할 수 있습니다. 이 이벤트를 통해 XML 브로커에서 업데이트 메시지 이외의 메시지를 처리할 수 있습니다.
- 2 *BeforeDispatch* 이벤트 핸들러가 메시지를 처리하지 않는 경우 XML 브로커는 *OnRequestUpdate* 이벤트를 받으며 여기서 기본 처리를 사용하지 않고 업데이트를 적용할 수 있습니다.
- 3 *OnRequestUpdate* 이벤트 핸들러가 요청을 처리하지 않는 경우 XML 브로커는 업데이트를 적용하고 업데이트 오류를 포함하는 델타 패킷을 받습니다.
- 4 업데이트 오류가 없으면 XML 브로커는 *OnGetResponse* 이벤트를 받으며 여기서 업데이트가 성공적으로 적용되었음을 나타내는 응답 메시지를 생성하거나 새로 고친 데이터를 브라우저에 보낼 수 있습니다. *OnGetResponse* 이벤트 핸들러가 응답을 완료하지 않으면 (*Handled* 매개변수를 *True*로 설정하지 않으면) XML 브로커는 델타 패킷을 생성했던 문서로 브라우저를 리디렉션하는 응답을 보냅니다.
- 5 업데이트 오류가 있으면 XML 브로커는 대신 *OnErrorResponse* 이벤트를 받습니다. 이 이벤트를 사용하여 업데이트 오류를 해결하거나 최종 사용자에게 업데이트 오류 내용을 알리는 웹 페이지를 생성할 수 있습니다. *OnErrorResponse* 이벤트 핸들러가 응답을 완료하지 않으면 (*Handled* 매개변수를 *True*로 설정하지 않으면) XML 브로커는 *ReconcileProducer*라는 특별한 콘텐츠 프로듀서를 호출하여 응답 메시지의 콘텐츠를 생성합니다.
- 6 마지막으로 XML 브로커는 *AfterDispatch* 이벤트를 받으며 여기서 응답을 웹 브라우저에 보내기 전에 최종 동작을 수행할 수 있습니다.

InternetExpress 페이지 프로듀서로 웹 페이지 생성

각 InternetExpress 페이지 프로듀서는 애플리케이션 클라이언트의 브라우저에 나타나는 HTML 문서를 생성합니다. 애플리케이션이 별도의 여러 웹 문서를 포함하면 각각의 문서에 대해 별도의 페이지 프로듀서를 사용합니다.

InternetExpress 페이지 프로듀서 (*TInetXPPageProducer*)는 특별한 페이지 프로듀서 컴포넌트입니다. 다른 페이지 프로듀서처럼 InternetExpress 페이지 프로듀서를 작업 항목의 *Producer* 속성으로 할당하거나 *OnAction* 이벤트 핸들러에서 명시적으로 호출할 수 있습니다. 작업 항목과 함께 콘텐츠 프로듀서를 사용하는 것에 대한 자세한 내용은

28-8 페이지의 "액션 항목으로 요청 메시지에 응답"을 참조하십시오. 페이지 프로듀서에 대한 자세한 내용은 28-14 페이지의 "페이지 프로듀서 컴포넌트 사용"을 참조하십시오.

대부분의 페이지 프로듀서와 달리 InternetExpress 페이지 프로듀서는 기본 템플릿을 *HTMLDoc* 속성의 값으로 갖습니다. 이 템플릿은 다른 컴포넌트에 의해 만들어진 콘텐츠를 포함하여 InternetExpress 페이지 프로듀서에서 HTML 문서를 포함한 자바스크립트 및 XML과 어셈블하는 데 사용하는 HTML 투명 태그 집합을 포함합니다. 이 템플릿에서 모든 HTML 투명 태그를 번역하고 해당 문서를 모으려면 페이지의 포함된 자바스크립트에 사용된 자바스크립트 라이브러리의 위치를 나타내야 합니다. *IncludePathURL* 속성을 설정하면 위치가 지정됩니다.

웹 페이지 에디터를 사용하여 웹 페이지의 일부를 생성하는 컴포넌트를 지정할 수 있습니다. 웹 페이지 컴포넌트를 더블 클릭하거나 Object Inspector에서 *WebPageItems* 속성 옆에 있는 생략 버튼을 클릭하여 웹 페이지 에디터를 표시합니다.

웹 페이지 에디터에 추가하는 컴포넌트는 InternetExpress 페이지 프로듀서의 기본 템플릿에서 HTML 투명 태그 중 하나를 대체하는 HTML을 생성합니다. 이러한 컴포넌트는 *WebPageItems* 속성의 값이 됩니다. 원하는 순서대로 컴포넌트를 추가한 후에 템플릿을 사용자 지정하여 사용자 고유의 HTML을 추가하거나 기본 태그를 변경할 수 있습니다.

웹 페이지 에디터 사용

웹 페이지 에디터를 통해 웹 항목을 InternetExpress 페이지 프로듀서에 추가하고 결과 HTML 페이지를 볼 수 있습니다. InternetExpress 페이지 프로듀서 컴포넌트를 더블 클릭하여 웹 페이지 에디터를 표시합니다.

참고 웹 페이지 에디터를 사용하려면 Internet Explorer 4 이상이 설치되어 있어야 합니다.

웹 페이지 에디터의 상단에 HTML 문서를 생성하는 웹 항목이 표시됩니다. 이러한 웹 항목은 중첩되며 각 웹 항목 타입은 해당 하위 항목에 의해 생성된 HTML을 어셈블합니다. 다른 타입의 항목은 다른 하위 항목을 포함할 수 있습니다. 왼쪽의 트리 뷰에는 중첩된 모양을 나타내는 모든 웹 항목이 표시됩니다. 오른쪽에는 현재 선택되어 있는 항목에 포함된 웹 항목이 표시됩니다. 웹 페이지 에디터의 상단에 있는 컴포넌트를 선택하면 Object Inspector에서 속성을 설정할 수 있습니다.

New Item 버튼을 클릭하여 현재 선택되어 있는 항목에 하위 항목을 추가합니다. Add Web Component 대화 상자는 현재 선택되어 있는 항목에 추가할 수 있는 항목들만 나열합니다.

InternetExpress 페이지 프로듀서는 두 가지 타입의 항목 중 하나를 포함할 수 있으며 이들 각각은 HTML 폼을 생성합니다.

- *TDataForm*은 데이터를 표시하기 위한 HTML 폼과 데이터를 처리하거나 업데이트를 보내는 제어를 생성합니다.

*TDataForm*에 추가하는 항목은 멀티레코드 그리드(*TDataGrid*) 또는 단일 레코드의 단일 필드를 각각 나타내는 제어 집합(*TFieldGroup*)으로 데이터를 표시합니다. 또한 데이터를 검색하거나 업데이트를 게시하는 버튼(*TDataNavigator*) 또는 업데이트를 웹 클라이언트에 적용하는 버튼(*TApplyUpdatesButton*) 집합을 추가할 수 있습니다.

이러한 각 항목은 개별 필드 또는 버튼을 나타내는 하위 항목을 포함합니다. 마지막으로 대부분의 웹 항목처럼 레이아웃 그리드(*TLayoutGroup*)를 추가하여 포함하는 항목의 레이아웃을 사용자 지정할 수 있습니다.

- *TQueryForm*은 애플리케이션 정의 값을 표시하거나 읽기 위한 HTML 폼을 생성합니다. 예를 들어, 매개변수 값을 표시하거나 보내기 위해 이 폼을 사용할 수 있습니다.

*TQueryForm*에 추가하는 항목은 애플리케이션 정의 값(*TQueryFieldGroup*) 또는 그러한 값을 제출하거나 재설정하는 버튼 집합(*TQueryButtons*)을 표시합니다. 이러한 각 항목은 개별 값 또는 버튼을 나타내는 하위 항목을 포함합니다. 데이터 폼에서처럼 레이아웃 그리드를 쿼리 폼에 추가할 수도 있습니다.

웹 페이지 에디터의 하단에 생성된 HTML 코드가 표시되므로 브라우저(IE4)에 나타날 모양을 확인할 수 있습니다.

웹 항목 속성 설정

웹 페이지 에디터를 사용하여 추가하는 웹 항목은 HTML을 생성하는 특수한 컴포넌트입니다. 각 웹 항목 클래스는 특정 컨트롤 또는 최종 HTML 문서의 섹션을 만들 수 있도록 디자인되었지만 일반 속성 집합은 최종 HTML의 외관에 영향을 미칩니다.

웹 항목이 XML 데이터 패킷의 정보를 나타낼 때(예를 들어, 필드 집합 또는 데이터를 처리하는 매개변수 표시 컨트롤이나 버튼을 생성할 때) *XMLBroker* 속성은 데이터 패킷을 관리하는 XML 브로커와 웹 항목을 연결합니다. *XMLDataSetField* 속성을 사용하여 데이터 패킷의 데이터셋 필드에 포함된 데이터셋을 추가로 지정할 수 있습니다. 웹 항목이 특정 필드 또는 매개변수 값을 나타내면 웹 항목은 *FieldName* 또는 *ParamName* 속성을 갖습니다.

스타일 속성을 웹 항목에 적용하여 생성되는 모든 HTML의 전체 외관에 영향을 줄 수 있습니다. 스타일 및 스타일 시트는 HTML 4 표준의 일부입니다. 이를 통해 HTML 문서에서 해당 범위 내에서 태그 및 모든 것에 적용되는 표시 속성 집합을 정의할 수 있습니다. 웹 항목은 이들을 사용하기 위한 유연성 있는 여러 방법을 제공합니다.

- 스타일을 사용하는 가장 간단한 방법은 웹 항목에서 직접 스타일 속성을 정의하는 것입니다. 이 작업을 수행하려면 *Style* 속성을 사용합니다. *Style*의 값은 다음과 같은 표준 HTML 스타일 정의 중에 속성 정의 부분입니다.

```
color: red.
```

- 스타일 정의 집합을 정의하는 스타일 시트를 정의할 수도 있습니다. 각 정의는 스타일 선택자(스타일이 항상 적용하는 태그의 이름 또는 사용자 정의 스타일 이름)와 속성 정의를 중괄호 안에 포함시킵니다.

```
H2 B {color: red}
.MyStyle {font-family: arial; font-weight: bold; font-size: 18px }
```

전체 정의 집합은 InternetExpress 페이지 프로듀서에 의해 *Styles* 속성으로 유지됩니다. 각 웹 항목은 *StyleRule* 속성을 설정하여 사용자 정의 이름으로 스타일을 참조할 수 있습니다.

- 스타일 시트를 다른 애플리케이션과 공유하는 경우 스타일 정의를 *Styles* 속성 대신 InternetExpress 페이지 프로듀서의 *StylesFile* 속성으로 제공할 수 있습니다. 개별 웹 항목은 *StyleRule* 속성을 사용하여 스타일을 계속 참조합니다.

웹 항목의 다른 일반적인 속성은 *Custom* 속성입니다. *Custom*은 생성된 HTML 태그에 추가하는 옵션 집합을 포함합니다. HTML은 각 타입의 태그에 대한 옵션 집합을 정의합니다. 대부분 웹 항목의 *Custom* 속성에 대한 VCL 참조는 가능한 옵션의 예제를 제공합니다. 가능한 옵션에 대한 자세한 내용은 HTML 참조를 사용합니다.

InternetExpress 페이지 프로듀서 템플릿 사용자 지정

InternetExpress 페이지 프로듀서의 템플릿은 애플리케이션이 동적으로 번역하는 추가 포함된 태그가 있는 HTML 문서입니다. 처음에 페이지 프로듀서는 기본 템플릿을 *HTMLDoc* 속성의 값으로 생성합니다. 이 기본 템플릿은 다음과 같은 형태를 갖습니다.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

기본 템플릿의 HTML 투명 태그는 다음과 같이 번역됩니다.

<#INCLUDES>는 자바스크립트 라이브러리를 포함하는 문장을 생성합니다. 이 문장은 다음과 같은 형태를 갖습니다.

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlDOM.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlDB.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlBind.js"> </SCRIPT>
```

<#STYLES>는 InternetExpress 페이지 프로듀서의 *Styles* 또는 *StylesFile* 속성에 나열된 정의에서 스타일 시트를 정의하는 문장을 생성합니다.

<#WARNINGS>는 런타임 시 아무것도 생성하지 않습니다. 디자인 타임 시 HTML 문서 생성 중 발견된 문제에 대한 경고 메시지를 추가합니다. 웹 페이지 에디터에서 이러한 메시지를 볼 수 있습니다.

<#FORMS>는 웹 페이지 에디터에 추가하는 컴포넌트에 의해 만들어진 HTML을 생성합니다. 각 컴포넌트의 HTML이 *WebPageItems*에 나타난 순서대로 생성됩니다.

<#SCRIPT>는 웹 페이지 에디터에 추가된 컴포넌트에 의해 생성된 HTML에서 사용하는 자바스크립트 선언 블록을 생성합니다.

*HTMLDoc*의 값을 변경하거나 *HTMLFile* 속성을 설정하여 기본 템플릿을 대체할 수 있습니다. 사용자 지정된 HTML 템플릿은 기본 템플릿을 구성하는 HTML 투명 태그를 포함할 수 있습니다. InternetExpress 페이지 프로듀서는 *Content* 메소드 호출 시 이러한 태그를 자동으로 번역합니다. 또한 InternetExpress 페이지 프로듀서는 다음의 세 가지 추가 태그를 자동으로 번역합니다.

<#BODYELEMENTS>는 기본 템플릿의 5 태그에 의해 HTML로 대체됩니다. 기본 템플릿을 사용하면서 에디터를 사용하여 추가 요소를 추가하려는 경우 HTML 에디터에서 템플릿을 생성할 때 유용합니다.

<#COMPONENT Name=WebComponentName>은 *WebComponentName*이라는 컴포넌트가 생성하는 HTML로 대체됩니다. 이 컴포넌트는 웹 페이지 에디터에 추가된 컴포넌트 중 하나

이거나 *IWebContent* 인터페이스를 지원하고 InternetExpress 페이지 프로듀서와 동일한 소유자를 갖는 컴포넌트일 수 있습니다.

`<#DATAPACKET XMLBroker=BrokerName>` 은 *BrokerName*에 의해 지정된 XML 브로커에서 얻은 XML 데이터 패킷으로 대체됩니다. 웹 페이지 에디터에 InternetExpress 페이지 프로듀서가 생성하는 HTML이 있으면 실제 XML 데이터 패킷 대신 이 태그가 표시됩니다.

또한 사용자 지정된 템플릿은 정의되는 다른 HTML 투명 태그를 포함할 수 있습니다. InternetExpress 페이지 프로듀서에 자동으로 번역되는 위의 일곱 가지 유형 중 하나가 아닌 태그가 나타나면 *OnHTMLTag* 이벤트가 생성되며 여기서 사용자 고유의 번역을 수행하는 코드를 쓸 수 있습니다. 일반적인 HTML 템플릿에 대한 자세한 내용은 28-14 페이지의 "HTML 템플릿"을 참조하십시오.

팁 웹 페이지 에디터에 나타나는 컴포넌트는 정적 코드를 생성합니다. 즉, 애플리케이션 서버에서 데이터 패킷에 나타나는 메타데이터를 변경하지 않는 한 HTML은 생성 시기와 상관 없이 항상 동일합니다. 웹 페이지 에디터에서 생성된 HTML을 복사한 후 이 HTML을 템플릿으로 사용하면 모든 요청 메시지에 응답하여 런타임 시 동적으로 이 코드를 생성하는 오버헤드를 피할 수 있습니다. 웹 페이지 에디터에 실제 XML 대신 `<#DATAPACKET>` 태그가 표시되므로 이 태그를 템플릿으로 사용하면 애플리케이션에서 애플리케이션 서버의 데이터 패킷을 동적으로 폐치할 수 있습니다.

26

데이터베이스 애플리케이션에서 XML 사용

Delphi를 사용하면 데이터베이스 서버에 대한 연결이 지원될 뿐만 아니라 데이터베이스 서버에 있는 것처럼 XML 문서로 작업할 수 있습니다. XML(Extensible Markup Language)은 구조화된 데이터를 나타내기 위한 마크업(markup) 랭귀지입니다. XML 문서는 웹 애플리케이션, business-to-business 통신 등에서 사용되는 데이터에 전송 가능한 표준 형식을 제공합니다. 직접적인 XML 문서 작업에 대한 Delphi의 지원에 대한 자세한 내용은 30장 "XML 문서 작업"을 참조하십시오.

데이터베이스 애플리케이션에서의 XML 문서 작업에 대한 Delphi의 지원은 데이터 패킷(클라이언트 데이터셋의 *Data* 속성)을 XML 문서로 변환하고 XML 문서를 데이터로 변환할 수 있는 컴포넌트 집합을 기반으로 합니다. 이러한 컴포넌트를 사용하려면 먼저 XML 문서와 데이터 패킷 사이의 변환을 정의해야 합니다. 일단 변환을 정의하면 특수 컴포넌트를 사용하여 다음 작업을 수행할 수 있습니다.

- XML 문서를 데이터 패킷으로 변환합니다.
- XML 문서로부터 데이터를 제공하고 업데이트를 해결합니다.
- 프로바이더의 클라이언트로 XML 문서를 사용합니다.

변환 정의

데이터 패킷과 XML 문서 사이에서 변환할 수 있게 하려면 데이터 패킷의 메타데이터와 해당 XML 문서 노드 사이의 관계를 정의해야 합니다. 이 관계에 대한 설명은 변환이라는 특수한 XML 문서에 저장됩니다.

각 변환 파일에는 XML 스키마의 노드와 데이터 패킷의 필드 사이의 매핑 및 변환의 결과에 대한 구조를 나타내는 골조 XML 문서라는 두 가지 항목이 포함됩니다. 변환은 XML 스키마나 문서에서 데이터 패킷으로 또는 데이터 패킷의 메타데이터에서 XML 스키마로 진행되는 단방향 매핑입니다. 변환 파일은 XML에서 데이터 패킷으로 매핑하는 경우와 데이터 패킷에서 XML로 매핑하는 경우처럼 쌍으로 만들어지기도 합니다.

매핑을 하기 위한 변환 파일을 만들려면 bin 디렉토리에 있는 XMLMapper 유틸리티를 사용하십시오.

XML 노드와 데이터 패킷 필드 사이의 매핑

XML은 구조화된 데이터를 저장하거나 설명하는 텍스트 기반 방법을 제공합니다. 데이터셋은 구조화된 데이터를 저장하고 설명하는 다른 방법을 제공합니다. 그러므로 XML 문서를 데이터셋으로 변환하려면 XML 문서의 노드와 데이터셋의 필드 사이에 있는 대응성을 식별해야 합니다.

예를 들어, 전자 메일 메시지 모음을 나타내는 XML 문서를 가정해 보십시오. 다음과 같이 하나의 메시지를 포함하는 것처럼 보일 수 있습니다.

```

<?xml version="1.0" standalone='yes' ?>
<email>
  <head>
    <from>
      <name>Dave Boss</name>
      <address>dboss@MyCo.com</address>
    </from>
    <to>
      <name>Joe Engineer</name>
      <address>jengineer@MyCo.com</address>
    <to>
    <cc>
      <name>Robin Smith</name>
      <address>rsmith@MyCo.com</address>
    </cc>
    <cc>
      <name>Leonard Devon</name>
      <address>ldevon@MyCo.com</address>
    </cc>
  </head>
  <body>
    <subject>XML components</subject>
    <content>
      Joe,
      Attached is the specification for the new XML component support in Delphi.
      This looks like a good solution to our buisness-to-buisness application!
      Also attached, please find the project schedule.Do you think its reasonable?
      Dave.
    </content>
    <attachment attachfile="XMLSpec.txt"/>
    <attachment attachfile="Schedule.txt"/>
  </body>
</email>

```

이 문서와 데이터셋 사이의 자연 매핑 하나는 각 전자 메일 메시지를 단일 레코드로 매핑합니다. 해당 레코드는 보낸 사람의 이름과 주소에 해당하는 필드를 갖게 됩니다. 전자 메일 메시지에는 받는 사람이 여러 명일 수 있기 때문에 받는 사람(<to>)은 중첩된 데이터셋으로 매핑됩니다. 마찬가지로 참조 목록도 중첩된 데이터셋으로 매핑됩니다. 메시지 자체(<content>)는 메모 필드가 될 수 있지만 제목 줄은 문자열 필드로 매핑됩니다.

니다. 하나의 메시지에 파일을 여러 개 첨부할 수 있으므로 첨부 파일의 이름은 중첩된 데이터셋으로 매핑됩니다. 따라서 위의 전자 메일은 다음과 같은 데이터셋으로 매핑됩니다.

SenderName	SenderAddress	To	CC	Subject	Content	Attach
Dave Boss	dboss@MyCo.Com	(DataSet)	(DataSet)	XML components	(MEMO)	(DataSet)

"To" 필드의 중첩된 데이터셋은 다음과 같습니다.

Name	Address
Joe Engineer	jengineer@MyCo.Com

"CC" 필드의 중첩된 데이터셋은 다음과 같습니다.

Name	Address
Robin Smith	rsmith@MyCo.Com
Leonard Devon	ldevon@MyCo.Com

"Attach" 필드의 중첩된 데이터셋은 다음과 같습니다.

Attachfile
XMLSpec.txt
Schedule.txt

이러한 매핑 정의에는 반복되어 중첩된 데이터셋으로 매핑되는 XML 문서의 해당 노드를 식별하는 작업이 포함됩니다. 해당하는 값이 있고 <content>...</content> 처럼 한 번만 나타나는 태그가 붙은 요소는 값으로 나타날 수 있는 데이터의 타입을 반영하는 데이터 타입을 가진 필드로 매핑됩니다. 첨부 태그의 AttachFile 속성과 같은 태그의 속성도 필드로 매핑됩니다.

XML 문서의 모든 태그가 해당 데이터셋에 나타나지는 않습니다. 예를 들어, <head>...</head> 요소에는 결과 데이터셋에 해당하는 요소가 없습니다. 일반적으로 값이 있는 요소, 반복될 수 있는 요소 또는 태그의 속성만 중첩된 데이터셋 필드를 포함하는 데이터셋의 필드로 매핑됩니다. XML 문서의 부모 노드가 그 값이 자식 노드의 값으로 작성되는 필드로 매핑되는 경우는 이 규칙에서 제외됩니다. 예를 들어, XML 문서에는 다음과 같은 태그 집합이 포함될 수 있습니다.

```
<FullName>
  <Title> Mr. </Title>
  <FirstName> John </FirstName>
  <LastName> Smith </LastName>
</FullName>
```

이것은 다음과 같은 값이 있는 단일 데이터셋 필드로 매핑됩니다.

```
Mr. John Smith
```

XMLMapper 사용

XML 맵퍼 유틸리티인 xmlmapper.exe를 사용하면 다음과 같은 세 가지 방법으로 매핑을 정의할 수 있습니다.

- 기존 XML 스키마 또는 문서에서 사용자가 정의하는 클라이언트 데이터셋으로 매핑합니다. 이것은 이미 XML 스키마가 있는 데이터로 작업하는 데이터베이스 애플리케이션을 만들려는 경우에 유용합니다.
- 기존 데이터 패킷에서 사용자가 정의하는 새로운 XML 스키마로 매핑합니다. 예를 들어, 새로운 business-to-business 통신 시스템을 만드는 경우처럼 이것은 XML의 기존 데이터베이스 정보를 표시하려는 경우에 유용합니다.
- 기존 XML 스키마와 기존 데이터 패킷 사이에서 매핑합니다. 이것은 같은 정보를 설명하는 XML 스키마와 데이터베이스가 있고 두 가지를 함께 사용하여 작업하려는 경우에 유용합니다.

일단 매핑을 정의하면 XML 문서를 데이터 패킷으로 변환하고 데이터 패킷을 XML 문서로 변환하는 데 사용되는 변환 파일을 생성할 수 있습니다. 변환 파일에만 방향을 정할 수 있습니다. 단일 매핑을 사용하면 XML에서 데이터 패킷으로의 변환과 데이터 패킷에서 XML로의 변환 모두를 생성할 수 있습니다.

참고 XML 맵퍼는 midas.dll과 msxml.dll이라는 두 개의 .DLL에 따라 정확히 작동합니다. xmlmapper.exe를 사용하려면 먼저 이 두 가지 .DLL을 설치해야 합니다. 또한 msxml.dll은 COM 서버로 등록해야 합니다. msxml.dll은 Regsvr32.exe를 사용하여 등록할 수 있습니다.

XML 스키마 또는 데이터 패킷 로딩

매핑을 정의하고 변환 파일을 생성할 수 있으려면 먼저 매핑하려는 XML 문서와 데이터 패킷에 대한 설명을 로드해야 합니다.

File|Open을 선택하고 나타나는 대화 상자에서 문서나 스키마를 선택하여 XML 문서나 스키마를 로드할 수 있습니다.

File|Open을 선택하고 나타나는 대화 상자에서 데이터 패킷 파일을 선택하면 데이터 패킷을 로드할 수 있습니다. (이 데이터 패킷은 클라이언트 데이터셋의 *SaveToFile* 메소드를 호출하면 생성되는 파일입니다.) 데이터 패킷을 디스크에 저장하지 않은 경우에는 Datapacket 창에서 마우스 오른쪽 버튼을 클릭하고 Connect To Remote Server를 선택하여 다계층 애플리케이션의 애플리케이션 서버에서 직접 데이터 패킷을 페치(fetch)할 수 있습니다.

XML 문서나 스키마만 또는 데이터 패킷만 로드하거나 둘 다 로드할 수 있습니다. 매핑의 한쪽만 로드하면 XML 맵퍼는 다른 쪽의 자연 매핑을 생성할 수 있습니다.

매핑 정의

XML 문서와 데이터 패킷 사이의 매핑은 데이터 패킷의 모든 필드나 XML 문서의 태그가 붙은 요소를 모두 포함하지 않아도 됩니다. 따라서 먼저 매핑되는 해당 요소를 지정해야 합니다. 이러한 요소를 지정하려면 먼저 대화 상자의 가운데 창에서 매핑 페이지를 선택합니다.

데이터 패킷의 필드로 매핑되는 XML 문서나 스키마의 요소를 지정하려면 XML 문서 창의 Sample 또는 Structure 탭을 선택하고 데이터 패킷 필드로 매핑할 요소의 노드를 더블 클릭합니다.

XML 문서의 속성 또는 태그가 붙은 요소로 매핑되는 데이터 패킷의 필드를 지정하려면 Datapacket 창에서 해당 필드의 노드를 더블 클릭합니다.

매핑의 한쪽(XML 문서 또는 데이터 패킷)만 로드한 경우에는 매핑되는 노드를 선택한 후 다른 쪽을 생성할 수 있습니다.

- XML 문서에서 데이터 패킷을 만드는 경우 먼저 데이터 패킷에서 해당하는 필드의 타입을 결정하는 선택한 노드의 속성을 정의합니다. 가운데 창에서 Node Repository 페이지를 선택합니다. 매핑에 참여하는 각 노드를 선택하고 해당 필드의 속성을 지정합니다. 매핑이 직접적이지 않은 경우(예를 들어, 필드 값이 하위 노드에서 작성되는 필드에 상응하는 노드 경우)에는 User Defined Translation 체크 박스를 검사합니다. 사용자 정의 노드에서 변환을 수행하려면 나중에 이벤트 핸들러를 생성해야 합니다.

노드가 매핑될 방법을 지정한 다음 Create|Datapacket from XML 을 선택합니다. 해당 데이터 패킷이 자동으로 생성되어 Datapacket 창에 표시됩니다.

- 데이터 패킷에서 XML 문서를 만드는 경우 Create|XML from Datapacket을 선택합니다. 데이터 패킷의 필드, 레코드 및 데이터셋에 해당하는 XML 문서의 속성 및 태그 이름을 지정할 수 있는 대화 상자가 나타납니다. 필드 값의 경우에는 값이 지정된 태그가 붙은 요소로 매핑할지, 이름을 지정한 방법을 사용하여 속성으로 매핑할지를 지정합니다. @ 기호로 시작되는 이름은 레코드에 해당하는 태그의 속성으로 매핑되지만 @ 기호로 시작되지 않는 이름은 값이 있고 레코드의 요소 내에서 중첩되는 태그가 붙은 요소로 매핑됩니다.
- XML 문서와 데이터 패킷(클라이언트 데이터셋 파일)을 모두 로드한 경우에는 해당 노드를 같은 순서로 선택했는지 확인하십시오. 해당 노드들은 Mapping 페이지의 맨 위에 있는 테이블에서 서로의 옆에 나타나야 합니다.

XML 문서와 데이터 패킷 모두를 로드하거나 만들고 매핑에 나타나는 노드를 선택하고 나면 Mapping 페이지의 맨 위에 있는 테이블은 정의한 매핑을 반영해야 합니다.

변환 파일 생성

변환 파일을 만들려면 다음 단계를 따르십시오.

- 1 먼저 만들려는 변환을 지정하는 라디오 버튼을 선택합니다.

- 데이터 패킷에서 XML 문서로 매핑을 진행하려는 경우에는 Datapacket to XML 버튼을 선택합니다.

- XML 문서에서 데이터 패킷으로 매핑을 진행하려는 경우에는 XML to Datapacket 버튼을 선택합니다.
- 2 데이터 패킷을 만들려는 경우에는 Create Datapacket As 구역의 라디오 버튼도 사용해야 합니다. 이 버튼을 사용하면 데이터 패킷이 사용되는 방법을 데이터셋으로, 업데이트를 적용하는 델타 패킷으로 또는 데이터를 폐치하기 전에 프로바이더에 제공하는 매개변수로 지정할 수 있습니다.
 - 3 변환의 메모리 내 버전을 만들려면 Create and Test Transformation을 클릭합니다. XML 맵퍼는 데이터 패킷에 대해 생성되는 XML 문서를 Datapacket 창에 표시하거나 XML 문서에 대해 생성되는 데이터 패킷을 XML Document 창에 표시합니다.
 - 4 마지막으로 File|Save|Transformation을 선택하여 변환 파일을 저장합니다. 변환 파일은 사용자가 정의한 변환을 설명하는, 확장명이 .xtr인 특수한 XML 파일입니다.

XML 문서를 데이터 패킷으로 변환

XML 문서를 데이터 패킷으로 변환하는 방법을 나타내는 변환 파일을 작성하면 변환에 사용된 스키마에 순응하는 XML 문서에 대한 데이터 패킷을 만들 수 있습니다. 그런 다음 이러한 데이터 패킷을 클라이언트 데이터셋에 할당하고 파일에 저장하여 파일 기반 데이터베이스 애플리케이션의 기초를 형성합니다.

TXML Transform 컴포넌트는 변환 파일의 매핑에 따라 XML 문서를 데이터 패킷으로 변환합니다.

참고 또한 *TXML Transform*을 사용하면 XML 형식으로 나타나는 데이터 패킷을 임의의 XML 문서로 변환할 수 있습니다.

소스 XML 문서 지정

다음과 같이 소스 XML 문서를 지정하는 세 가지 방법이 있습니다.

- 소스 문서가 디스크에 있는 .xml 파일이면 *SourceXmlFile* 속성을 사용할 수 있습니다.
- 소스 문서가 XML의 메모리 내 문자열이면 *SourceXml* 속성을 사용할 수 있습니다.
- 소스 문서에 대한 IDOMDocument 인터페이스가 있으면 *SourceXmlDocument* 속성을 사용할 수 있습니다.

*TXML Transform*은 위에 나열된 순서로 이 속성들을 검사합니다. 즉, 먼저 *SourceXmlFile* 속성에서 파일 이름을 검사합니다. *SourceXmlFile*이 빈 문자열인 경우에만 *SourceXml* 속성을 검사합니다. 마찬가지로 *SourceXml*이 빈 문자열인 경우에만 *SourceXmlDocument* 속성을 검사합니다.

변환 지정

XML 문서를 데이터 패킷으로 변환하는 변환을 지정하는 방법에는 다음과 같은 두 가지 방법이 있습니다.

- xmlmapper.exe를 사용하여 만든 변환 파일을 나타내도록 *TransformationFile* 속성을 설정합니다.
- 변환에 대한 *IDOMDocument* 인터페이스가 있는 경우에는 *TransformationDocument* 속성을 설정합니다.

*TXMLTransform*은 위에 나열된 순서로 이 속성들을 검사합니다. 즉, 먼저 *TransformationFile* 속성에서 파일 이름을 검사합니다. *TransformationFile*이 빈 문자열인 경우에만 *TransformationDocument* 속성을 검사합니다.

결과 데이터 패킷 가져오기

*TXMLTransform*이 변환을 수행하고 데이터 패킷을 생성하도록 하려면 *Data* 속성만 읽으면 됩니다. 예를 들어, 다음 코드는 XML 문서와 변환 파일을 사용하여 데이터 패킷을 생성한 다음 클라이언트 데이터셋에 할당됩니다.

```
XMLTransform1.SourceXMLFile := 'CustomerDocument.xml';
XMLTransform1.TransformationFile := 'CustXMLToCustTable.xtr';
ClientDataSet1.XMLData := XMLTransform1.Data;
```

사용자 정의 노드 변환

xmlmapper.exe를 사용하여 변환을 정의하면 XML 문서의 일부 노드가 "사용자 정의" 되도록 지정할 수 있습니다. 사용자 정의 노드는 직접적인 '노드 값에서 필드 값'으로의 변환에 의존하지 않고 코드에서 변환을 제공하는 노드입니다.

OnTranslate 이벤트를 사용하는 사용자 정의 노드를 번역하도록 노드를 제공할 수 있습니다. *OnTranslate*는 *TXMLTransform* 컴포넌트가 XML 문서에서 사용자 정의 노드를 만날 때마다 호출됩니다. *OnTranslate* 이벤트 핸들러에서는 소스 문서를 읽고 데이터 패킷의 필드에 대한 결과 값을 지정할 수 있습니다.

예를 들어, *OnTranslate* 이벤트 핸들러는 다음과 같은 형식의 XML 문서의 노드를

```
<FullName>
  <Title> </Title>
  <FirstName> </FirstName>
  <LastName> </LastName>
</FullName>
```

다음과 같은 단일 필드 값으로 변환합니다.

```
procedure TForm1.XMLTransform1Translate(Sender:TObject; Id:String; SrcNode:IDOMNode;
var Value:String; DestNode:IDOMNode);
var
  CurNode:IDOMNode;
begin
```

```
if Id = 'FullName' then
begin
  Value = '';
  if SrcNode.hasChildNodes then
  begin
    CurNode := SrcNode.firstChild;
    Value := Value + CurNode.nodeValue;
    while CurNode <> SrcNode.lastChild do
    begin
      CurNode := CurNode.nextSibling;
      Value := Value + ' ';
      Value := Value + CurNode.nodeValue;
    end;
  end;
end;
end;
```

XML 문서를 프로바이더의 소스로 사용

TXMLTransformProvider 컴포넌트를 사용하면 XML 문서를 데이터베이스 테이블처럼 사용할 수 있습니다. *TXMLTransformProvider*는 XML 문서의 데이터를 패키지로 만들고 클라이언트의 업데이트를 해당 XML 문서에 다시 적용합니다. *TXMLTransformProvider*는 다른 프로바이더 컴포넌트처럼 클라이언트 데이터셋이나 XML 브로커와 같은 클라이언트에 나타납니다. 프로바이더 컴포넌트에 대한 자세한 내용은 24 장 "프로바이더 컴포넌트 사용"을 참조하십시오. 프로바이더 컴포넌트를 클라이언트 데이터셋과 사용하는 것에 대한 자세한 내용은 23-25 페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

XML 프로바이더가 데이터를 제공하고 *XMLDataFile* 속성을 사용하여 업데이트를 적용하는 XML 문서를 지정할 수 있습니다.

TXMLTransformProvider 컴포넌트는 내부 *TXMLTransform* 컴포넌트를 사용하여 데이터 패킷과 소스 XML 문서 사이를 번역합니다. 즉 XML 문서를 데이터 패킷으로 번역하는 경우와 업데이트를 적용한 후 데이터 패킷을 다시 XML 형식의 소스 문서로 번역하는 경우가 있습니다. 이러한 두 가지 *TXMLTransform* 컴포넌트는 각각 *TransformRead* 및 *TransformWrite* 속성을 사용하여 액세스할 수 있습니다.

*TXMLTransformProvider*를 사용할 때는 이 두 가지 *TXMLTransform* 컴포넌트가 데이터 패킷과 소스 XML 문서 사이를 번역할 때 사용하는 변환을 지정해야 합니다. 독립형 *TXMLTransform* 컴포넌트를 사용할 때와 같이 *TXMLTransform* 컴포넌트의 *TransformationFile* 또는 *TransformationDocument* 속성을 설정하여 이 작업을 수행합니다.

또한 변환에 사용자 정의 노드가 포함되면 *OnTranslate* 이벤트 핸들러를 내부 *TXMLTransform* 컴포넌트에 제공해야 합니다.

TransformRead 및 *TransformWrite*의 값인 *TXMLTransform* 컴포넌트에서 소스 문서를 지정할 필요는 없습니다. *TransformRead*의 경우 소스는 프로바이더의 *XMLDataFile* 속성에서 지정한 파일이지만 *XMLDataFile*을 빈 문자열로 설정한 경우에는 *TransformRead*, *XmlSource* 또는 *TransformRead*, *XmlSourceDocument*를 사용하여 소스 문서를 제공할 수 있습니다. *TransformWrite*의 경우 소스는 업데이트를 적용할 때 프로바이더에서 내부적으로 생성됩니다.

프로바이더의 클라이언트로 XML 문서 사용

TXMLTransformClient 컴포넌트는 어댑터로 작동하여 사용자가 XML 문서나 문서 집합을 애플리케이션 서버의 클라이언트 또는 단순히 *TDataSetProvider* 컴포넌트를 통해 연결되는 데이터셋의 클라이언트로 사용할 수 있게 합니다. 즉, *TXMLTransform* 클라이언트를 사용하면 데이터베이스 데이터를 XML 문서로 게시할 수 있으며 XML 문서의 형태로 제공하는 외부 애플리케이션에서의 업데이트 요청(삽입 또는 삭제)을 사용할 수 있습니다.

TXMLTransformClient 객체가 데이터를 페치하고 업데이트를 적용하는 프로바이더를 지정하려면 *ProviderName* 속성을 설정합니다. 클라이언트 데이터셋의 *ProviderName* 속성을 사용하는 경우처럼 *ProviderName*은 원격 애플리케이션 서버의 프로바이더 이름이 되거나 *TXMLTransformClient* 객체와 같이 같은 형태나 데이터 모듈에서 로컬 프로바이더가 될 수 있습니다. 프로바이더에 대한 자세한 내용은 24장 "프로바이더 컴포넌트 사용"을 참조하십시오.

프로바이더가 원격 애플리케이션 서버에 있으면 *DataSnap* 연결 컴포넌트를 사용하여 해당 애플리케이션 서버에 연결해야 합니다. *RemoteServer* 속성을 사용하여 연결 컴포넌트를 지정합니다. *DataSnap* 연결 컴포넌트에 대한 자세한 내용은 25-24 페이지의 "애플리케이션 서버에 연결"을 참조하십시오.

프로바이더에서 XML 문서 페치(fetch)

*TXMLTransformClient*는 내부 *TXMLTransform* 컴포넌트를 사용하여 프로바이더에서 XML 문서로 데이터 패킷을 번역합니다. *TransformGetData* 속성의 값으로 이 *TXMLTransform* 컴포넌트를 액세스할 수 있습니다.

프로바이더의 데이터를 나타내는 XML 문서를 생성할 수 있으려면 *TransformGetData*를 사용하여 데이터 패킷을 적절한 XML 형식으로 번역하는 변환 파일을 지정해야 합니다. 독립형 *TXMLTransform* 컴포넌트를 사용할 때와 같이 *TXMLTransform* 컴포넌트의 *TransformationFile* 또는 *TransformationDocument* 속성을 설정하여 이 작업을 수행합니다. 이 변환에 사용자 정의 노드가 포함되면 *OnTranslate* 이벤트 핸들러와 함께 *TransformGetData*도 제공해야 합니다.

*TransformGetData*의 소스 문서를 지정할 필요가 없으며 *TXMLTransformClient*가 프로바이더에서 소스 문서를 페치합니다. 그러나 프로바이더가 입력 매개변수를 예상하면 데이터를 페치하기 전에 입력 매개변수를 설정해야 합니다. *SetParams* 메소드를 사용하여 프로바이더에서 데이터를 페치하기 전에 이러한 입력 매개변수를 제공합니다. *SetParams*는 두 가지 인수가 있는데 매개변수 값을 추출할 XML 문자열과 이 XML을

데이터 패킷으로 번역하는 변환 파일의 이름입니다. *SetParams*는 변환 파일을 사용하여 XML 문자열을 데이터 패킷으로 변환한 다음 데이터 패킷에서 매개변수 값을 추출합니다.

참고 매개변수 문서나 변환을 다른 방법으로 지정하려면 이 인수 중 하나를 오버라이드할 수 있습니다. *TransformSetParams* 속성의 여러 속성 중 하나를 설정하여 매개변수 또는 매개변수를 변환할 때 사용하는 변환이 포함된 문서를 지정한 다음 *SetParams*를 호출할 때 오버라이드할 인수를 빈 문자열로 설정합니다. 사용할 수 있는 속성에 대한 자세한 내용은 26-6 페이지의 "XML 문서를 데이터 패킷으로 변환"을 참조하십시오.

*TransformGetData*를 구성하고 입력 매개변수를 제공하면 *GetDataAsXml* 메소드를 호출하여 XML을 페치할 수 있습니다. *GetDataAsXml*은 현재의 매개변수 값을 프로바이더로 보내고, 데이터 패킷을 페치하여 XML 문서로 변환하고, 해당 문서를 문자열로 반환합니다. 이 문자열을 다음과 같이 파일에 저장할 수 있습니다.

```
var
  XMLDoc:TFileStream;
  XMLstring;
begin
  XMLTransformClient1.ProviderName := 'Provider1';
  XMLTransformClient1.TransformGetData.TransformationFile := 'CustTableToCustXML.xtr';
  XMLTransformClient1.TransformSetParams.SourceXmlFile := 'InputParams.xml';
  XMLTransformClient1.SetParams('', 'InputParamsToDP.xtr');
  XML := XMLTransformClient1.GetDataAsXml;
  XMLDoc := TFileStream.Create('Customers.xml', fmCreate or fmOpenWrite);
  try
    XMLDoc.Write(XML, Length(XML));
  finally
    XMLDoc.Free;
  end;
end;
```

XML 문서에서 프로바이더로 업데이트 적용

또한 *TXML TransformClient*를 사용하여 XML 문서의 모든 데이터를 프로바이더의 데이터셋으로 삽입하거나 XML 문서에 있는 모든 레코드를 프로바이더의 데이터셋에서 삭제할 수 있습니다. 이러한 업데이트를 수행하려면 다음으로 전달하려는 *ApplyUpdates* 메소드를 호출합니다.

- 그 값이 삽입하거나 삭제할 데이터가 있는 XML 문서의 내용인 문자열.
- XML 데이터를 삽입 또는 삭제 델타 패킷으로 변환할 수 있는 변환 파일의 이름. (XML 맵퍼 유틸리티를 사용하여 변환 파일을 지정할 때는 이 변환이 삽입 델타 패킷에 대한 것인지, 삭제 델타 패킷에 대한 것인지 지정합니다.)
- 업데이트 작업이 종료되기 전에 허용할 수 있는 업데이트 오류의 수. 지정한 수보다 적은 레코드를 삽입하거나 삭제할 수 없으면 *ApplyUpdates*는 실제 오류 수를 반환합니다. 지정한 수보다 많은 레코드를 삽입하거나 삭제할 수 없으면 전체 업데이트 작업이 롤백되고 업데이트가 수행되지 않습니다.

다음 호출에서는 XML 문서 Customers.xml을 델타 패킷으로 변환하고 오류 수에 관계 없이 모든 업데이트를 적용합니다.

```
StringList1.LoadFromFile('Customers.xml');  
nErrors := ApplyUpdates(StringList1.Text, 'CustXMLToInsert.xtr', -1);
```


III

인터넷 애플리케이션 작성

"인터넷 애플리케이션 작성"에 포함된 장에서는 인터넷에서 배포된 애플리케이션을 작성하는 데 필요한 개념 및 기술을 설명합니다.

참고 이 단원에 설명되어 있는 컴포넌트들을 모든 Delphi 에디션에서 사용할 수 있는 것은 아닙니다.

인터넷 애플리케이션 생성

웹 서버 애플리케이션은 기존 웹 서버의 기능을 확장합니다. 웹 서버 애플리케이션은 웹 서버로부터 HTTP 요청 메시지를 받아 이 메시지에서 요청된 모든 작업을 수행하고 웹 서버로 다시 보낼 응답을 작성합니다. Delphi 애플리케이션을 사용하여 수행할 수 있는 모든 작업은 웹 서버 애플리케이션에 포함시킬 수 있습니다.

Delphi는 웹 서버 애플리케이션 개발을 위한 Web broker와 WebSnap이라는 두 개의 다른 아키텍처를 제공합니다. WebSnap과 Web broker가 서로 다르기는 하지만 공통 점이 많습니다. 공통 요소를 갖습니다. WebSnap 아키텍처는 Web Broker의 상위 집합으로 작용합니다. WebSnap 아키텍처는 개발자가 애플리케이션을 실행하지 않고도 페이지의 콘텐츠를 표시할 수 있게 하는 WebSnap Surface Designer와 같은 새로운 기능 및 추가적인 컴포넌트를 제공합니다. WebSnap을 사용하여 개발된 애플리케이션은 Web Broker 컴포넌트를 포함할 수 있는 반면 Web Broker를 사용하여 개발된 애플리케이션은 WebSnap 컴포넌트를 포함할 수 없습니다.

이 장에서는 Web Broker와 WebSnap 기술의 특징에 대해 설명하고 인터넷 기반 클라이언트/서버 애플리케이션에 대한 일반적인 정보를 제공합니다.

Web Broker 및 WebSnap 정보

웹 서버 애플리케이션을 구축하는 첫 번째 단계는 사용할 아키텍처를 선택하는 것입니다. Web broker와 WebSnap은 다음과 같은 동일한 기능을 많이 갖고 있습니다.

- ISAPI, NSAPI, CGI, Win CGI 및 Apache를 포함하여 다양한 타입의 웹 서버 애플리케이션을 지원합니다. 이 내용은 27-6 페이지의 "웹 서버 애플리케이션의 유형"에서 설명합니다.
- 수신된 클라이언트 요청을 별도의 스레드에서 처리되도록 하는 다중 스레드를 지원합니다.
- 더 신속한 응답을 위해 웹 모듈을 캐시로 저장합니다.

하지만 이 두 가지 방식에는 장단점이 있습니다. 이 두 가지 방식의 주요한 차이점은 다음 표에서 개괄적으로 설명합니다.

표 27.1 Web Broker 와 WebSnap 비교

Web Broker	WebSnap
이전 버전과 호환됩니다.	WebSnap 애플리케이션은 콘텐츠를 만드는 모든 Web Broker 컴포넌트를 사용할 수 있지만 이를 포함하는 웹 모듈과 디스패처는 새로운 개념입니다.
크로스 플랫폼(CLX) 애플리케이션에서 사용할 수 있습니다.	현재 WebSnap은 Windows에서만 사용 가능합니다.
하나의 웹 모듈만 애플리케이션에서 사용할 수 있습니다.	다중 웹 모듈은 애플리케이션을 유닛으로 분할할 수 있는데 이는 여러 개발자가 거의 충돌 없이 동일한 프로젝트에서 작업할 수 있게 합니다.
하나의 웹 디스패처만 애플리케이션에서 사용할 수 있습니다.	여러 특수한 용도의 디스패처는 다양한 종류의 요청을 처리합니다.
콘텐츠를 만드는 특수한 컴포넌트로는 페이지 프로듀서, InternetExpress 컴포넌트 및 Web Services 컴포넌트가 있습니다.	Web broker 애플리케이션에서 나타날 수 있는 모든 콘텐츠 프로듀서 및 복잡한 데이터 방식 웹 페이지를 신속하게 구축할 수 있도록 디자인된 기타 많은 것들을 지원합니다.
스크립트를 지원하지 않습니다.	서버측 스크립트(JScript 또는 VBscript)에 대한 지원은 HTML 생성 로직이 비즈니스 로직으로부터 분리되게 합니다.
명명된 페이지를 기본 제공하지 않습니다.	명명된 페이지는 페이지 디스패처에 의해 자동으로 검색되어 서버측 스크립트에서 나타낼 수 있습니다.
세션을 지원하지 않습니다.	세션은 단기간 동안 필요한 최종 사용자에 대한 정보를 저장합니다. 이는 로그인/로그아웃 지원과 같은 작업에 사용될 수 있습니다.
모든 요청은 작업 항목이나 자동 디스패칭 컴포넌트 중의 하나를 사용하여 명시적으로 처리되어야 합니다.	디스패치 컴포넌트는 다양한 요청에 자동으로 응답합니다.
일부 특수한 컴포넌트에서만 콘텐츠의 미리 보기를 제공합니다. 대부분의 개발에서는 볼 수 없습니다.	WebSnap surface designer는 사용자가 웹 페이지를 비주얼하게 구축하고 디자인 타임에 결과를 볼 수 있도록 합니다. 미리 보기는 모든 컴포넌트에서 사용할 수 있습니다.

Web Broker에 대한 자세한 내용은 28장 "Web Broker 사용"을 참조하십시오. WebSnap에 대한 자세한 내용은 29장 "WebSnap 사용"을 참조하십시오.

용어 및 표준

인터넷 상의 활동을 제어하는 많은 프로토콜은 RFC(Request for Comment) 문서에서 정의되며 이러한 문서는 인터넷 프로토콜 엔지니어링 및 개발 조직인 IETF(Internet Engineering Task Force)에 의해 작성, 업데이트 및 유지 관리됩니다. 인터넷 애플리케이션 작성 시 유용한 몇 가지 중요한 RFC가 있습니다.

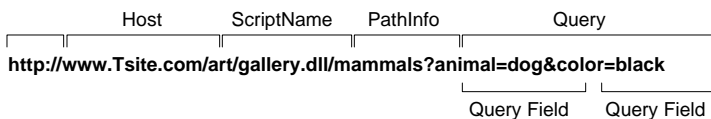
- RFC822, "인터넷 텍스트 메시지의 포맷에 관한 표준"에서는 메시지 헤더의 구조 및 내용을 설명합니다.
- RFC1521, "MIME(Multipurpose Internet Mail Extensions) 1부: 인터넷 메시지 몸체(Body)의 포맷을 지정하고 설명하기 위한 메커니즘"에서는 multipart와 multiformat 메시지를 캡슐화하고 전송하는 데 사용되는 방법을 설명합니다.
- RFC1945, "Hypertext Transfer Protocol-HTTP/1.0"에서는 공동 하이퍼미디어 문서를 배포하는 데 사용되는 전송 메커니즘을 설명합니다.

IETF에서는 자체 웹 사이트인 www.ietf.cnri.reston.va.us에서 RFC의 라이브러리를 유지 관리합니다.

URL(Uniform Resource Locator)의 각 부분

URL(Uniform Resource Locator)은 네트워크에서 사용할 수 있는 리소스 위치에 대해 완전하게 설명한 것으로 애플리케이션에서 액세스할 수 있는 여러 부분들로 구성되어 있습니다. 그림 27.1은 URL을 구성하는 부분들을 보여 줍니다.

그림 27.1 URL(Uniform Resource Locator)의 각 부분



첫 번째 부분은 기술적으로 URL의 일부는 아니지만 프로토콜(http)을 식별합니다. 이 부분은 https(보안 http), ftp 등의 다른 프로토콜을 지정할 수 있습니다.

Host 부분은 웹 서버와 웹 서버 애플리케이션을 실행하는 시스템을 식별합니다. 위의 그림에는 나와 있지 않지만 이 부분은 메시지를 받는 포트를 오버라이드할 수 있습니다. 일반적으로 포트 번호가 프로토콜에 함축되어 있으므로 포트를 지정할 필요가 없습니다.

ScriptName 부분은 웹 서버 애플리케이션의 이름을 지정합니다. 웹 서버는 이 애플리케이션에 메시지를 전달합니다.

스크립트 이름 다음에는 PathInfo가 옵니다. 이 부분은 웹 서버 애플리케이션 내의 메시지의 대상을 식별합니다. 경로 정보 값은 호스트 시스템의 디렉토리, 특정 메시지에 응답하는 컴포넌트의 이름 또는 웹 서버 애플리케이션이 수신 메시지의 처리를 분할하는 데 사용하는 기타 다른 메커니즘도 참조할 수 있습니다.

Query 부분에는 명명된 값 집합이 포함됩니다. 이러한 값과 그 이름은 웹 서버 애플리케이션에 의해 정의됩니다.

URI와 URL 비교

URL은 HTTP 표준인 RFC1945에 정의된 URI(Uniform Resource Identifier)의 하위 집합입니다. 최종 결과가 특정 위치에 있지는 않지만 필요에 따라 만들어지는 많은 소스로부터 웹 서버 애플리케이션에서 콘텐츠를 만드는 경우가 많습니다. URI는 특정 위치와 관련되지 않은 리소스를 정의할 수 있습니다.

HTTP 요청 헤더 정보

HTTP 요청 메시지에는 클라이언트, 요청 대상, 요청 처리 방법, 요청과 함께 보내진 콘텐츠 등에 대한 정보를 설명하는 여러 개의 헤더가 들어 있습니다. 각 헤더는 문자열 값이 따라오는 "Host"와 같은 이름으로 식별합니다. 예를 들어, 다음과 같은 HTTP 요청이 있다고 가정해 봅시다.

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com: 1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

첫 번째 줄은 요청을 GET으로 식별합니다. GET 요청 메시지는 GET 다음의 URI(여기서는 /art/gallery.dll/animals?animal=dog&color=black)에 연결된 콘텐츠를 반환하도록 웹 서버 애플리케이션에 요청합니다. 첫 번째 줄의 마지막 부분은 클라이언트가 HTTP 1.0 표준을 사용하고 있음을 나타냅니다.

두 번째 줄은 Connection 헤더이며 요청이 서비스되었을 때에도 연결을 끊어서는 안 된다는 것을 나타냅니다. 세 번째 줄은 User-Agent 헤더이며 요청을 생성한 프로그램에 대한 정보를 제공합니다. 네 번째 줄은 Host 헤더이며 Host 이름과 연결을 설정하기 위해 연결되는 서버의 포트를 제공합니다. 마지막 줄은 Accept 헤더로 클라이언트가 유효한 응답으로 받을 수 있는 미디어 유형을 나열합니다.

HTTP 서버 활동

웹 브라우저의 클라이언트/서버 특성은 언뜻 보기엔 간단합니다. 링크만 클릭하면 정보가 화면에 표시되기 때문에 웹에서의 정보 검색은 대부분의 사용자에게 매우 단순한 절차입니다. 반면 좀 더 경험있는 사용자들은 HTML 문법의 특성과 프로토콜의 클라이언트/서버 특성을 어느 정도는 이해하고 있습니다. 이러한 정도면 일반적으로 간단한 페이지 지향 웹 사이트 콘텐츠를 충분히 만들 수 있습니다. 그러나 더 복잡한 웹 페이지를 만드는 사람은 HTML을 통해 정보 수집 및 표시를 자동화하는 다양한 옵션을 여러 가지 사용합니다.

웹 서버 애플리케이션을 제작하기 전에 클라이언트가 요청을 하는 방법과 서버가 클라이언트 요청에 응답하는 방법을 이해하는 것이 좋습니다.

클라이언트 요청 구성

HTML 하이퍼텍스트 링크가 선택되거나 사용자가 URL을 지정하면 브라우저는 프로토콜, 지정된 도메인, 정보 경로, 날짜 및 시간, 운영 환경, 브라우저 자체 등에 대한 정보와 기타 콘텐츠 정보를 수집한 다음 요청을 구성합니다.

예를 들어, 폼에서 버튼을 클릭하여 선택된 기준에 따라 이미지 페이지를 표시하려면 클라이언트는 다음과 같은 URL을 만듭니다.

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

이 URL은 www.TSite.com 도메인에 있는 HTTP 서버를 지정합니다. 클라이언트는 www.TSite.com에 접속하고 HTTP 서버에 연결하여 요청을 전달합니다. 이 요청은 다음과 같이 나타납니다.

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection:Keep-Alive
User-Agent:Mozilla/3.0b4Gold (WinNT; I)
Host:www.TSite.com:1024
Accept:image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

클라이언트 요청에 대한 작업

웹 서버는 클라이언트 요청을 받고 그 구성에 따른 여러 작업을 수행할 수 있습니다. 서버가 요청의 /gallery.dll 부분을 프로그램으로서 인지하도록 구성된 경우, 요청에 대한 정보를 해당 프로그램에 전달합니다. 요청에 대한 정보를 프로그램에 전달하는 방법은 다음과 같이 웹 서버 애플리케이션 타입에 따라 다릅니다.

- 프로그램이 CGI(Common Gateway Interface) 프로그램인 경우, 서버는 요청에 포함된 정보를 CGI 프로그램에 직접 전달합니다. 웹 서버는 CGI 프로그램이 실행되는 동안 대기합니다. CGI 프로그램을 종료할 때 콘텐츠를 서버에 직접 다시 전달합니다.
- 프로그램이 WinCGI인 경우, 서버는 파일을 열고 요청 정보를 상세히 기록합니다. 그런 다음 Win-CGI 프로그램을 실행하여 클라이언트 정보가 들어 있는 파일의 위치와 Win-CGI 프로그램이 콘텐츠를 생성하는 데 사용할 파일의 위치를 전달합니다. 웹 서버는 CGI 프로그램이 실행되는 동안 대기합니다. 프로그램이 종료할 때 서버는 Win-CGI 프로그램에서 작성된 콘텐츠 파일의 데이터를 읽습니다.
- 프로그램이 동적 연결 라이브러리(DLL)인 경우, 서버는 필요한 경우 DLL을 로드하고 요청에 포함된 정보를 하나의 구조로서 DLL에 전달합니다. 웹 서버는 CGI 프로그램이 실행되는 동안 대기합니다. DLL을 종료할 때 콘텐츠를 서버에 직접 다시 전달합니다.

CGI 프로그램은 언제나 프로그래머의 요청에 따라 작동하고 데이터베이스 액세스, 간단한 테이블 조회 또는 계산, HTML 문서 만들기 또는 선택 등과 같이 프로그래머가 지정한 작업을 수행합니다.

클라이언트 요청에 응답

웹 서버 애플리케이션은 클라이언트 요청을 완료할 때 HTML 코드 페이지 또는 기타 MIME 콘텐츠를 생성하고 이를 표시할 수 있도록 서버를 통해 클라이언트로 다시 전달합니다. 응답이 전달되는 방식 또한 프로그램의 타입에 따라 다릅니다.

- Win-CGI 스크립트를 완료할 때 HTML 페이지를 생성하고, 그것을 파일로 작성하고, 또 다른 파일에는 응답 정보를 작성하며, 두 파일의 위치를 서버에 다시 전달합니다. 서버는 두 파일을 모두 열고 HTML 페이지를 클라이언트에 다시 전달합니다.
- DLL을 완료할 때 HTML 페이지와 응답 정보를 서버에 직접 다시 전달하는데 이는 클라이언트로 다시 전달됩니다.

웹 서버 애플리케이션을 DLL로 만들면 개별 요청을 서비스하는 데 필요한 프로세스 및 디스크 액세스 수가 감소하여 시스템 로드 및 리소스 사용이 줄어듭니다.

웹 서버 애플리케이션의 유형

Web Broker와 WebSnap 중 어느 것을 사용하든지 다섯 가지 유형의 웹 서버 애플리케이션을 만들 수 있습니다. 또한 애플리케이션 로직을 디버그할 수 있도록 애플리케이션에 웹 서버를 통합하는 웹 애플리케이션 디버거 실행 파일을 만들 수 있습니다. 웹 애플리케이션 디버거 실행 파일은 디버깅을 위해서만 쓰입니다. 애플리케이션 배포 시 다섯 가지 유형 중 하나로 마이그레이션해야 합니다.

다섯 가지 유형의 웹 서버 애플리케이션 모두 *TWebApplication*, *TWebRequest* 및 *TWebResponse*의 타입 특정 자손을 사용합니다.

표 27.2 웹 서버 애플리케이션 컴포넌트

애플리케이션 유형	애플리케이션 객체	요청 객체	응답 객체
Microsoft Server DLL (ISAPI)	<i>TISAPIApplication</i>	<i>TISAPIRequest</i>	<i>TISAPIResponse</i>
Netscape Server DLL (NSAPI)	<i>TISAPIApplication</i>	<i>TISAPIRequest</i>	<i>TISAPIResponse</i>
Apache Server DLL	<i>TApacheApplication</i>	<i>TApacheRequest</i>	<i>TApacheResponse</i>
콘솔 CGI 애플리케이션	<i>TCGIApplication</i>	<i>TCGIRequest</i>	<i>TCGIResponse</i>
Windows CGI 애플리케이션	<i>TCGIApplication</i>	<i>TWinCGIRequest</i>	<i>TWinCGIResponse</i>

ISAPI 및 NSAPI

ISAPI 또는 NSAPI 웹 서버 애플리케이션은 웹 서버에 의해 로드되는 DLL입니다. 클라이언트 요청 정보는 *TISAPIRequest* 및 *TISAPIResponse* 객체를 만드는 *TISAPIApplication*에 의해 하나의 구조로서 DLL에 전달되고 평가됩니다. 각각의 요청 메시지는 별도의 실행 스레드에서 자동으로 처리됩니다.

Apache

Apache 웹 서버 애플리케이션은 웹 서버에 의해 로드되는 DLL입니다. 클라이언트 요청 정보는 *TApacheRequest* 및 *TApacheResponse* 객체를 만드는 *TApacheApplication*에 의해 하나의 구조로서 DLL에 전달되고 평가됩니다. 각각의 요청 메시지는 별도의 실행 스레드에서 자동으로 처리됩니다.

CGI 독립형

CGI 독립형 웹 서버 애플리케이션은 표준 입력으로 클라이언트 요청 정보를 받아 표준 출력으로 결과를 서버에 전달하는 콘솔 애플리케이션입니다. 이 데이터는 *TCGIRequest* 및 *TCGIResponse* 객체를 만드는 *TCGIApplication*에 의해 평가됩니다. 각각의 요청 메시지는 개별 애플리케이션 인스턴스에 의해 처리됩니다.

Win-CGI 독립형

Win-CGI 독립형 웹 서버 애플리케이션은 서버에 의해 작성된 구성 설정 파일(INI 파일)로부터 클라이언트 요청 정보를 받고 서버가 클라이언트에 다시 전달하는 파일에 결과를 기록하는 Windows 애플리케이션입니다. INI 파일은 *TWinCGIRequest* 및 *TWinCGIResponse* 객체를 만드는 *TCGIApplication*에 의해 평가됩니다. 각각의 요청 메시지는 개별 애플리케이션 인스턴스에 의해 처리됩니다.

서버 애플리케이션 디버깅

웹 서버 애플리케이션은 웹 서버에서 온 메시지에 응답하여 실행되므로 이 애플리케이션에 대한 디버깅은 몇 가지 특수한 문제를 야기합니다. 또한 웹 서버가 루프를 벗어나게 되고 애플리케이션이 예상 요청 메시지를 찾지 않기 때문에 간단하게 IDE에서 애플리케이션을 시작할 수 없습니다.

다음 항목은 웹 서버 애플리케이션을 디버깅하는 데 사용할 수 있는 기술을 설명합니다.

웹 애플리케이션 디버거 사용

웹 애플리케이션 디버거는 HTTP 요청, 응답 및 응답 시간을 모니터할 수 있는 손쉬운 방법을 제공합니다. 웹 애플리케이션 디버거는 웹 서버를 대신합니다. 일단 애플리케이션을 디버깅했다면 지원되는 웹 애플리케이션의 타입 중 하나로 변환하고 상용 웹 서버를 사용하여 설치합니다.

웹 애플리케이션 디버거를 사용하려면 먼저 사용자의 웹 애플리케이션을 웹 애플리케이션 디버거 실행 파일로 만들어야 합니다. Web Broker와 WebSnap 중 어느 것을 사용하든지 웹 서버 애플리케이션을 만드는 마법사는 사용자가 애플리케이션을 처음 시작할 때 이를 옵션에 포함시킵니다. 이로 인해 COM 서버이기도 한 웹 서버 애플리케이션이 만들어집니다.

Web Broker를 사용한 웹 서버 애플리케이션 작성 방법에 대한 내용은 28장 "Web Broker 사용"을 참조하십시오. WebSnap 사용에 대한 자세한 내용은 29장 "WebSnap 사용"을 참조하십시오.

웹 애플리케이션 디버거를 사용한 애플리케이션 실행

일단 웹 서버 애플리케이션을 개발했다면 다음과 같은 방법으로 실행하고 디버깅할 수 있습니다.

- 1 IDE에서 로드된 프로젝트에서 브레이크포인트를 설정하면 기타 다른 실행 파일처럼 애플리케이션을 디버깅할 수 있습니다.
- 2 Run|Run을 선택합니다. 그러면 웹 서버 애플리케이션인 COM 서버의 콘솔 윈도우가 나타납니다. 애플리케이션을 처음 실행할 때 COM 서버를 등록하여 웹 애플리케이션 디버거가 액세스할 수 있게 합니다.
- 3 Tools|Web App Debugger를 선택합니다.
- 4 Start 버튼을 클릭합니다. 그러면 기본 브라우저에 Serverinfo 페이지가 나타납니다.
- 5 ServerInfo 페이지는 모든 등록된 웹 애플리케이션 디버거 실행 파일의 드롭다운 목록을 제공합니다. 드롭다운 목록에서 애플리케이션을 선택합니다. 이 드롭다운 목록에서 애플리케이션을 찾지 못하면 애플리케이션을 실행 파일로서 실행해 봅니다. 애플리케이션에서 스스로 등록할 수 있도록 한 번 실행해야 합니다. 그래도 드롭다운 목록에서 애플리케이션을 찾지 못하면 웹 페이지를 새로 고쳐 봅니다.(때때로 웹 브라우저가 이 페이지를 캐시로 저장하여 사용자가 가장 최근에 변경된 내용을 보는 것을 막습니다.)
- 6 드롭다운 목록에서 애플리케이션을 선택했다면 Go 버튼을 누릅니다. 그러면 웹 애플리케이션 디버거에서 사용자의 애플리케이션이 실행되어 사용자의 애플리케이션과 웹 애플리케이션 디버거 사이에 전달되는 요청과 응답 메시지에 관한 세부 사항을 사용자에게 제공합니다.

사용자의 애플리케이션을 다른 유형의 웹 서버 애플리케이션으로 변환

웹 서버 애플리케이션 디버깅을 종료하면 상용 웹 서버를 사용하여 설치하기 전에 다른 유형의 웹 애플리케이션으로 변환해야 합니다. 다음 단계들은 이러한 변환 방법에 대해 설명합니다.

- 1 IDE에서 Project|Add New Project를 선택합니다. 그러면 현재 애플리케이션(사용자의 웹 서버 애플리케이션)을 닫지 않고 새 프로젝트를 엽니다.
- 2 Web Broker 또는 WebSnap 애플리케이션을 시작하기 위해 마법사를 실행하여 만들려는 애플리케이션의 유형을 선택합니다.
- 3 View|Project Manager를 선택하여 프로젝트 관리자를 표시합니다.
- 4 프로젝트 관리자에서 웹 서버 애플리케이션을 구성하는 모든 유닛을 이전 프로젝트에서 방금 추가한 프로젝트로 끌어다 놓습니다. 콘솔 윈도우를 구현하는 유닛은 생략합니다.

이제 해당 유형의 웹 서버 애플리케이션을 갖게 되었습니다.

DLL인 웹 애플리케이션 디버깅

ISAPI, NSAPI 및 Apache 애플리케이션은 사실 이미 정의된 엔트리 포인트를 포함하고 있는 DLL입니다. 웹 서버는 이러한 엔트리 포인트를 호출하여 요청 메시지를 애플리케이션에 전달합니다. 이러한 애플리케이션은 DLL이므로 서버를 실행하기 위해 애플리케이션의 실행 매개변수를 설정하여 디버깅할 수 있습니다.

애플리케이션의 실행 매개변수를 설정하려면 RunParameters를 선택하고 Host Application과 Run Parameters를 웹 서버용 실행 파일과 실행 시 필요한 매개변수를 지정하도록 설정합니다. 웹 서버에 대한 이러한 값들에 관한 자세한 내용은 웹 서버 공급업체가 제공하는 설명서를 참조하십시오.

참고 일부 웹 서버는 사용자가 이러한 방식으로 호스트 애플리케이션을 실행할 수 있는 권한을 갖기 전에 추가적인 변경 사항을 필요로 합니다. 자세한 내용은 웹 서버 공급업체에 문의하십시오.

팁 IIS 5와 함께 Windows 2000을 사용하는 경우, 사용자의 권리를 적절하게 설정하는 데 필요한 모든 변경 사항에 대한 자세한 내용은 다음의 웹 사이트에 설명되어 있습니다.

<http://community.borland.com/article/0,1410,23024,00.html>

Host Application과 Run Parameters를 설정했으면 브레이크포인트를 설정하여 서버가 요청 메시지를 DLL에 전달할 때 브레이크포인트 중 하나에 이르면 정상적으로 디버깅할 수 있습니다.

참고 애플리케이션의 실행 매개변수를 사용하여 웹 서버를 실행하기 전에 서버가 이미 실행되고 있는지는 않은지 확인하십시오.

Windows NT에서의 디버깅

Windows NT에서 DLL을 디버깅하려면 올바른 사용자 권한을 가져야 합니다. User Manager에서 다음과 같은 권한을 부여하는 목록에 사용자 이름을 추가합니다.

- 서비스로 로그인
- 운영 체제의 일부로 활동
- 보안 감사 생성

Windows 2000에서의 디버깅

Windows 2000에서 다음과 같은 권한을 부여 받습니다.

- 1 제어판의 관리 도구에서 로컬 보안 정책을 클릭합니다. 로컬 정책을 확장하고 사용자 권한 할당을 클릭합니다. 오른쪽 패널에 있는 운영 체제의 일부로 활동을 더블 클릭합니다.
- 2 추가를 선택하여 사용자를 목록에 추가합니다. 현재 사용자를 추가합니다.
- 3 변경 사항이 반영되도록 재부팅합니다.

28

Web Broker 사용

컴포넌트 팔레트의 인터넷 탭에 있는 Web Broker 컴포넌트를 통해 특정 Uniform Resource Identifier (URI) 와 연결된 이벤트 핸들러를 작성할 수 있습니다. 처리가 완료되면 프로그램에 따른 HTML 또는 XML 문서들을 만들고 클라이언트로 전송할 수 있습니다. Web Broker 컴포넌트를 사용하면 크로스 플랫폼 애플리케이션을 개발할 수 있습니다.

웹 페이지의 콘텐츠는 종종 데이터베이스에서 가져옵니다. 데이터베이스와의 연결을 자동 관리하기 위해 인터넷 컴포넌트를 사용할 수 있고 하나의 DLL이 동시에 스레드에 대해 안전하게 여러 데이터베이스를 연결할 수 있게 합니다.

이 장의 다음 단원은 Web Broker 컴포넌트를 사용하여 웹 서버 애플리케이션을 만드는 방법을 설명합니다.

Web Broker로 웹 서버 애플리케이션 생성

다음과 같은 Web Broker 아키텍처를 사용하여 새로운 웹 서버 애플리케이션을 생성할 수 있습니다.

- 1 File|New|Other를 선택합니다.
 - 2 New Items 대화 상자에서 New tab을 선택하여 웹 서버 애플리케이션을 선택합니다.
 - 3 그러면 다음 웹 서버 애플리케이션 유형 중 하나를 선택할 수 있는 대화 상자가 나타납니다.
- ISAPI와 NSAPI: 이 애플리케이션 유형을 선택하면 웹 서버에서 원하는 export 된 메소드로 사용자의 프로젝트를 DLL로 설정합니다. 프로젝트 파일에 라이브러리 헤더를 추가하고 사용 목록에 필수 항목을 추가한 다음 프로젝트 파일의 절을 export 합니다.

- Apache: 이 애플리케이션 유형을 선택하면 Apache 웹 서버에서 원하는 export 된 메소드로 사용자의 프로젝트를 DLL로 설정합니다. 프로젝트 파일에 라이브러리 헤더를 추가하고 사용 목록에 필수 항목을 추가한 다음 프로젝트 파일의 절을 export 합니다.
- CGI 독립형: 이런 유형의 애플리케이션을 선택하면 프로젝트가 콘솔 애플리케이션으로 설정되고 프로젝트 파일의 uses 절에 필수 항목이 추가됩니다.
- Win-CGI 독립형: 이 애플리케이션 유형을 선택하면 프로젝트가 Windows 애플리케이션으로 설정되고 프로젝트 파일의 uses 절에 필요한 항목이 추가됩니다.
- 실행 가능한 웹 애플리케이션 디버거 독립형이 애플리케이션 유형을 선택하면 웹 서버 애플리케이션을 개발하고 테스트하기 위한 환경이 설정됩니다. 이런 유형의 애플리케이션은 배포용이 아닙니다.

애플리케이션이 사용하게 될 웹 서버 유형과 통신하는 웹 서버 애플리케이션의 유형을 선택합니다. 그러면 인터넷 컴포넌트를 사용하도록 구성되고 빈 웹 모듈을 포함한 새 프로젝트가 만들어집니다.

웹 모듈

웹 모듈 (*TWebModule*)은 *TDataModule*의 자손이며 웹 애플리케이션의 비즈니스 룰과 년비주얼 컴포넌트에 대해서 중앙에서 일괄적으로 제어하기 위해 조상과 동일한 방법으로 사용됩니다.

애플리케이션이 응답 메시지를 생성하는 데 사용하는 콘텐츠 프로듀서를 추가합니다. 기본 제공 콘텐츠 프로듀서로는 *TPageProducer*, *TDataSetPageProducer*, *TDataSetTableProducer*, *TQueryTableProducer*, *TInetXPageProducer* 또는 사용자 스스로 작성한 *TCustomContentProducer*의 자손 등이 있습니다. 사용자의 애플리케이션이 데이터베이스에서 끌어낸 자료를 포함한 응답 메시지를 생성하면 사용자는 데이터 액세스 컴포넌트 또는 다계층 데이터베이스 애플리케이션에서의 클라이언트처럼 작동하는 웹 서버를 생성하는 특별한 컴포넌트를 추가할 수 있습니다.

웹 모듈은 년비주얼 컴포넌트와 비즈니스 룰을 저장하는 작업뿐만 아니라 들어오는 HTTP 요청 메시지를 이러한 요청에 대한 응답을 생성하는 액션 항목에 일치시키는 디스패처 역할도 수행합니다.

웹 애플리케이션에서 사용할 많은 년비주얼 컴포넌트와 비즈니스 룰과 함께 설정된 데이터 모듈을 이미 갖고 있는 경우가 있습니다. 이럴 경우 웹 모듈을 기존 데이터 모듈로 바꿀 수 있습니다. 자동으로 생성된 웹 모듈을 단순히 삭제하여 사용자의 데이터 모듈로 바꿉니다. 그런 다음 웹 모듈의 경우처럼 데이터 모듈에 *TWebDispatcher* 컴포넌트를 추가하여 요청 메시지를 액션 항목에 디스패치할 수 있도록 합니다. 들어오는 HTTP 요청 메시지에 응답하기 위해 액션 항목이 선택되는 방법을 변경하려면 *TCustomWebDispatcher*에서 새 디스패처 컴포넌트를 파생시켜 데이터 모듈에 추가합니다.

프로젝트는 단 하나의 디스패처만 포함할 수 있습니다. 디스패처는 프로젝트를 만들 때 자동으로 생성된 웹 모듈이 되거나 웹 모듈을 대체하는 데이터 모듈에 추가된 *TWebDispatcher* 컴포넌트가 될 수 있습니다. 실행 중에 디스패처를 포함한 두 번째 데이터 모듈이 만들어지면 웹 서버 애플리케이션은 런타임 오류를 생성합니다.

- 참고** 디자인 타임에 설정한 웹 모듈이 실제로는 템플릿이 됩니다. ISAPI와 NSAPI 애플리케이션에서 각 요청 메시지는 별도의 스레드를 생성하고 웹 모듈의 별도 인스턴스와 그 내용이 각 스레드에 대해 동적으로 생성됩니다.
- 경고** DLL 기반 웹 서버 애플리케이션의 웹 모듈은 나중의 재사용을 위해 캐쉬에 저장되어 응답 시간을 증가시킵니다. 디스패처의 상태와 액션 리스트는 요청들 사이에서 다시 초기화되지 않습니다. 실행 중에 액션 항목을 사용 가능 또는 사용 불가능하게 하는 것은 그 모듈이 다음의 클라이언트 요청에 사용될 때 예기치 못한 결과를 가져올 수 있습니다.

웹 애플리케이션 객체

웹 애플리케이션을 위해 설정되는 프로젝트에는 *Application*이라는 전역 변수가 포함됩니다. *Application*은 사용자가 만들고 있는 애플리케이션 유형에 해당하는 *TWebApplication* (*TISAPIApplication* 또는 *TCGIApplication*)의 자손입니다. 웹 서버로부터 받은 HTTP 요청 메시지에 응답하여 실행됩니다.

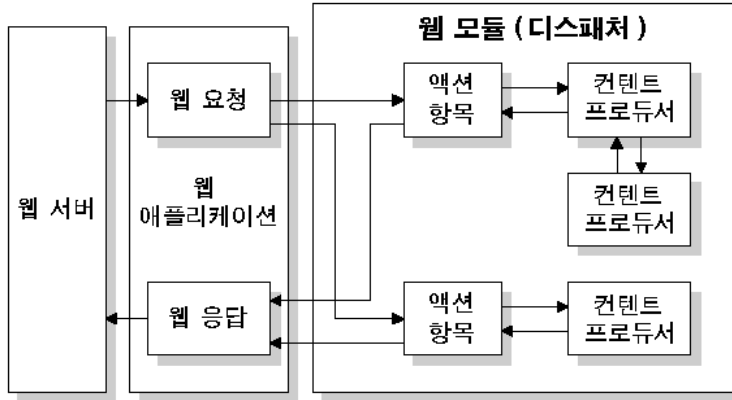
- 경고** 프로젝트의 **uses** 절에서 CGIApp 또는 ISAPIApp 유닛 뒤에 폼 유닛을 포함해서는 안 됩니다. 폼도 *Application*이라는 전역 변수를 선언하기 때문에 CGIApp 또는 ISAPIApp 유닛 뒤에 올 경우 *Application*이 잘못된 유형의 객체로 초기화됩니다.

Web Broker 애플리케이션의 구조

웹 애플리케이션은 HTTP 요청 메시지를 받으면 *TWebRequest* 객체를 만들어 HTTP 요청 메시지를 나타내고 *TWebResponse* 객체를 만들어 반환해야 할 응답을 나타냅니다. 그런 다음 웹 애플리케이션은 이 객체들을 웹 디스패처(웹 모듈 또는 *TWebDispatcher* 컴포넌트)에 전달합니다.

웹 디스패처는 웹 서버 애플리케이션의 흐름을 제어합니다. 디스패처는 특정 유형의 HTTP 요청 메시지에 대한 처리 방법을 알고 있는 액션 항목(*TWebActionItem*)의 컬렉션을 유지 관리합니다. 디스패처는 적절한 액션 항목 또는 자동 디스패칭 컴포넌트를 식별하여 HTTP 요청 메시지를 처리하며 요청된 모든 액션을 수행하거나 응답 메시지를 생성할 수 있도록 요청 및 응답 객체를 식별된 핸들러에 전달합니다. 자세한 내용은 28-4 페이지의 "웹 디스패처"에 설명되어 있습니다.

그림 28.1 서버 애플리케이션의 구조



액션 항목은 요청을 읽고 응답 메시지를 결합합니다. 특수한 컨텐츠 프로듀서 컴포넌트는 액션 항목이 사용자 지정 HTML 코드 또는 다른 MIME 콘텐츠를 포함할 수 있는 응답 메시지의 콘텐츠를 동적으로 생성하는 데 도움을 줍니다. 컨텐츠 프로듀서는 다른 컨텐츠 프로듀서나 *THTMLTagAttributes*의 자손을 사용하여 액션 항목이 응답 메시지의 콘텐츠를 만들 수 있게 합니다. 컨텐츠 프로듀서에 대한 자세한 내용은 28-13 페이지의 "응답 메시지 콘텐츠 생성"을 참조하십시오.

다계층 데이터베이스 애플리케이션 안에 Web Client를 만들면 웹 서버 애플리케이션은 XML로 인코딩된 데이터베이스 정보와 자바스크립트로 인코딩된 데이터베이스 조작 클래스를 나타내는, 부가적이고 자동으로 디스패치되는 컴포넌트를 포함할 수 있습니다. Web Service를 구현하는 서버를 만들면 웹 서버 애플리케이션에 SOAP 기반 메시지를 변환하고 실행시키는 invoker로 전달하는 자동 디스패칭 컴포넌트가 포함됩니다. 디스패처는 액션 항목을 모두 시도해 본 후 요청 메시지를 처리하기 위한 자동 디스패칭 컴포넌트를 호출합니다.

모든 액션 항목 또는 자동 디스패칭 컴포넌트가 *TWebResponse* 객체를 채워 응답 생성을 끝내면 디스패처는 결과를 웹 애플리케이션에 전달합니다. 그러면 웹 애플리케이션은 웹 서버를 통해 클라이언트에게 응답을 보냅니다.

웹 디스패처

웹 모듈을 사용하고 있다면 웹 모듈이 웹 디스패처 역할도 합니다. 이미 존재하는 데이터 모듈을 사용하는 중이라면 *TWebDispatcher* 디스패처 컴포넌트 하나를 그 데이터 모듈에 추가해야 합니다. 디스패처는 특정 유형의 HTTP 요청 메시지에 대한 처리 방법을 알고 있는 액션 항목의 컬렉션을 유지 관리합니다. 웹 애플리케이션이 요청 객체와 응답 객체를 디스패처에 전달하면 디스패처는 요청에 응답하기 위해 하나 이상의 액션 항목을 선택합니다.

디스패처에 액션 추가

Object Inspector에서 디스패처의 *Actions* 속성의 생략 버튼을 클릭하여 액션 에디터를 엽니다. 이 액션 에디터에서 Add 버튼을 클릭하면 디스패처에 액션 항목을 추가할 수 있습니다.

다른 요청 메소드나 대상 URI에 응답하려면 디스패처에 액션을 추가합니다. 액션 항목은 다양한 방법으로 설정할 수 있습니다. 요청을 미리 처리하는 액션 항목에서 시작하여 응답을 보냈는지, 오류 코드를 반환했는지의 응답 완료 여부를 검사하는 기본 액션으로 마무리할 수 있습니다. 또는 각 액션 항목이 요청을 완벽하게 처리할 수 있도록 모든 유형의 요청에 대해 별도의 액션 항목을 추가할 수 있습니다.

액션 항목은 28-6 페이지의 "액션 항목"에서 더 자세히 설명되어 있습니다.

요청 메시지 디스패칭

클라이언트 요청을 받으면 디스패처는 *BeforeDispatch* 이벤트를 생성합니다. 이 이벤트가 생성되면 사용자가 액션 항목으로 확인하기 전에 애플리케이션에서 요청 메시지를 미리 처리할 수 있는 기회를 제공합니다.

그런 다음 디스패처는 요청 메시지에 있는 대상 URL의 경로 정보 부분과 일치하고 요청 메시지의 메소드로서 지정된 서비스를 제공할 수 있는 항목을 액션 항목 목록에서 검색합니다. 즉, *TWebRequest* 객체의 *PathInfo* 및 *MethodType* 속성을 액션 항목에 있는 같은 이름의 속성과 비교합니다.

적절한 액션 항목을 찾으면 디스패처는 해당 액션 항목을 실행시키며 이 때 액션 항목은 다음 중 하나를 실행합니다.

- 응답 콘텐츠를 완성하고 요청이 완전히 처리된 응답 또는 신호를 보냅니다.
- 응답을 추가하고 다른 액션 항목이 작업을 완료할 수 있게 해줍니다.
- 요청을 다른 액션 항목에 전달합니다.

액션 항목을 모두 검사한 다음 메시지가 처리되지 않았을 경우, 디스패처는 액션 항목을 사용하지 않는 특별하게 등록된 자동 디스패칭 컴포넌트를 점검합니다. 자동 디스패칭 컴포넌트는 다계층 데이터베이스 애플리케이션에만 사용됩니다. 자세한 내용은 25-34 페이지의 "InternetExpress를 사용하여 웹 애플리케이션 구축"을 참조하십시오.

모든 액션 항목과 특별하게 등록된 자동 디스패칭 컴포넌트를 점검한 후에도 여전히 요청 메시지가 모두 처리되지 않으면 디스패처는 기본 액션 항목을 호출합니다. 기본 액션 항목은 대상 URL이나 요청 메소드와 반드시 일치할 필요는 없습니다.

기본 액션이 있어서 디스패처가 기본 액션을 포함한 액션 리스트의 끝에 도달했고 트리거된 액션이 없을 경우 서버에는 아무 것도 전달되지 않습니다. 이 경우 서버는 클라이언트와의 연결을 끊습니다.

요청이 액션 항목에 의해 처리될 경우 디스패처는 *AfterDispatch* 이벤트를 생성합니다. 이 이벤트가 생성되면 애플리케이션은 생성된 응답을 검사하여 최종적으로 필요한 내용을 변경할 수 있습니다.

액션 항목

각 액션 항목 (*TWebActionItem*)은 주어진 유형의 요청 메시지에 응답하여 특정 작업을 수행합니다.

액션 항목은 요청에 완전하게 응답하거나 응답의 일부를 수행하고 다른 액션 항목이 작업을 완료하도록 허용할 수 있습니다. 액션 항목은 요청에 대해 HTTP 응답 메시지를 보내거나 다른 액션 항목이 완료할 수 있도록 응답의 일부를 설정할 수 있습니다. 액션 항목이 응답을 완료하고 보내지 않았을 경우 웹 서버 애플리케이션이 응답 메시지를 보냅니다.

액션 항목 실행 시기 결정

대부분의 액션 항목 속성은 HTTP 요청 메시지를 처리하기 위해 디스패처가 액션 항목을 선택하는 시기를 결정합니다. 액션 항목의 속성을 설정하려면 Object Inspector에서 디스패처의 *Actions* 속성을 선택하고 생략 버튼을 클릭하여 액션 에디터를 열어야 합니다. 액션 에디터에서 액션을 선택하면 Object Inspector에서 해당 액션의 속성을 수정할 수 있습니다.

대상 URL

디스패처는 액션 항목의 *PathInfo* 속성과 요청 메시지의 *PathInfo*를 비교합니다. 이 속성 값은 액션 항목이 처리할 준비가 된 모든 요청의 URL에 있는 경로 정보 부분이어야 합니다. 예를 들어, 다음과 같은 URL이 있다고 가정해 보십시오.

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

여기서 */gallery.dll* 부분이 웹 서버 애플리케이션을 나타낸다고 가정하면 다음 부분이 경로 정보입니다.

```
/mammals
```

요청을 서비스할 때 웹 애플리케이션이 정보를 검색해야 할 장소를 나타내거나 웹 애플리케이션을 논리적 하위 서비스로 나누기 위해 경로 정보를 사용합니다.

요청 메소드 유형

액션 항목의 *MethodType* 속성은 처리할 수 있는 요청 메시지 유형을 나타냅니다. 디스패처는 액션 항목의 *MethodType* 속성과 요청 메시지의 *MethodType*을 비교합니다. *MethodType*은 다음 값 중 하나를 가질 수 있습니다.

표 28.1 MethodType 값

값	의미
<i>mtGet</i>	요청은 응답 메시지로 반환될 대상 URI와 연결된 정보를 요구합니다.
<i>mtHead</i>	요청은 <i>mtGet</i> 요청에 서비스하지만 응답 콘텐츠를 생략하는 경우처럼 응답의 헤더 속성을 요구합니다.
<i>mtPost</i>	요청은 웹 애플리케이션에 포스트될 정보를 제공합니다.

표 28.1 MethodType 값 (계속)

값	의미
<i>mtPut</i>	요청은 대상 URI와 연결된 리소스를 요청 메시지의 콘텐츠로 바꾸도록 요구합니다.
<i>mtAny</i>	<i>mtGet</i> , <i>mtHead</i> , <i>mtPut</i> 및 <i>mtPost</i> 를 비롯한 모든 요청 메소드 유형을 일치시킵니다.

액션 항목 활성화 및 비활성화

각 액션 항목에는 해당 액션 항목을 활성화 또는 비활성화하는 데 사용할 수 있는 *Enabled* 속성이 있습니다. *Enabled*를 *False*로 설정하면 액션 항목이 비활성화되므로 디스패처가 요청 처리를 위해 액션 항목을 검색할 때 그 액션 항목은 고려하지 않습니다.

BeforeDispatch 이벤트 핸들러는 디스패처가 액션 항목을 요청 메시지와 일치시키기 전에 액션 항목의 *Enabled* 속성을 변경하여 액션 항목이 처리해야 할 요청을 제어할 수 있습니다.

주의 실행 도중에 액션의 *Enabled* 속성을 변경하면 후속 요청에 대해 예기치 않은 결과가 발생할 수 있습니다. 웹 서버 애플리케이션이 웹 모듈을 캐시에 저장하는 DLL이면 초기 상태가 다음 요청을 위해 다시 초기화되지 않습니다. 모든 액션 항목을 해당 시작 상태로 올바르게 초기화하려면 *BeforeDispatch* 이벤트를 사용합니다.

기본 액션 항목 선택

액션 항목 중 하나만 기본 액션 항목이 될 수 있습니다. 기본 액션 항목을 선택하려면 해당 액션 항목의 *Default* 속성을 *True*로 설정합니다. 특정 액션 항목의 *Default* 속성이 *True*로 설정되면 이전 기본 액션 항목이 있을 경우에는 *Default* 속성은 *False*로 설정됩니다.

디스패처는 요청 처리를 위해 액션 항목 목록을 검색하여 특정 액션 항목을 선택할 때 기본 액션 항목의 이름을 저장합니다. 액션 항목 목록의 끝에 도달했지만 요청이 완전히 처리되지 않은 경우 디스패처는 기본 액션 항목을 실행합니다.

디스패처는 기본 액션 항목의 *PathInfo* 또는 *MethodType*을 검사하지 않습니다. 또한 디스패처는 기본 액션 항목의 *Enabled* 속성도 검사하지 않습니다. 따라서 기본 액션 항목의 *Enabled* 속성을 *False*로 설정함으로써 맨 끝 부분에서만 기본 액션 항목을 호출할 수 있습니다.

잘못된 URI 또는 *MethodType*을 나타내는 오류 코드만 반환될 경우에도 기본 액션 항목은 발생하는 모든 요청을 처리하도록 준비되어야 합니다. 기본 액션 항목이 요청을 처리하지 않을 경우 웹 클라이언트에 응답이 보내지지 않습니다.

주의 실행 도중에 액션의 *Default* 속성을 변경하면 현재 요청에 대해 예기치 않은 결과가 생길 수 있습니다. 이미 트리거된 액션의 *Default* 속성을 *True*로 설정할 경우 해당 액션은 다시 평가되지 않고 액션 리스트의 끝에 도달했을 때 디스패처는 이 액션을 실행하지 않습니다.

액션 항목으로 요청 메시지에 응답

액션 항목은 실행 시 웹 서버 애플리케이션의 실제 작업을 수행합니다. 웹 디스패처가 액션 항목을 실행시키면 해당 액션 항목은 다음 두 가지 방법으로 현재 요청 메시지에 응답할 수 있습니다.

- 액션 항목이 *Producer* 속성 값으로 연결된 프로듀서 컴포넌트를 가질 경우 해당 프로듀서는 *Content* 메소드를 사용하여 응답 메시지의 *Content*를 자동으로 지정합니다. 컴포넌트 팔레트의 인터넷 페이지에는 응답 메시지 콘텐츠의 HTML 페이지를 만드는 데 도움이 되는 여러 콘텐츠 프로듀서 컴포넌트가 포함되어 있습니다.
- 연결된 프로듀서가 있고 이 프로듀서가 응답을 지정한 후 액션 항목은 *OnAction* 이벤트를 받습니다. *OnAction* 이벤트 핸들러에는 모든 응답 정보를 채울 *TWebResponse* 객체와 HTTP 요청 메시지를 나타내는 *TWebRequest* 객체가 전달됩니다.

하나의 콘텐츠 프로듀서가 액션 항목의 내용을 생성할 수 있을 경우 가장 간단한 방법은 콘텐츠 프로듀서를 액션 항목의 *Producer* 속성으로 지정하는 것입니다. 그러나 *OnAction* 이벤트 핸들러에서 언제든지 모든 콘텐츠 프로듀서에 액세스할 수 있습니다. *OnAction* 이벤트 핸들러는 여러 콘텐츠 프로듀서를 사용하고 응답 메시지 속성을 지정하는 등의 작업을 수행할 수 있도록 더 많은 유연성을 부여합니다.

콘텐츠 프로듀서 컴포넌트와 *OnAction* 이벤트 핸들러는 모두 객체 또는 런타임 라이브러리 메소드를 사용하여 요청 메시지에 응답합니다. 또한 데이터베이스 액세스, 계산 수행, HTML 문서 만들기 또는 선택 등의 작업을 할 수 있습니다. 콘텐츠 프로듀서 컴포넌트를 사용한 응답 콘텐츠 생성에 대한 자세한 내용은 28-13 페이지의 "응답 메시지 콘텐츠 생성"을 참조하십시오.

응답 보내기

OnAction 이벤트 핸들러는 *TWebResponse* 객체의 메소드를 사용하여 웹 클라이언트에게 응답을 돌려 보낼 수 있습니다. 그러나 클라이언트에게 응답을 보내는 액션 항목이 없으면 마지막 액션 항목이 요청이 처리되었다는 것을 알 때까지 웹 서버 애플리케이션에서 계속하여 요청을 보냅니다.

여러 액션 항목 사용

단일 액션 항목에서 요청에 응답하거나 여러 액션 항목 간에 작업을 분할할 수 있습니다. 액션 항목이 응답 메시지 설정을 완전히 끝내지 않았을 경우 액션 항목은 *Handled* 매개변수를 *False*로 설정하여 *OnAction* 이벤트 핸들러에서 이러한 상태를 알려야 합니다.

요청 메시지에 응답하는 작업이 여러 액션 항목 간에 분할되면 다른 액션 항목이 작업을 계속할 수 있도록 각 *Handled* 매개변수를 *False*로 설정합니다. 이 때 기본 액션 항목에서는 *Handled* 매개변수를 *True*로 설정된 대로 두어야 합니다. 그렇지 않으면 웹 클라이언트에게 응답이 보내지지 않습니다.

여러 액션 항목 간에 작업을 분할할 때 기본 액션 항목의 *OnAction* 이벤트 핸들러 또는 디스패처의 *AfterDispatch* 이벤트 핸들러는 모든 작업이 수행되었는지 검사하고 그렇지 않을 경우 적절한 오류 코드를 설정해야 합니다.

클라이언트 요청 정보 액세스

웹 서버 애플리케이션이 HTTP 요청 메시지를 받으면 클라이언트 요청의 헤더가 *TWebRequest* 객체의 속성으로 로드됩니다. NSAPI 및 ISAPI 애플리케이션에서 요청 메시지는 *TISAPIRequest* 객체로 인해 캡슐화됩니다. 콘솔 CGI 애플리케이션은 *TCGIRequest* 객체를 사용하며 Windows CGI 애플리케이션은 *TWinCGIRequest* 객체를 사용합니다.

요청 객체의 속성은 읽기 전용입니다. 이러한 속성을 사용하면 클라이언트 요청에서 사용할 수 있는 모든 정보를 수집할 수 있습니다.

요청 헤더 정보를 포함하는 속성

요청 객체 속성은 대부분 HTTP 요청 헤더에서 가져온 요청에 대한 정보를 포함합니다. 그러나 모든 요청이 이러한 속성 모두에 대해 값을 제공하는 것은 아닙니다. 특히 HTTP 표준이 계속 향상됨에 따라 일부 요청은 요청 객체의 속성으로 표시되지 않는 헤더 필드를 포함할 수도 있습니다. 요청 객체 속성 중 하나로 표시되지 않는 요청 헤더 필드의 값을 얻으려면 *GetFieldByName* 메소드를 사용하십시오.

대상을 식별하는 속성

요청 메시지의 전체 대상은 *URL* 속성에 의해 주어집니다. 일반적으로 요청 메시지의 전체 대상은 프로토콜(HTTP), *Host*(서버 시스템), *ScriptName*(서버 애플리케이션), *PathInfo*(호스트 상의 위치) 및 *Query*로 나눌 수 있는 URL입니다.

이러한 각 부분은 고유한 속성으로 표시됩니다. 프로토콜은 항상 HTTP이고 *Host* 및 *ScriptName*은 웹 서버 애플리케이션을 식별합니다. 디스패처는 액션 항목을 요청 메시지에 일치시킬 때 *PathInfo* 부분을 사용합니다. 일부 요청에서는 요청된 정보의 세부 사항을 지정하기 위해 *Query*가 사용되며 이 부분의 값도 *QueryFields* 속성으로 구분 분석됩니다.

웹 클라이언트를 설명하는 속성

또한 요청에는 요청이 시작된 장소에 대한 정보를 제공하는 여러 속성이 포함되어 있습니다. 이러한 속성은 보낸 사람의 전자 메일 주소(*From* 속성)에서 메시지가 시작된 URI(*Referer* 또는 *RemoteHost* 속성)에 이르기까지 모든 정보를 포함합니다. 요청에 콘텐츠가 포함되어 있고 이 콘텐츠가 요청과 동일한 URI에서 가져온 것이 아닌 경우 *DerivedFrom* 속성에 의해 콘텐츠의 소스가 주어집니다. 이외에도 클라이언트의 IP 주소(*RemoteAddr* 속성) 및 요청을 보낸 애플리케이션의 이름과 버전(*UserAgent* 속성)을 확인할 수 있습니다.

요청 목적을 식별하는 속성

Method 속성은 요청 메시지가 서버 애플리케이션에 요구하는 작업을 설명하는 문자열입니다. HTTP 1.1 표준은 다음 메소드를 정의합니다.

값	메시지가 요청하는 내용
<i>OPTIONS</i>	사용할 수 있는 통신 옵션에 대한 정보
<i>GET</i>	<i>URL</i> 속성에 의해 식별된 정보
<i>HEAD</i>	응답 콘텐츠 없이 해당되는 <i>GET</i> 메시지에서 가져온 헤더 정보
<i>POST</i>	<i>Content</i> 속성에 포함된 데이터를 적절하게 포스트하도록 서버 애플리케이션에 요청
<i>PUT</i>	<i>URL</i> 속성이 나타낸 리소스를 <i>Content</i> 속성에 포함된 데이터로 바꾸도록 서버 애플리케이션에 요청
<i>DELETE</i>	<i>URL</i> 속성에 의해 식별되는 리소스를 삭제하거나 숨기도록 서버 애플리케이션에 요청
<i>TRACE</i>	요청 수신을 확인하는 루프백을 보내도록 서버 애플리케이션에 요청

Method 속성은 웹 클라이언트가 요청하는 서버의 다른 메소드를 나타낼 수 있습니다. 웹 서버 애플리케이션은 *Method*의 모든 가능한 값에 대해 응답을 제공할 필요가 없습니다. 단, HTTP 표준에서는 *GET* 및 *HEAD* 요청 모두에 대해 서비스할 것을 요구합니다.

MethodType 속성은 *Method* 값이 *GET*(*mtGet*), *HEAD*(*mtHead*), *POST*(*mtPost*), *PUT*(*mtPut*) 또는 다른 문자열 (*mtAny*) 중에서 무엇인지 나타냅니다. 디스패처는 *MethodType* 속성의 값을 각 액션 항목의 *MethodType*과 일치시킵니다.

예상 응답을 설명하는 속성

Accept 속성은 웹 클라이언트가 응답 메시지의 콘텐츠로 적용하는 미디어 유형을 나타냅니다. *IfModifiedSince* 속성은 클라이언트가 최근에 변경된 정보만 원하는지 지정합니다. *Cookie* 속성은 일반적으로 애플리케이션에서 이전에 추가한 상태 정보를 포함하고 있으며 여기서 응답을 수정할 수 있습니다.

컨텐츠를 설명하는 속성

대부분의 요청은 정보에 대한 요청이기 때문에 콘텐츠를 포함하지 않습니다. 그러나 *POST* 요청과 같은 일부 요청은 웹 서버 애플리케이션이 사용할 콘텐츠를 제공합니다. 콘텐츠의 미디어 유형은 *ContentType* 속성에서 제공되고 그 길이는 *ContentLength* 속성에서 제공됩니다. 데이터 압축과 같이 메시지 콘텐츠가 인코딩된 경우에는 *ContentEncoding* 속성에 정보가 들어 있습니다. 콘텐츠를 만든 애플리케이션의 이름과 버전 번호는 *ContentVersion* 속성에 의해 지정됩니다. 또한 *Title* 속성이 콘텐츠에 대한 정보를 제공할 수 있습니다.

HTTP 요청 메시지의 내용

헤더 필드 이외에 일부 요청 메시지는 웹 서버 애플리케이션이 몇몇 방법으로 처리해야 할 콘텐츠 부분을 포함합니다. 예를 들어, POST 요청은 웹 서버 애플리케이션이 유지 관리하는 데이터베이스에 추가해야 할 정보를 포함할 수 있습니다.

컨텐츠의 처리되지 않은 값은 *Content* 속성에 의해 주어집니다. 앰퍼샌드(&)로 분리된 필드로 콘텐츠를 구문 분석할 수 있는 경우 *ContentFields* 속성에서 구문 분석된 버전을 사용할 수 있습니다.

HTTP 응답 메시지 만들기

들어오는 HTTP 요청 메시지에 대해 *TWebRequest* 객체를 만들 경우 웹 서버 애플리케이션은 반환 시 보내질 응답 메시지를 나타내는 해당 *TWebResponse* 객체도 함께 만듭니다. NSAPI 및 ISAPI 애플리케이션에서 응답 메시지는 *TISAPIResponse* 객체에 의해 캡슐화됩니다. 콘솔 CGI 애플리케이션은 *TCGIResponse* 객체를 사용하고 Windows CGI 애플리케이션은 *TWinCGIResponse* 객체를 사용합니다.

웹 클라이언트 요청에 대한 응답을 생성하는 액션 항목은 응답 객체의 속성을 채웁니다. 어떤 경우는 오류 코드를 반환하거나 요청을 다른 URI에 리디렉션하는 것만큼 작업이 간단할 수 있지만, 또 다른 경우 다른 소스에서 정보를 페치하여 완성된 폼으로 결합하도록 액션 항목에 요청하는 복잡한 계산을 포함할 수 있습니다. 요청 메시지가 단지 요청된 액션이 수행되었다는 사실을 알리는 것에 불과할지라도 대부분의 요청 메시지는 응답을 필요로 합니다.

응답 헤더 채우기

TWebResponse 객체의 속성 대부분은 웹 클라이언트로 돌려 보내지는 HTTP 응답 메시지의 헤더 정보를 나타냅니다. 액션 항목은 자신의 *OnAction* 이벤트 핸들러에서 이러한 속성을 설정합니다.

모든 응답 메시지가 헤더 속성 모두에 대해 값을 지정할 필요는 없습니다. 설정해야 할 속성은 요청의 특성과 응답 상태에 따라 달라집니다.

응답 상태 표시

모든 응답 메시지는 응답 상태를 나타내는 상태 코드를 포함해야 합니다. 상태 코드는 *StatusCode* 속성을 설정하여 지정할 수 있습니다. HTTP 표준은 이미 정의된 의미로 다수의 표준 상태 코드를 정의합니다. 이외에도 사용자는 사용되지 않은 가능한 값을 사용하여 고유한 상태 코드를 정의할 수 있습니다.

각 상태 코드는 다음과 같이 세 자리 숫자이며 여기서 최상위 숫자가 응답의 종류를 나타냅니다.

- 1xx: 정보(요청을 받았지만 완전히 처리하지 않았습니다.)
- 2xx: 성공(요청을 받아 이해하고 승인하였습니다.)

- 3xx: 리디렉션(요청을 완료하기 위해 클라이언트가 추가 액션을 수행해야 합니다.)
- 4xx: 클라이언트 오류(요청을 이해하거나 서비스할 수 없습니다.)
- 5xx: 서버 오류(요청은 유효하지만 서버가 요청을 처리할 수 없습니다.)

각 상태 코드에 연결된 문자열은 상태 코드의 의미를 설명합니다. 이 문자열은 *ReasonString* 속성에 의해 주어집니다. 이미 정의된 상태 코드의 경우 *ReasonString* 속성을 설정할 필요가 없습니다. 고유한 상태 코드를 정의할 경우에는 *ReasonString* 속성도 설정해야 합니다.

클라이언트 액션에 대한 요구 표시

상태 코드의 범위가 300-399 사이인 경우 웹 서버 애플리케이션이 요청을 완료할 수 있으려면 클라이언트가 추가 액션을 수행해야 합니다. 클라이언트를 다른 URI로 리디렉션해야 하거나 요청 처리를 위해 새 URI를 만들었다는 것을 나타내야 한다면 *Location* 속성을 설정합니다. 클라이언트가 암호를 제공해야 사용자가 작업을 계속 진행할 수 있는 경우에는 *WWWAuthenticate* 속성을 설정합니다.

서버 애플리케이션 설명

일부 응답 헤더 속성은 웹 서버 애플리케이션의 기능을 설명합니다. *Allow* 속성은 애플리케이션이 응답할 수 있는 메소드를 나타냅니다. *Server* 속성은 응답을 생성하는 데 사용되는 애플리케이션의 이름과 버전 번호를 제공합니다. *Cookies* 속성은 후속 요청 메시지에 포함되는 클라이언트의 서버 애플리케이션 사용에 대한 상태 정보를 유지할 수 있습니다.

컨텐츠 설명

몇 가지 속성들은 응답 콘텐츠를 설명합니다. *ContentType*은 응답의 미디어 유형을 제공하고 *ContentVersion*은 해당 미디어 유형에 대한 버전 번호이며 *ContentLength*는 응답의 길이를 제공합니다. 데이터 압축의 경우와 같이 콘텐츠가 인코딩될 경우 *ContentEncoding* 속성을 사용하여 나타냅니다. 콘텐츠를 다른 URI에서 가져온 경우 *DerivedFrom* 속성에서 이를 나타내야 합니다. 콘텐츠 값이 시간에 민감한 경우에는 해당 값이 여전히 유효한지 여부를 *LastModified* 및 *Expires* 속성이 나타냅니다. *Title* 속성은 콘텐츠에 대한 설명 정보를 제공할 수 있습니다.

응답 콘텐츠 설정

일부 요청의 경우 요청 메시지에 대한 응답이 응답의 헤더 속성에 전부 포함됩니다. 그러나 대부분의 경우 액션 항목은 응답 메시지에 일부 콘텐츠를 지정합니다. 파일에 저장된 정적 정보나 액션 항목 또는 콘텐츠 프로듀서에 의해 동적으로 만들어진 정보도 콘텐츠가 될 수 있습니다.

Content 속성이나 *ContentStream* 속성을 사용하면 응답 메시지의 콘텐츠를 설정할 수 있습니다.

Content 속성은 문자열입니다. Delphi 문자열은 텍스트 값에만 국한되는 것이 아니라 *Content* 속성 값은 HTML 명령의 문자열 또는 비트 스트림과 같은 그래픽 콘텐츠나 다른 MIME 콘텐츠 형식이 될 수 있습니다.

응답 메시지의 콘텐츠를 스트림에서 읽을 수 있는 경우 *ContentStream* 속성을 사용합니다. 예를 들어, 응답 메시지가 파일 콘텐츠를 보내야 할 경우에는 *ContentStream* 속성에 대해 *TFileStream* 객체를 사용합니다. *Content* 속성과 마찬가지로 *ContentStream* 은 HTML 명령의 문자열이나 다른 MIME 콘텐츠 형식을 제공할 수 있습니다. *ContentStream* 속성을 사용할 경우에는 웹 응답 객체가 자동으로 스트림을 해제하므로 사용자가 직접 스트림을 해제하지 마십시오.

참고 *ContentStream* 속성 값이 **nil**이 아닌 경우 *Content* 속성은 무시됩니다.

응답 보내기

요청 메시지에 대한 응답에서 더 이상 수행할 작업이 없다는 것이 확실하면 *OnAction* 이벤트 핸들러에서 응답을 직접 보낼 수 있습니다. 응답 객체는 응답을 보내기 위한 두 개의 메소드, 즉 *SendResponse* 및 *SendRedirect*를 제공합니다. *TWebResponse* 객체의 모든 헤더 속성과 지정된 콘텐츠를 사용하는 응답을 보내려면 *SendResponse*를 호출합니다. 단지 웹 클라이언트를 다른 URI로 리디렉션해야 할 경우에는 *SendRedirect* 메소드가 더 효율적입니다.

응답을 보내는 이벤트 핸들러가 없을 경우 웹 애플리케이션 객체는 디스패처가 종료된 후 응답을 보냅니다. 그러나 응답을 처리했다는 사실을 나타내는 액션 항목이 없으면 웹 애플리케이션은 응답을 보내지 않고 웹 클라이언트와의 연결을 끊습니다.

응답 메시지 콘텐츠 생성

Delphi에서는 액션 항목이 HTTP 응답 메시지의 콘텐츠를 만들 수 있도록 도와 줄 여러 가지 객체를 제공합니다. 이 객체들을 사용하면 파일에 저장되거나 웹 클라이언트에게 직접 돌려 보내지는 HTML 명령의 문자열을 생성할 수 있습니다. *TCustomContentProducer* 또는 그 자손 중 하나에서 콘텐츠 프로듀서를 파생시켜 자신만의 고유한 콘텐츠 프로듀서를 작성할 수 있습니다.

*TCustomContentProducer*는 MIME 형식을 만들기 위한 일반 인터페이스를 HTTP 응답 메시지로 제공하며 자손으로는 페이지 프로듀서와 테이블 프로듀서를 포함합니다.

- 페이지 프로듀서는 사용자 지정 HTML 코드로 교체할 특수 태그를 HTML 문서에서 스캔합니다. 페이지 프로듀서에 대한 자세한 내용은 다음 단원에서 설명합니다.
- 테이블 프로듀서는 데이터셋의 정보에 기초하여 HTML 명령을 만듭니다. 자세한 내용은 28-17 페이지의 "응답에서 데이터베이스 정보 사용"에 설명되어 있습니다.

페이지 프로듀서 컴포넌트 사용

페이지 프로듀서(*TPageProducer*와 그 자손)는 HTML 템플릿을 가져온 다음 특수 HTML 투명 태그를 사용자 지정 HTML 코드로 교체하여 템플릿을 변환합니다. 사용자는 HTTP 요청 메시지에 대한 응답을 생성해야 할 경우 페이지 프로듀서가 채우는 표준 응답 템플릿 집합을 저장할 수 있습니다. 또한 페이지 프로듀서를 서로 연결하면 HTML 투명 태그를 계속 구체화하여 HTML 문서를 반복적으로 작성할 수 있습니다.

HTML 템플릿

HTML 템플릿은 연속적인 HTML 명령과 HTML 투명 태그들입니다. HTML 투명 태그는 다음과 같은 형태를 갖습니다.

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

꺼쇠 괄호(< 및 >)는 태그의 전체 유효 범위(scope)를 정의합니다. 열린 꺼쇠 괄호(<) 바로 뒤에는 공백 없이 파운드 기호(#)가 오며 꺼쇠 괄호와 구분됩니다. 파운드 기호는 페이지 프로듀서에 대한 문자열을 HTML 투명 태그로 식별합니다. 파운드 기호 바로 뒤에는 공백 없이 태그 이름이 오며 파운드 기호와 구분됩니다. 태그 이름은 유효한 식별자가 될 수 있고 태그가 나타내는 변환 유형을 식별합니다.

태그 이름 다음에 오는 HTML 투명 태그는 수행할 변환에 대한 세부 정보를 지정하는 매개변수를 포함할 수 있습니다. 각 매개변수는 *ParamName=Value*의 형태를 가지며 매개변수 이름, 등호 기호(=) 및 값 사이에는 공백이 오지 않습니다. 각 매개변수들은 공백으로 구분합니다.

꺼쇠 괄호(< 및 >)는 #TagName 구조를 인식하지 않는 HTML 브라우저가 태그를 알 수 있게 해줍니다.

자신의 고유한 HTML 투명 태그를 만들어 페이지 프로듀서가 처리하는 모든 종류의 정보를 나타낼 수도 있지만 *TTag* 데이터 형식 값에 연결된 여러 가지 태그 이름들이 이미 정의되어 있습니다. 이와 같이 이미 정의된 태그 이름은 응답 메시지에 따라 달라질 수 있는 HTML 명령에 해당합니다. 다음 표는 이러한 태그 이름을 나열한 것입니다.

태그 이름	TTag 값	태그가 변환되어야 하는 대상
<i>Link</i>	<i>tgLink</i>	하이퍼텍스트 링크. 결과는 <A> 태그로 시작하여 태그로 끝나는 HTML입니다.
<i>Image</i>	<i>tgImage</i>	그래픽 이미지. 결과는 HTML 태그입니다.
<i>Table</i>	<i>tgTable</i>	HTML 테이블. 결과는 <TABLE> 태그로 시작하여 </TABLE> 태그로 끝나는 HTML입니다.
<i>ImageMap</i>	<i>tgImageMap</i>	연결된 핫 존(hot zone)이 있는 그래픽 이미지. 결과는 <MAP> 태그로 시작하여 </MAP> 태그로 끝나는 HTML입니다.
<i>Object</i>	<i>tgObject</i>	포함된 ActiveX 객체. 결과는 <OBJECT> 태그로 시작하여 </OBJECT> 태그로 끝나는 HTML입니다.
<i>Embed</i>	<i>tgEmbed</i>	Netscape 호환 추가 DLL. 결과는 <EMBED> 태그로 시작하여 </EMBED> 태그로 끝나는 HTML입니다.

다른 모든 태그 이름은 *tgCustom*에 연결됩니다. 페이지 프로듀서는 이미 정의된 태그 이름의 기본 제공 처리를 제공하지 않습니다. 이미 정의된 태그 이름은 애플리케이션이 변환 프로세스를 더 일반적인 작업으로 구성할 수 있도록 하기 위해 제공됩니다.

참고 이미 정의된 태그 이름은 대소문자를 구분하지 않습니다.

HTML 템플릿 지정

페이지 프로듀서는 HTML 템플릿을 지정하는 방법에 대해 많은 선택을 제공합니다. *HTMLFile* 속성을 HTML 템플릿을 포함하는 파일의 이름으로 설정할 수 있고, *HTMLDoc* 속성을 HTML 템플릿을 포함하는 *TStrings* 객체로 설정할 수 있습니다. *HTMLFile* 속성 또는 *HTMLDoc* 속성을 사용하여 템플릿을 지정하는 경우 *Content* 메소드를 호출하여 변환된 HTML 명령을 생성할 수 있습니다.

이 외에도 *ContentFromString* 메소드를 호출하여 매개변수로 전달되는 단일 문자열인 HTML 템플릿을 직접 변환할 수 있습니다. 또한 *ContentFromStream* 메소드를 호출하여 스트림에서 HTML 템플릿을 읽을 수 있습니다. 따라서 예를 들어 모든 HTML 템플릿을 데이터베이스의 메모 필드에 저장하고 *ContentFromStream* 메소드를 사용하여 변환된 HTML 명령을 얻을 수 있으며, 이 때 *TBlobStream* 객체에서 템플릿을 직접 읽을 수 있습니다.

HTML 투명 태그 변환

페이지 프로듀서는 사용자가 *Content* 메소드 중 하나를 호출할 때 HTML 템플릿을 변환합니다. *Content* 메소드는 HTML 투명 태그를 만나면 *OnHTMLTag* 이벤트를 트리거합니다. 확인되는 태그 유형을 결정하고 이를 사용자 지정 콘텐츠로 바꾸기 위해 이벤트 핸들러를 작성해야 합니다.

페이지 프로듀서에 *OnHTMLTag* 이벤트 핸들러를 만들지 않은 경우 HTML 투명 태그는 빈 문자열로 바뀝니다.

액션 항목에서 페이지 프로듀서 사용

페이지 프로듀서 컴포넌트는 일반적으로 HTML 템플릿이 포함된 파일을 지정하기 위해 *HTMLFile* 속성을 사용합니다. *OnAction* 이벤트 핸들러는 다음과 같이 *Content* 메소드를 호출하여 템플릿을 최종적인 HTML로 변환합니다.

```
procedure WebModule1.MyActionEventHandler(Sender:TObject; Request:TWebRequest;
    Response:TWebResponse; var Handled:Boolean);
begin
    PageProducer1.HTMLFile := 'Greeting.html';
    Response.Content := PageProducer1.Content;
end;
```

Greeting.html은 다음과 같은 HTML 템플릿이 포함된 파일입니다.

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello <#UserName>!Welcome to our web site.
</BODY>
</HTML>
```

OnHTMLTag 이벤트는 다음과 같은 방법으로 실행 도중에 HTML에 있는 사용자 지정 태그(<#UserName>)를 바꿉니다.

```
procedure WebModule1.PageProducer1HTMLTag(Sender :TObject;Tag:TTag;
const TagString:string; TagParams:TStrings; var ReplaceText:string);
begin
  if CompareText(TagString,'UserName') = 0 then
    ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

요청 메시지의 콘텐츠가 문자열 *Mr.Ed*일 경우 *Response.Content*의 값은 다음과 같습니다.

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!Welcome to our web site.
</BODY>
</HTML>
```

참고 이 예제는 *OnAction* 이벤트 핸들러를 사용하여 콘텐츠 프로듀서를 호출하고 응답 메시지의 콘텐츠를 할당합니다. 디자인 타임에 페이지 프로듀서의 *HTMLFile* 속성을 지정하는 경우 *OnAction* 이벤트 핸들러를 작성할 필요가 없습니다. 이 경우에는 간단히 *PageProducer1*을 액션 항목의 *Producer* 속성으로 지정하면 위의 *OnAction* 이벤트 핸들러와 동일한 효과를 거둘 수 있습니다.

페이지 프로듀서 간의 연결

OnHTMLTag 이벤트 핸들러에서 가져온 대체 텍스트는 HTTP 응답 메시지에서 사용할 최종적인 HTML이 아니어도 됩니다. 특정 페이지 프로듀서의 출력이 다음 페이지의 입력이 되는 페이지 프로듀서를 여러 개 사용할 수도 있습니다.

페이지 프로듀서를 서로 연결하는 가장 간단한 방법은 각 페이지 프로듀서를 별도의 액션 항목에 연결하는 것이며, 이 때 모든 액션 항목은 동일한 *PathInfo*와 *MethodType*을 가집니다. 첫 번째 액션 항목은 콘텐츠 프로듀서에 가져온 웹 응답 메시지를 설정하지만 이 액션 항목의 *OnAction* 이벤트 핸들러는 메시지가 처리되지 않은 것으로 간주합니다. 다음 액션 항목은 연결된 프로듀서의 *ContentFromString* 메소드를 사용하여 웹 응답 메시지 콘텐츠를 조작하는 등의 작업을 수행합니다. 첫 번째 이후의 액션 항목은 다음과 같이 *OnAction* 이벤트 핸들러를 사용합니다.

```
procedure WebModule1.Action2Action(Sender:TObject; Request:TWebRequest;
  Response:TWebResponse; var Handled:Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

예를 들어, 원하는 페이지의 월과 연도를 지정하는 요청 메시지에 대한 응답으로 달력 페이지를 반환하는 애플리케이션을 가정해 보십시오. 각 달력 페이지에는 우선 그림이 있고 이전 월과 다음 월의 이미지 사이에 월 이름과 연도가 있으며 마지막으로 실제 달력이 나옵니다. 결과 이미지는 다음과 같이 나타납니다.



이 달력의 일반 폼은 템플릿 파일에 저장되며 다음과 같이 나타납니다.

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

첫 번째 페이지 프로듀서의 *OnHTMLTag* 이벤트 핸들러는 요청 메시지에서 월과 연도를 조회합니다. 조회한 정보와 템플릿 파일을 사용하여 이벤트 핸들러는 다음을 수행합니다.

- <#MonthlyImage>를 <#Image Month=January Year=1997>로 바꿉니다.
- <#TitleLine>을 <#Calendar Month=December Year=1996 Size=Small> January 1997 <#Calendar Month=February Year=1997 Size=Small>로 바꿉니다.
- <#MainBody>를 <#Calendar Month=January Year=1997 Size=Large>로 바꿉니다.

다음 페이지 프로듀서의 *OnHTMLTag* 이벤트 핸들러는 첫 번째 페이지 프로듀서가 만든 콘텐츠를 사용하고 <#Image Month=January Year=1997> 태그를 해당 HTML 태그로 바꿉니다. 다른 페이지 프로듀서는 적절한 HTML 테이블을 사용하여 #Calendar 태그를 해결합니다.

응답에서 데이터베이스 정보 사용

HTTP 요청 메시지에 대한 응답에는 데이터베이스에서 가져온 정보를 포함할 수 있습니다. 인터넷 팔레트 페이지에 있는 특수한 콘텐츠 프로듀서는 HTML을 생성하여 데이터베이스의 레코드를 HTML 테이블에 나타낼 수 있습니다.

또 다른 방법으로 컴포넌트 팔레트의 InternetExpress 페이지 상의 특별한 컴포넌트로 다계층 데이터베이스 애플리케이션의 일부분인 웹 서버를 구축할 수 있습니다. 자세한 내용은 25-34 페이지의 "InternetExpress를 사용하여 웹 애플리케이션 구축"을 참조하십시오.

웹 모듈에 세션 추가

콘솔 CGI 애플리케이션 및 Win-CGI 애플리케이션은 HTTP 요청 메시지에 응답하여 시작됩니다. 이러한 유형의 애플리케이션에서 데이터베이스 작업을 수행할 경우 각각의 요청 메시지가 고유한 애플리케이션 인스턴스를 갖기 때문에 사용자는 기본 세션을 사용하여 데이터베이스 연결을 관리할 수 있습니다. 애플리케이션의 각 인스턴스는 고유한 차이로 기본 세션을 가집니다.

그러나 ISAPI 애플리케이션이나 NSAPI 애플리케이션을 작성할 때 각각의 요청 메시지는 하나의 애플리케이션 인스턴스의 별도 스레드에서 처리됩니다. 다른 스레드의 데이터베이스 연결을 서로 방해하지 않도록 하기 위해서는 각각의 스레드에 고유한 세션을 주어야 합니다.

ISAPI 또는 NSAPI 애플리케이션의 각 요청 메시지는 새로운 스레드를 생성합니다. 새로운 스레드의 웹 모듈은 런타임 시 동적으로 생성됩니다. *TSession* 객체를 웹 모듈에 더해 웹 모듈을 포함하고 있는 스레드에 대한 데이터베이스 연결을 처리합니다.

웹 모듈의 별도 인스턴스가 런타임 시 각 스레드에 대해 생성됩니다. 각 모듈들은 세션 객체를 포함하고 있습니다. 각각의 세션이 별도의 이름을 갖고 있어야 별도의 요청 메시지를 처리하는 스레드가 서로의 데이터베이스 연결을 방해하지 않습니다. 각 모듈의 세션 객체가 고유한 이름을 동적으로 생성하도록 세션 객체의 *AutoSessionName* 속성을 설정해야 합니다. 각 세션 객체는 자신의 고유한 이름을 동적으로 생성하며 그 고유한 이름을 참고하도록 모듈의 모든 데이터셋의 *SessionName* 속성을 설정합니다. 이것은 각 요청 스레드가 다른 요청 메시지를 방해하지 않고 데이터베이스와 상호 작용을 할 수 있게 합니다. 세션에 대한 자세한 내용은 20-17 페이지의 "데이터베이스 세션 관리"를 참조하십시오.

HTML로 데이터베이스 정보 표시

인터넷 팔레트 페이지의 특수한 콘텐츠 프로듀서 컴포넌트는 데이터셋의 레코드에 기초하여 HTML 명령을 제공합니다. 다음과 같은 두 가지 유형의 data-aware 콘텐츠 프로듀서가 있습니다.

- 데이터셋 필드 서식을 HTML 문서의 텍스트로 설정하는 데이터셋 페이지 프로듀서
- 데이터셋 레코드 서식을 HTML 테이블로 설정하는 테이블 프로듀서

데이터셋 페이지 프로듀서 사용

데이터셋 페이지 프로듀서는 다른 페이지 프로듀서 컴포넌트와 같은 방식으로 사용됩니다. 즉, HTML 투명 태그를 포함하는 템플릿을 최종 HTML 형태로 변환합니다. 그러나 데이터셋 페이지 프로듀서는 데이터셋에 있는 필드 이름과 일치하는 태그 이름을 가진 태그를 해당 필드의 현재 값으로 변환하는 특수 기능을 포함합니다. 페이지 프로듀서의 일반적인 사용에 관한 자세한 내용은 28-14 페이지의 "페이지 프로듀서 컴포넌트 사용"을 참조하십시오.

데이터셋 페이지 프로듀서를 사용하려면 *TDataSetPageProducer* 컴포넌트를 웹 모듈에 추가하고 *DataSet* 속성을 필드 값이 HTML 콘텐츠로 표시되어야 하는 데이터셋으로 설정합니다. 데이터셋 페이지 프로듀서의 출력을 설명하는 HTML 템플릿을 만듭니다. 그리고 표시하려는 모든 필드 값에 대해 다음과 같은 형태의 태그를 HTML 템플릿에 포함시킵니다.

```
<#FieldName>
```

여기서 *FieldName*은 값을 표시해야 하는 데이터셋의 필드 이름을 지정합니다.

애플리케이션이 *Content*, *ContentFromString* 또는 *ContentFromStream* 메소드를 호출하면 데이터셋 페이지 프로듀서는 필드를 나타내는 태그를 현재 필드 값으로 대체합니다.

테이블 프로듀서 사용

인터넷 팔레트 페이지에는 HTML 테이블을 만들어 데이터셋 레코드를 나타내는 다음과 같은 두 개의 컴포넌트가 들어 있습니다.

- 데이터셋 필드 서식을 HTML 문서의 텍스트로 설정하는 데이터셋 테이블 프로듀서
- 요청 메시지가 제공한 매개변수를 설정한 후 쿼리를 실행하고 결과 데이터셋의 서식을 HTML 테이블로 설정하는 쿼리 테이블 프로듀서

두 테이블 프로듀서 중 하나를 사용하면 테이블 색, 테두리, 구분자 두께 등의 속성을 지정하여 결과 HTML 테이블의 모양을 사용자 지정할 수 있습니다. 디자인 타임에 테이블 프로듀서의 속성을 설정하려면 테이블 프로듀서 컴포넌트를 더블 클릭하여 Response Editor 대화 상자를 표시합니다.

테이블 속성 지정

테이블 프로듀서는 *THTMLTableAttributes* 객체를 사용하여 데이터셋의 레코드를 표시하는 HTML 테이블의 시각적 모양을 설명합니다. *THTMLTableAttributes* 객체에는 HTML 문서에서의 테이블 너비와 간격, 배경 색, 테두리 두께, 셀 안쪽 여백, 셀 간격 등에 대한 속성이 포함되어 있습니다. 이러한 모든 속성은 테이블 프로듀서가 만든 HTML <TABLE> 태그에서 옵션으로 전환됩니다.

디자인 타임에 Object Inspector로 이러한 속성을 지정합니다. Object Inspector에서 테이블 프로듀서 객체를 선택하고 *TableAttributes* 속성을 확장하여 *THTMLTableAttributes* 객체의 표시 속성에 액세스합니다.

또한 이러한 속성을 런타임에 프로그램에서 지정할 수도 있습니다.

행 속성 지정

데이터를 표시하는 테이블의 행에서 테이블 속성처럼 셀의 배경 색 및 정렬을 지정할 수 있습니다. 이 때 사용되는 *RowAttributes* 속성은 *THTMLTableRowAttributes* 객체입니다.

디자인 타임에 Object Inspector에서 *RowAttributes* 속성을 확장하여 이러한 속성을 지정합니다. 또한 이러한 속성을 런타임에 프로그램에서 지정할 수도 있습니다.

이 밖에도 *MaxRows* 속성을 설정하면 HTML 테이블에 표시되는 행 수를 조정할 수 있습니다.

열 지정

디자인 타임에 테이블의 데이터셋을 알고 있다면 Columns 에디터를 사용하여 열의 필드 바인딩을 사용자 지정하고 속성을 표시할 수 있습니다. 이렇게 하려면 테이블 프로듀서 컴포넌트를 선택하고 마우스 오른쪽 버튼을 클릭한 다음 컨텍스트 메뉴에서 Columns 에디터를 선택합니다. 이 에디터는 테이블 열을 추가, 삭제 또는 재정렬할 수 있게 해줍니다. Columns 에디터에서 열을 선택한 후 Object Inspector에서 개별 열의 필드 바인딩을 설정하고 속성을 표시할 수 있습니다.

HTTP 요청 메시지에서 데이터셋의 이름을 가져오는 중이면 디자인 타임에 Columns 에디터에서 필드를 바인딩할 수 없습니다. 그러나 해당 *THTMLTableColumn* 객체를 설정하고 *Columns* 속성의 메소드를 통해 테이블에 열을 추가하여 런타임에 프로그램에서 열을 사용자 지정할 수 있습니다. *Columns* 속성을 설정하지 않을 경우, 테이블 프로듀서는 데이터셋 필드와 일치하는 열의 기본 집합을 만들고 특별한 표시 특성들을 지정하지 않습니다.

HTML 문서에 테이블 포함시키기

테이블 프로듀서의 *Header* 및 *Footer* 속성을 사용하면 더 큰 문서의 데이터셋을 나타내는 HTML 테이블을 포함할 수 있습니다. *Header*는 테이블 앞에 오는 모든 내용을 지정하기 위하여, *Footer*는 테이블 뒤에 오는 모든 내용을 지정하기 위하여 사용합니다.

페이지 프로듀서와 같은 다른 콘텐츠 프로듀서를 사용하여 *Header* 및 *Footer* 속성에 대한 값을 만들 수도 있습니다.

더 큰 문서의 테이블을 포함할 경우 테이블에 캡션을 추가할 수 있습니다. 테이블에 캡션을 제공하려면 *Caption* 및 *CaptionAlignment* 속성을 사용합니다.

데이터셋 테이블 프로듀서 설정

*TDataSetTableProducer*는 데이터셋의 HTML 테이블을 만드는 테이블 프로듀서입니다. 표시할 레코드를 포함하는 데이터셋을 지정하려면 *TDataSetTableProducer*의 *DataSet* 속성을 설정합니다. 기본 데이터베이스 애플리케이션에서의 대부분의 data-aware 객체와 마찬가지로 *DataSource* 속성은 설정하지 않습니다. 이 속성을 설정하지 않는 이유는 *TDataSetTableProducer*가 고유한 데이터 소스를 내부적으로 생성하기 때문입니다.

웹 애플리케이션이 항상 같은 데이터셋에서 가져온 레코드를 표시하는 경우 디자인 타임에 *DataSet*의 값을 설정할 수 있습니다. 데이터셋이 HTTP 요청 메시지에 있는 정보에 기초한다면 런타임에 *DataSet* 속성을 설정해야 합니다.

쿼리 테이블 프로듀서 설정

HTML 테이블을 만들어 HTTP 요청 메시지에서 매개변수를 가져온 쿼리 결과를 표시할 수 있습니다. 이렇게 하려면 쿼리 매개변수를 *TQueryTableProducer* 컴포넌트의 *Query* 속성으로 사용하는 *TQuery* 객체를 지정합니다.

요청 메시지가 GET 요청인 경우 HTTP 요청 메시지의 대상으로 제공된 URL의 *Query* 필드에서 쿼리의 매개변수를 가져옵니다. 요청 메시지가 POST 요청인 경우에는 요청 메시지의 콘텐츠에서 쿼리 매개변수를 가져옵니다.

*TQueryTableProducer*의 *Content* 메소드는 호출 시 요청 객체에서 찾은 매개변수를 사용하여 쿼리를 실행합니다. HTML 테이블 서식을 설정하여 결과 데이터셋의 레코드를 표시합니다.

다른 테이블 프로듀서와 마찬가지로 사용자는 HTML 테이블의 표시 속성이나 열 바인딩을 사용자 지정하거나 더 큰 HTML 문서에 테이블을 포함할 수 있습니다.

29

WebSnap 사용

WebSnap은 복잡한 데이터 방식 웹 페이지를 포함하는 웹 애플리케이션을 보다 쉽게 만들 수 있게 하는 컴포넌트, 마법사 및 뷰로 WebBroker를 확장시킵니다. 여러 모듈과 서버측 스크립트에 대한 WebSnap의 지원으로 팀 개발이 쉬워집니다.

디스패처 컴포넌트는 페이지 콘텐츠 요청, HTML 폼 전송, 동영상 요청을 자동으로 처리합니다. 어댑터라고 하는 새로운 컴포넌트는 애플리케이션의 비즈니스 룰에 스크립트 가능한 인터페이스를 정의하는 수단을 제공합니다. 예를 들어, *TDataSetAdapter* 객체를 사용하면 데이터셋 컴포넌트를 스크립트 가능하게 만들 수 있습니다. 새로운 프로듀서 컴포넌트를 사용하면 복잡한 데이터 방식 폼과 테이블을 쉽게 만들 수 있고 XSL을 사용하여 페이지를 생성할 수 있습니다. 세션 컴포넌트를 사용하면 최종 사용자를 추적할 수 있습니다. 사용자 목록 컴포넌트를 사용하면 사용자 이름, 암호, 액세스 권한에 액세스할 수 있습니다.

웹 애플리케이션 마법사를 사용하면 필요한 컴포넌트를 가지고 사용자 지정된 애플리케이션을 쉽게 작성할 수 있습니다. 웹 페이지 모듈 마법사를 사용하면 애플리케이션의 새로운 페이지를 정의하는 모듈을 작성할 수 있습니다. 또는 웹 데이터 모듈 마법사를 사용하여 웹 애플리케이션에서 공유하는 컴포넌트들의 컨테이너를 만들 수 있습니다.

페이지 모듈 뷰를 통해 애플리케이션을 실행하지 않고도 서버측 스크립트의 결과를 볼 수 있습니다. Preview 탭을 클릭하면 내장 브라우저에 페이지가 표시됩니다. HTML Result 뷰는 생성된 HTML을 보여 줍니다. XSL 트리 뷰와 XML 트리 뷰를 통해 XML과 XSL을 더 쉽게 작업할 수 있습니다.

개발자 팀을 지원하도록 WebSnap의 다중 모듈 지원을 사용하여 애플리케이션을 개별적으로 작업할 수 있는 유닛으로 분할할 수 있습니다. 새로운 페이지 모듈을 만들 때 페이지 모듈 마법사로 외부 템플릿 파일을 작성할 수 있습니다. IDE 외부에서 템플릿 파일을 편집하여 애플리케이션을 재컴파일하지 않고 테스트할 수 있습니다.

이 장의 다음 단원에서는 WebSnap 컴포넌트를 사용하여 웹 서버 애플리케이션을 작성하는 방법에 대해서 설명합니다.

WebSnap을 사용한 웹 서버 애플리케이션 생성

WebSnap 아키텍처를 사용하여 새로운 웹 서버 애플리케이션을 만드는 방법은 다음과 같습니다.

- 1 File|New|Other를 선택합니다.
- 2 New Items 대화 상자에서 WebSnap 탭을 선택한 후 WebSnap Application을 선택합니다.
- 3 다음과 같은 타입의 정보를 요구하는 대화 상자가 나타납니다.
 - 서버 유형
 - 웹 애플리케이션 모듈 유형
 - 웹 애플리케이션 모듈 옵션
 - 애플리케이션 컴포넌트

서버 유형

애플리케이션의 웹 서버 유형에 따라 다음 웹 서버 애플리케이션 유형 중 하나를 선택합니다.

- ISAPI와 NSAPI. 이 애플리케이션 유형을 선택하면 프로젝트는 웹 서버에서 요구하는 export된 메소드를 가진 DLL로 설정됩니다. 프로젝트 파일에 라이브러리 헤더를 추가하고 사용 목록과 프로젝트 파일의 exports 절에 필수 항목을 추가합니다.
- Apache. 이 애플리케이션 유형을 선택하면 프로젝트는 Apache 웹 서버에서 요구하는 export된 메소드를 가진 DLL로 설정됩니다. 프로젝트 파일에 라이브러리 헤더를 추가하고 사용 목록과 프로젝트 파일의 exports 절에 필수 항목을 추가합니다.
- CGI 독립형. 이 애플리케이션 유형을 선택하면 프로젝트가 콘솔 애플리케이션으로 설정되고 프로젝트 파일의 uses 절에 필수 항목이 추가됩니다.
- Win-CGI 독립형. 이 애플리케이션 유형을 선택하면 프로젝트가 Windows 애플리케이션으로 설정되고 프로젝트 파일의 uses 절에 필수 항목이 추가됩니다.
- 웹 애플리케이션 디버거 실행 파일. 이 애플리케이션 유형을 선택하면 웹 서버 애플리케이션을 개발하고 테스트하기 위한 환경이 설정됩니다. 이 유형의 애플리케이션은 배포용이 아닙니다.

애플리케이션이 사용할 웹 서버 유형과 통신하는 웹 서버 애플리케이션의 유형을 선택합니다.

웹 애플리케이션 모듈 유형

웹 애플리케이션 모듈은 웹 애플리케이션의 비즈니스 룰과 런타임 컴포넌트를 중앙에서 제어하도록 해줍니다. 웹 애플리케이션 모듈에는 다음 두 가지 유형이 있습니다.

- 페이지 모듈. 이 모듈 유형을 선택하면 콘텐츠 페이지가 만들어집니다. 페이지 모듈에는 페이지의 콘텐츠를 생성하는 페이지 프로듀서가 있습니다. 페이지 프로듀서는 HTTP 요청 경로 정보가 페이지 이름과 같을 때 연결된 페이지를 표시합니다. 페이지는 경로 정보가 비어 있을 때 기본 페이지로 작용할 수 있습니다.
- 데이터 모듈. 이 모듈 유형을 선택하면 콘텐츠 페이지가 만들어지지 않습니다. 이 모듈은 다른 모듈들이 공유하는 컴포넌트, 예를 들면 두 웹 페이지 모듈에서 사용하는 데이터베이스 컴포넌트의 컨테이너로 사용됩니다.

웹 애플리케이션 모듈 옵션

선택한 애플리케이션 모듈 유형이 페이지 모듈이면 대화 상자의 Page Name 필드에 이름을 입력하여 이름을 페이지와 연결할 수 있습니다. 런타임 시 이 모듈의 인스턴스는 캐시에 보관되거나 요청이 서비스될 때 메모리에서 제거될 수 있습니다. Caching 필드에서 옵션 하나를 선택합니다. Page Options 버튼을 통해 더 많은 페이지 모듈 옵션을 선택할 수 있습니다. 다음 범주를 설정할 수 있습니다.

- Producer: 페이지의 프로듀서 타입은 *AdapterPageProducer*, *DataSetPageProducer*, *InetXPageProducer*, *PageProducer*, *XSLPageProducer* 중 하나로 설정할 수 있습니다. 선택한 페이지 프로듀서가 스크립팅을 지원하면 Script Engine 드롭다운 목록을 사용하여 페이지를 스크립트할 때 사용되는 언어를 선택하십시오.

참고 *AdapterPageProducer*는 JScript만 지원합니다.

- HTML: 선택한 프로듀서가 HTML 템플릿을 사용할 경우 이 그룹이 보입니다.
- XSL: 선택한 프로듀서가 *TXSLPageProducer*와 같은 XSL 템플릿을 사용하면 이 그룹이 보입니다.
- New File: 유닛의 일부로 작성되고 관리되는 템플릿 파일을 원하면 New File을 선택합니다. 프로젝트 관리자에 관리되는 템플릿 파일이 나타나고 이는 유닛 소스 파일과 동일한 파일 이름 및 위치를 갖습니다. 프로듀서 컴포넌트의 속성(일반적으로 *HTMLDoc* 또는 *HTMLFile* 속성)을 사용하려면 New File을 선택 해제합니다.
- Template: New File이 선택되었으면 Template 드롭다운 목록에서 템플릿 파일의 기본 콘텐츠를 선택합니다. "Default" 템플릿은 애플리케이션의 제목, 페이지의 제목, 게시된 페이지에 대한 하이퍼링크를 표시합니다.
- Page: 페이지 모듈의 페이지 이름과 제목을 입력합니다. 페이지 이름은 HTTP 요청 또는 애플리케이션의 로직 내에 있는 페이지를 참조할 때 사용되는 반면 제목은 페이지가 브라우저에 표시될 때 최종 사용자가 보게 되는 이름입니다.
- Published: 페이지 이름이 요청 메시지의 경로 정보와 일치하는 HTTP 요청에 페이지가 자동으로 응답하게 하려면 Published를 선택합니다.

- Login Required: 사용자가 로그인한 뒤에 페이지를 액세스할 수 있게 하려면 Login Required를 선택합니다.

애플리케이션 컴포넌트

애플리케이션 컴포넌트는 웹 애플리케이션의 기능을 제공합니다. 예를 들어, 어댑터 디스패처 컴포넌트를 포함하면 HTML 폼 전송과 동적으로 생성된 이미지의 반환이 자동으로 처리됩니다. 페이지 디스패처를 포함하면 HTTP 요청 경로 정보에 페이지의 이름이 포함될 때 페이지의 콘텐츠가 자동으로 표시됩니다.

Components 버튼을 선택하면 하나 이상의 애플리케이션 컴포넌트를 선택할 수 있는 대화 상자가 표시됩니다.

- Application Adapter: 제목과 같은 애플리케이션에 관한 정보를 포함하고 있습니다. 기본 타입은 *TApplicationAdapter*입니다.
- End User Adapter: 사용자의 이름, 액세스 권한 및 로그인 여부와 같은 사용자에 관한 정보를 포함하고 있습니다. 기본 타입은 *TEndUserAdapter*입니다. *TEndUserSessionAdapter*를 선택할 수도 있습니다.
- Page Dispatcher: HTTP 요청의 경로 정보를 검토하고 페이지의 콘텐츠를 반환하는 해당 페이지 모듈을 호출합니다. 기본 타입은 *TPageDispatcher*입니다.
- Adapter Dispatcher: 어댑터 액션과 필드 컴포넌트를 호출하여 HTML 폼 전송 및 동적 이미지 요청을 자동으로 처리합니다. 기본 타입은 *TAdapterDispatcher*입니다.
- Dispatch Actions: 경로 정보와 메소드 타입에 기반하여 요청을 처리하는 액션 항목의 모음을 정의할 수 있게 합니다. 액션 항목은 사용자가 정의한 이벤트를 호출하거나 페이지 프로듀서 컴포넌트의 콘텐츠를 요청합니다. 기본 타입은 *TWebDispatcher*입니다.
- Locate File Service: 웹 애플리케이션이 실행 중일 때 템플릿 파일 및 스크립트 포함 파일의 로드를 제어할 수 있습니다. 기본 타입은 *TLocateFileService*입니다.
- Sessions Service: 짧은 시간 주기 동안 필요한 최종 사용자 정보를 저장하기 위해 사용됩니다. 예를 들면, 세션을 사용하여 로그인한 사용자를 추적하고 무동작(inactivity) 기간이 지나면 자동으로 사용자를 로그아웃할 수 있습니다. 기본 타입은 *TSessionService*입니다.
- User List Service: 승인된 사용자 및 그들의 암호와 액세스 권한을 추적합니다. 기본 타입은 *TWebUserList*입니다.

위의 각 컴포넌트에 대한 컴포넌트 타입은 Delphi 소프트웨어 제품과 함께 제공되는 기본 타입입니다. 사용자는 사용자 고유의 컴포넌트 타입을 만들거나 협력업체 컴포넌트 타입을 사용할 수 있습니다.

웹 모듈

웹 모듈 유형에는 다음 네 가지가 있습니다.

- *TWebAppPageModule*
- *TWebAppDataModule*
- *TWebPageModule*
- *TWebDataModule*

웹 애플리케이션 모듈 (*TWebAppPageModule* 또는 *TWebAppDataModule*)은 요청 디스패치, 세션 관리, 사용자 목록 유지 관리와 같은 전체적인 애플리케이션에 대한 기능을 수행하는 애플리케이션 컴포넌트에 대한 컨테이너입니다. 사용자의 프로젝트는 이러한 유형의 애플리케이션 모듈 중 하나만 포함할 수 있습니다.

웹 페이지 모듈 (*TWebPageModule*)은 페이지에 콘텐츠를 제공하고, 웹 데이터 모듈 (*TWebDataModule*)은 사용자의 애플리케이션에서 공유되는 컴포넌트에 대한 컨테이너로 작용합니다. 옵션으로서 웹 애플리케이션 모듈에 하나 이상의 웹 페이지와 웹 데이터 모듈을 포함할 수 있습니다.

웹 데이터 모듈

표준 데이터 모듈처럼 웹 데이터 모듈은 팔레트의 컴포넌트에 대한 컨테이너입니다. 데이터 모듈은 컴포넌트 추가, 제거, 선택에 대한 디자인 외관을 제공합니다. 애플리케이션이 실행 중인 경우 데이터 모듈은 창을 만들지 않습니다.

웹 데이터 모듈은 유닛의 구조 및 웹 데이터 모듈이 구현하는 인터페이스에 있는 표준 데이터 모듈과 다릅니다.

웹 데이터 모듈을 애플리케이션에서 공유되는 컴포넌트에 대한 컨테이너로 사용합니다. 예를 들면, 데이터셋 컴포넌트를 데이터 모듈에 넣고 다음 모듈에서 데이터셋에 액세스할 수 있습니다.

- 그리드를 표시하는 페이지 모듈
- 입력 폼을 표시하는 페이지 모듈

웹 데이터 모듈 유닛의 구조

표준 데이터 모듈에는 데이터 모듈 객체에 액세스할 때 사용하는 폼 변수라고 하는 변수가 있습니다. 웹 데이터 모듈은 이 변수를 함수로 대체합니다. 용도는 동일합니다. 하지만 WebSnap 애플리케이션은 멀티스레드될 수 있고 여러 요청을 동시에 서비스하는 특정 모듈의 여러 인스턴스를 가질 수 있기 때문에 이 함수는 정확한 인스턴스를 반환하도록 구현됩니다.

유닛은 또한 팩토리를 등록합니다. 팩토리에서는 WebSnap 애플리케이션이 모듈을 관리하는 방법을 지정합니다. 예를 들면 플래그는 모듈을 캐시할 것인지 여부, 다른 요청을 위한 모듈 재사용 여부, 요청이 서비스된 후 모듈을 소멸시킬 것인지 여부를 나타냅니다.

웹 데이터 모듈에 의해 구현되는 인터페이스

웹 데이터 모듈은 다음과 같은 인터페이스를 구현합니다.

INotifyWebActivate: 이 인터페이스는 모듈이 웹 요청을 서비스하기 위해 활성화될 때와 웹 요청이 서비스되고 나서 모듈이 비활성화되기 전에 호출됩니다.

IWebVariablesContainer: 이 인터페이스는 스크립트 실행 중 변수 참조를 해석하기 위해 호출됩니다. 어댑터 디스패처 또한 이 인터페이스를 호출하여 HTTP 요청에서 참조되는 어댑터 액션과 필드를 찾아냅니다.

IGetScriptObject: 이 인터페이스는 모듈의 IDispatch 구현을 검색하기 위해 호출됩니다. 반환된 객체는 모듈과 활성 스크립팅 엔진 간의 인터페이스입니다.

IIteratorObjectSupport: 이 인터페이스는 활성 스크립팅에 의해 액세스될 수 있는 모듈 내에 있는 모든 객체를 반복하기 위해 사용됩니다.

웹 페이지 모듈

페이지 모듈에는 연결된 페이지 프로듀서 컴포넌트가 있습니다. 요청이 받아들여지면 페이지 디스패처는 요청을 분석하고 해당 페이지 모듈을 호출하여 요청을 처리하고 페이지의 콘텐츠를 반환합니다.

웹 데이터 모듈과 마찬가지로 웹 페이지 모듈은 컴포넌트에 대한 컨테이너입니다. 웹 데이터 모듈과 웹 페이지 모듈 간의 차이점은 웹 페이지 모듈은 웹 페이지를 만들기 위해 사용된다는 점입니다.

페이지 프로듀서 컴포넌트

웹 페이지 모듈은 페이지의 콘텐츠를 생성하는 데 쓰이는 페이지 프로듀서 컴포넌트를 식별하는 속성을 갖고 있습니다. WebSnap 마법사는 웹 페이지 모듈을 만들 때 자동으로 프로듀서를 추가합니다. 나중에 WebSnap 팔레트에서 다른 프로듀서를 가져다 놓아 페이지 프로듀서 컴포넌트를 바꿀 수 있습니다. 하지만 페이지 모듈이 템플릿 파일을 가지고 있는 경우 이 파일의 콘텐츠는 프로듀서 컴포넌트에 적합합니다.

페이지 이름

웹 페이지 모듈은 HTTP 요청 또는 애플리케이션의 로직 내에서 페이지를 참조하기 위해 사용할 수 있는 페이지 이름을 가지고 있습니다. 웹 페이지 모듈의 유닛에 있는 팩토리는 웹 페이지 모듈의 페이지 이름을 지정합니다.

프로듀서 템플릿

대부분의 페이지 프로듀서는 템플릿을 사용합니다. HTML 템플릿에는 대개 투명 태그 또는 서버측 스크립트와 함께 혼합된 몇몇 정적인 HTML이 들어 있습니다. 페이지 프로듀서가 콘텐츠를 만들 때 투명 태그를 해당 값으로 바꾸고 서버측 스크립트를 실행하여 클라이언트 브라우저에 의해 표시되는 HTML을 만들어냅니다. XSLPageProducer는 예외입니다. HTML이 아닌 XSL이 들어 있는 XSL 템플릿을 사용합니다. XSL 템플릿은 투명 태그나 서버측 스크립트를 지원하지 않습니다.

웹 페이지 모듈은 유닛의 일부로 관리되는 연결된 템플릿 파일을 가질 수도 있습니다. 관리되는 템플릿 파일은 프로젝트 관리자에 나타나고 유닛 서비스 파일과 동일한 기준 파일 이름 및 위치를 갖습니다. 웹 페이지 모듈에서 연결된 템플릿 파일을 갖고 있지 않은 경우, 페이지 프로듀서 컴포넌트의 속성에서 템플릿을 지정합니다.

웹 페이지 모듈이 구현하는 인터페이스

웹 페이지 모듈은 다음과 같은 인터페이스 외에도 웹 데이터 모듈의 모든 인터페이스를 구현합니다.

IDefaultPageFileName: WebSnap 외관 디자이너는 이 인터페이스를 사용하여 페이지 모듈이 적절한 템플릿 파일을 사용하는지 확인합니다.

ISetWebContentOptions: WebSnap 외관 디자이너는 이 인터페이스를 사용하여 페이지 모듈이 생성하는 콘텐츠를 제어합니다. 예를 들어, Active Script가 실행되기 전에 콘텐츠를 검색하는 옵션으로 사용될 수 있습니다.

IGetProducerComponent: WebSnap 외관 디자이너는 이 인터페이스를 사용하여 페이지 모듈과 연결된 프로듀서 컴포넌트를 검색합니다.

IProducerEditorViewSupport: WebSnap 외관 디자이너는 이 인터페이스를 사용하여 페이지 모듈이 활성화될 때 표시되어야 하는 에디터 뷰에 관한 정보를 검색합니다. 사용할 수 있는 에디터 뷰에는 HTML Script, Preview, HTML Result, XSL Tree, XML Tree가 있습니다.

IPageResult: 이 인터페이스로 페이지 모듈이 만들어 내는 결과를 액세스할 수 있습니다. 이 인터페이스는 HTTP Content, HTTP Redirection, Page Include 등 세 가지 타입의 결과를 지원합니다.

IGetDefaultAction: 이 인터페이스는 페이지 모듈에 연결된 기본 어댑터 액션이 있는 경우 이를 검색합니다.

웹 애플리케이션 모듈

웹 애플리케이션 모듈은 *TWebPageModule* 또는 *TWebDataModule*이 구현하지 않는 인터페이스를 구현합니다.

웹 애플리케이션 데이터 모듈이 구현하는 인터페이스

웹 애플리케이션 데이터 모듈은 다음과 같은 인터페이스 외에도 웹 데이터 모듈의 모든 인터페이스를 구현합니다.

IGetWebAppServices: 애플리케이션의 웹 요청 핸들러에 인터페이스를 가져옵니다.

IGetWebAppComponents: AdapterDispatcher, PageDispatcher, SessionsService, EndUserAdapter를 포함한 애플리케이션 레벨 컴포넌트 인터페이스를 가져옵니다.

웹 애플리케이션 페이지 모듈이 구현하는 인터페이스

웹 애플리케이션 페이지 모듈은 웹 애플리케이션 데이터 모듈과 웹 페이지 모듈의 모든 인터페이스를 구현합니다.

어댑터(Adapter)

어댑터는 애플리케이션 데이터에 대해 인터페이스를 만드는 방법을 제공합니다. 이 인터페이스로 페이지에 스크립트 언어를 삽입하고 스크립트 코드에서 어댑터로 호출하여 정보를 가져올 수 있습니다.

예를 들어, HTML 페이지에 표시될 데이터 필드를 정의하는 데 어댑터를 사용할 수 있습니다. 스크립트된 HTML 페이지에는 해당 데이터 필드의 값을 가져오는 HTML 콘텐츠와 스크립트 문이 포함되기도 합니다.

Fields

Fields는 페이지 프로듀서가 애플리케이션에서 데이터를 가져오고 웹 페이지에 그 콘텐츠를 표시하기 위해 사용하는 컴포넌트입니다. 필드는 또한 이미지를 가져오는 데에도 사용될 수 있습니다. 이 경우에 필드는 웹 페이지에 작성된 이미지의 주소를 반환합니다. 페이지가 콘텐츠를 표시할 때 웹 애플리케이션에 보내진 요청이 어댑터 디스패처를 호출하여 필드 컴포넌트로부터 실제 이미지를 가져옵니다.

Actions

Actions는 어댑터를 대신하여 명령을 실행하는 컴포넌트입니다. 페이지 프로듀서가 페이지를 생성할 때 스크립트 언어는 어댑터 액션 컴포넌트를 호출하여 명령을 실행하는 데 필요한 매개변수와 함께 액션의 이름을 반환합니다. 예를 들어, HTML 폼 상의 버튼을 클릭함으로써 테이블의 행을 삭제하는 것을 생각해 봅니다. 여기서는 HTTP 요청에서 버튼과 연결된 액션 이름과 행 번호를 나타내는 매개변수가 반환됩니다. 어댑터 디스패처는 명명된 액션 컴포넌트를 찾고 행 번호를 매개변수로서 액션에 전달합니다.

Errors

어댑터는 액션 실행 중 발생한 오류의 목록을 가지고 있습니다. 페이지 프로듀서는 이 오류 목록을 액세스하고 애플리케이션이 최종 사용자에게 반환하는 웹 페이지에 표시합니다.

Records

*TDataSetAdapter*와 같은 일부 어댑터 컴포넌트는 여러 행을 나타냅니다. 어댑터는 여러 행을 반복할 수 있는 스크립트 인터페이스를 제공합니다. 일부 어댑터는 페이지를 지원하고 현재 페이지의 행들만 반복합니다.

페이지 프로듀서

웹 페이지 모듈 대신 페이지 프로듀서를 사용하여 콘텐츠를 생성할 수 있습니다. 프로듀서를 웹 디스패치 액션 항목에 연결하면 WebBroker 애플리케이션에서 사용하던 방법과 동일하게 프로듀서를 사용할 수도 있습니다. 웹 페이지 모듈을 사용함으로써 얻는 이점은 다음과 같습니다.

- 애플리케이션을 실행하지 않고 페이지의 레이아웃을 미리 볼 수 있습니다.
- 페이지 이름을 모듈과 연결할 수 있어서 페이지 디스패처가 자동으로 페이지 프로듀서를 호출할 수 있습니다.

템플릿

프로듀서는 다음과 같은 기능을 제공합니다.

- HTML 콘텐츠를 생성합니다.
- HTMLfile 속성을 사용하여 외부 파일을 참조하거나 HTMLDoc 속성을 사용하여 내부 문자열을 참조할 수 있습니다.
- 프로듀서가 웹 페이지 모듈과 함께 사용되는 경우 템플릿은 유닛에 연결된 파일일 수 있습니다.
- 프로듀서는 투명 태그 또는 활성 스크립팅을 사용하여 템플릿에 삽입될 수 있는 HTML을 동적으로 생성합니다. 투명 태그는 WebBroker 애플리케이션과 동일한 방식으로 사용될 수 있습니다. 자세한 내용은 28-15 페이지의 "HTML 투명 태그 변환"을 참조하십시오. 활성 스크립팅 지원을 통해 HTML 페이지 내부에 JavaScript 또는 VBScript를 포함할 수 있습니다.

WebSnap의 서버측 스크립팅

페이지 프로듀서 템플릿에 JScript 또는 VBScript와 같은 스크립트 언어가 포함될 수 있습니다. 페이지 프로듀서는 프로듀서의 콘텐츠 요청에 응답하여 스크립트를 실행합니다. 웹 애플리케이션이 스크립트를 평가하기 때문에 (브라우저가 평가하는) 클라이언트측 스크립트와 반대로 서버측 스크립트라고 합니다.

활성 스크립팅

WebSnap은 활성 스크립팅에 의존하여 서버측 스크립트를 구현합니다. 활성 스크립팅은 스크립팅 언어가 COM 인터페이스를 통해 애플리케이션 객체에 사용될 수 있게 하는 Microsoft가 만든 기술입니다. Microsoft는 두 가지 활성 스크립트 언어인 VBScript와 JScript를 제공합니다. 다른 랭귀지에 대한 지원은 협력업체를 통해 사용할 수 있습니다.

스크립트 엔진

페이지 프로듀서의 *ScriptEngine* 속성은 템플릿 내의 스크립트를 평가하는 활성 스크립트 엔진을 식별합니다.

스크립트 블록

스크립트 블록은 <%와 %>로 구분됩니다. 스크립트 엔진은 스크립트 블록 안에 있는 텍스트를 평가합니다. 그 결과는 페이지 프로듀서 콘텐츠의 일부가 됩니다. 페이지 프로듀서는 내포된 투명 태그를 변환한 후 스크립트 블록 외부에 텍스트를 작성합니다. 스크립트 블록은 또한 조건부 논리 및 루프에서 텍스트의 출력을 지시할 수 있도록 텍스트를 괄호로 묶을 수 있습니다. 예를 들어, 다음과 같은 JScript 블록은 번호가 매겨진 5개의 줄을 생성합니다.

```
<ul>
  <% for (i=0;i<5;i++) { %>
    <li>Item <% Response.Write(i) %></li>
  <% } %>
</ul>
```

다음의 스크립트 블록과 동일합니다.

```
<ul>
  <% for (i=0;i<5;i++) { %>
    <li>Item <%=i %></li>
  <% } %>
</ul>
```

<%= delimiter는 *Response.Write*의 단축형입니다.

스크립트 작성

개발자는 WebSnap 기능을 활용하여 자동으로 스크립트를 생성할 수 있습니다.

템플릿 마법사

새로운 WebSnap 애플리케이션이나 페이지 모듈을 작성할 때 WebSnap 마법사는 페이지 모듈 템플릿의 초기 콘텐츠를 선택하는 데 사용되는 템플릿 필드를 제공합니다. 예를 들어, "Default"라는 템플릿은 애플리케이션 제목, 페이지 이름, 게시된 페이지에 대한 링크를 표시하는 JScript를 생성합니다.

TAdapterPageProducer

*TAdapterPageProducer*는 HTML과 JScript를 생성하여 폼과 테이블을 만듭니다. 생성된 JScript는 어댑터 객체를 호출하여 필드 값, 필드 이미지 매개변수, 액션 매개변수를 가져옵니다.

스크립트 편집과 보기

WebSnap 외관 디자이너는 스크립트된 페이지를 미리 볼 수 있도록 웹 페이지 모듈 뷰를 제공합니다. HTML Result 탭을 사용하면 실행된 스크립트에서 HTML 결과를 볼 수 있습니다. Preview 탭을 사용하면 브라우저에서 결과를 볼 수 있습니다. HTML Script 탭은 웹 페이지 모듈이 *TAdapterPageProducer*를 사용하는 경우에 사용할 수 있습니다. HTML Script 탭은 *TAdapterPageProducer* 객체에 의해 생성된 HTML과 JScript를 표시합니다. 어댑터 필드를 표시하고 어댑터 액션을 실행하는 HTML 폼을 만드는 스크립트를 작성하는 방법은 이 뷰를 참조하십시오.

페이지에 스크립트 포함

템플릿은 파일이나 다른 페이지의 스크립트를 포함할 수 있습니다. 파일에서 스크립트를 포함하려면 다음과 같은 코드 문을 사용합니다.

```
<!-- #include file="filename.html" -->
```

템플릿이 다른 페이지의 스크립트를 포함하고 스크립트가 포함하고 있는 페이지에 의해 평가되는 경우, 다음과 같은 코드 문을 사용하여 page1의 평가되지 않은 콘텐츠를 포함시킵니다.

```
<!-- #include page="page1" -- >
```

스크립트 객체

스크립트 객체는 스크립트가 참조할 수 있는 VCL 또는 CLX 객체입니다. 활성 스크립트 엔진으로 객체에 대해 *IDispatch* 인터페이스를 등록하여 VCL 또는 CLX 객체를 스크립트할 수 있게 만듭니다. 다음 객체들을 스크립트에 사용할 수 있습니다.

- **Application** – 애플리케이션 객체 (Null일 수도 있음)로 웹 애플리케이션 모듈의 애플리케이션 어댑터에 액세스할 수 있습니다. 다음 JScript 블록에 애플리케이션 제목을 씁니다.

```
<%= Application.Title %>
```

- **EndUser** – EndUser 객체를 통해 웹 애플리케이션 모듈의 최종 사용자 어댑터에 액세스할 수 있습니다. 다음 JScript 블록에 최종 사용자 이름을 씁니다.

```
<%= EndUser.DisplayName %>
```

- **Session** – 세션 객체를 통해 웹 애플리케이션 모듈의 세션 객체에 액세스할 수 있습니다. 다음 JScript 블록에 세션 ID를 씁니다.

```
<%= Session.SessionID %>
```

- **Pages** – 페이지 객체 (*Pages*)로 애플리케이션 페이지에 액세스할 수 있습니다. 다음 JScript 블록에 게시된 모든 페이지에 대한 링크를 씁니다.

```

<% e = new Enumerator(Pages)
   for ( ; !e.atEnd(); e.moveNext() )
   {
       if (e.item().Published)
       {
           Response.Write('<A HREF="' + e.item().HREF + '>' + e.item().Title + '</A>')
       }
   }
%>

```

에디터의 Preview 탭은 스크립트 블록의 정확한 결과를 표시하지는 않습니다. 페이지 객체는 항상 디자인 타임 시 비어 있는데 웹 페이지 모듈 팩토리가 등록되지 않았기 때문입니다.

- **Modules** - 모듈 객체는 애플리케이션 모듈에 대한 액세스를 제공합니다. 다음 JScript 블록에 DM이라는 모듈에 어댑터 필드의 콘텐츠를 씁니다.

```
<%= Modules.DM.Adapter1.Field1.DisplayText %>
```

- **Page** - Page 객체로 현재 페이지에 액세스할 수 있습니다. 다음 JScript 블록에 현재 페이지의 제목을 씁니다.

```
<%= Page.Title %>
```

- **Producer** - Producer 객체로 웹 페이지 모듈의 페이지 프로듀서에 액세스할 수 있습니다. 다음 JScript 블록에 콘텐츠를 작성하기 전에 투명 태그를 평가합니다.

```
<% Producer.Write('Here is a tag <#TAG>') %>
```

에디터의 Preview 탭은 이 스크립트 블록의 정확한 결과를 표시하지는 않습니다. 대개 투명 태그를 바꾸는 이벤트 핸들러는 애플리케이션이 실행 중이 아닌 경우에는 실행되지 않습니다.

- **Response** - Response 객체로 WebResponse에 액세스할 수 있습니다. 태그 교환을 원하지 않을 때는 이 객체를 사용하지 마세요.

```
<% Response.Write('Hello World!') %>
```

- **Request** - Request 객체로 WebRequest에 액세스할 수 있습니다. 다음 JScript 블록에 pathinfo를 표시합니다.

```
<%= Request.PathInfo %>
```

- **Adapter Objects** - 현재 페이지에 있는 모든 어댑터의 컴포넌트는 제한 없이 참조될 수 있습니다. 다른 모듈에 있는 어댑터 객체들은 Modules 객체를 사용하여 제한되어야 합니다. 다음 이 JScript 블록은 Adapter1의 모든 행에서 *FirstName* 필드의 텍스트 값을 표시합니다.

```

<% e = new Enumerator(Adapter1.Records) %>
<% for ( ; !e.atEnd(); e.moveNext() ) %>
<% { %>
    <p><%= Adapter1.FirstName.DisplayText %>
<% } %>

```

요청(Request) 디스패치

WebSnap 애플리케이션은 HTTP 요청 메시지를 받으면 *TWebRequest* 객체를 만들어 HTTP 요청 메시지를 나타내고 *TWebResponse* 객체를 만들어 반환해야 할 응답을 나타냅니다.

WebContext

요청을 처리하기 전에 웹 애플리케이션 모듈은 *TWebContext* 타입의 *WebContext* 객체를 초기화합니다. *WebContext*는 요청을 서비스하는 컴포넌트에 의해 사용되는 변수에 액세스할 수 있게 하는 스레드 변수입니다. 예를 들어, *WebContext*는 *TWebResponse*와 *TWebRequest* 객체뿐만 아니라 나중에 다루게 될 어댑터 요청과 어댑터 응답 객체를 포함합니다.

디스패치 컴포넌트

웹 애플리케이션 모듈 내에서 디스패치 컴포넌트는 애플리케이션의 흐름을 제어합니다. 디스패치는 HTTP 요청을 검사하여 HTTP 요청 메시지의 특정 타입을 처리하는 방법을 결정합니다.

어댑터 디스패치 컴포넌트 (*TAdapterDispatcher*)는 어댑터 액션 컴포넌트 또는 어댑터 이미지 필드 컴포넌트를 식별하는 콘텐츠 필드나 쿼리 필드를 찾습니다. 어댑터 디스패처가 컴포넌트를 찾으면 그 컴포넌트에 제어를 넘깁니다.

웹 디스패치 컴포넌트 (*TWebDispatcher*)는 HTTP 요청 메시지의 특정 타입을 처리하는 방법을 알고 있는 액션 항목 (*TWebActionItem*) 모음을 유지 관리합니다. 웹 디스패처는 요청에 일치하는 액션 항목을 찾습니다. 찾은 경우 그 액션 항목에 제어를 넘깁니다. 웹 디스패처는 또한 요청을 처리할 수 있는 자동 디스패칭 컴포넌트를 찾습니다.

페이지 디스패치 컴포넌트 (*TPageDispatcher*)는 *TWebRequest* 객체의 *pathInfo* 속성을 검사하여 등록된 웹 페이지 모듈의 이름을 찾습니다. 디스패처가 웹 페이지 모듈 이름을 찾은 경우 그 웹 페이지 모듈에 제어를 넘깁니다.

어댑터 디스패치 작업

어댑터 디스패치 컴포넌트는 어댑터 액션과 필드 컴포넌트를 호출하여 HTML 폼 전송, 동적 이미지 요청을 자동으로 처리합니다.

어댑터 컴포넌트를 사용하여 콘텐츠 생성

WebSnap 애플리케이션이 자동으로 어댑터 액션을 실행하고 어댑터 필드에서 동적 이미지를 가져오려면 HTML 콘텐츠가 제대로 구성되어 있어야 합니다. HTML 콘텐츠가 제대로 구성되지 않은 경우, 결과 HTTP 요청은 어댑터 디스패처가 어댑터 액션과 필드 컴포넌트를 호출하기 위해 필요로 하는 정보를 포함하지 않습니다.

HTML 페이지를 구성할 때 오류를 줄이기 위해서 어댑터 컴포넌트는 HTML 요소의 이름과 값을 나타냅니다. 어댑터 컴포넌트에는 어댑터 필드 업데이트 시 사용하는 HTML 폼 상에 표시되어야 하는 숨겨진 필드의 이름과 값을 가져오는 메소드가 있습니다. 대개 페이지 프로듀서는 서버측 스크립트를 사용하여 어댑터 컴포넌트에서 이름과 값을 가져 오고 이 이름과 값으로 HTML을 생성합니다. 예를 들어, 다음의 스크립트는 Adapter1에서 필드 Graphic을 참조하는 요소를 구성합니다.

```

```

웹 애플리케이션이 스크립트를 평가하면 HTML src 속성은 필드 컴포넌트가 이미지를 가져오기 위해 필요로 하는 필드와 매개변수를 식별하는 데 필요한 정보를 담고 있습니다. 결과 HTML은 다음과 같습니다.

```

```

브라우저가 이 이미지를 웹 애플리케이션에 가져오기 위해 HTTP 요청을 보내면 어댑터 디스패처는 모듈 DM에서 Adapter1의 Graphic 필드가 매개변수 "Species No=90090"으로 호출되어야 한다고 결정할 수 있습니다. 어댑터 디스패처는 Graphic 필드를 호출하여 적절한 HTTP 응답을 씁니다.

다음의 스크립트는 Adapter1의 EditRow 액션과 페이지 "Details"를 참조하는 <A> 요소를 구성합니다.

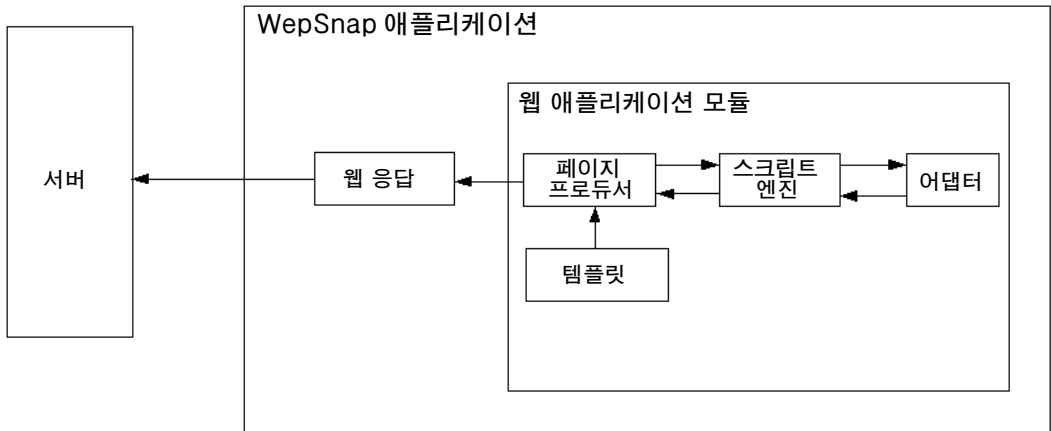
```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).ASHREF%">Edit...</a>
```

결과 HTML은 다음과 같습니다.

```
<a href="?&_lSpecies No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

최종 사용자가 하이퍼링크를 클릭하면 브라우저가 HTTP 요청을 보내고 어댑터 디스패처는 모듈 DM에서 Adapter1의 EditRow 액션이 매개변수 "Species No=90310"으로 호출되어야 한다고 결정할 수 있습니다. 또한 어댑터 디스패처는 액션이 성공적으로 실행될 경우 Edit 페이지가 표시되고 액션 실행이 실패할 경우 Grid 페이지가 표시되는 것을 지시합니다. 그런 다음 EditRow 액션을 호출하여 편집할 행을 찾고 Edit라는 이름의 페이지가 호출되어 HTTP 응답을 생성합니다. 그림 29.1은 어댑터 컴포넌트를 사용하여 콘텐츠를 생성하는 방법을 보여 줍니다.

그림 29.1 콘텐츠 흐름 생성



어댑터 요청(Request) 및 응답(Response)

어댑터 디스패처는 클라이언트 요청을 받으면 어댑터 요청과 어댑터 응답 객체를 만들어 HTTP 요청에 관한 정보를 보유합니다. 어댑터 요청과 어댑터 응답 객체는 WebContext 에 저장되어 요청을 처리하는 동안 액세스할 수 있습니다.

어댑터 디스패처는 액션과 이미지라는 두 가지 타입의 어댑터 요청 객체를 만듭니다. 어댑터 액션을 실행할 때 액션 요청 객체를 만듭니다. 어댑터 필드에서 이미지를 가져올 때 이미지 요청 객체를 만듭니다.

어댑터 응답 객체는 어댑터 컴포넌트가 어댑터 액션 또는 어댑터 이미지 요청에 대한 응답을 나타내기 위해 사용합니다. 어댑터 응답 객체에는 액션과 이미지, 두 가지 타입이 있습니다.

액션 요청(Action request)

액션 요청 객체에는 HTTP 요청을 어댑터 액션을 실행하는 데 필요한 정보로 나눌 책임이 있습니다. 어댑터 액션을 실행하는 데 필요한 정보의 타입은 다음과 같습니다.

- **Component Name** - 어댑터 액션 컴포넌트를 식별합니다.
- **Adapter Mode** - 어댑터는 모드를 정의할 수 있습니다. 예를 들어, *TDataSetAdapter* 는 편집, 삽입, 찾기 모드를 지원합니다. 어댑터 액션은 모드에 따라 다르게 실행할 수도 있습니다. 예를 들어, *TDataSetAdapter* Apply 액션은 삽입 모드에서 새 레코드를 추가하고 편집 모드에서는 레코드를 업데이트합니다.
- **Success Page** - 성공 페이지는 액션을 성공적으로 실행한 후에 표시되는 페이지를 식별합니다.
- **Failure Page** - 실패 페이지는 액션을 실행하는 동안 오류가 발생할 경우 표시되는 페이지를 식별합니다.

- **Action Request Parameters** - 어댑터 액션에서 필요로 하는 매개변수를 나타냅니다. 예를 들어, *TDataSetAdapter* Apply 액션은 업데이트될 레코드를 나타내는 키 값을 포함합니다.
- **Adapter Field Values** - HTML 폼이 전송될 때 HTTP 요청에 전달되는 어댑터 필드의 값입니다. 필드 값은 최종 사용자가 입력한 새로운 값, 어댑터 필드의 원래 값, 업로드된 파일을 포함할 수 있습니다.
- **Record Keys** - HTML 폼이 여러 레코드의 변경 사항을 전송할 경우 어댑터 액션이 각 레코드에서 수행될 수 있도록 각 레코드를 고유하게 나타내기 위해서는 어댑터 액션 컴포넌트에 의해 사용되는 키가 필요합니다. 예를 들어, *TDataSetAdapter* Apply 액션이 여러 레코드에서 수행되면 데이터셋 필드를 업데이트하기 전에 레코드 키를 사용하여 데이터셋의 각 레코드를 찾을 수 있습니다.

액션 응답(Action response)

액션 응답 객체는 어댑터 액션 컴포넌트를 대신하여 HTTP 응답을 생성합니다. 어댑터 액션은 객체 내에서 속성을 설정하거나 액션 응답 객체에서 메소드를 호출하여 응답의 타입을 나타냅니다. 속성은 다음과 같습니다.

- *Redirect Options* - 리디렉션 옵션은 HTML 콘텐츠를 반환하는 것 대신 HTTP 리디렉션을 수행할지 여부를 나타냅니다.
- *Execution Status* - 상태를 success로 설정하면 기본 액션 응답은 Action Request에 확인된 성공 페이지의 콘텐츠가 됩니다.

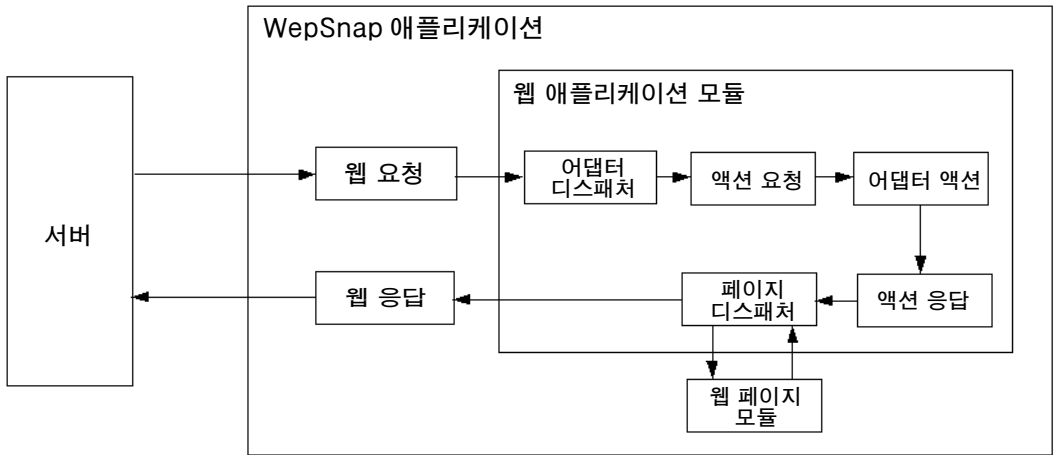
액션 응답 메소드는 다음과 같습니다.

- *RespondWithPag* - 어댑터 액션은 특정 웹 페이지 모듈이 응답을 생성해야 할 때 이 메소드를 호출합니다.
- *RespondWithComponent* - 어댑터 액션은 이 컴포넌트가 들어 있는 웹 페이지 모듈에서 응답이 나와야 할 때 이 메소드를 호출합니다.
- *RespondWithURL* - 어댑터 액션은 지정된 URL에 대한 리디렉션이 응답일 때 이 메소드를 호출합니다.

페이지로 응답할 때 액션 응답 객체는 페이지 디스패처를 사용하여 페이지 콘텐츠를 생성하려고 합니다. 페이지 디스패처를 찾지 못하면 직접 웹 페이지 모듈을 호출합니다.

그림 29.2는 액션 요청과 액션 응답 객체가 요청을 처리하는 방법을 보여 줍니다.

그림 29.2 액션 요청과 응답



이미지 요청(Image request)

이미지 요청 객체는 HTTP 요청을 어댑터 이미지 필드가 이미지를 생성하기 위해 필요한 정보로 나눌 책임이 있습니다. 이미지 요청이 나타내는 정보의 타입은 다음과 같습니다.

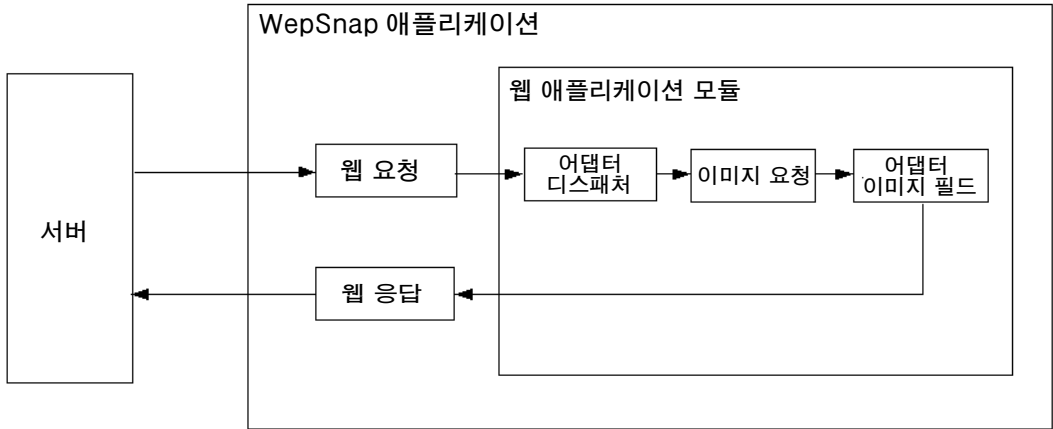
- Component Name - 어댑터 필드 컴포넌트를 나타냅니다.
- Image Request Parameters - 어댑터 이미지가 필요로 하는 매개변수를 식별합니다. 예를 들어, *TDataSetAdapterImageField* 객체에는 이미지가 포함된 레코드를 나타내는 키 값이 필요합니다.

이미지 응답(Image response)

이미지 응답 객체는 *TWebResponse* 객체를 포함합니다. 어댑터 필드는 웹 응답 객체에 이미지를 써서 어댑터 요청에 응답합니다.

그림 29.3은 어댑터 이미지 필드가 요청에 어떻게 응답하는지 보여 줍니다.

그림 29.3 요청에 대한 이미지 응답



액션 항목 디스패칭

웹 디스패처(*TWebDispatcher*)는 액션 항목의 목록을 검색합니다.

- 대상 URL 요청 메시지의 Pathinfo 부분과 일치합니다.
- 요청 메시지의 메소드로 지정된 서비스를 제공할 수 있습니다.

즉, *TWebRequest* 객체의 *PathInfo* 및 *MethodType* 속성을 액션 항목에 있는 동일한 이름의 속성과 비교함으로써 이것을 수행합니다.

적절한 액션 항목을 찾으면 디스패처는 해당 액션 항목을 실행시키고 액션 항목은 다음 중 하나를 실행합니다.

- 응답 콘텐츠를 완성하고 요청이 완전히 처리된 응답 또는 신호를 보냅니다.
- 응답을 추가하고 다른 액션 항목이 작업을 완료할 수 있게 해줍니다.
- 요청을 다른 액션 항목에 전달합니다.

디스패처가 모든 액션 항목을 확인한 뒤에 메시지가 정확히 처리되지 않았으면 디스패처는 액션 항목을 사용하지 않고 특별히 등록된 자동 디스패칭 컴포넌트를 확인합니다. 이 컴포넌트들은 다계층 데이터베이스 애플리케이션에 특정적입니다. 여전히 요청 메시지가 완전히 처리되지 않았으면 디스패처는 기본 액션 항목을 호출합니다. 기본 액션 항목은 대상 URL 또는 요청의 메소드와 일치할 필요는 없습니다.

페이지 디스패처 작업

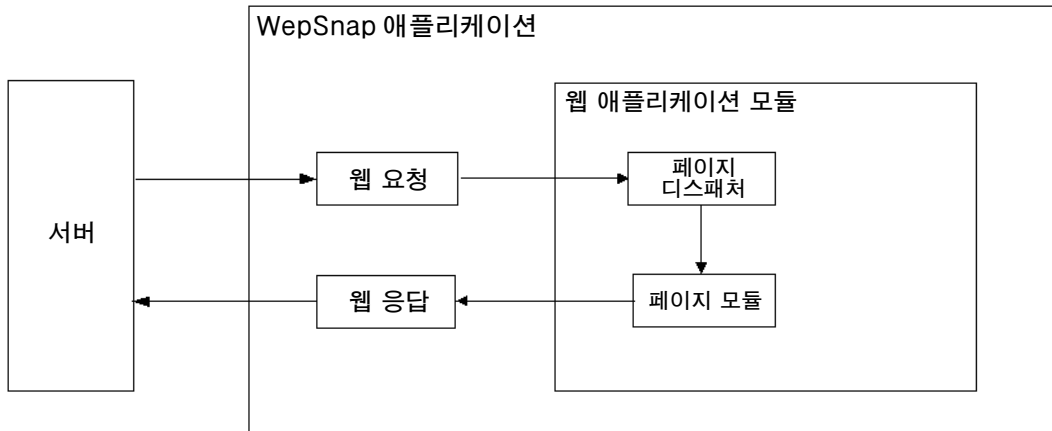
페이지 디스패처가 클라이언트 요청을 받으면 대상 URL의 요청 메시지의 Pathinfo 부분을 확인하여 페이지 이름을 알아냅니다. 경로 정보 부분이 비어 있지 않으면 페이지 디스패처는 경로 정보의 끝 단어를 페이지 이름으로 사용합니다. 경로 정보 부분이 비어 있으면 페이지 디스패처는 기본 페이지 이름을 알아내려고 합니다.

페이지 디스패처의 *DefaultPage* 속성에 페이지 이름이 있으면 페이지 디스패처는 이 이름을 기본 페이지 이름으로 사용합니다. *DefaultPage* 속성이 비어 있고 웹 애플리케이션 모듈이 페이지 모듈인 경우 페이지 디스패처는 웹 애플리케이션 모듈의 이름을 기본 페이지 이름으로 사용합니다.

페이지 이름이 비어 있지 않으면 페이지 디스패처는 일치하는 이름을 갖는 웹 페이지 모듈을 찾습니다. 웹 페이지 모듈을 찾으면 그 모듈을 호출하여 응답을 생성합니다. 페이지 이름이 비어 있거나 페이지 디스패처가 웹 페이지 모듈을 찾지 못하면 페이지 디스패처는 예외를 생성합니다.

그림 29.4는 페이지 디스패처가 요청에 어떻게 응답하는지 보여 줍니다.

그림 29.4 페이지 디스패칭



WebSnap 자습서

다음 단원에서는 WebSnap 애플리케이션을 만드는 단계를 설명합니다. 새 디스패처와 어댑터 컴포넌트를 새 웹 페이지 모듈에 사용함으로써 자습서를 마치면 사용자는 WebSnap 아키텍처와 새로운 개념에 친숙해질 것입니다. WebSnap 애플리케이션은 WebSnap HTML 컴포넌트를 사용하여 테이블의 콘텐츠를 편집하는 애플리케이션을 만드는 방법을 보여 줍니다.

새 애플리케이션 생성

다음과 같은 방법으로 새 WebSnap 애플리케이션을 만듭니다.

1 단계. WebSnap 애플리케이션 마법사 시작

- 1 Delphi 애플리케이션을 실행한 후 File|New|Other를 선택합니다.
- 2 New Items 대화 상자에서 WebSnap 탭을 선택한 후 WebSnap Application을 선택합니다.

- 3 New WebSnap Application 대화 상자에서 다음과 같이 합니다.
 - Web App Debugger Executable을 선택합니다.
 - CoClass 필드에 **CountryTutorial**을 입력합니다.
 - Page Module을 컴포넌트 타입으로 선택합니다.
 - Page Name 필드에 **Home**을 입력합니다.
- 4 OK를 클릭합니다.

2 단계. 생성된 파일과 프로젝트 저장

다음과 같은 방법으로 파스칼 유닛 파일과 프로젝트를 저장합니다.

- 1 File|Save All을 선택합니다.
- 2 File name 필드에 **HomeU.pas**를 입력하고 OK를 클릭합니다.
- 3 File name 필드에 **CountryU.pas**를 입력하고 OK를 클릭합니다.
- 4 File name 필드에 **CountryTutorial.dpr**을 입력하고 OK를 클릭합니다.
- 5 OK를 클릭합니다.

참고 애플리케이션이 실행 파일과 동일한 디렉토리에서 HomeU.html 파일을 찾기 때문에 동일한 디렉토리에 유닛과 프로젝트를 저장합니다.

3 단계. 애플리케이션 제목 지정

애플리케이션 제목은 최종 사용자에게 나타내는 이름입니다. 다음과 같은 방법으로 애플리케이션 제목을 지정합니다.

- 1 View|Project Manager를 선택합니다.
- 2 Project Manager 창에서 CountryTutorial.exe를 확장하고 HomeU 항목을 더블 클릭합니다.
- 3 Object Inspector 창 (왼쪽 아래)의 폴다운 목록에서 ApplicationAdapter를 선택합니다.
- 4 Properties 탭에서 ApplicationTitle 필드에 **Country Tutorial**을 입력합니다.
- 5 에디터 창에서 Preview 탭을 클릭합니다. 애플리케이션 제목이 페이지의 위쪽에 표시됩니다.

CountryTable 페이지 생성

웹 페이지 모듈은 게시된 페이지를 데이터 컴포넌트의 컨테이너로 정의하기 위해 사용됩니다.

1 단계. 새 모듈 추가

다음과 같은 방법으로 새로운 모듈을 추가합니다.

- 1 File|New|Other를 선택합니다.
- 2 New Items 대화 상자에서 WebSnap 탭을 선택한 후 WebSnap Page Module을 선택합니다.
- 3 대화 상자에서 Producer Type을 목록에 있는 AdapterPageProducer로 설정합니다.
- 4 Page Name 필드에 **CountryTable**을 입력합니다.
- 5 나머지 필드와 선택은 기본값으로 둡니다.
- 6 OK를 클릭합니다.

2 단계. 새 모듈 저장

유닛을 프로젝트 파일의 디렉토리에 저장합니다. 애플리케이션이 실행될 때 실행 파일과 동일한 디렉토리에서 CountryTableU.html 파일을 찾습니다.

- 1 File|Save를 선택합니다.
- 2 File 이름 필드에 **CountryTableU.pas**를 입력하고 OK를 클릭합니다.

CountryTable 모듈에 데이터 컴포넌트 추가

TTable 컴포넌트는 HTML 테이블에 데이터를 제공합니다. *TDataSetAdapter* 컴포넌트를 통해 서버측 스크립트는 *TTable* 컴포넌트에 액세스할 수 있습니다.

1 단계. data-aware 컴포넌트 추가

- 1 View|Project Manager를 선택합니다.
- 2 Project Manager 창에서 CountryTutorial.exe를 확장하고 CountryTableU 항목을 더블 클릭합니다.
- 3 View|Object TreeView를 선택합니다. 왼쪽에 있는 Object TreeView 창이 활성화됩니다.
- 4 도구 팔레트에서 Data Access 탭을 선택합니다.
- 5 Table 컴포넌트를 선택한 후 (마우스 왼쪽 버튼을 클릭한 채로) Object TreeView 창으로 끌어다 놓습니다.
- 6 Session 컴포넌트를 선택한 후 (마우스 왼쪽 버튼을 클릭한 채로) Object TreeView 창으로 끌어다 놓습니다.
- 7 Object TreeView 창에서 Session 컴포넌트를 선택합니다. 그러면 Object Inspector 창에 Session 컴포넌트 값이 표시됩니다.
- 8 Object Inspector 창에서 AutoSessionName 속성을 *True*로 설정합니다.

- 9 Object TreeView 창에서 Table 컴포넌트를 선택합니다. 그러면 Object Inspector 창에 Table 컴포넌트 값이 표시됩니다.
- 10 Object Inspector 창에서 다음 속성을 변경합니다.
 - *Active* 속성을 *True*로 설정합니다.
 - *DatabaseName* 속성을 DBDEMOS로 설정합니다.
 - *Name* 속성에 **Country**를 입력합니다.
 - *TableName* 속성을 country.db로 설정합니다.

참고 멀티스레드된 애플리케이션에서는 BDE 컴포넌트(*TTable*)를 사용하므로 Session 컴포넌트가 필요합니다.

2 단계. 키 필드 지정

키 필드는 테이블 내의 레코드를 나타낼 때 사용합니다. 이것은 애플리케이션에 페이지를 추가하고 편집할 때 중요해집니다. 다음과 같은 방법으로 키 필드를 지정합니다.

- 1 Object TreeView 창에서 Session과 DBDemos 노드를 확장하고 country.db 노드를 선택합니다. 이 노드는 Country Table 컴포넌트입니다.
- 2 country.db 노드를 마우스 오른쪽 버튼으로 클릭하고 Fields Editor를 선택합니다.
- 3 CountryTable.Country 에디터 창을 마우스 오른쪽 버튼으로 클릭하고 Add All Fields 명령을 선택합니다.
- 4 추가된 필드 목록에서 Name 필드를 선택합니다.
- 5 Object Inspector 창에서 *ProviderFlags* 속성을 확장합니다.
- 6 *pfInKey* 속성 값을 *True*로 설정합니다.

3 단계. 어댑터 컴포넌트 추가

TTable 서버측 스크립팅에서 데이터를 나타내려면 *DataSetAdapter* (*TDataSetAdapter*) 컴포넌트를 포함해야 합니다. 다음과 같은 방법으로 해당 컴포넌트를 추가합니다.

- 1 도구 팔레트에서 WebSnap 탭을 선택합니다.
- 2 *DataSetAdapter* 컴포넌트를 선택한 후 (마우스 왼쪽 버튼을 클릭한 채로) Object TreeView 창으로 끌어다 놓습니다.
- 3 Object Inspector 창에서 다음 속성을 변경합니다.
 - *DataSet* 필드를 Country로 설정합니다.
 - *Name* 필드에 **Adapter**를 입력합니다.

데이터를 표시할 그리드 생성

*AdapterPageProducer*는 서버측 스크립트를 활용하여 쉽게 HTML 테이블을 만듭니다.

1 단계. 그리드 추가

다음과 같은 방법으로 Country 테이블에서 데이터를 표시하는 그리드를 추가합니다.

- 1 View|Project Manager를 선택합니다.
- 2 Project Manager 창에서 CountryTutorial.exe를 확장하여 CountryTableU 항목을 더블 클릭합니다.
- 3 View|Object TreeView를 선택합니다. 왼쪽에 있는 Object TreeView 창이 활성화됩니다.
- 4 *AdapterPageProducer* 컴포넌트를 확장합니다.
- 5 WebPageItems 항목을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
- 6 Add Web Component 창에서 AdapterForm을 선택한 다음 OK를 클릭합니다. AdapterForm1 컴포넌트가 Object TreeView 창에 나타납니다.
- 7 AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
- 8 Add Web Component 창에서 AdapterGrid를 선택한 다음 OK를 클릭합니다. AdapterGrid1 컴포넌트가 Object TreeView 창에 나타납니다.
- 9 Object Inspector 창에서 Adapter 속성을 Adapter로 설정합니다.
- 10 Page를 미리 보려면 코드 에디터 창에서 CountryTableU.pas 탭을 선택하고 아래쪽에 있는 Preview 탭을 선택합니다. Preview 탭이 보이지 않으면 아래쪽에 있는 오른쪽 화살표를 사용하여 탭을 스크롤합니다.
- 11 HTML Script 탭을 선택하여 WebSnap 컴포넌트가 생성한 JScript를 봅니다.

2 단계. 그리드에 편집 명령 추가

사용자는 테이블의 콘텐츠를 업데이트해야 합니다. 사용자가 행을 삭제하거나 삽입하는 등의 업데이트를 할 수 있게 하려면 명령 컴포넌트를 추가합니다.

다음과 같은 방법으로 명령 컴포넌트를 추가합니다.

- 1 CountryTable의 Object TreeView 창에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장합니다.
- 2 AdapterGrid1 컴포넌트를 마우스 오른쪽 버튼으로 클릭하고 Add All Columns를 선택합니다.
- 3 AdapterGrid1 컴포넌트를 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다. AdapterCommandColumn1 항목이 AdapterGrid1 컴포넌트에 추가됩니다.
- 4 AdapterCommandColumn1을 마우스 오른쪽 버튼으로 클릭하고 Add Commands를 선택합니다.
- 5 DeleteRow, EditRow, NewRow 중에서 해당 명령을 선택한 다음 OK를 클릭합니다.

- 6 페이지를 미리 보려면 코드 에디터의 아래쪽에서 Preview 탭을 클릭합니다.

편집 폼 추가

country 테이블의 편집 폼이 될 웹 페이지 모듈을 작성합니다.

1 단계. 새로운 모듈 추가

다음과 같은 방법으로 새로운 WebSnap 페이지 모듈을 추가합니다.

- 1 File|New|Other를 선택합니다.
- 2 New Items 대화 상자에서 WebSnap 탭을 선택한 후 WebSnap Page Module을 선택합니다.
- 3 대화 상자에서 Producer Type을 목록에 있는 AdapterPageProducer로 설정합니다.
- 4 Page Name 필드에 **CountryForm**을 입력합니다.
- 5 Published 상자를 선택 해제합니다.
- 6 나머지 필드와 선택은 기본값으로 둡니다.
- 7 OK를 클릭합니다.

2 단계. 새 모듈 저장

유닛을 프로젝트 파일로 디렉토리에 저장합니다. 애플리케이션이 실행되면 실행 파일과 동일한 디렉토리에서 CountryFormU.html 파일을 찾습니다.

- 1 File|Save를 선택합니다.
- 2 File name 필드에 **CountryFormU.pas**를 입력하고 OK를 클릭합니다.

3 단계. CountryTableU 유닛 사용

Add CountryTableU 유닛을 **uses** 절에 추가하여 모듈이 Adapter 컴포넌트에 액세스할 수 있게 합니다.

- 1 File|Use Unit을 선택합니다.
- 2 목록에서 CountryTableU를 선택한 다음 OK를 클릭합니다.

4 단계. 입력 필드 추가

컴포넌트를 *AdapterPageProducer* 컴포넌트에 추가하여 데이터 입력 필드를 HTML 형태로 생성합니다.

다음과 같은 방법으로 입력 필드를 추가합니다.

- 1 View|Project Manager를 선택합니다.

- 2 Project Manager 창에서 CountryTutorial.exe를 확장하고 CountryTableU 항목을 더블 클릭합니다.
- 3 View|Object TreeView를 선택합니다. 왼쪽에 있는 Object TreeView 창이 활성화됩니다.
- 4 Object TreeView 창에서 *AdapterPageProducer* 컴포넌트를 확장한 다음 WebPageItems를 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
- 5 AdapterForm을 선택한 다음 OK를 클릭합니다. AdapterForm1 항목이 Object TreeView 창에 나타납니다.
- 6 AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
- 7 AdapterFieldGroup을 선택한 다음 OK를 클릭합니다. AdapterFieldGroup1 항목이 Object TreeView 창에 나타납니다.
- 8 Object Inspector 창에서 Adapter 속성을 CountryTable.Adapter로 설정합니다.
- 9 페이지를 미리 보려면 코드 에디터의 아래쪽에서 Preview 탭을 클릭합니다.

5 단계. 버튼 추가

컴포넌트를 *AdapterPageProducer* 컴포넌트에 추가하고 HTML 폼에 전송 버튼을 생성합니다. 다음과 같은 방법으로 컴포넌트를 추가합니다.

- 1 Object TreeView에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장합니다.
- 2 AdapterForm1 항목을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
- 3 AdapterCommandGroup을 선택한 다음 OK를 클릭합니다. AdapterCommandGroup1 항목이 Object TreeView 창에 나타납니다.
- 4 Object Inspector 창에서 DisplayComponent 속성을 AdapterFieldGroup1로 설정합니다.
- 5 AdapterCommandGroup1 항목을 마우스 오른쪽 버튼으로 클릭하고 Add Commands를 선택합니다.
- 6 Cancel, Apply, Refresh Row 중에서 해당 명령을 선택한 다음 OK를 클릭합니다.
- 7 페이지를 미리 보려면 코드 에디터 창의 아래쪽에 있는 Preview 탭을 클릭합니다. 미리 보기로 country 폼을 볼 수 없으면 Code 탭을 클릭한 다음 Preview 탭을 다시 클릭합니다.

6 단계. 그리드 페이지에 폼 액션 연결

사용자가 버튼을 클릭하면 어댑터 액션이 실행됩니다. 다음과 같은 방법으로 어댑터 액션이 실행된 후에 표시할 페이지를 지정합니다.

- 1 Object TreeView에서 AdapterCommandGroup1을 확장하여 CmdCancel, CmdApply, CmdRefreshRow 항목을 표시합니다.
- 2 CmdCancel을 선택합니다. Object Inspector 창에서 PageName 속성에 **CountryTable**을 입력합니다.
- 3 Select CmdApply.Object Inspector 창에서 PageName 속성에 **CountryTable**을 입력합니다.

7 단계. 폼 페이지에 그리드 액션 연결

다음과 같은 방법으로 그리드에서 버튼을 클릭하여 어댑터 액션이 실행된 후에 표시할 페이지를 지정합니다.

- 1 View|Project Manager를 선택합니다.
- 2 Project Manager 창에서 CountryTutorial.exe를 확장하여 CountryTableU 항목을 더블 클릭합니다.
- 3 Object TreeView 창에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장하고 CmdNewRow, CmdEditRow, CmdDeleteRow 항목을 표시합니다. 이 항목들이 AdapterCommandColumn1 항목 아래에 나타납니다.
- 4 CmdNewRow를 선택합니다. Object Inspector 창에서 PageName 속성에 **CountryForm**을 입력합니다.
- 5 CmdEditRow를 선택합니다. Object Inspector 창에서 PageName 속성에 **CountryForm**을 입력합니다.
- 6 애플리케이션이 작업 중이고 모든 버튼이 몇몇 액션을 수행하는지 확인하려면 애플리케이션을 실행하십시오.

참고 잘못된 타입 같은 데이터베이스 오류 표시는 없습니다. 예를 들어, Area 필드에 잘못된 값(예: 'abc')을 가진 나라 이름을 추가해 보십시오.

오류 보고 추가

최종 사용자에게 오류를 보고하기 위해 AdapterErrorList 컴포넌트를 사용하여 country 테이블을 편집하는 어댑터 액션을 실행할 때 발생하는 오류를 표시합니다.

1 단계. 그리드에 오류 지원 추가

- 1 CountryTable에 대한 Object TreeView에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장하여 AdapterForm1을 표시합니다.
- 2 AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
- 3 목록에서 AdapterErrorList를 선택한 다음 OK를 클릭합니다. AdapterErrorList1 항목이 Object TreeView 창에 나타납니다.

- 4 마우스로 직접 끌거나 Object TreeView 툴바의 위쪽 화살표를 사용하여 끌어서 AdapterErrorList1을 AdapterGrid1 위로 이동합니다.
- 5 Object Inspector 창에서 Adapter 속성을 Adapter로 설정합니다.

2 단계. 폼에 오류 지원 추가

- 1 CountryForm에 대한 Object TreeView에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장하여 AdapterForm1을 표시합니다.
- 2 AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
- 3 목록에서 AdapterErrorList를 선택한 다음 OK를 클릭합니다. AdapterErrorList1 항목이 Object TreeView 창에 나타납니다.
- 4 직접 마우스로 끌거나 Object TreeView 툴바의 위쪽 화살표를 사용하여 끌어서 AdapterErrorList1을 AdapterGrid1 위로 이동합니다.
- 5 Object Inspector 창에서 Adapter 속성을 CountryTable.Adapter로 설정합니다.

3 단계. 오류 보고 메커니즘 테스트

다음과 같은 방법으로 Grid Errors를 테스트합니다.

- 1 애플리케이션을 실행한 다음 CountryTable 페이지로 이동합니다.
- 2 브라우저의 다른 인스턴스를 시작하여 CountryTable 페이지를 찾습니다.
- 3 그리드의 첫 행에서 DeleteRow 버튼을 클릭합니다.
- 4 두 번째 브라우저를 새로 고치지 않고 그리드의 첫 행에서 DeleteRow 버튼을 클릭합니다.
- 5 그리드 위에 오류 메시지가 표시됩니다.

다음과 같은 방법으로 폼 오류를 테스트합니다.

- 1 애플리케이션을 실행한 다음 CountryTable 페이지로 이동합니다.
- 2 EditRow 버튼을 클릭합니다.
- 3 CountryForm 페이지가 표시됩니다.
- 4 Area 필드를 'abc'로 변경하고 Apply 버튼을 클릭합니다.
- 5 첫 필드 위에 오류 메시지가 표시됩니다.

완성된 애플리케이션 실행

다음과 같은 방법으로 완성된 애플리케이션을 실행합니다.

- 1 Run|Run을 선택합니다. 폼이 표시됩니다. Web App Debugger 실행 가능 웹 애플리케이션은 COM 서버이고 사용자가 보는 폼은 COM 서버의 콘솔 창입니다. 처음 프로젝트를 실행할 때 Web App Debugger가 직접 액세스할 수 있는 COM 객체가 등록됩니다.
- 2 Tools|Web App Debugger를 선택합니다.

- 3** 기본 URL 링크를 클릭하여 ServerInfo 페이지를 표시합니다. ServerInfo 페이지는 등록된 모든 Web Application Debugger 실행 파일의 이름을 표시합니다.
- 4** 드롭다운 목록에서 CountryTutorial을 선택하고 Go 버튼을 클릭합니다.

30

XML 문서 작업

XML(Extensible Markup Language)은 구조화된 데이터를 나타내기 위한 마크업(markup) 언어입니다. 태그가 표시 특성보다는 정보의 구조를 나타낸다는 것을 제외하고는 HTML과 유사합니다. XML 문서는 쉽게 검색하고 편집되도록 정보를 저장하는 간단한 텍스트 기반 방식을 제공합니다. XML은 웹 애플리케이션, B2B 통신 등에서 데이터에 대한 표준 전송 포맷으로 사용되기도 합니다.

XML 문서는 데이터 몸체의 계층 구조 뷰를 제공합니다. XML 문서의 태그는 주식의 모습을 나타내는 다음과 같은 문서에서처럼 각 데이터 요소의 역할이나 의미를 나타냅니다.

```
<?xml version="1.0" standalone='yes' ?>
<!DOCTYPE stockholdings SYSTEM "sth.dtd">
<StockList>
  <Stock exchange=NASDAQ>
    <name>Inprise (Borland)</name>
    <price>15.375</price>
    <symbol>INPR</symbol>
    <shares>100</shares>
  </Stock>
  <Stock exchange=NYSE>
    <name>Pfizer</name>
    <price>42.75</price>
    <symbol>PFE</symbol>
    <shares type=preferred>25</shares>
  </Stock>
</StockList>
```

이 예제는 XML 문서의 여러 가지 일반적인 요소를 설명합니다. 첫 번째 줄은 처리 명령입니다. 파일 해석 방법에 대한 정보를 제공하지만 데이터를 포함하지 않습니다.

<!DOCTYPE> 태그로 시작하는 두 번째 줄은 문서 타입 선언입니다. 문서의 구조를 명명하고 구조를 나타내는 다른 파일(sth.dtd)을 참조합니다. 이 경우에 구조는 Document Type Definition(DTD) 파일에 나타납니다. XML 문서의 구조를 나타내는 다른 파일 형식에는 Reduced XML Data(XDR)와 XML 스키마(XSD)가 있습니다.

나머지 줄은 단일 루트 노드(<StockList> 태그)를 갖는 계층 구조로 구성됩니다. 이 계층 구조의 각 노드는 자식 노드의 집합이나 텍스트 값을 포함합니다. 태그 중 일부는 (<Stock> 및 <shares> 태그) 태그 해석 방법에 대한 세부 사항을 제공하는 Name=Value 쌍인 속성을 포함합니다.

XML 문서의 텍스트에서 직접 작업할 수 있기는 있지만 일반적으로 애플리케이션은 데이터 구문 분석과 편집을 위한 몇 가지 추가 툴을 사용합니다. W3C는 Document Object Model(DOM)이라고 하는 구문 분석된 XML 문서를 나타내는 표준 인터페이스 집합을 정의합니다. 여러 공급업체들은 사용자가 XML 문서를 더 쉽게 해석하고 편집할 수 있도록 DOM 인터페이스를 구현하는 XML 파서를 제공합니다.

Delphi는 XML 문서 작업을 위한 여러 가지 추가 툴을 제공합니다. 이러한 툴이 다른 공급업체가 제공하는 DOM 파서를 사용하므로 XML 문서 작업이 훨씬 쉬워집니다. 이 장에서는 그러한 툴에 대해 설명합니다.

참고 이 장에 설명된 툴 이외에도 Delphi에는 XML 문서를 Delphi 데이터베이스 아키텍처로 통합하는 데이터 패킷으로 변환하는 툴과 컴포넌트가 있습니다. XML 문서를 데이터베이스 애플리케이션으로 통합하는 툴에 대한 자세한 내용은 26장 "데이터베이스 애플리케이션에서 XML 사용"을 참조하십시오.

Document Object Model 사용

DOM(Document Object Model)은 구문 분석된 XML 문서를 나타내는 표준 인터페이스 집합입니다. Delphi에는 (Microsoft와 IBM에서 만든) 두 개의 DOM 구현이 함께 들어 있습니다. 또한 사용자가 다른 공급업체의 추가 DOM 구현을 Delphi XML 프레임워크로 통합할 수 있게 하는 등록 메커니즘을 포함합니다.

XMLDOM 유닛은 W3C XML DOM 레벨 2 사양에 정의된 모든 DOM 인터페이스에 대한 선언을 포함합니다. 각각의 DOM 공급업체는 이러한 인터페이스의 구현을 제공합니다.

- Microsoft 구현을 사용하려면 사용자의 uses 절에 MSXMLDOM 유닛을 포함시킵니다. Microsoft 구현이 COM 기반이기 때문에 또한 msxml.dll 라이브러리를 COM 서버로 등록해야 합니다. Regsvr32.exe를 사용하여 이 DLL을 등록할 수 있습니다.
- IBM 구현을 사용하려면 사용자의 uses 절에 IBMXMLDOM 유닛을 포함시킵니다.
- 또 다른 DOM 구현을 사용하려면 상위 레벨 인터페이스(*IDOMImplementation*)를 반환하는 함수를 포함하는 유닛을 만들어야 합니다. 사용자의 유닛은 전역 *RegisterDOMImplementation* 프로시저를 호출하여 이러한 함수를 등록해야 합니다.

일부 공급업체들은 표준 DOM 인터페이스에 확장을 제공합니다. 사용자가 이러한 확장을 사용할 수 있도록 하려면 XMLDOM 유닛에서 *IDOMNodeEx* 인터페이스를 정의합니다. *IDOMNodeEx*는 가장 유용한 이러한 확장자를 포함하는 표준 *IDOMNode*의 자손입니다.

DOM 인터페이스를 직접 사용하여 XML 문서를 구문 분석하고 편집할 수 있습니다. 간단하게 *GetDOM* 함수를 호출하면 시작점으로 사용할 수 있는 *IDOMImplementation* 인터페이스를 얻을 수 있습니다.

참고 DOM 인터페이스에 대한 자세한 설명은 DOM 공급업체가 제공하는 문서인 XMLDOM 유닛 안에 있는 선언 또는 W3C 웹 사이트(www.w3.org)에서 제공되는 규격을 참조하십시오.

DOM 인터페이스를 직접 작업하는 것보다 Delphi의 XML 절을 사용하는 것이 더 편리할 수도 있습니다. 이러한 내용은 아래에서 설명됩니다.

XML 컴포넌트 작업

Delphi는 XML 문서 작업을 위한 많은 클래스와 인터페이스를 정의합니다. 이는 XML 문서의 로드, 편집 및 저장 프로세스를 단순화시킵니다.

TXMLDocument 사용

XML 문서 작업의 시작점은 *TXMLDocument* 컴포넌트입니다. 다음 단계들은 *TXMLDocument*를 사용하여 XML 문서에서 직접 작업하는 방법에 대해 설명합니다.

- 1 *TXMLDocument* 컴포넌트를 사용자의 폼이나 데이터 모듈에 추가합니다. *TXMLDocument*는 컴포넌트 팔레트의 Internet 페이지에 나타납니다.
- 2 *DOMVendor* 속성을 설정하여 컴포넌트가 XML 문서의 구문 분석과 편집에 사용하고자 하는 DOM 구현을 지정합니다. Object Inspector에 현재 등록된 모든 DOM 공급업체가 나열되어 있습니다. DOM 구현에 대한 내용은 30-2 페이지의 "Document Object Model 사용"을 참조하십시오.
- 3 사용자의 구현에 따라 *ParseOptions* 속성을 설정하여 원본으로 사용하는 DOM 구현이 XML 문서를 구문 분석하는 방법을 구성할 수도 있습니다.
- 4 기존 XML 문서에서 작업하려면 다음과 같이 문서를 지정합니다.
 - XML 문서를 파일에 저장하려면 *FileName* 속성을 해당 파일의 이름으로 설정합니다.
 - *XML* 속성을 사용하는 대신 XML 문서를 문자열로 지정할 수 있습니다.
- 5 *Active* 속성을 *True*로 설정합니다.

일단 활성 *TXMLDocument* 객체를 가지면 노드의 계층 구조를 통과하여 값을 읽거나 설정할 수 있습니다. 이 계층 구조의 루트 노드는 *DocumentElement* 속성으로 사용할 수 있습니다.

XML 노드 작업

일단 XML 문서가 DOM 구현에 의해 구문 분석되었으면 문서가 나타내는 데이터는 노드의 계층 구조로 사용될 수 있습니다. 각 노드는 문서의 태그로 표시된 요소에 해당합니다. 예를 들어, 다음과 같은 XML이 있습니다.

```
Component palette<?xml version="1.0" standalone='yes' ?>
<!DOCTYPE stockholdings SYSTEM "sth.dtd">
<StockList>
  <Stock exchange=NASDAQ>
    <name>Inprise (Borland)</name>
    <price>15.375</price>
    <symbol>INPR</symbol>
    <shares>100</shares>
  </Stock>
  <Stock exchange=NYSE>
    <name>Pfizer</name>
    <price>42.75</price>
    <symbol>PFE</symbol>
    <shares type=preferred>25</shares>
  </Stock>
</StockList>
```

*TXMLDocument*는 다음과 같이 노드의 계층 구조를 생성합니다. 계층 구조의 루트는 *StockList* 노드입니다. *StockList*는 두 개의 *Stock* 태그에 해당하는 두 개의 자식 노드를 갖습니다. 이 두 개의 자식 노드 각각은 자신의 네 개의 자식 노드 (*name*, *price*, *symbol* 및 *shares*)를 갖습니다. 이 네 개의 자식 노드는 잎 노드 (leaf node)로서 동작합니다. 그들이 포함하는 텍스트는 각 잎 노드의 값으로 나타납니다.

참고 이러한 노드 분할은 DOM 구현이 XML 문서에 대한 노드를 생성하는 방식과 약간 다릅니다. 특히 DOM 파서는 태그로 표시된 요소 모두를 내부 노드로 처리합니다. 타입 텍스트 노드의 추가 노드는 *name*, *price*, *symbol* 및 *shares* 노드 값에 대해 생성됩니다. 그런 다음 이 텍스트 노드는 *name*, *price*, *symbol* 및 *shares* 노드의 자식으로 나타납니다.

각 노드는 XML 문서 컴포넌트의 *DocumentElement* 속성 값인 루트 노드에서 시작하여 *IXMLNode* 인터페이스를 통해 액세스됩니다.

노드의 값 사용

IXMLNode 인터페이스가 주어진 경우, *IsTextElement* 속성을 확인하여 내부 노드 (internal node) 또는 잎 노드를 나타내는지 여부를 확인할 수 있습니다.

- 잎 노드 (leaf node)를 나타낼 경우, *Text* 속성을 사용하여 그 값을 읽거나 설정할 수 있습니다.
- 내부 노드를 나타낼 경우, *ChildNodes* 속성을 사용하여 자식 노드에 액세스할 수 있습니다.

따라서 예를 들면 위의 XML 문서를 사용하여 다음과 같이 Inprise의 주식 가격을 읽을 수 있습니다.

```
InpriseStock := XMLDocument1.DocumentElement.ChildNodes[0];
```

```
Price := InpriseStock.ChildNodes['price'].Text;
```

노드의 속성 작업

노드가 속성을 포함하는 경우, *Attributes* 속성을 사용하여 작업할 수 있습니다. 기존 속성 이름을 지정하여 속성 값을 읽거나 변경할 수 있습니다. 다음과 같이 *Attributes* 속성을 설정할 때 새로운 속성을 지정하여 새로운 속성들을 추가할 수 있습니다.

```
InpriseStock := XMLDocument1.DocumentElement.ChildNodes[0];
InpriseStock.ChildNodes['shares'].Attributes['type'] := 'common';
```

자식 노드 추가 및 삭제

AddChild 메소드를 사용하여 자식 노드를 추가할 수 있습니다. *AddChild*는 XML 문서의 태그로 표시된 요소에 해당하는 새 노드를 생성합니다. 이러한 노드를 요소 노드라고 합니다.

새 요소 노드를 생성하려면 새 태그에 나타나는 이름을 지정하고 옵션으로 새 노드를 나타내는 위치를 지정합니다. 예를 들어, 다음 코드는 위 문서에 새로운 주식 목록을 추가합니다.

```
var
  NewStock:IXMLNode;
  ValueNode:IXMLNode;
begin
  NewStock := XMLDocument1.DocumentElement.AddChild('stock');
  NewStock.Attributes['exchange'] := 'NASDAQ';
  ValueNode := NewStock.AddChild('name');
  ValueNode.Text := 'Cisco Systems';
  ValueNode := NewStock.AddChild('price');
  ValueNode.Text := '62.375';
  ValueNode := NewStock.AddChild('symbol');
  ValueNode.Text := 'CSCO';
  ValueNode := NewStock.AddChild('shares');
  ValueNode.Text := '25';
end;
```

*AddChild*의 오버로드된 버전은 태그 이름이 정의된 네임스페이스(namespace)를 사용자가 지정할 수 있게 해줍니다.

ChildNodes 속성의 메소드를 사용하여 자식 노드를 삭제할 수 있습니다. *ChildNodes*는 노드의 자식들을 관리하는 *IXMLNodeList* 인터페이스입니다. *Delete* 메소드를 사용하여 위치나 이름으로 식별되는 하나의 자식 노드를 삭제할 수 있습니다. 예를 들어, 다음 코드는 위 문서에 나열된 마지막 주식을 삭제합니다.

```
StockList := XMLDocument1.DocumentElement;
StockList.ChildNodes.Delete(StockList.ChildNodes.Count - 1);
```

데이터 바인딩 마법사를 이용한 XML 문서 추출

TXMLDocument 컴포넌트 및 문서의 노드에 대해 표면화하는 *IXMLNode* 인터페이스만 사용하여 XML 문서 작업을 하거나 *TXMLDocument*도 없이 DOM 인터페이스를 사용하여 독점적으로 작업할 수도 있지만 XML 데이터 바인딩 마법사를 사용하면 훨씬 더 간단하고 읽기 쉬운 코드를 작성할 수 있습니다.

데이터 바인딩 마법사는 XML 스키마 또는 데이터 파일을 취하고 그 위에 매핑하는 인터페이스 집합을 생성합니다. 예를 들어, 주어진 XML 데이터의 경우 다음과 같습니다.

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

데이터 바인딩 마법사는 구현할 클래스와 함께 다음과 같은 인터페이스를 생성합니다.

```
ICustomer = interface(IXMLNode)
  property id: Integer read Getid write Setid;
  property name: DOMString read Getname write Setname;
  property phone: DOMString read Getphone write Setphone;
  function Getid:Integer;
  function Getname:DOMString;
  function Getphone:DOMString;
  procedure Setid(Value:Integer);
  procedure Setname(Value:DOMString);
  procedure Setphone(Value:DOMString);
end;
```

모든 자식 노드는 이름이 자식 노드의 태그 이름과 일치하고 값이 자식 노드의 인터페이스(자식이 내부 노드인 경우)이거나 앞 노드에 대한 자식 노드의 값인 속성에 매핑됩니다. 또한 모든 node attribute는 속성에 매핑됩니다. 즉, 속성 이름은 attribute 이름이고 속성 값은 attribute 값입니다.

XML 문서의 태그로 표시된 각 요소에 대한 인터페이스(및 구현 클래스) 생성 이외에 마법사는 루트 노드에 대한 인터페이스를 얻기 위해 전역 함수를 생성합니다. 예를 들어, 위의 XML이 루트 노드에 <Customers> 태그가 있는 문서에서 유래된 경우, 데이터 바인딩 마법사는 다음과 같은 전역 루틴을 생성합니다.

```
function GetCustomers(XMLDoc: TXMLDocument): ICustomers;
```

생성된 인터페이스를 사용하면 XML 문서의 구조를 더 직접적으로 반영하기 때문에 사용자의 코드를 단순화시킵니다. 예를 들어 다음과 같은 코드를 작성하는 대신,

```
CustName := XMLDocument1.DocumentElement.ChildNodes[0].ChildNodes['name'].Value;
```

사용자의 코드는 다음과 같습니다.

```
CustName := GetCustomers(XMLDocument1)[0].Name;
```

데이터 바인딩 마법사에서 생성된 인터페이스는 모두 *IXMLNode*에서 파생된다는 점에 유의하십시오. 이는 데이터 바인딩 마법사를 사용하지 않을 때와 동일한 방식으로 자식 노드를 추가하고 삭제할 수 있다는 것을 의미합니다(30-5 페이지의 "자식 노드 추가 및 삭제" 참조). 또한 자식 노드가 반복적인 요소를 나타낼 때(노드의 모든 자식이 동일한 타입인 경우) 부모 노드에는 추가적인 반복을 추가하기 위한 두 개의 메소드 *Add* 및

*Insert*가 주어집니다. 이 메소드들은 생성할 노드의 타입을 지정할 필요가 없기 때문에 *AddChild*를 사용하는 것보다 더 간단합니다.

XML 데이터 바인딩 마법사 사용

다음과 같은 방법으로 데이터 바인딩 마법사를 사용합니다.

- 1 File|New|Other를 선택하고 New 페이지의 아래쪽에서 XML Data Binding 레이블이 붙은 아이콘을 선택합니다.
- 2 XML 데이터 바인딩 마법사가 나타납니다.
- 3 마법사의 첫 번째 페이지에서 다음을 수행합니다.
 - 인터페이스를 생성하려는 XML 문서 또는 스키마를 지정합니다. 이는 예제 XML 문서, Document Type Definition(.dtd) 파일, Reduced XML Data(.xdr) 파일 또는 XML 스키마(.xsd) 파일일 수 있습니다.
- 4 Options 버튼을 클릭하여 마법사에서 인터페이스와 구현 클래스 생성 시 사용하고 자 하는 이름 지정 방식 및 파스칼 데이터 타입에 대해 스키마에서 정의된 타입의 기본 매핑을 지정합니다.
- 5 마법사의 두 번째 페이지로 이동합니다. 이 페이지는 사용자가 문서나 스키마의 모든 노드 타입에 대한 상세한 정보를 제공할 수 있도록 합니다. 문서의 노드 타입을 모두 보여 주는 트리 뷰가 왼쪽에 있습니다. 복잡한 노드(자식을 갖는 노드)의 경우, 트리 뷰는 자식 요소를 표시하도록 확장될 수 있습니다. 이 트리 뷰에서 노드를 선택하면 대화 상자의 오른쪽에 노드에 대한 정보가 나타나고 사용자는 마법사에서 노드를 처리하는 방법을 지정할 수 있습니다.
 - Source Name 컨트롤은 XML 스키마에 있는 노드 타입의 이름을 표시합니다.
 - Source Type 컨트롤은 XML 스키마에서 지정된 노드 값의 타입을 표시합니다.
 - Documentation 컨트롤은 스키마에 주석을 첨가하여 사용자가 노드의 사용이나 용도를 설명할 수 있게 합니다.
 - 마법사에서 선택된 노드에 대한 코드를 생성할 경우(즉, 마법사가 인터페이스 및 구현 클래스를 생성하는 복잡한 타입일 경우 또는 마법사가 복잡한 타입의 인터페이스에 속성을 생성하는 복잡한 타입의 자식 요소 중 하나인 경우), Generate Binding 체크 박스를 사용하여 마법사에서 노드에 대한 코드를 생성할지 여부를 지정할 수 있습니다. Generate Binding을 선택 해제한 경우, 마법사에서는 복잡한 타입에 대한 인터페이스나 구현 클래스를 생성하지 않거나 자식 요소 또는 속성에 대해서 부모 인터페이스에 속성을 만들지 않습니다.
 - Binding Options 섹션을 통해 사용자는 선택된 요소에 대해 마법사에서 생성한 코드에 영향을 줄 수 있습니다. 어떤 노드에서는 Identifier Name(생성된 인터페이스나 속성의 이름)을 지정할 수 있습니다. 또한 인터페이스의 경우 문서의 루트 노드를 나타내도록 지시해야 합니다. 속성을 나타내는 노드의 경우 속성의 타입을 지정할 수 있고 속성이 인터페이스가 아니면 읽기 전용 속성 여부를 지정할 수 있습니다.

- 6 일단 마법사에서 각 노드에 대해 생성하고자 하는 코드를 지정했으면 세 번째 페이지로 이동합니다. 이 페이지에서는 마법사에서 코드를 생성하는 방법에 대한 일부 전역 옵션을 선택할 수 있고 생성될 코드를 미리 볼 수 있으며 향후 사용을 위해 사용자의 선택을 저장하는 방법을 마법사에 알려 줄 수 있습니다.
 - 마법사에서 생성하는 코드를 미리 보려면 Binding Summary 목록에 있는 인터페이스를 선택하고 Code Preview 컨트롤에서 결과 인터페이스 정의를 봅니다.
 - Data Binding Settings를 사용하여 마법사에서 선택 사항을 저장하는 방법을 지시합니다. 그러한 설정을 문서에 연결된 스키마 파일(대화 상자의 첫 페이지에 지정된 스키마 파일)에 주석으로 저장하거나 마법사에서만 사용되는 독립적인 스키마 파일의 이름을 지정할 수 있습니다.
- 7 Finish를 클릭하면 데이터 바인딩 마법사에서는 XML 문서의 모든 노드 타입에 대한 인터페이스 및 구현 클래스를 정의하는 새로운 유닛을 생성합니다. 또한 *TXMLDocument* 객체를 가지고 데이터 계층 구조의 루트 노드에 대한 인터페이스를 반환하는 전역 함수를 만듭니다.

XML 데이터 바인딩 마법사가 생성하는 코드 사용

일단 마법사가 인터페이스와 구현 클래스 집합을 생성하면 문서의 구조 또는 마법사에 사용자가 제공한 스키마와 일치하는 XML 문서 작업을 할 수 있습니다. Delphi에 들어 있는 기본 제공 XML 컴포넌트만 사용할 때와 마찬가지로 사용자의 시작점은 컴포넌트 팔레트의 Internet 페이지에 있는 *TXMLDocument* 컴포넌트입니다.

XML 문서 작업을 하려면 다음 단계를 따릅니다.

- 1 *TXMLDocument* 컴포넌트를 폼이나 데이터 모듈에 놓습니다.
- 2 *FileName* 속성을 설정하여 *TXMLDocument*를 XML 문서에 바인딩합니다. (또 다른 방법으로는 런타임 시 *XML* 속성을 설정하여 XML의 문자열을 사용할 수 있습니다.)
- 3 사용자의 코드에서 XML 문서의 루트 노드에 대한 인터페이스를 얻기 위해 마법사가 생성한 전역 함수를 호출합니다. 예를 들어, XML 문서의 루트 요소가 <StockList> 태그였다면 기본적으로 마법사는 다음과 같은 *IStockList Type* 인터페이스를 반환하는 *GetStockList Type* 함수를 생성합니다.

```
var  
    StockList: IStockListType;  
begin  
    StockList := GetStockListType(XMLDocument1);
```

- 4 이 인터페이스는 루트 요소의 속성 (attribute) 들에 해당하는 속성 (property) 들뿐만 아니라 문서 루트 요소의 하위 노드에 해당하는 속성 (property) 들을 갖습니다. 이를 사용하여 XML 문서의 계층 구조를 통과하고 문서의 데이터 수정 등을 할 수 있습니다.
- 5 마법사에 의해 생성된 인터페이스를 사용하여 변경 내용을 저장하려면 *TXMLDocument* 컴포넌트의 *SaveToFile* 메소드를 호출하거나 *XML* 속성을 읽습니다.

Web Services 사용

Web Services는 World Wide Web과 같은 네트워크 상에서 게시 및 호출할 수 있는 모듈이 자체적으로 포함되어 있는 애플리케이션입니다. Web Services는 제공된 서비스를 설명하는 명확한 인터페이스를 제공합니다.

Web Services는 클라이언트와 서버 간의 느슨한 결합을 허용하도록 디자인되었습니다. 즉, 서버 구현에서 클라이언트가 특정 플랫폼이나 특정 프로그램 언어만 사용해야 하는 것은 아닙니다. 랭귀지-중립 방식으로 인터페이스를 정의하는 것 외에 여러 통신 메커니즘도 사용할 수 있도록 디자인되었습니다.

Delphi의 Web Services 지원은 SOAP(Simple Object Access Protocol)를 사용하여 작동하도록 디자인되었습니다. SOAP는 분산 환경에서 정보를 교환하기 위한 표준 경량급(lightweight) 프로토콜입니다. XML을 사용하여 원격 프로시저 호출을 인코딩하고 통신 프로토콜로서 대개 HTTP를 사용합니다. SOAP에 대한 자세한 내용은 다음 주소에서 사용 가능한 SOAP 사양을 참조하십시오.

<http://www.w3.org/TR/SOAP/>

참고 Delphi의 Web Services 지원은 SOAP 및 HTTP를 기반으로 하지만 프레임워크가 매우 일반적이므로 다른 인코딩 방식 및 통신 프로토콜을 사용하도록 확장할 수 있습니다.

Delphi의 SOAP 기반 기술은 Windows에서 사용할 수 있으며 나중에 Linux에서 구현될 것이므로 크로스 플랫폼의 분산 애플리케이션을 기초로 하여 만들 수 있습니다. 런타임 소프트웨어는 CORBA를 사용하여 애플리케이션을 배포할 때 필요하므로 특별한 클라이언트 런타임 소프트웨어를 설치하지 않아도 됩니다. 이 기술은 HTTP 메시지를 기반으로 하기 때문에 다양한 시스템에서 광범위하게 사용할 수 있다는 장점이 있습니다. Web Services 지원은 Delphi의 크로스 플랫폼 웹 서버 애플리케이션 아키텍처의 맨 위에서 만들어집니다.

Delphi를 사용하여 Web Services를 구현하는 서버와 그러한 서비스를 호출하는 클라이언트를 모두 만들 수 있습니다. Delphi로 서버 및 클라이언트 애플리케이션을 만들면 Web Services에 대한 인터페이스를 정의하는 단일 유닛을 공유할 수 있습니다. 또한 SOAP 메시지에 응답하는 Web Services를 구현하는 임의의 서버에 대한 Delphi 클라이언트와 임의의 클라이언트가 사용할 수 있는 Web Services를 게시하는 Delphi 서버를 작성할 수 있습니다.

Delphi를 사용하여 클라이언트나 서버를 작성하지 않으면 사용 가능한 인터페이스에 관한 정보와 WSDL(Web Services Definition Language) 문서를 사용하여 인터페이스를 호출하는 방법에 관한 정보를 게시하거나 import할 수 있습니다. 서버측에서는 애플리케이션이 Web Services를 설명하는 WSDL 문서를 게시할 수 있습니다. 클라이언트측에서는 마법사가 게시된 WSDL 문서를 import하여 필요한 인터페이스 정의 및 연결 정보를 제공할 수 있습니다.

Web Services를 지원하는 서버 작성

Delphi에서 Web Services를 지원하는 서버는 호출 가능한 인터페이스를 사용하여 만듭니다. 호출 가능한 인터페이스는 런타임 타입 정보(RTTI)를 포함하도록 컴파일된 인터페이스입니다. 이 RTTI는 클라이언트로부터 들어오는 메소드 호출을 제대로 마샬링(marshaling)할 수 있도록 이들을 해석할 때 사용합니다.

호출 가능한 인터페이스와 이를 구현하는 클래스 외에도 서버는 디스패처 및 호출자(invoker)라는 두 개의 컴포넌트를 필요로 합니다. 디스패처(*THTTTPSoapDispatcher*)는 들어오는 SOAP 메시지를 받아 호출자에 전달하는 자동 디스패칭 컴포넌트입니다. 호출자(*THTTTPSoapPascalInvoker*)는 SOAP 메시지를 해석하고, 호출 가능한 인터페이스를 식별하고, 해당 호출을 실행하고, 응답 메시지를 어셈블합니다.

참고 *THTTTPSoapDispatcher* 및 *THTTTPSoapPascalInvoker*는 SOAP 요청을 포함하는 HTTP 메시지에 응답할 수 있도록 디자인되었습니다. 하지만 기본 아키텍처는 매우 보편적이어서 다른 디스패처 및 호출자 컴포넌트의 대체 컴포넌트를 사용하여 그 밖의 프로토콜을 지원할 수 있습니다.

일단 호출 가능한 인터페이스 및 구현 클래스를 등록하면 디스패처와 호출자는 HTTP 요청 메시지의 SOAP Action 헤더에 있는 인터페이스들을 식별하는 모든 메시지를 자동으로 처리합니다.

Web Services 서버 구축

다음 단계에 따라 Web Services를 구현하는 서버 애플리케이션을 만듭니다.

- 1 Web Services를 구성하는 인터페이스를 정의합니다. 이러한 인터페이스 정의는 반드시 호출 가능한 인터페이스여야 합니다. 구현 클래스를 포함하는 유닛으로부터 분리된 사용자의 유닛에 인터페이스 정의를 만드는 것이 좋습니다. 이러한 방법으로 인터페이스를 정의하는 유닛은 서버 및 클라이언트 애플리케이션 모두에 포함될 수 있습니다. 이 유닛의 초기화 섹션에서 인터페이스를 등록하기 위한 코드를 추가합니다. 호출 가능한 인터페이스 작성 및 등록에 대한 자세한 내용은 31-3 페이지의 "호출 가능한 인터페이스 정의"를 참조하십시오.
- 2 인터페이스가 스칼라가 아닌 타입을 사용하는 경우, 이 타입이 올바르게 마샬링될 수 있는지 확인해야 합니다. Web Services 애플리케이션은 그 구조를 설명하는 런타임 타입 정보(RTTI)를 포함하는 특별한 객체를 사용하여 인터페이스만 처리할 수 있습니다. 복잡한 타입을 나타내기 위한 객체 만들기 및 등록에 대한 자세한 내용은 31-5 페이지의 "호출 가능한 인터페이스에서 복잡한 타입 사용"을 참조하십시오.

- 3 1 단계에서 정의한 호출 가능한 인터페이스를 구현하는 클래스를 정의하고 구현합니다. 또한 각 구현 클래스에 대해 클래스를 인스턴트화하는 팩토리 프로시저를 만들어야 할 경우도 있습니다. 이 유닛의 초기화 섹션에서 구현 클래스를 등록하기 위한 코드를 추가합니다. 이 프로세스는 31-6 페이지의 "구현 생성 및 등록"에서 다룹니다.
- 4 SOAP 요청을 실행하려고 할 때 애플리케이션이 예외를 발생시키면 예외는 메소드 호출 결과 대신 반환되는 SOAP 폴트 패킷 (fault packet) 안에서 자동으로 인코딩됩니다. 간단한 오류 메시지 이상의 많은 정보를 전달하려는 경우, 인코딩되어 클라이언트에게 전달되는 사용자 고유의 예외 클래스를 만들 수 있습니다. 이 내용은 31-7 페이지의 "Web Services에 대한 사용자 지정 예외 클래스 생성"에 설명되어 있습니다.
- 5 File|New|Other를 선택하고 Web Services 페이지에서 Web Services 애플리케이션 아이콘을 더블 클릭합니다. Web Services를 구현하려는 웹 서버 애플리케이션의 유형을 선택합니다. 다른 유형의 Web Services 애플리케이션에 대한 자세한 내용은 27-6 페이지의 "웹 서버 애플리케이션의 유형"을 참조하십시오.
- 6 마법사는 호출자 컴포넌트 (*THttpSOAPascalInvoker*) 및 디스패처 컴포넌트 (*THttpSoapDispatcher*)를 포함하는 새로운 Web Services 애플리케이션을 생성합니다. 호출자는 1 단계에서 등록한 인터페이스의 메소드와 SOAP 메시지 간에 변환합니다. 디스패처는 들어오는 SOAP 메시지에 자동으로 응답하고 이를 호출자에게 전달합니다. *WebDispatch* 속성을 사용하여 애플리케이션이 응답하는 HTTP 요청 메시지를 식별할 수 있습니다. 이는 애플리케이션을 향한 URL의 경로 부분을 나타내는 *PathInfo* 속성 및 요청 메시지에 대한 메소드 헤더를 나타내는 *MethodType* 속성을 설정해야 합니다.
- 7 Project|Add To Project를 선택하고 1~4 단계에서 만든 유닛을 웹 서버 애플리케이션에 추가합니다.
- 8 애플리케이션이 Delphi로 작성되지 않은 클라이언트를 사용하도록 하려면 호출 방법과 인터페이스를 정의하는 WSDL 문서를 게시합니다. Web Services 애플리케이션을 설명하는 WSDL 문서 생성 방법에 대한 자세한 내용은 31-7 페이지의 "Web Services 애플리케이션에 대한 WSDL 문서 생성"을 참조하십시오.

호출 가능한 인터페이스 정의

호출 가능한 인터페이스를 만들기 위해서는 단지 {\$M+} 컴파일러 옵션으로 인터페이스를 컴파일하면 됩니다. 호출 가능한 인터페이스의 자손 또한 호출 가능합니다. 하지만 호출 가능한 인터페이스가 호출 불가능한 다른 인터페이스의 자손이라면 Web Services 서버의 클라이언트는 호출 가능한 인터페이스와 그 자손에서 정의한 메소드만 호출할 수 있습니다. 호출 불가능한 조상에서 상속된 메소드는 타입 정보와 함께 컴파일되지 않으므로 클라이언트에 의해 호출될 수 없습니다.

Delphi는 Web Services 서버에 의해 클라이언트에게 노출되는 인터페이스의 기초로 사용할 수 있는 호출 가능한 기본 인터페이스인 *IInvokable*을 정의합니다. *IInvokable*은 {\$M+} 컴파일러 옵션을 사용하여 컴파일되어 그 자손과 함께 RTTI를 포함하도록 컴파일된다는 것을 제외하면 기본 인터페이스 (*IInterface*)와 동일합니다.

예를 들어, 다음 코드는 숫자 값을 인코딩 및 디코딩하는 두 개의 메소드가 포함된 호출 가능한 인터페이스를 정의합니다.

```
IEncodeDecode = interface(IInvokable)
  ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
  function EncodeValue(Value:Integer):Double; stdcall;
  function DecodeValue(Value:Double):Integer; stdcall;
end;
```

Web Services 애플리케이션이 호출 가능한 인터페이스를 사용하기 전에 호출 레지스트리에 등록해야 합니다. 서버에서 호출 레지스트리 항목은 호출자 컴포넌트 (*THHTPSOAPPascalInvoker*)가 인터페이스 호출을 실행하기 위해 사용하는 구현 클래스를 식별할 수 있도록 해줍니다. 클라이언트 애플리케이션에서 호출 레지스트리 항목은 컴포넌트가 호출 가능한 인터페이스를 식별하고 이를 호출하는 방법에 대한 정보를 찾을 수 있도록 해줍니다.

인터페이스를 정의하는 유닛의 초기화 섹션에서 인터페이스를 호출 레지스트리에 등록하기 위한 코드를 추가합니다. 호출 레지스트리에 액세스하려면 유닛의 `uses` 절에 `InvokeRegistry` 유닛을 추가합니다. `InvokeRegistry` 유닛은 호출 가능한 등록된 모든 인터페이스의 카탈로그, 그 구현 클래스, 구현 클래스의 인스턴스를 만드는 팩토리를 메모리에 유지하는 전역 변수 *InvRegistry*를 선언합니다.

작성을 완료하면 인터페이스를 정의하는 유닛은 다음과 같습니다.

```
unit EncodeDecode;

interface
type

IEncodeDecode = interface(IInvokable)
  ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
  function EncodeValue(Value:Integer):Double; stdcall;
  function DecodeValue(Value:Double):Integer; stdcall;
end;

implementation
uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(IEncodeDecode));
end.
```

Web Services의 인터페이스에는 가능한 모든 Web Services에 있는 모든 인터페이스 중에 이를 식별하는 네임스페이스가 있어야 하기 때문에 인터페이스를 등록할 때 호출 레지스트리는 인터페이스에 대한 네임스페이스를 자동으로 생성합니다. 기본 네임스페이스는 애플리케이션 (*AppNamespacePrefix* 변수), 인터페이스 이름, 이를 정의한 유닛의 이름을 고유하게 식별하는 문자열로부터 만들어집니다.

팁 호출 가능한 인터페이스를 정의하는 유닛은 이를 구현하는 클래스를 작성한 유닛과 별도로 두는 것이 좋습니다. 그러면 이 유닛은 클라이언트와 서버 애플리케이션 모두에 포함될 수 있습니다. 생성된 네임스페이스에는 인터페이스를 정의한 유닛의 이름이 포함되기 때문에 클라이언트 및 서버 애플리케이션 모두에서 동일한 유닛을 공유하면 자동으로 동일한 네임스페이스를 사용할 수 있습니다.

호출 가능한 인터페이스에서 복잡한 타입 사용

호출자 컴포넌트(*THTTPSOAPPascalInvoker*)는 호출 가능한 인터페이스에 스칼라 타입을 마샬링하는 방법을 자동적으로 인식합니다. 원격 가능 클래스 레지스트리(아래 참조)에 등록되어 있는 한 동적 배열도 처리할 수 있습니다. 하지만 정적 배열, 인터페이스, 레코드, 집합, 클래스와 같은 더 복잡한 타입으로 데이터를 전송하려면 추가 지원을 제공해야 합니다. 이러한 지원은 호출자가 SOAP 스트림의 데이터와 타입 값 사이의 변환에 사용할 수 있는 런타임 타입 정보(RTTI)를 포함하는 클래스의 형태여야 합니다.

호출 가능한 인터페이스에 복잡한 데이터 타입을 나타내기 위해 클래스를 정의하는 경우, 기본 클래스로 *TRemotable*을 사용합니다. 예를 들어, 일반적으로 매개변수로서 레코드를 전달할 경우에는 레코드의 모든 멤버가 새로운 클래스에서 *published* 속성인 *TRemotable* 자손을 대신 정의합니다.

새로운 *TRemotable* 자손의 값이 오브젝트 파스칼 스칼라 타입과 일치하지 않는 WSDL 문서의 스칼라 타입에 해당하면 새로운 클래스와 문자열 표현 방식 간에 변환할 수 있는 메소드를 제공할 수 있습니다. *XSToNative* 및 *NativeToXS* 메소드를 오버라이드하여 이러한 메소드를 제공합니다.

TRemotable 자손을 정의하는 유닛의 초기화 섹션에서 이 클래스를 원격 가능 클래스 레지스트리에 등록해야 합니다. *InvokeRegistry* 유닛을 *uses* 절에 추가하여 원격 가능 클래스 레지스트리에 액세스합니다. 이 유닛은 등록된 모든 원격 가능 클래스의 카탈로그와 그 값을 문자열로 전송할 수 있는지 여부의 표시를 유지하는 전역 변수인 *RemClassRegistry*를 선언합니다. 예를 들어, *XSBuiltIns* 유닛의 다음과 같은 줄에서 *TDateTime* 값을 나타내는 *TRemotable* 자손인 *TXSDateTime*을 등록합니다.

```
RemClassRegistry.RegisterXSClass(TXSDateTime, XMLSchemaNamespace, 'dateTime', True);
```

첫 번째 매개변수는 *TRemotable* 자손의 이름입니다. 두 번째 매개변수는 새 클래스의 네임스페이스를 고유하게 식별하는 URI(Uniform Resource Identifier)입니다. 빈 문자열인 경우, 레지스트리는 URI를 생성할 수 있습니다. 세 번째 매개변수는 클래스가 나타내는 데이터 타입의 이름입니다. 빈 문자열인 경우, 레지스트리는 단순히 클래스 이름을 사용합니다. 마지막 매개변수는 클래스 인스턴스의 값이 문자열로 전송될 수 있는지 여부(*XSToNative* 및 *NativeToXS* 메소드 구현 여부)를 나타냅니다.

팁 호출 가능한 인터페이스를 선언하고 등록하는 유닛을 비롯한 나머지 서버 애플리케이션과 분리된 유닛에 *TRemotable* 자손을 구현하고 등록하는 것이 좋습니다. 이러한 방법으로 클라이언트와 서버 모두에서 사용자의 타입을 정의하는 유닛을 사용할 수 있고 둘 이상의 인터페이스에 대한 타입을 사용할 수 있습니다.

매개변수로 동적 배열을 사용하는 경우, 이를 나타내기 위해 원격 가능 클래스를 만들 필요는 없지만 원격 가능 클래스 레지스트리에 등록해야 합니다. 예를 들어, 인터페이스가 다음과 같은 타입을 사용하는 경우입니다.

```
type
  TDateTimeArray = array of TXSDateTime;
```

동적 배열을 선언한 유닛의 초기화 섹션에 다음과 같은 등록을 추가해야 합니다.

```
RemClassRegistry.RegisterXSInfo(TypeInfo(TDateTimeArray), MyNameSpace, 'DTarray', False);
```

클래스 참조보다는 동적 배열의 타입 정보에 대한 포인터를 취하는 첫 번째 매개변수를 제외하면 이 매개변수들은 *RegisterXSClass*가 사용하는 것과 동일합니다.

구현 생성 및 등록

호출 가능 인터페이스에 대한 구현을 작성하는 가장 간단한 방법은 *TInvokableClass*의 자손인 클래스를 만드는 것입니다. 지원하는 호출 가능한 인터페이스를 포함하여 클래스 선언을 추가한 다음, *Ctrl+Shift+C*를 입력하여 class completion을 호출합니다. 인터페이스 멤버는 클래스 선언에 나타나고 빈 메소드는 유닛의 구현 섹션에 나타납니다.

예를 들어, "호출 가능한 인터페이스 정의" 위에서 선언된 인터페이스를 구현하는 구현 클래스에 대한 선언은 다음과 같습니다.

```
TEncodeDecode = class(TInvokableClass, IEncodeDecode)
protected
  function EncodeValue(Value:Integer):Double; stdcall;
  function DecodeValue(Value:Double):Integer; stdcall;
end;
```

이 클래스를 선언하는 유닛의 구현 섹션에서 *EncodeValue* 및 *DecodeValue* 메소드를 채워십시오.

일단 구현 클래스를 만들었으면 이 클래스를 호출 레지스트리에 등록해야 합니다. 호출 레지스트리는 이를 사용하여 게시된 인터페이스를 구현하는 클래스를 식별하고 호출자가 인터페이스를 호출해야 할 때 호출자 컴포넌트에 사용 가능하도록 합니다. 구현 클래스를 등록하려면 구현 유닛의 초기화 섹션에 대한 *InvRegistry* 전역 변수의 *RegisterInvokableClass* 메소드에 호출을 추가합니다.

```
InvRegistry.RegisterInvokableClass(TEncodeDecode);
```

또한 *TInvokableClass*의 자손이 아닌 구현 클래스를 만들 수도 있습니다. 하지만 이 경우에는 호출 레지스트리가 클래스의 인스턴스를 만들기 위해 호출할 수 있는 팩토리 프로시저를 제공해야 합니다.

팩토리 프로시저는 *TCreateInstanceProc* 타입이어야 합니다. 이는 구현 클래스의 인스턴스를 반환합니다. 프로시저가 새로운 인스턴스를 만들면 호출 레지스트리가 객체 인스턴스를 명시적으로 해제하지 않듯이 인터페이스의 참조 카운트가 0으로 떨어질 때 객체가 자신을 스스로 해제해야 합니다. 그에 대한 대안으로 팩토리 프로시저는 모든 호출자가 공유하는 전역 인스턴스에 대한 참조를 반환할 수 있습니다. 다음 코드는 후자의 방법을 설명합니다.

```
procedure CreateEncodeDecode(out obj:TObject);
begin
  if FEncodeDecode = nil then
  begin
    FEncodeDecode := TEncodeDecode.Create;
    {save a reference to the interface so that the global instance doesn't free itself }
    FEncodeDecodeInterface := FEncodeDecode as IEncodeDecode;
  end;
  obj := FEncodeDecode; { return global instance }
end;
```

팩토리 프로시저를 사용하는 경우 *RegisterInvokableClass* 메소드에 대한 두 번째 매개변수로 팩토리 프로시저를 제공합니다.

```
InvRegistry.RegisterInvokableClass(TEncodeDecode, CreateEncodeDecode);
```

Web Services에 대한 사용자 지정 예외 클래스 생성

SOAP 요청을 실행하려고 하는 동안 Web Services 애플리케이션이 예외를 발생시키면 메소드 호출의 결과 대신 반환하는 SOAP 폴트 패킷에서의 예외에 대한 정보를 자동으로 인코딩합니다. 그러면 클라이언트 애플리케이션이 예외를 발생시킵니다.

기본적으로 클라이언트 애플리케이션은 SOAP 폴트 패킷의 오류 메시지와 함께 단지 일반 예외 (*Exception*) 를 발생시킵니다. 하지만 *ERemotableException*의 자손인 예외 클래스를 사용하여 추가적인 예외 정보를 전송할 수 있습니다. 예외 클래스에 추가한 *published* 속성 값은 클라이언트가 동등한 예외를 발생시킬 수 있도록 SOAP 폴트 패킷에 포함됩니다.

ERemotableException 자손을 사용하려면 원격 가능 클래스 레지스트리에 이를 등록해야 합니다. 따라서 사용자의 *ERemotableException* 자손을 정의하는 유닛에서 *uses* 절에 *InvokeRegistry* 유닛을 추가하고 *RemClassRegistry* 전역 변수의 *RegisterXSClass* 메소드에 호출을 추가해야 합니다.

클라이언트가 사용자의 *ERemotableException* 자손을 정의하고 등록하는 동일한 유닛을 사용한다면 SOAP 폴트 패킷을 받을 때 클라이언트는 SOAP 폴트 패킷의 값에 설정된 모든 속성과 함께 적절한 예외 클래스의 인스턴스를 자동으로 발생시킵니다.

Web Services 애플리케이션에 대한 WSDL 문서 생성

호출 가능한 인터페이스, 복잡한 타입 정보를 나타내는 클래스, 사용자의 원격 가능 예외를 정의하고 등록하는 동일한 유닛을 Delphi 클라이언트 애플리케이션에 포함시키면 호출을 생성하여 Web Services 를 사용할 수 있습니다. 필요한 모든 작업은 Web Services 애플리케이션을 설치할 URL을 제공하는 것입니다.

하지만 Web Services 를 더 많은 클라이언트들이 사용하도록 만들 수 있습니다. 예를 들어, Delphi로 작성되지 않은 클라이언트를 가질 수도 있습니다. 여러 버전의 서버 애플리케이션을 배포하는 경우, 서버에 대해 하드 코딩된 단일 URL을 사용하기보다는 클라이언트가 서버 위치를 동적으로 찾도록 할 수도 있습니다. 이 경우 Web Services의 타입과 인터페이스를 설명하는 WSDL 문서를 이를 호출하는 방법에 관한 정보와 함께 게시할 수도 있습니다.

Web Services를 설명하는 WSDL 문서를 게시하기 위해서는 단지 웹 모듈에 *TWSDLHTMLPublish* 컴포넌트를 추가하면 됩니다. *TWSDLHTMLPublish*는 Web Services에 대한 WSDL 문서의 목록을 요청하는 수신 메시지에 자동으로 응답하는 자동 디스패칭 컴포넌트입니다. WSDL 문서의 목록을 액세스하는 데 URL 클라이언트의 경로 정보를 사용해야 한다고 지정하려면 *WebDispatch* 속성을 사용합니다. 그러면 웹 브라우저는 *WebDispatch* 속성의 경로 앞에 오는 서버 애플리케이션의 위치를 구성하는 URL을 지정하여 WSDL 문서의 목록을 요청할 수 있습니다. 이 URL은 다음과 같습니다.

```
http://www.myco.com/MyService.dll/WSDL
```

팁 그 대신 물리적 WSDL 파일을 원하면 웹 브라우저에 WSDL 문서를 표시한 다음 이를 저장하여 WSDL 문서 파일을 생성할 수 있습니다.

Web Services를 구현하는 동일한 애플리케이션의 WSDL 문서를 반드시 게시할 필요는 없습니다. WSDL 문서를 게시하는 간단한 애플리케이션을 만들려면 구현 객체를 포함하는 유닛을 생략하고 호출 가능한 인터페이스, 복잡한 타입을 나타내는 원격 가능 클래스, 모든 원격 가능 예외를 정의하고 등록하는 유닛만 포함시킵니다.

기본적으로 WSDL 문서를 게시할 경우, 경로가 달라도 WSDL 문서를 게시했던 곳과 동일한 URL에서 서비스가 사용 가능함을 나타냅니다. 여러 버전의 Web Services 애플리케이션을 배포하는 경우 또는 Web Services를 구현하는 것보다 다른 애플리케이션의 WSDL 문서를 게시하는 경우, Web Services의 위치에 관한 업데이트된 정보를 포함하는 WSDL 문서를 변경해야 합니다.

URL을 변경하기 위해 WSDL 관리자를 사용합니다. 첫 번째 단계는 관리자를 사용할 수 있도록 하는 것입니다. *TWSDLHTMLPublish* 컴포넌트의 *AdminEnabled* 속성을 *True*로 설정함으로써 이 작업을 수행합니다. 그런 다음 브라우저를 사용하여 WSDL 문서의 목록을 표시하면 문서를 관리할 버튼도 포함됩니다. WSDL 관리자를 사용하여 Web Services 애플리케이션을 배포했던 위치(URL)를 지정하십시오.

Web Services용 클라이언트 작성

Delphi는 SOAP 기반 바인딩을 사용하는 Web Services 호출에 대한 클라이언트측 지원을 제공합니다. 이러한 Web Services는 Delphi로 작성된 서버 또는 WSDL 문서에 있는 Web Services를 정의하는 다른 서버에 의해 제공될 수 있습니다.

서버가 Delphi로 작성되지 않은 경우, 서버를 설명하는 WSDL 문서를 먼저 import 할 수 있습니다. 이 과정은 아래에서 설명합니다. 서버가 Delphi를 사용하여 작성되었으면 WSDL 문서를 사용하지 않아도 됩니다. 복잡한 타입을 나타내는 원격 가능 클래스를 정의하고 Web Services 애플리케이션이 발생시킬 수 있는 원격 가능 예외를 정의하는 모든 유닛 외에도 프로젝트에 사용할 호출 가능한 인터페이스를 정의하는 모든 유닛을 추가할 수 있습니다.

참고 Delphi Web Services 서버에서 호출 가능한 인터페이스를 정의하는 유닛 작성에 대한 자세한 내용은 31-3 페이지의 "호출 가능한 인터페이스 정의"를 참조하십시오. 복잡한 타입에 대한 원격 가능 클래스를 정의하는 유닛 작성에 대한 자세한 내용은 31-5 페이지의 "호출 가능한 인터페이스에서 복잡한 타입 사용"을 참조하십시오. 원격 가능 예외를 정의하는 유닛 만들기에 대한 자세한 내용은 31-7 페이지의 "Web Services에 대한 사용자 지정 예외 클래스 생성"을 참조하십시오.

WSDL 문서 import하기

Delphi로 작성되지 않은 Web Services를 사용하려면 우선 서비스를 정의하는 WSDL 문서 또는 XML 스키마 파일을 import해야 합니다. Web Services importer는 사용하여 할 인터페이스와 타입을 정의하고 등록하는 유닛을 만듭니다.

Web Services importer를 사용하려면 File|New|Other를 선택하고 Web Services 페이지에서 Web Services importer라는 레이블이 있는 아이콘을 더블 클릭합니다. 나타나는 대화 상자에서 WSDL 문서 또는 XML 스키마 파일의 이름을 지정하거나 또는 문서가 게시된 URL을 제공합니다. Generate를 클릭하면 importer는 문서에 정의된 연산에 대해 호출 가능한 인터페이스를 정의하고 등록하는 새로운 유닛과 문서가 정의하는 타입에 대해 원격 가능 클래스를 정의하고 등록하는 새로운 유닛을 만듭니다.

WSDL 문서 또는 XML 스키마 파일이 오브젝트 파스칼 키워드이기도 한 식별자를 사용하면 importer는 생성 코드가 컴파일할 수 있도록 해당 이름을 자동으로 조정합니다. 복잡한 타입이 인라인에서 선언되면 importer는 호출 가능한 인터페이스와 동일한 유닛에서 해당 원격 가능 클래스를 정의하고 등록하는 코드를 추가합니다. 그렇지 않으면 타입은 별도의 유닛에서 정의되고 등록됩니다.

호출 가능 인터페이스 호출

호출 가능 인터페이스를 호출하려면 클라이언트 애플리케이션은 복잡한 타입을 구현하는 모든 원격 가능 클래스 및 호출 가능 인터페이스를 정의하는 유닛을 포함해야 합니다. 서버가 Delphi에서 작성된 경우, 서버 애플리케이션이 이러한 인터페이스와 클래스를 정의하고 등록하기 위해 사용하는 유닛과 동일한 유닛이어야 합니다. 호출 가능한 인터페이스 또는 원격 가능 클래스를 등록할 때 이를 고유하게 식별하는 URI(Uniform Resource Identifier)가 주어지기 때문에 동일한 유닛을 사용하는 것이 가장 좋습니다. 해당 URI는 인터페이스(또는 클래스)의 이름 및 이것이 정의되어 있는 유닛의 이름에서 파생됩니다. 클라이언트 및 서버가 동일한 URI를 사용하여 인터페이스(또는 클래스)를 등록하지 않으면 통신할 수 없습니다. 동일한 유닛을 사용하지 않는 경우, 인터페이스 및 구현 클래스를 등록하는 코드는 클라이언트와 서버가 동일한 네임스페이스를 사용하도록 네임스페이스 URI를 명시적으로 지정해야 합니다.

서버가 Delphi로 작성되지 않았거나 서버에서 사용한 동일한 유닛을 클라이언트에서 사용하지 않으려면 Web Services importer로 유닛을 만들 수 있습니다.

일단 클라이언트 애플리케이션에 호출 가능한 인터페이스의 선언이 있으면 원하는 인터페이스에 대해 *THTTPrIo*의 인스턴스를 만듭니다.

```
X := THTTPrIo.Create(nil);
```

그런 다음 서버 인터페이스를 식별하고 서버의 위치를 찾는 데 필요한 정보와 함께 *THTTPrIo* 객체를 제공하십시오. 이 정보를 제공하는 데는 두 가지 방법이 있습니다.

- 서버가 Delphi로 작성된 경우, 서버의 인터페이스는 인터페이스가 등록될 때 생성된 URI를 기반으로 자동으로 식별됩니다. 서버의 위치를 나타내려면 *URL* 속성을 설정하기만 하면 됩니다. URL의 경로 부분은 서버의 웹 모듈에 있는 디스패처 컴포넌트의 경로와 일치해야 합니다.

```
X.URL := 'http://www.myco.com/MyService.dll/SOAP/';
```

- 서버가 Delphi로 작성되지 않은 경우 *THTTPrIo*는 인터페이스에 대한 URI, Soap Action 헤더에 포함시켜야 하는 정보, WSDL 문서로부터의 서버 위치를 반드시 찾아야 합니다. *WSDLLocation*, *Service*, *Port* 속성을 사용하여 이를 수행하는 방법을 지정합니다.

```
X.WSDLLocation := 'Cryptography.wsdl';  
X.Service := 'Cryptography';  
X.Port := 'SoapEncodeDecode';
```

그런 다음 연산자 **as**를 사용하면 *THHTTPRIO*의 인스턴스를 호출 가능한 인터페이스로 변환할 수 있습니다. 이렇게 함으로써 연결된 인터페이스에 대한 *vtable*을 메모리에 동적으로 생성하여 인터페이스 호출이 만들어집니다.

```
InterfaceVariable := X as IEncodeDecode;  
Code := InterfaceVariable.EncodeValue(5);
```

*THHTTPRIO*는 호출 레지스트리를 사용하여 호출 가능한 인터페이스에 대한 정보를 얻습니다. 클라이언트 애플리케이션에 호출 레지스트리가 없거나 호출 가능한 인터페이스가 등록되어 있지 않으면 *THHTTPRIO*는 인메모리 *vtable*을 만들 수 없습니다.

32

소켓 작업

이 장에서는 TCP/IP 및 관련 프로토콜을 사용하여 다른 시스템과 통신할 수 있는 애플리케이션을 만들 수 있게 해주는 소켓 컴포넌트에 대하여 설명합니다. 소켓을 사용하면 필요한 네트워크 소프트웨어에 대해 자세히 모르더라도 다른 컴퓨터들과 연결해서 읽고 쓸 수 있습니다. 소켓은 TCP/IP 프로토콜에 기반한 연결을 제공하지만 매우 일반적으로 사용되는 것이므로 UDP (User Datagram Protocol), XNS (Xerox Network System), Digital의 DECnet 또는 Novell의 IPX/SPX 제품군과 같은 연관된 프로토콜과도 작동할 수 있습니다.

소켓을 사용하면 다른 시스템에 대해 읽기와 쓰기를 수행하는 네트워크 서버 또는 클라이언트 애플리케이션을 작성할 수 있습니다. 서버 또는 클라이언트 애플리케이션은 일반적으로 HTTP (Hypertext Transfer Protocol) 나 FTP (File Transfer Protocol) 와 같은 단일 서비스에만 사용됩니다. 서버 소켓을 사용하면 이러한 서비스 중 하나를 제공하는 애플리케이션을 해당 서비스를 사용하고자 하는 클라이언트 애플리케이션에 연결할 수 있습니다. 클라이언트 소켓을 사용하여 이러한 서비스 중 하나를 사용하는 애플리케이션은 해당 서비스를 제공하는 서버 애플리케이션에 연결할 수 있습니다.

서비스 구현

소켓은 네트워크 서버 또는 클라이언트 애플리케이션을 작성하는 데 필요한 서비스 중 하나를 제공합니다. 외부 서버에서도 HTTP 또는 FTP와 같은 대부분의 서비스를 쉽게 사용할 수 있습니다. 운영 체제와 함께 제공되는 경우도 있기 때문에 사용자가 직접 작성할 필요가 없습니다. 그러나 서비스가 구현되는 방법을 더 자세히 제어해야 하거나, 애플리케이션과 네트워크 통신 간의 보다 긴밀한 통합이 필요하거나 또는 특정 서비스에 사용할 수 있는 서버가 없는 경우에는 사용자의 고유한 서버나 클라이언트 애플리케이션을 만들 수 있습니다. 예를 들어, 분산 데이터셋에서 작업하는 경우 다른 시스템의 데이터베이스와 통신하기 위해 계층을 작성할 수 있습니다.

서비스 프로토콜 이해

네트워크 서버 또는 클라이언트를 작성하기 전에 애플리케이션에서 제공하거나 사용 중인 서비스를 이해해야 합니다. 대부분의 서비스는 네트워크 애플리케이션이 지원해야 하는 표준 프로토콜을 가집니다. HTTP, FTP, finger 또는 time과 같은 표준 서비스에 대한 네트워크 애플리케이션을 작성 중인 경우, 먼저 다른 시스템과 통신하는 데 사용할 프로토콜을 잘 알아야 합니다. 이를 위해서는 사용자가 제공하거나 사용하려는 특정 서비스에 대한 설명서를 참조하는 것이 좋습니다.

다른 시스템과 통신하는 애플리케이션에 대해 새로운 서비스를 제공하려면 우선 해당 서비스의 서버와 클라이언트를 위한 통신 프로토콜을 디자인하는 것이 필요합니다. 어떤 메시지들이 보내지는가? 이러한 메시지들이 어떻게 통합되는가? 정보를 인코딩하는 방법은 무엇인가?

애플리케이션과 통신

종종 네트워크 서버 또는 클라이언트 애플리케이션은 네트워크 소프트웨어와 서비스를 사용하는 애플리케이션 간의 계층(layer)을 제공합니다. 예를 들어, HTTP 서버는 콘텐츠를 제공하고 HTTP 요청 메시지에 응답하는 웹 서버 애플리케이션과 인터넷 사이에 자리합니다.

소켓은 네트워크 서버 또는 클라이언트 애플리케이션과 네트워크 소프트웨어 간의 인터페이스를 제공합니다. 사용자는 사용자의 애플리케이션과 이를 사용하는 클라이언트 간의 인터페이스를 제공해야 합니다. 사용자는 표준 협력업체 서버(예: Apache)의 API를 복사하거나 사용자 고유의 API를 디자인하고 게시할 수 있습니다.

서비스와 포트

일반적으로 대부분의 표준 서비스는 특정 포트 번호와 연결됩니다. 포트 번호에 대한 자세한 내용은 나중에 설명이 되며 여기서는 포트 번호를 서비스에 대한 숫자 코드로 간주합니다.

사용자가 표준 서비스를 구현한 경우, Linux 소켓 객체는 사용자가 서비스의 포트 번호를 찾을 수 있게 하는 메소드를 제공합니다. 사용자가 새로운 서비스를 제공한 경우, 사용자는 /etc/services 파일에 연결된 포트 번호를 지정할 수 있습니다. services 파일에 대한 더 자세한 내용은 Linux 문서를 참조하십시오.

소켓 연결 유형

소켓의 연결이 시작되는 방법과 로컬 소켓이 연결되는 대상에 따라 소켓 연결을 다음과 같이 세 가지 기본 유형으로 구분할 수 있습니다.

- 클라이언트 연결
- 리스닝(listening) 연결
- 서버 연결

일단 클라이언트 소켓과의 연결이 완료되면 서버 연결은 클라이언트 연결과 구분될 수 없습니다. 두 끝점은 그 기능이 동일하며 같은 타입의 이벤트를 받습니다. 단, 리스닝 연결은 끝점을 하나만 갖기 때문에 근본적으로 다릅니다.

클라이언트 연결

클라이언트 연결은 로컬 시스템의 클라이언트 소켓을 원격 시스템의 서버 소켓에 연결합니다. 클라이언트 연결은 클라이언트 소켓에 의해 시작됩니다. 먼저 클라이언트 소켓은 연결하고자 하는 서버 소켓을 기술해야 합니다. 그 다음, 클라이언트 소켓은 서버 소켓을 조회하고 서버를 찾을 때 연결을 요청합니다. 서버 소켓은 연결을 즉시 완료하지 않을 수도 있습니다. 서버 소켓은 클라이언트 요청의 대기열을 유지 관리하다가 시간이 되면 연결을 완료합니다. 클라이언트 연결을 승인하면 서버 소켓은 연결하고자 하는 서버 소켓에 대한 완전한 정보를 클라이언트 소켓으로 보내고 클라이언트가 연결을 완료합니다.

리스닝(listening) 연결

서버 소켓은 클라이언트를 찾지 않습니다. 그 대신 클라이언트 요청을 리스닝하는 수동적인 "반 연결(half connection)"을 구성합니다. 서버 소켓은 대기열을 리스닝 연결에 연결하며 대기열은 클라이언트 연결 요청이 들어올 때 그것을 기록합니다. 클라이언트 연결 요청을 승인하면 서버 소켓은 리스닝 연결이 계속 열린 상태에서 다른 클라이언트 요청을 받을 수 있도록 새 소켓을 구성하여 클라이언트에 연결합니다.

서버 연결

서버 연결은 리스닝 소켓이 클라이언트 요청을 승인할 때 서버 소켓에 의해 구성됩니다. 서버가 연결을 승인하면 클라이언트에 대한 연결을 완료하는 서버 소켓의 정보가 클라이언트에게 보내집니다. 클라이언트 소켓이 이 정보를 받고 연결을 완료하면 연결이 설정됩니다.

소켓 설명

네트워크 애플리케이션은 소켓을 사용하여 네트워크 상에서 다른 시스템과 통신할 수 있습니다. 각 소켓은 네트워크 연결의 끝점으로 보여질 수 있으며 다음 사항을 지정하는 주소를 가집니다.

- 실행되는 시스템
- 인식하는 인터페이스의 타입
- 연결을 위해 사용하는 포트

소켓 연결에 대한 전체 설명에는 연결의 두 끝점에 있는 소켓 주소가 포함됩니다. IP 주소나 호스트와 포트 번호로 소켓 양 끝점의 주소를 설명할 수 있습니다.

소켓 연결을 하기 전에 먼저 끝점을 구성하는 소켓을 완전히 설명해야 합니다. 일부 정보는 애플리케이션이 실행 중인 시스템에서 얻을 수 있습니다. 예를 들어, 클라이언트 소켓의 로컬 IP 주소는 운영 체제에서 얻을 수 있으므로 설명할 필요가 없습니다.

사용자가 제공해야 할 정보는 사용 중인 소켓 타입에 따라 달라집니다. 클라이언트 소켓은 연결하고자 하는 서버 정보를 제공해야 합니다. 리스닝 소켓은 제공하는 서비스를 나타내는 포트 정보를 제공해야 합니다.

호스트 설명

호스트는 소켓이 포함된 애플리케이션을 실행 중인 시스템입니다. 다음과 같이 표준 인터넷 도트 표시법으로 된 4개의 숫자(바이트) 값의 문자열인 IP 주소를 제공하여 소켓의 호스트 정보를 제공할 수 있습니다.

```
123.197.1.2
```

하나의 시스템이 하나 이상의 IP 주소를 지원할 수도 있습니다.

IP 주소는 종종 기억하기가 어려워서 잘못 입력하는 경우가 있습니다. 이 경우 대안은 호스트 이름을 사용하는 것입니다. 호스트 이름은 URL (Uniform Resource Locator) 에서 자주 볼 수 있는 IP 주소의 별칭입니다. 이것은 다음과 같이 도메인 이름 및 서비스를 포함한 문자열입니다.

```
http://www.ASite.com
```

대부분의 인터넷은 인터넷에 있는 시스템의 IP 주소에 대한 호스트 이름을 제공합니다. 사용자는 명령 프롬프트에서 다음의 명령을 실행하여 IP 주소(만약 이미 존재한다면)와 연결된 호스트 이름을 알 수 있습니다.

```
nslookup IPADDRESS
```

*IPADDRESS*는 사용자가 알고자 하는 IP 주소입니다. 사용자의 로컬 IP 주소에 호스트 이름이 없고 이를 갖고자 한다면 네트워크 관리자에게 문의하십시오.

서버 소켓은 호스트를 지정할 필요가 없습니다. 로컬 IP 주소는 시스템에서 읽을 수 있습니다. 로컬 시스템이 하나 이상의 IP 주소를 지원할 경우 서버 소켓은 모든 IP 주소에서 동시에 클라이언트 요청을 리스닝합니다. 서버 소켓이 연결을 승인하면 클라이언트 소켓은 원격 IP 주소를 제공합니다.

클라이언트 소켓은 호스트 이름이나 IP 주소로 원격 호스트를 지정해야 합니다.

호스트 이름 또는 IP 주소 선택

대부분의 애플리케이션은 호스트 이름을 사용하여 시스템을 지정합니다. 호스트 이름은 기억하기가 쉽고 입력 오류를 쉽게 확인할 수 있습니다. 또한 서버는 특정 호스트 이름과 연결된 시스템 또는 IP 주소를 변경할 수 있습니다. 호스트 이름을 사용하면 클라이언트 소켓은 호스트 이름이 나타내는 추상적 사이트가 새 IP 주소로 이동한 경우에도 해당 사이트를 찾을 수 있습니다.

호스트 이름이 없는 경우 클라이언트 소켓은 IP 주소를 사용하여 서버 시스템을 지정해야 합니다. IP 주소를 제공하면 서버 시스템을 더 빠르게 지정할 수 있습니다. 호스트 이

름이 제공되는 경우에는 소켓이 호스트 이름과 연결된 IP 주소를 검색한 후에만 서버 시스템을 찾을 수 있습니다.

포트 사용

IP 주소가 소켓 연결의 다른 끝에 있는 시스템을 찾는 데 필요한 정보를 제공하더라도 해당 시스템의 포트 번호가 필요합니다. 포트 번호가 없으면 시스템은 한 번에 하나의 연결만 구성할 수 있습니다. 포트 번호는 단일 시스템이 각 연결에 별도의 포트 번호를 제공하여 여러 연결을 동시에 호스트할 수 있도록 하는 고유 식별자입니다.

앞에서 이미 포트 번호는 네트워크 애플리케이션에 의해 구현된 서비스에 대한 숫자 코드를 설명한 바 있습니다. 이는 사실상 고정된 포트 번호에서 리스닝 서버 연결을 사용하여 클라이언트 소켓이 리스닝 서버를 찾을 수 있도록 하는 규칙일 뿐입니다. 서버 소켓은 자신이 제공하는 서비스와 연결된 포트 번호를 리스닝합니다. 서버 소켓이 클라이언트 소켓에 대한 연결을 승인하면 다른 임의의 포트 번호를 사용하는 별도의 소켓 연결을 만듭니다. 이와 같은 방법으로 리스닝 연결은 서비스와 연결된 포트 번호를 계속 리스닝할 수 있습니다.

클라이언트 소켓은 다른 소켓에 의해 발견되지 않아야 하므로 임의의 로컬 포트 번호를 사용합니다. 클라이언트 소켓은 서버 애플리케이션을 찾을 수 있도록 연결하고자 하는 서버 소켓의 포트 번호를 지정합니다. 종종 이 포트 번호는 원하는 서비스 이름을 지정하여 간접적으로 지정됩니다.

소켓 컴포넌트 사용

Internet 팔레트 페이지에는 사용자의 네트워크 애플리케이션이 다른 시스템과 연결을 구성할 수 있게 하고, 사용자가 그 연결을 통해 정보를 읽고 작성할 수 있게 하는 세 가지 소켓 컴포넌트가 들어 있습니다. 이 세 가지 소켓 컴포넌트는 다음과 같습니다.

- *TcpServer*
- *TcpClient*
- *UdpSocket*

각 소켓 컴포넌트에는 실제 소켓 연결의 끝점을 나타내는 소켓 객체가 연결됩니다. 소켓 컴포넌트가 소켓 객체를 사용하여 소켓 서버 호출을 캡슐화하므로 애플리케이션에서는 연결을 구성하거나 소켓 메시지를 관리하는 세부 사항에 대해 신경 쓸 필요가 없습니다.

사용자 대신 소켓 컴포넌트가 만든 연결의 세부 사항에 대해 사용자 지정하려는 경우, 소켓 객체의 속성, 이벤트 및 메소드를 사용할 수 있습니다.

연결 정보 얻기

클라이언트 또는 서버 소켓에 대한 연결을 완료한 후, 연결에 대한 정보를 얻기 위해 사용자의 소켓 컴포넌트와 연결된 클라이언트 또는 서버 소켓 객체를 사용할 수 있습니다. 로컬 클라이언트 또는 서버 소켓에 사용되는 주소와 포트 번호를 결정하는 데 *LocalHost* 및 *LocalPort* 속성을 사용하고, 원격 클라이언트 또는 서버 소켓에 의해 사

용되는 주소와 포트 번호를 결정하는 데 *RemoteHost* 및 *RemotePort* 속성을 사용합니다. *GetSocketAddr* 메소드를 사용하면 호스트 이름과 포트 번호에 기반한 유효한 소켓 주소를 빌드합니다. *LookupPort* 메소드를 사용하면 포트 번호를 찾을 수 있습니다. *LookupProtocol* 메소드를 사용하면 프로토콜 번호를 찾을 수 있습니다. *LookupHostName* 메소드를 사용하면 호스트 시스템의 IP 주소에 기반한 호스트 이름을 찾을 수 있습니다.

소켓을 오고 가는 네트워크 트래픽을 보려면 *BytesSent* 및 *BytesReceived* 속성을 사용합니다.

클라이언트 소켓 사용

TcpClient 또는 *UdpSocket* 컴포넌트를 폼이나 데이터 모듈에 추가하여 사용자의 애플리케이션을 TCP/IP 또는 UDP 클라이언트로 바꿉니다. 클라이언트 소켓을 사용하면 연결하고자 하는 서버 소켓과 서버가 제공하게 할 서비스를 지정할 수 있습니다. 원하는 연결 정보를 제공했다면 클라이언트 소켓 컴포넌트를 사용하여 서버에 대한 연결을 완료할 수 있습니다.

각 클라이언트 소켓 컴포넌트는 연결에서의 클라이언트 끝점을 나타내기 위해 하나의 클라이언트 소켓 객체를 사용합니다.

원하는 서버 지정

클라이언트 소켓 컴포넌트에는 사용자가 연결하고자 하는 서버 시스템과 포트를 지정할 수 있게 해주는 여러 속성들이 있습니다. *RemoteHost* 속성을 사용하여 호스트 이름 또는 IP 주소로 원격 호스트 서버를 지정합니다.

서버 시스템 지정에 추가해서 사용자의 클라이언트 소켓이 연결하게 될 서버 시스템의 포트를 지정해야 합니다. *RemotePort* 속성을 사용하여 대상 서비스의 이름을 지정함으로써 서버 포트 번호를 직접적 또는 간접적으로 지정할 수 있습니다.

연결 구성

일단 클라이언트 소켓 컴포넌트의 속성을 설정하여 연결하고자 하는 서버 정보를 제공했다면 *Open* 메소드를 호출하여 런타임에 연결을 구성할 수 있습니다. 애플리케이션이 시작될 때 자동으로 연결을 구성하도록 하려면 디자인 타임에 Object Inspector에서 *Active* 속성을 *True*로 설정합니다.

연결 정보 얻기

서버 소켓에 대한 연결을 완료한 후 클라이언트 소켓 컴포넌트와 연결된 클라이언트 소켓 객체를 사용하여 연결에 대한 정보를 얻을 수 있습니다. *LocalHost* 및 *LocalPort* 속성을 사용하면 클라이언트 및 서버 소켓이 연결의 끝점을 구성하기 위해 사용한 주소와 포트 번호를 정할 수 있습니다. *Handle* 속성을 사용하면 소켓 호출을 생성할 때 사용하는 소켓 연결에 대한 핸들을 얻을 수 있습니다.

연결 끊기

소켓 연결을 통한 서버 애플리케이션과의 통신이 끝나면 *Close* 메소드를 호출하여 연결을 종료할 수 있습니다. 서버측에서 연결이 끊어질 수도 있습니다. 이 경우에 사용자는 *OnDisconnect* 이벤트를 통해 공지를 받게 됩니다.

서버 소켓 사용

서버 소켓 컴포넌트 (*TcpServer* 또는 *UdpSocket*)를 폼이나 데이터 모듈에 추가하여 사용자의 애플리케이션을 IP 서버로 바꿉니다. 서버 소켓을 사용하면 제공 중인 서비스를 지정하거나 클라이언트 요청을 리스닝하는 데 사용할 포트를 지정할 수 있습니다. 서버 소켓 컴포넌트를 사용하면 클라이언트 연결 요청을 리스닝하고 승인할 수 있습니다.

각각의 서버 소켓 컴포넌트는 단일의 서버 소켓 객체를 사용하여 리스닝 연결의 서버 끝점을 나타냅니다. 또한 각 컴포넌트는 서버가 받아들이는 클라이언트 소켓에 대한 각 활성 연결의 서버 끝점을 위해 서버 클라이언트 소켓 객체를 사용합니다.

포트 지정

서버 소켓이 클라이언트 요청을 리스닝하려면 서버가 리스닝할 포트를 지정해야 합니다. 이 포트를 지정하려면 *LocalPort* 속성을 사용합니다. 서버 애플리케이션이 일반적으로 특정 포트 번호와 연결된 표준 서비스를 제공하는 경우에 *LocalPort* 속성을 사용하여 서비스 이름을 지정할 수도 있습니다. 포트 번호를 지정할 때에는 입력 오류가 발생하기 쉬우므로 포트 번호 대신 서비스 이름을 사용하는 것이 좋습니다.

클라이언트 요청 리스닝(listening)

일단 서버 소켓 컴포넌트의 포트 번호를 설정했다면 *Open* 메소드를 호출하여 런타임 시 리스닝 연결을 구성할 수 있습니다. 애플리케이션이 시작될 때 자동으로 리스닝 연결을 구성하도록 하려면 디자인 타임에 Object Inspector에서 *Active* 속성을 *True*로 설정합니다.

클라이언트에 연결

리스닝(listening) 중인 서버 소켓 컴포넌트는 자동으로 클라이언트 연결 요청을 승인합니다. 사용자는 이러한 자동 승인이 *OnAccept* 이벤트에서 발생할 때마다 공지를 받습니다.

서버 연결 닫기

리스닝 연결을 종료하려면 *Close* 메소드를 호출하거나 *Active* 속성을 *False*로 설정합니다. 그러면 클라이언트 애플리케이션에 대해 열려 있는 모든 연결이 종료되고 아직 승인되지 않고 보류 중인 연결이 취소되고 나서 리스닝 연결이 종료됩니다. 그런 다음 서버 소켓 컴포넌트는 더 이상 새로운 연결을 승인하지 않습니다.

TCP 클라이언트가 사용자의 서버 소켓에 대한 개별적인 연결을 종료하면 *OnDisconnect* 이벤트에서 사용자에게 이러한 사실을 알립니다.

소켓 이벤트에 응답

소켓을 사용하는 애플리케이션을 작성할 때 프로그램의 어디에서든지 소켓을 읽거나 쓸 수 있습니다. 소켓이 열린 후 사용자의 프로그램에서 *SendBuf*, *SendStream* 또는 *Sendln* 메소드를 사용하여 소켓에 작성할 수 있습니다. 유사한 이름의 메소드, 즉 *ReceiveBuf* 및 *ReceiveIn*을 이용하여 소켓을 읽을 수 있습니다. *OnSend* 및 *OnReceive* 이벤트는 소켓에 무언가 작성하고 읽을 때마다 트리거됩니다. 필터링을 위해 이러한 이벤트를 사용할 수 있습니다. 사용자가 읽기 또는 쓰기 작업을 할 때마다 읽기 또는 쓰기 이벤트가 트리거됩니다.

연결로부터 오류 메시지를 받으면 클라이언트 소켓과 서버 소켓은 모두 오류 이벤트를 생성합니다.

소켓 컴포넌트는 또한 연결을 열고 완료하는 과정에서 두 개의 이벤트를 받습니다. 애플리케이션이 소켓이 열리는 방법에 대해 영향을 줄 필요가 있을 경우, *SendBuf* 및 *ReceiveBuf* 메소드를 사용하여 이러한 클라이언트 이벤트 또는 서버 이벤트에 응답합니다.

오류 이벤트

연결 중에 오류 메시지를 받으면 클라이언트와 서버 소켓은 *OnError* 이벤트를 생성합니다. *OnError* 이벤트 핸들러를 작성하면 이러한 오류 메시지에 응답할 수 있습니다. 이 이벤트 핸들러에는 다음에 관한 정보가 전달됩니다.

- 오류 공지를 받은 소켓 객체
- 오류 발생 시 소켓이 시도하던 작업
- 오류 메시지가 제공한 오류 코드

이벤트 핸들러에서 오류에 응답할 수 있고 오류 코드를 0으로 변경하여 소켓이 예외를 일으키는 것을 방지할 수 있습니다.

클라이언트 이벤트

클라이언트 소켓이 연결되면 다음 이벤트가 발생합니다.

- 이벤트 공지를 위해 소켓이 설정되고 초기화됩니다.
- 서버와 서버 소켓이 만들어진 후 *OnCreateHandle* 이벤트가 발생합니다. *Handle* 속성을 통해 사용할 수 있는 소켓 객체는 연결의 다른 끝을 구성할 서버나 클라이언트 소켓에 대한 정보를 제공할 수 있습니다. 이 때 연결에 사용되는 실제 포트를 처음으로 얻을 수 있으며 실제 포트는 연결을 승인한 리스닝 소켓의 포트와 다를 수도 있습니다.
- 연결 요청이 서버에 의해 승인되고 클라이언트 소켓에 의해 완료됩니다.
- 연결이 설정되면 *OnConnect* 공지 이벤트가 발생합니다.

서버 이벤트

서버 소켓 컴포넌트는 두 가지 타입의 연결, 즉 리스닝 연결과 클라이언트 애플리케이션에 대한 연결을 구성합니다. 이러한 각 연결이 구성되는 동안 서버 소켓은 이벤트를 받습니다.

리스닝 시의 이벤트

리스닝 연결이 구성되기 바로 전에 *OnListening* 이벤트가 발생합니다. *Handle* 속성을 사용하면 리스닝에 대해 소켓이 열리기 전에 소켓을 변경할 수 있습니다. 예를 들어, 서버가 리스닝을 위해 사용하는 IP 주소를 제한하려면 *OnListening* 이벤트 핸들러에서 이 작업을 처리합니다.

클라이언트 연결 시의 이벤트

서버 소켓이 클라이언트 연결 요청을 승인하면 다음 이벤트가 발생합니다.

- *OnAccept* 이벤트가 발생하고 새 *TTcpClient* 객체가 이벤트 핸들러에 전달됩니다. 이것은 *TTcpClient*의 속성을 사용하여 클라이언트에 대한 연결의 서버 끝점에 대한 정보를 얻을 수 있는 첫 번째 시점입니다.
- *BlockMode*가 *bmThreadBlocking*일 경우 *OnGetThread* 이벤트가 발생합니다. *TServerSocketThread*의 고유한 사용자 지정 자산을 제공하려는 경우 *OnGetThread* 이벤트 핸들러에서 사용자 지정 자산을 만들어서 *TServerSocketThread* 대신 사용할 수 있습니다. 스레드 초기화를 수행하거나 스레드가 연결을 통해 읽거나 쓰기를 시작하기 전에 소켓 API를 호출하려면 *OnGetThread* 이벤트 핸들러도 사용해야 합니다.
- 클라이언트가 연결을 완료한 후 *OnAccept* 이벤트가 발생합니다. 비차단 서버의 경우에는 이 때 소켓 연결을 통해 읽기 또는 쓰기를 시작할 수도 있습니다.

소켓 연결을 통한 읽기 및 쓰기

소켓 연결을 통해 정보를 읽거나 쓰기 위하여 다른 시스템에 대한 소켓 연결을 구성합니다. 읽거나 쓸 대상이 되는 정보 또는 정보를 읽거나 쓰는 시기는 소켓 연결과 연결된 서비스에 따라 달라집니다.

소켓을 통한 읽기 및 쓰기는 비동기적으로 발생할 수 있기 때문에 네트워크 애플리케이션에서 다른 코드의 실행이 차단되지 않습니다. 이를 비차단 연결 (non-blocking connection)이라고 합니다. 또한 차단 연결을 구성할 수도 있는데 다음 코드 줄을 실행하기 전에 애플리케이션은 읽기 또는 쓰기가 완료되기를 기다립니다.

비차단 연결(Non-blocking connections)

비차단 연결은 비동기적으로 읽고 쓰기 때문에 네트워크 애플리케이션에서 데이터 전송이 다른 코드의 실행을 차단하지 않습니다. 클라이언트 또는 서버 소켓에 대해 비차단 연결을 만들려면 *BlockMode* 속성을 *bmNonBlocking*으로 설정합니다.

비차단 연결인 경우, 소켓은 읽기 및 쓰기 이벤트를 통해 연결의 다른 끝에 있는 소켓이 정보를 읽거나 쓰려는 시기를 알 수 있습니다.

읽기 및 쓰기 이벤트

비차단 소켓은 연결을 통해 읽거나 써야 할 경우 읽기 및 쓰기 이벤트를 생성합니다. 사용자는 *OnReceive* 또는 *OnSend* 이벤트 핸들러에서 이러한 공지에 응답할 수 있습니다.

소켓 연결과 연결된 소켓 객체는 이벤트 핸들러를 읽거나 쓰기 위한 매개변수로 제공됩니다. 소켓 객체는 사용자가 연결을 통해 정보를 읽거나 쓸 수 있도록 하는 여러 메소드를 제공합니다.

소켓 연결에서 정보를 읽으려면 *ReceiveBuf* 또는 *ReceiveIn* 메소드를 사용합니다. 소켓 연결에 정보를 쓰려면 *SendBuf*, *SendStream* 또는 *SendIn* 메소드를 사용합니다.

차단 연결(Blocking connections)

연결이 차단된 경우, 소켓은 소켓 연결로부터 공지를 수동적으로 기다리는 대신 연결을 통해 읽기 또는 쓰기를 시작해야 합니다. 읽기 및 쓰기가 발생하는 시기가 연결의 끝에서 결정될 경우 차단 소켓을 사용합니다.

클라이언트 또는 서버 소켓의 경우, *BlockMode* 속성을 *bmBlocking*으로 설정하여 차단 연결을 구성합니다. 사용자는 애플리케이션이 연결을 통한 읽기 또는 쓰기가 완료되기를 기다리는 동안 다른 스레드에서 코드를 계속 실행할 수 있도록 클라이언트 애플리케이션이 수행하는 작업에 따라 읽기 또는 쓰기를 위한 새 실행 스레드를 만들 수 있습니다.

서버 소켓의 경우에는 *BlockMode* 속성을 *bmBlocking* 또는 *bmThreadBlocking*으로 설정하여 차단 연결을 구성합니다. 소켓이 연결을 통한 정보 읽기 또는 쓰기가 완료되기를 기다리는 동안 차단 연결이 다른 모든 코드의 실행을 보류하기 때문에 서버 소켓 컴포넌트가 *BlockMode*가 *bmThreadBlocking*일 경우 항상 모든 클라이언트 연결에 대해 새 실행 스레드를 생성합니다. *BlockMode*가 *bmBlocking*이면 새 연결이 설정될 때까지 프로그램 실행이 차단됩니다.

IV

COM 기반 애플리케이션 개발

"COM 기반 애플리케이션 개발"에 포함된 장에서는 Automation 컨트롤러, Automation 서버, ActiveX 컨트롤 및 COM+ 애플리케이션 등의 COM 기반 애플리케이션을 작성하는 데 필요한 개념을 설명합니다.

참고 COM 클라이언트에 대한 지원은 모든 Delphi 에디션에서 사용할 수 있습니다. 하지만 서버를 생성하려면 전문가용 또는 기업용 에디션이 필요합니다.

33

COM 기술 개요

Delphi는 마법사 및 클래스를 제공하여 Microsoft의 COM(Component Object Model)을 기반으로 하는 애플리케이션을 쉽게 구현할 수 있게 합니다. 이러한 마법사를 이용하여 COM 기반 클래스 및 컴포넌트를 만들어 애플리케이션 내에서 사용하거나 COM 객체, Active Server Object를 포함하는 Automation 서버, ActiveX 컨트롤 또는 ActiveForm를 구현하는 완전한 기능을 갖는 COM 클라이언트나 서버를 만들 수 있습니다.

참고 ActiveX, COM+ 및 컴포넌트 팔레트의 Servers 탭에 있는 것과 같은 COM 컴포넌트는 CLX 애플리케이션에서 사용할 수 없습니다. 이 기술은 Windows에서만 사용할 수 있으며 크로스 플랫폼을 지원하지 않습니다.

COM은 랭귀지와 무관하게 사용할 수 있는 소프트웨어 컴포넌트 모델로서 Windows 플랫폼에서 실행되는 소프트웨어 컴포넌트와 애플리케이션 간의 상호 작용을 가능하게 합니다. COM의 핵심은 명확하게 정의되어 있는 인터페이스를 통해 컴포넌트들 사이에서, 애플리케이션들 사이에서 또 클라이언트와 서버 사이에서의 통신을 가능하게 한다는 것입니다. 인터페이스를 통해 클라이언트는 COM 컴포넌트가 런타임 시 지원하는 기능을 알아볼 수 있습니다. 사용자 컴포넌트에 기능을 추가하려면 원하는 기능에 대한 추가 인터페이스를 간단히 추가하면 됩니다.

애플리케이션은 애플리케이션과 동일한 컴퓨터에 있거나 DCOM(Distributed COM)이라는 메커니즘을 사용하여 네트워크 상의 다른 컴퓨터에 있는 COM 컴포넌트의 인터페이스에 액세스할 수 있습니다. 클라이언트, 서버 및 인터페이스에 대한 자세한 내용은 33-3 페이지의 "COM 애플리케이션의 구성 요소"를 참조하십시오.

이 장에서는 Automation 및 ActiveX 컨트롤이 작성되는 기본 기술에 대한 개념적인 개요를 제공합니다. 그 이후의 장에서는 Delphi에서 Automation 객체 및 ActiveX 컨트롤을 만드는 것에 대한 자세한 정보를 제공합니다.

사양 및 구현으로서의 COM

COM은 사양이면서 동시에 구현입니다. COM 사양은 객체를 만드는 방법과 객체가 서로 통신하는 방법을 정의합니다. 이러한 사양에 따라서 COM 객체는 다른 랭귀지로 작성될 수 있으며 다른 프로세스 공간 및 다른 플랫폼에서 실행될 수 있습니다. 객체는 작

성된 사양만 준수하면 통신할 수 있습니다. 이렇게 하면 컴포넌트로서의 레거시 코드를 객체 지향 랭귀지로 구현되는 새 컴포넌트와 통합할 수 있습니다.

COM 구현은 Win32 하위 시스템에 기본 제공되며 작성된 사양을 지원하는 많은 핵심 서비스가 제공됩니다. COM 라이브러리에는 COM 객체의 핵심 기능을 정의하는 표준 인터페이스 집합과 COM 객체를 만들고 관리할 목적으로 디자인된 작은 API 함수 집합이 들어 있습니다.

애플리케이션에서 Delphi 마법사와 VCL 객체를 사용할 때는 COM 사양의 Delphi 구현을 사용하고 있습니다. 또한 Delphi는 Active Documents와 같이 직접 구현되지 않는 기능에 COM 서비스에 대한 래퍼를 제공합니다. ComObj 유닛에서 정의된 이러한 래퍼와 API 정의는 AxCtrls 유닛에서 찾을 수 있습니다.

참고 Delphi의 인터페이스 및 랭귀지는 COM 사양을 준수합니다. Delphi는 Delphi ActiveX 프레임워크(DAX)라는 클래스 집합을 사용하여 COM 사양을 준수하는 객체를 구현합니다. 이러한 클래스는 AxCtrls, OleCtrls 및 OleServer 유닛에 있습니다. 또한 COM API에 대한 파스칼 인터페이스는 ActiveX.pas 및 ComSvcs.pas에 있습니다.

COM 확장

COM이 발전하면서 기본 COM 서비스 수준을 넘어 확장되었습니다. COM은 Automation, ActiveX 컨트롤, Active Documents 및 Active Directories와 같은 다른 기술들의 기본 역할을 합니다. COM 확장에 대한 자세한 내용은 33-10 페이지의 "COM 확장"을 참조하십시오.

또한 대규모 분산 환경에서 작업할 때는 트랜잭션 COM 객체를 만들 수 있습니다. Windows 2000 이전 버전에서 이 객체는 COM 설계에 포함되지 않았으며 오히려 MTS (Microsoft Transaction Server) 환경에서 실행되었습니다. Windows 2000의 출현으로 이러한 지원은 COM+로 통합되었습니다. 트랜잭션 객체에 대해서는 39장 "MTS 또는 COM+ 객체 생성"에서 자세히 설명합니다.

Delphi는 마법사를 제공하여 Delphi 환경에서 위의 기술을 통합하는 애플리케이션을 쉽게 구현합니다. 자세한 내용은 33-18 페이지의 "마법사로 COM 객체 구현"을 참조하십시오.

COM 애플리케이션의 구성 요소

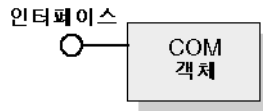
COM 애플리케이션을 구현할 때는 다음을 제공해야 합니다.

- COM 인터페이스** 객체가 외부에서 그 서비스를 클라이언트에 표시하는 방식입니다. COM 객체는 관련된 메소드 및 속성의 각 집합에 대한 인터페이스를 제공합니다. COM 속성은 VCL 객체에서의 속성과 다릅니다. COM 속성은 항상 읽기 및 쓰기 액세스 메소드를 사용합니다.
- COM 서버** COM 객체에 대한 코드를 포함하는 EXE, DLL 또는 OCX와 같은 모듈입니다. 객체 구현은 서버에 상주합니다. COM 객체는 하나 이상의 인터페이스를 구현합니다.
- COM 클라이언트** 서버에서 요청한 서비스를 가져오도록 인터페이스를 호출하는 코드입니다. 클라이언트는 인터페이스를 통해 서버에서 무엇을 가져오려는 것인지는 알지만 서버가 어떻게 서비스를 제공하는가 하는 내부적인 내용은 알 수 없습니다. Delphi를 통해 사용자는 Word 문서나 PowerPoint 슬라이드와 같은 COM 서버를 컴포넌트 팔레트의 컴포넌트로 설치하여 클라이언트를 쉽게 만들 수 있습니다. 이렇게 하면 사용자는 서버에 연결하여 Object Inspector를 통해 이벤트를 훔칠 수 있습니다.

COM 인터페이스

COM 클라이언트는 COM 인터페이스를 통해 객체와 통신합니다. 인터페이스는 서비스의 프로바이더(서버 객체)와 그 클라이언트 사이의 통신을 제공하는 논리적으로나 의미상으로 연관 있는 루틴의 그룹입니다. 그림 33.1은 COM 인터페이스를 표현하는 표준 방식을 보여 줍니다.

그림 33.1 COM 인터페이스



예를 들어, 모든 COM 객체는 COM 객체에서 사용할 수 있는 인터페이스를 클라이언트에게 알려 주는 기본 인터페이스, *IUnknown*을 구현합니다.

객체에는 인터페이스가 여러 개 있을 수 있으며 각 인터페이스는 기능을 구현합니다. 인터페이스는 인터페이스가 제공하는 서비스를 클라이언트에게 전달하는 방법을 제공하는데 객체가 이 서비스를 제공하는 방법 또는 위치에 대한 구현 정보는 제공하지 않습니다.

COM 인터페이스의 핵심적인 특징은 다음과 같습니다.

- 일단 게시되면 인터페이스는 '불변', 즉 변경되지 않습니다. 인터페이스에 의존하여 특정 기능 집합을 제공할 수 있습니다. 추가 기능은 추가 인터페이스에서 제공합니다.

- 규칙에 따라 COM 인터페이스 식별자는 대문자 I로 시작하고 *IMalloc*이나 *IPersist*와 같이 인터페이스를 정의하는 상징적인 이름입니다.
- 인터페이스는 임의로 생성되는 128비트 숫자인 **GUID(Globally Unique Identifier)**라는 고유한 ID를 갖도록 되어 있습니다. 인터페이스 GUID는 **인터페이스 식별자(IID)**라고 합니다. 인터페이스 GUID는 한 제품의 다른 버전이나 다른 제품 사이에서 발생하는 이름 충돌을 없앱니다.
- 인터페이스는 랭귀지 독립적입니다. 랭귀지에서 포인터의 구조체를 지원한다면 COM 인터페이스를 구현하는 데 랭귀지를 사용할 수 있고 포인터를 통해 명시적으로나 암시적으로 함수를 호출할 수 있습니다.
- 인터페이스는 객체 자체는 아니며 객체를 액세스하는 방법을 제공합니다. 따라서 클라이언트는 데이터를 직접 액세스하지 못하고 인터페이스 포인터를 통해 데이터를 액세스합니다. Windows 2000은 just-in-time 활성화 및 객체 풀링과 같은 COM+ 기능을 제공하는 인터셉터로 알려진 간접의 추가적인 계층을 추가합니다.
- 인터페이스는 항상 기본 인터페이스인 *IUnknown*에서 상속됩니다.
- 프록시를 통한 COM에 의해 인터페이스를 리디렉션하여 리디렉션을 인식하는 클라이언트나 서버 객체 없이도 인터페이스 메소드 호출을 사용하여 스레드, 프로세스 및 네트워크로 연결된 컴퓨터 사이에서 호출할 수 있습니다. 자세한 내용은 33-6 페이지의 "In-process, Out-of-process 및 원격 서버"를 참조하십시오.

기본 COM 인터페이스, IUnknown

모든 COM 객체는 *IUnknown*이라는 기본 인터페이스, 기본 인터페이스 타입 *IInterface*에 대한 **typedef**를 지원해야 합니다. *IUnknown*에는 다음과 같은 루틴이 포함됩니다.

- | | |
|------------------|---|
| QueryInterface | 객체가 지원하는 다른 인터페이스에 대한 포인터를 제공합니다. |
| AddRef 및 Release | 클라이언트가 더 이상 서비스를 사용하지 않을 때 객체가 자신을 삭제할 수 있도록 객체의 수명을 추적하는 일반 참조 카운팅 메소드입니다. |

클라이언트는 *IUnknown* 메소드, *QueryInterface*를 통해 다른 인터페이스에 대한 포인터를 가져옵니다. *QueryInterface*는 서버 객체의 모든 인터페이스에 대해 알고 있으며 요청한 인터페이스에 대한 포인터를 클라이언트에게 제공할 수 있습니다. 인터페이스에 대한 포인터를 받으면 클라이언트는 인터페이스의 모든 메소드를 호출할 수 있게 됩니다.

객체는 일반 참조 카운팅 메소드인 *AddRef* 및 *Release*라는 *IUnknown* 메소드를 통해 자신의 수명을 추적합니다. 객체의 참조 카운트가 0이 아니면 객체는 메모리에 남습니다. 참조 카운트가 0이 되면 인터페이스 구현은 안전하게 객체를 삭제할 수 있습니다.

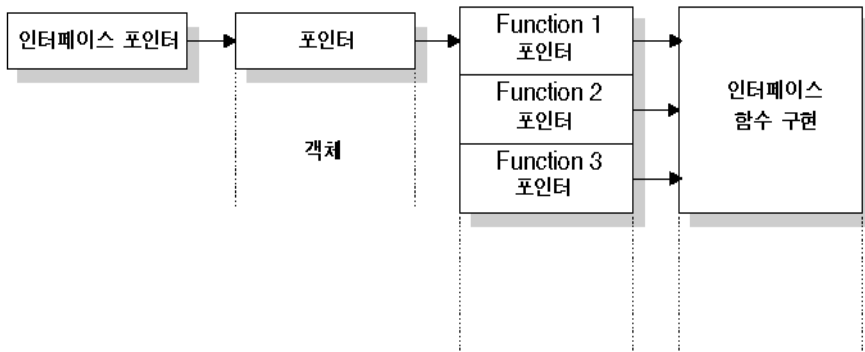
COM 인터페이스 포인터

인터페이스 포인터는 인터페이스에서 각 메소드의 구현을 가리키는 객체 인스턴스에 대한 32비트 포인터입니다. 구현은 **vtable**이라는 메소드에 대한 포인터 배열을 통해 액세스됩니다. vtable은 오브젝트 파스칼의 가상 함수를 지원하는 데 사용되는 메커니즘

과 유사합니다. 이러한 유사성으로 인해 컴파일러는 오브젝트 파스칼 클래스의 메소드에 대한 호출을 해결하는 것과 동일한 방법으로 인터페이스의 메소드에 대한 호출을 해결할 수 있습니다.

vtable은 객체 클래스의 모든 인스턴스에서 공유되므로 각 객체 인스턴스에 대한 객체 코드는 private 데이터를 포함하는 두 번째 구조체를 할당합니다. 그러면 클라이언트의 인터페이스 포인터는 다음의 다이어그램에서 보듯이 vtable 포인터에 대한 포인터가 됩니다.

그림 33.2 인터페이스 vtable



Windows 2000 및 이후 버전의 Windows에서는 객체가 COM+에서 실행될 때 간접적인 추가 레벨이 인터페이스 포인터와 vtable 포인터 사이에서 제공됩니다. 클라이언트에 사용할 수 있는 인터페이스 포인터는 인터셉터를 가리키며, 인터셉터는 다시 vtable을 가리킵니다. 인터셉터를 사용하여 COM+에서는 just-in-time 활성화와 같은 서비스를 제공할 수 있으므로 서버는 클라이언트에 불투명한 방식으로 동적으로 비활성화되고 재활성화될 수 있습니다. 이렇게 하려면 COM+는 인터셉터가 일반적인 vtable 포인터인 것처럼 동작하도록 해야 합니다.

COM 서버

COM 서버는 클라이언트 애플리케이션이나 라이브러리에 서비스를 제공하는 애플리케이션 또는 라이브러리입니다. COM 서버는 하나 이상의 COM 객체로 구성되며, 이 때 COM 객체는 속성 및 메소드의 집합입니다.

클라이언트는 COM 객체가 서비스를 수행하는 *방법*을 모르며 객체의 구현은 캡슐화되어 있습니다. 객체는 앞에서 설명한 대로 인터페이스를 통해 서비스를 사용할 수 있게 합니다.

또한 클라이언트는 COM 객체가 상주하는 *위치*도 알 필요가 없습니다. COM은 객체의 위치에 관계 없이 투명한 액세스를 제공합니다.

클라이언트가 COM 객체로부터 서비스를 요청하면 클라이언트는 클래스 식별자(CLSID)를 COM으로 전달합니다. CLSID는 COM 객체를 식별하는 GUID입니다. COM은 시스템 레지스트리에 등록된 이 CLSID를 사용하여 적절한 서버 구현을 찾습니다. 일단 서버를 찾으면 COM은 코드를 메모리로 가져와 서버가 클라이언트에 대한 객체

인스턴스를 인스턴스화하게 합니다. 이 프로세스는 요구에 따라 객체의 인스턴스를 만드는, 인터페이스를 기반으로 하는 클래스 팩토리라는 특수한 객체를 통해 간접적으로 처리됩니다.

COM 서버는 최소한 다음 작업을 수행해야 합니다.

- 서버 모듈과 클래스 식별자 (CLSID) 를 연결하는 시스템 레지스트리에 항목을 등록합니다.
- 특정 CLSID의 다른 객체를 만드는 클래스 팩토리 객체를 구현합니다.
- 클래스 팩토리를 COM에 노출합니다.
- 클라이언트에게 서비스를 제공하지 않는 서버를 메모리에서 제거할 수 있는 언로드 메커니즘을 제공합니다.

참고 Delphi 마법사는 33-18 페이지의 "마법사로 COM 객체 구현"에서 설명한 대로 COM 객체 및 서버 생성을 자동화합니다.

CoClass 및 클래스 팩토리

COM 객체는 하나 이상의 COM 인터페이스를 구현하는 **CoClass**의 인스턴스입니다. COM 객체는 인터페이스에서 정의한 대로 서비스를 제공합니다.

CoClass는 *클래스 팩토리*라는 객체의 특수한 형식으로 인스턴스화됩니다. 클라이언트가 객체의 서비스를 요청할 때마다 클래스 팩토리는 특정 클라이언트에 대한 객체 인스턴스를 만듭니다. 일반적으로 다른 클라이언트가 객체의 서비스를 요청하면 클래스 팩토리는 두 번째 클라이언트에게 서비스를 제공하기 위한 다른 객체 인스턴스를 만듭니다. 또한 클라이언트는 인스턴스를 지원하도록 자신을 등록하는 실행 COM 객체로 바인딩할 수 있습니다.

CoClass에 클래스 팩토리과 클래스 식별자 (CLSID)가 반드시 있어야 외부적으로, 즉 다른 모듈에서 인스턴스화할 수 있습니다. CoClass에 이러한 고유한 식별자를 사용하면 클래스에서 새 인터페이스를 구현할 때마다 CoClass를 업데이트할 수 있습니다. 새 인터페이스는 DLL을 사용할 때 발생하는 일반적인 문제인 이전 버전에 영향을 주지 않고 메소드를 수정하거나 추가할 수 있습니다.

Delphi 마법사는 클래스 식별자 할당 및 클래스 팩토리의 구현과 인스턴스화를 처리합니다.

In-process, Out-of-process 및 원격 서버

COM을 사용하면 클라이언트는 객체가 상주하는 위치를 알 필요가 없으며 객체의 인터페이스만 호출하면 됩니다. COM은 호출하는 데 필요한 단계를 수행합니다. 이 단계는 객체가 클라이언트와 동일한 프로세스에 상주하는지, 클라이언트 컴퓨터의 다른 프로

세스에 상주하는지, 네트워크 상의 다른 시스템에 상주하는지에 따라 달라집니다. 이렇게 다양한 서버들의 유형은 다음과 같습니다.

In-process 서버

예를 들어, Internet Explorer나 Netscape에서 볼 수 있는 웹 페이지에 포함된 ActiveX 컨트롤과 같이 클라이언트와 동일한 프로세스 공간에서 실행되는 라이브러리(DLL)입니다. 즉 ActiveX 컨트롤은 클라이언트 컴퓨터로 다운로드되고 웹 브라우저와 동일한 프로세스 내에서 호출됩니다.

클라이언트는 COM 인터페이스를 직접 호출하여 in-process 서버와 통신합니다.

Out-of-process 서버 또는 로컬 서버

다른 프로세스 공간이지만 클라이언트와 동일한 컴퓨터에서 실행되는 다른 애플리케이션(EXE)입니다. 예를 들어, Word 문서에 포함된 Excel 스프레드시트는 동일한 컴퓨터에서 실행되는 두 가지 별개의 애플리케이션입니다.

로컬 서버는 COM을 사용하여 클라이언트와 통신합니다.

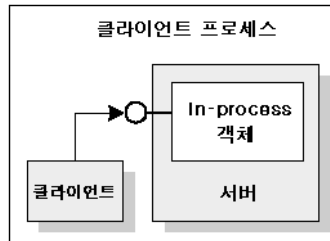
원격 서버

클라이언트와 다른 컴퓨터에서 실행되는 DLL이나 다른 애플리케이션입니다. 예를 들어, Delphi 데이터베이스 애플리케이션은 네트워크의 다른 컴퓨터에 있는 애플리케이션 서버에 연결됩니다.

원격 서버는 분산 COM(DCOM)을 사용하여 인터페이스를 액세스하고 애플리케이션 서버와 통신합니다.

그림 33.3에서 보듯이 in-process 서버의 경우 객체 인터페이스에 대한 포인터가 클라이언트와 동일한 프로세스 공간에 있으므로 COM은 객체 구현을 직접 호출합니다.

그림 33.3 In-process 서버

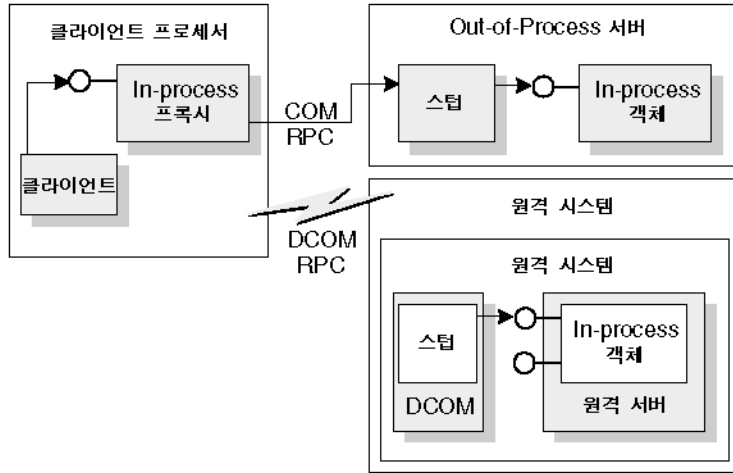


참고 이 프로세서가 COM+에서 항상 적용되는 것은 아닙니다. 클라이언트가 다른 컨텍스트에서 객체를 호출하면 COM+는 호출을 인터셉트하여 서버가 in-process인 경우에도 out-of-process 서버(아래 참조)에 대한 호출인 것처럼 동작합니다. COM+ 작업에 대한 자세한 내용은 39장 "MTS 또는 COM+ 객체 생성"을 참조하십시오.

그림 33.4에서 보듯이 프로세스가 다른 프로세스나 다른 컴퓨터에 있으면 COM은 프록시를 사용하여 원격 프로시저 호출을 초기화합니다. **프록시**는 클라이언트와 동일한 프로세스에 상주하므로 클라이언트 관점에서 보면 모든 인터페이스 호출은 비슷해 보입니다. 프록시는 클라이언트의 호출을 인터셉트하여 실제 객체가 실행되는 위치로 전달합니다. 클라이언트가 다른 프로세스 공간이나 다른 컴퓨터에 있는 객체를 마치 자신과 동

일한 프로세스에 있는 것처럼 액세스할 수 있는 메커니즘을 **마샬링 (marshaling)** 이라고 합니다.

그림 33.4 Out-of-process 및 원격 서버



Out-of-process와 원격 서버의 차이점은 사용되는 프로세스 간 통신의 유형입니다. 프록시는 COM을 사용하여 Out-of-process 서버와 통신하고, 분산 COM(DCOM)을 사용하여 원격 컴퓨터와 통신합니다. DCOM은 로컬 객체 요청을 다른 컴퓨터에서 실행되는 원격 객체로 투명하게 전달합니다.

참고 DCOM은 원격 프로시저 호출에 Open Group의 DCE (Distributed Computing Environment)에서 제공하는 RPC 프로토콜을 사용합니다. DCOM은 분산 보안을 위해 NT LAN 관리자 (NTLM) 보안 프로토콜을 사용하고 디렉토리 서비스에 DNS (Domain Name System)를 사용합니다.

마샬링 메커니즘(Marshaling mechanism)

마샬링은 클라이언트가 다른 프로세스나 다른 컴퓨터에 있는 원격 객체에 대해 인터페이스 함수를 호출할 수 있는 메커니즘입니다. 마샬링은 다음과 같은 작업을 수행합니다.

- 서버 프로세스에서 인터페이스 포인터를 가져오고 클라이언트 프로세스의 코드에 프록시 포인터를 사용할 수 있게 합니다.
- 클라이언트에서 전달된 인터페이스 호출의 인수를 전달하여 원격 객체의 프로세스 공간에 인수를 둡니다.

모든 인터페이스 호출에서 클라이언트는 인수를 스택으로 푸시하고 인터페이스 포인터를 통해 함수를 호출합니다. 객체에 대한 호출이 in-process가 아닌 경우 호출은 프록시로 전달됩니다. 프록시는 인수를 마샬링 패킷으로 압축하고 구조체를 원격 객체로 전송합니다. 객체의 스텝(stub)은 패킷의 압축을 풀고, 인수를 스택으로 푸시하며, 객체의 구현을 호출합니다. 실제로 객체는 클라이언트의 호출을 자신의 주소 공간에 다시 만듭니다.

발생하는 마샬링의 타입은 구현되는 COM 객체에 따라 달라집니다. 객체는 *IDispatch* 인터페이스에서 제공하는 표준 마샬링 메커니즘을 사용할 수 있습니다. 이는 시스템 표준 원격 프로시저 호출(RPC)을 통해 통신할 수 있는 일반적 마샬링 메커니즘입니다. *IDispatch* 인터페이스에 대한 자세한 내용은 36-12 페이지의 "Automation 인터페이스"를 참조하십시오. 객체가 *IDispatch*를 구현하지 않더라도 객체 자신을 Automation 호환 타입으로 제한하고 등록된 타입 라이브러리가 있는 경우 COM은 자동으로 마샬링을 지원합니다.

자신을 Automation 호환 타입으로 제한하지 않거나 타입 라이브러리를 등록하지 않은 애플리케이션은 고유한 마샬링을 제공해야 합니다. 마샬링은 *IMarshal* 인터페이스의 구현을 통해 제공되거나 개별적으로 생성된 프록시/스텝 DLL을 사용하여 제공됩니다. Delphi는 프록시/스텝 DLL의 자동 생성을 지원하지 않습니다.

집합체

경우에 따라 서버 객체는 다른 COM 객체를 사용하여 기능 중 일부를 수행하기도 합니다. 예를 들어, 재고 관리 객체는 별도의 송장 객체를 사용하여 고객 송장을 처리할 수 있습니다. 그러나 재고 관리 객체가 송장 인터페이스를 클라이언트에게 제공하려면 다음과 같은 문제가 발생합니다. 재고 인터페이스를 가지고 있는 클라이언트가 *QueryInterface*를 호출하여 송장 인터페이스를 가져올 수 있지만 송장 객체가 만들어질 때는 재고 관리 객체에 대해 모르며 *QueryInterface* 호출에 대한 응답으로 재고 인터페이스를 반환할 수 없습니다. 송장 인터페이스를 가지는 클라이언트는 재고 인터페이스로 돌아갈 수 없습니다.

이 문제를 방지하기 위해 일부 COM 객체는 **집합체**를 지원합니다. 재고 관리 객체가 송장 객체의 인스턴스를 만들면 자신의 *IUnknown* 인터페이스 복사본을 전달합니다. 그런 다음 송장 객체는 해당 *IUnknown* 인터페이스를 사용하여 지원되지 않는, 재고 인터페이스와 같은 인터페이스를 요청하는 모든 *QueryInterface* 호출을 처리할 수 있습니다. 이런 경우 두 객체 모두를 집합체라고 합니다. 송장 객체는 집합체의 내부 또는 포함된 객체라고 하며 재고 객체는 외부 객체라고 합니다.

참고 집합체의 외부 객체처럼 동작하려면 COM 객체가 Windows API *CoCreateInstance* 또는 *CoCreateInstanceEx*를 사용하여 내부 객체를 만들어 내부 객체가 *QueryInterface* 호출에 사용할 수 있는 매개 변수로 *IUnknown* 포인터를 전달해야 합니다.

집합체의 내부 객체처럼 동작할 수 있는 객체를 만들려면 *TContainedObject*의 자손이어야 합니다. 객체를 만들면 내부 객체가 처리할 수 없는 호출에서 *QueryInterface* 메소드가 사용될 수 있도록 외부 객체의 *IUnknown* 인터페이스를 생성자에게 전달합니다.

COM 클라이언트

클라이언트는 항상 COM 객체의 인터페이스를 쿼리하여 제공 가능한지 알아볼 수 있습니다. 모든 COM 객체는 클라이언트가 알려진 인터페이스를 요청할 수 있도록 합니다. 또한 서버가 *IDispatch* 인터페이스를 지원하면 클라이언트는 인터페이스가 지원하는 메소드에 대한 정보를 서버에 쿼리합니다. 서버 객체는 그 객체를 사용하는 클라이언트에 대해 예상할 수 없습니다. 마찬가지로 클라이언트는 객체가 서비스를 제공하는 방법

또는 위치를 알 필요가 없으며 인터페이스를 통해 알려 주는 서비스를 제공하는 서버 객체에 의존합니다.

COM 클라이언트에는 컨트롤러와 컨테이너라는 두 가지 타입이 있습니다. 컨트롤러는 서버를 시작하고 인터페이스를 통해 상호 작용합니다. 또한 컨트롤러는 COM 객체로부터 서비스를 요청하거나 별도의 프로세스로 처리합니다. 컨테이너는 컨테이너의 사용자 인터페이스에 나타나는 비주얼 컨트롤이나 객체를 호스트하고 이미 정의된 인터페이스를 사용하여 서버 객체와의 디스플레이 문제를 해결합니다. DCOM에서는 컨테이너 관계를 가질 수 없습니다. 예를 들어, 컨테이너의 사용자 인터페이스에 나타나는 비주얼 컨트롤은 로컬에 있어야 합니다. 그 이유는 컨트롤이 자신을 그리도록 되어 있기 때문이며 컨트롤이 로컬 GDI 리소스로 액세스해야 합니다.

Delphi는 서버 객체가 다른 VCL 컴포넌트처럼 보이도록 사용자가 타입 라이브러리나 ActiveX 컨트롤을 컴포넌트 래퍼로 가져오게 하여 COM 클라이언트 개발을 쉽게 합니다. 이 프로세스에 대한 자세한 내용은 35장 "COM 클라이언트 생성"을 참조하십시오.

COM 확장

COM은 원래 핵심적인 통신 기능을 제공하고 확장을 통해 이러한 기능을 확장할 수 있도록 디자인되었습니다. COM 자체는 특정 용도로 특화된 인터페이스 집합을 정의하여 핵심 기능을 확장해 왔습니다.

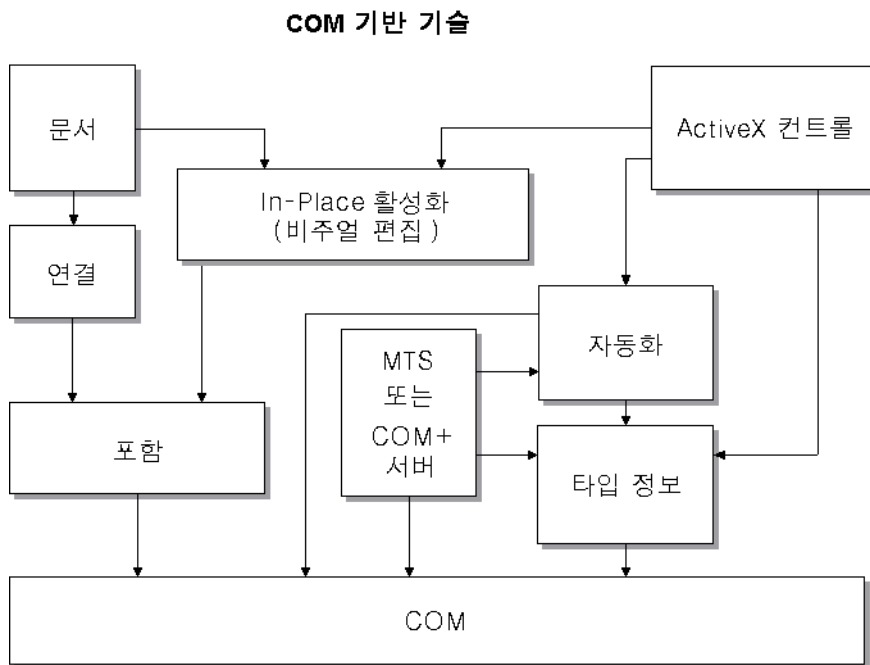
다음은 COM 확장이 현재 제공하는 서비스의 일부 목록입니다. 다음 단원에서 이러한 서비스에 대해 더 자세히 설명합니다.

- Automation 서버** Automation은 애플리케이션의 기능을 참조하여 다른 애플리케이션의 객체를 프로그램에서 제어합니다. Automation 서버는 런타임 시 다른 실행 파일에 의해 제어될 수 있는 객체입니다.
- ActiveX 컨트롤** ActiveX 컨트롤은 일반적으로 클라이언트 애플리케이션에 포함시키기 위한 특수한 in-process 서버입니다. 컨트롤은 이벤트뿐만 아니라 디자인 및 런타임 동작을 제공합니다.
- Active Server Page** Active Server Page는 HTML 페이지를 생성하는 스크립트입니다. 스크립트 랭귀지는 Automation 객체를 만들고 실행하는 구문을 포함합니다. 즉, Active Server Page는 Automation 컨트롤러처럼 동작합니다.
- Active Document** 연결 및 포함, 드래그 앤 드롭, 비주얼 편집 및 in-place 활성화를 지원하는 객체입니다. Word 문서 및 Excel 스프레드시트가 Active Document의 예입니다.

- 트랜잭션 객체** 많은 클라이언트에 응답하기 위한 추가 지원을 포함하는 객체입니다. just-in-time 활성화, 트랜잭션, 리소스 풀링 및 보안 서비스와 같은 기능이 포함됩니다. 이러한 기능은 원래 MTS에 의해 처리되지만 COM+가 등장하면서 COM에 내장되었습니다.
- 타입 라이브러리** 객체 및 인터페이스에 대한 자세한 타입 정보를 제공하는, 경우에 따라 리소스로 저장되는 정적인 데이터 구조의 모음입니다. Automation 서버의 클라이언트, ActiveX 컨트롤 및 트랜잭션 객체는 타입 정보를 사용할 수 있을 것으로 예상합니다.

다음 다이어그램은 COM 확장의 관계와 COM에 기반을 두고 이들이 어떻게 만들어지는지 보여 줍니다.

그림 33.5 COM 기반 기술



COM 객체는 비주얼일 수도 있고 닌비주얼일 수도 있습니다. 객체가 마샬링을 지원하면 어떤 객체는 클라이언트와 동일한 프로세스 공간에서 실행되며 또 어떤 객체는 다른 프로세스나 원격 컴퓨터에서 실행될 수 있습니다. 표 33.1은 비주얼일지 여부에 관계 없이 사용자가 만들 수 있는 COM 객체의 타입, 객체가 실행될 수 있는 프로세스 공간,

객체가 마샬링을 제공하는 방법 및 타입 라이브러리가 필요한지 여부를 요약한 것입니다.

표 33.1 COM 객체 요구 사항

객체	비주얼 객체 인지 여부	프로세스 공간	통신	타입 라이브러리
Active Document	일반적으로	In-process 또는 out-of-process	OLE Verbs	아니오
Automation	경우에 따라	In-process, out-of-process 또는 원격	out-of process 및 원격 서버의 <i>IDispatch</i> 인터페이스를 사용하여 자동으로 마샬링	자동 마샬링에 필요
ActiveX 컨트롤	일반적으로	In-process	<i>IDispatch</i> 인터페이스를 사용하여 자동으로 마샬링	필수
MTS 또는 COM+	경우에 따라	MTS의 In-process, COM+의 모든 프로세스	타입 라이브러리를 통해 자동으로 마샬링	필수
In-process 사용자 지정 인터페이스 객체	선택적으로	In-process	in-process 서버에 필요한 마샬링 없음	권장
다른 사용자 지정 인터페이스 객체	선택적으로	In-process, out-of-process 또는 원격	타입 라이브러리를 통해 자동으로 마샬링되거나 그렇지 않으면 사용자 지정 인터페이스를 사용하여 마샬링	권장

Automation 서버

Automation은 애플리케이션의 기능을 참조하여 동시에 둘 이상의 애플리케이션을 처리할 수 있는 매크로처럼 다른 애플리케이션의 객체를 프로그램에서 제어합니다. 처리되는 서버 객체는 Automation 객체라고 하며 Automation 객체의 클라이언트는 Automation 컨트롤러라고 합니다.

Automation은 in-process, 로컬 및 원격 서버에서 사용할 수 있습니다.

Automation은 다음과 같은 두 가지 주요한 특징을 가집니다.

- Automation 객체는 속성 및 명령 집합을 정의하고 타입 설명을 통해 그 기능을 설명합니다. 이렇게 하려면 인터페이스, 인터페이스 메소드 및 해당 메소드의 인수에 대한 정보를 제공하는 방법이 있어야 합니다. 일반적으로 이 정보는 타입 라이브러리에 있습니다. 또한 Automation 서버는 *IDispatch* 인터페이스(다음 참조)를 통해 쿼리할 때 동적으로 타입 정보를 생성할 수 있습니다.
- Automation 객체는 메소드를 액세스할 수 있게 하여 다른 애플리케이션이 메소드를 사용할 수 있도록 합니다. 이를 위해 Automation 객체는 *IDispatch* 인터페이스를 구현합니다. 이 인터페이스를 통해 객체는 모든 메소드 및 속성을 표시할 수 있습니다. 타입 정보를 통해 일단 식별되면 이 인터페이스의 기본 메소드를 통해 객체의 메소드를 호출할 수 있습니다.

Automation *IDispatch* 인터페이스가 마샬링 프로세스를 자동화하기 때문에 개발자는 종종 Automation으로 모든 프로세스 공간에서 실행되는 보이지 않는 OLE 객체를 만들어 사용합니다. 그러나 Automation은 사용할 수 있는 타입을 제한합니다.

일반적으로 타입 라이브러리에 유효한 타입 목록 및 특히 Automation 인터페이스에 대해서는 34-11 페이지의 "유효한 타입"을 참조하십시오.

Automation 서버 작성에 대한 자세한 내용은 36장 "일반 COM 서버 생성"을 참조하십시오.

Active Server Page

ASP(Active Server Page) 기술을 사용하면 웹 서버를 통해 클라이언트에 의해 시작할 수 있는 Active Server Pages라는 간단한 스크립트를 작성할 수 있습니다. 클라이언트에서 실행되는 ActiveX 컨트롤과 달리 Active Server Pages는 서버에서 실행되고 클라이언트에 결과 HTML 페이지를 반환합니다.

Active Server Pages는 Jscript 또는 VBScript로 작성됩니다. 서버가 웹 페이지를 로드할 때마다 스크립트가 실행됩니다. 그런 다음 이 스크립트는 포함된 Automation 서버 또는 Enterprise Java Bean을 시작할 수 있습니다. 예를 들어, 비트맵을 만들거나 데이터베이스에 연결하는 것처럼 Automation 서버를 작성할 수 있으며 이 서버는 클라이언트가 웹 페이지를 로드할 때마다 업데이트되는 데이터를 액세스합니다.

Active Server Pages는 웹 페이지를 서비스하기 위해 Microsoft Internet Information Server(IIS) 환경에 의존합니다.

Delphi 마법사를 사용하면 Active Server Page로 작업하도록 특별히 디자인된 Automation 객체인 Active Server Object를 만듭니다. 이러한 타입의 객체 생성 및 사용에 대한 자세한 내용은 37장 "Active Server Page 생성"을 참조하십시오.

ActiveX 컨트롤

ActiveX는 COM 컴포넌트, 특히 컨트롤을 더 간단하고 효율적으로 만드는 기술입니다. ActiveX는 사용하기 전에 클라이언트에서 다운로드해야 하는 인터넷 애플리케이션에 사용하도록 제작된 컨트롤에 특히 필요합니다.

ActiveX 컨트롤은 in-process 서버로만 실행되는 비주얼 컨트롤이며 ActiveX 컨트롤 컨테이너 애플리케이션에 연결할 수 있습니다. ActiveX 컨트롤은 그 자체로는 완전한 애플리케이션이 아니지만 여러 애플리케이션에서 다시 사용할 수 있는 조립식 OLE 컨트롤로 생각할 수 있습니다. ActiveX 컨트롤에는 볼 수 있는 사용자 인터페이스가 있으며 미리 정의된 인터페이스에 의존하여 I/O를 결정하고 호스트 컨테이너와의 문제를 표시합니다.

ActiveX 컨트롤은 Automation을 사용하여 속성, 메소드 및 이벤트를 표시합니다. ActiveX 컨트롤의 기능으로는 이벤트를 발생시키는 기능, 데이터 소스로 바인딩하는 기능 및 라이선스 지원 기능이 있습니다.

ActiveX 컨트롤의 사용 중 하나는 웹 페이지의 대화형 객체로써 웹 사이트에서 사용되는 것입니다. 따라서 ActiveX는 웹 브라우저로 HTML이 아닌 문서를 보는 데 사용되는

ActiveX Documents의 사용을 포함하여 World Wide Web의 대화형 콘텐츠를 대상으로 하는 표준입니다. ActiveX 기술에 대한 자세한 내용은 Microsoft ActiveX 웹 사이트를 참조하십시오.

Delphi 마법사를 사용하면 쉽게 ActiveX 컨트롤을 만들 수 있습니다. 이러한 타입의 객체 생성 및 사용에 대한 자세한 내용은 38장 "ActiveX 컨트롤 생성"을 참조하십시오.

활성 문서

이전에 OLE 문서로 알려진 활성 문서는 연결 및 포함, 드래그 앤 드롭 및 시각적 편집을 지원하는 COM 서비스의 집합입니다. 활성 문서는 사운드 클립, 스프레드시트, 텍스트 및 비트맵과 같은 다른 형식의 데이터나 객체를 완전히 통합할 수 있습니다.

ActiveX 컨트롤과 달리 활성 문서는 in-process 서버로 제한되지 않으며 cross-process 애플리케이션에서 사용할 수 있습니다.

거의 보이지 않는 Automation 객체와 달리 활성 문서 객체는 다른 애플리케이션에서 볼 수 있도록 활성화할 수 있습니다. 따라서 활성 문서 객체는 다음과 같은 두 가지 데이터와 연결됩니다. 즉, 디스플레이 또는 출력 장치에서 볼 수 있도록 객체를 표시하는 데 사용되는 표시 데이터와 객체를 편집하는 데 사용되는 원시 데이터입니다.

활성 문서 객체는 문서 컨테이너 또는 문서 서버가 될 수 있습니다. Delphi가 활성 문서를 만드는 자동 마법사를 제공하지 않으면 VCL 클래스, *TOleContainer*를 사용하여 기존 활성 문서의 연결 및 포함을 지원할 수 있습니다.

활성 문서 컨테이너의 기초로 *TOleContainer*를 사용할 수도 있습니다. 활성 문서 서버의 객체를 만들려면 객체가 지원해야 하는 서비스에 따라 COM 객체 마법사를 사용하여 적절한 인터페이스를 추가합니다. 활성 문서 서버를 만들고 사용하는 것에 대한 자세한 내용은 Microsoft ActiveX 웹 사이트를 참조하십시오.

참고 활성 문서의 사양에 cross-process 애플리케이션의 마샬링에 대한 지원이 기본 제공되면 활성 문서는 창 핸들, 메뉴 핸들 등과 같이 주어진 컴퓨터의 특정 시스템에 지정된 타입을 사용하기 때문에 원격 서버에서는 실행되지 않습니다.

트랜잭션 객체

Delphi는 "트랜잭션 객체"라는 용어를 사용하여 Windows 2000 이전의 Windows 버전의 경우에는 MTS(Microsoft Transaction Server)에서, Windows 2000 이후 버전은 COM+에서 제공하는 트랜잭션 서비스, 보안 및 리소스 관리를 사용하는 객체를 참조합니다. 이 객체들은 대규모 분산 환경에서 작동하도록 디자인되었습니다.

트랜잭션 서비스는 견고하므로 동작이 항상 완료되거나 롤백됩니다. 서버는 작업을 부분적으로 완료하지 않습니다. 보안 서비스를 사용하면 다른 수준의 지원을 클라이언트의 다른 클래스에 노출할 수 있습니다. 리소스 관리를 사용하면 객체는 리소스를 풀링하거나 객체가 사용 중일 때만 활성으로 유지하여 더 많은 클라이언트를 처리할 수 있습니다. 시스템에서 이러한 서비스를 제공할 수 있게 하려면 객체는 *IObjectControl* 인터페이스를 구현해야 합니다. 서비스를 액세스하기 위해 트랜잭션 객체는 MTS나 COM+에서 자신을 위해 만든 *IObjectContext*라는 인터페이스를 사용합니다.

MTS에서 서버 객체는 라이브러리(DLL)로 작성된 다음 MTS 런타임 환경에 설치되어야 합니다. 즉, 서버 객체는 MTS 런타임 프로세스 공간에서 실행되는 in-process 서버입니다. COM+에서 모든 COM 호출은 인터셉터를 통해 라우트되기 때문에 이러한 제한이 적용되지 않습니다. 클라이언트에서 MTS와 COM+의 차이는 분명합니다.

MTS나 COM+ 서버는 같은 프로세스 공간에서 실행되는 트랜잭션 객체를 그룹화합니다. MTS에서는 이 그룹을 MTS 패키지라고 하며, COM+에서는 COM+ 애플리케이션이라고 합니다. 하나의 컴퓨터는 각각 별도의 프로세스 공간에서 실행되고 있는 여러 개의 다른 MTS 패키지 또는 COM+ 애플리케이션에서 실행될 수 있습니다.

클라이언트에 있어서 트랜잭션 객체는 다른 COM 서버 객체처럼 나타날 수 있습니다. 클라이언트가 트랜잭션 자체를 초기화하지 않는 한 트랜잭션, 보안 또는 just-in-time 활성화에 대해 알 필요가 없습니다.

MTS와 COM+ 모두 트랜잭션 객체를 관리하는 별도의 도구를 제공합니다. 이 도구를 사용하면 객체를 패키지 또는 COM+ 애플리케이션으로 구성하고, 컴퓨터에 설치된 패키지나 COM+ 애플리케이션을 보고, 포함된 객체의 속성을 보거나 변경하고, 트랜잭션을 모니터링하고 관리하며, 클라이언트에서 객체를 사용할 수 있게 하는 등의 작업을 수행할 수 있습니다. MTS에서 이 도구는 MTS 탐색기이고, COM+에서는 COM+ 컴포넌트 관리자입니다.

타입 라이브러리

타입 라이브러리는 객체의 인터페이스에서 결정할 수 있는 것보다 객체에 대한 더 많은 타입 정보를 얻을 수 있는 방법을 제공합니다. 타입 라이브러리에 포함된 타입 정보는 어떤 인터페이스가 어떤 객체에 있는지(CLSID가 주어짐), 각 인터페이스에 어떤 구성원 함수가 있는지 및 해당 함수에 어떤 인수가 필요한지 등 객체 및 인터페이스에 대해 필요한 정보를 제공합니다.

객체의 실행 중인 인스턴스를 쿼리하거나 타입 라이브러리를 로드하고 읽어서 타입 정보를 얻을 수 있습니다. 이 정보를 사용하면 필요한 구성원 함수가 무엇인지와 해당 구성원 함수를 전달할 객체를 특히 잘 알게 되어 원하는 객체를 사용하는 클라이언트를 구현할 수 있습니다.

Automation 서버의 클라이언트, ActiveX 컨트롤 및 트랜잭션 객체는 타입 정보를 사용할 수 있을 것으로 예상합니다. COM 객체 마법사는 타입 라이브러리를 옵션으로 만들지만 Delphi의 모든 마법사는 타입 라이브러리를 자동으로 생성합니다. 34장 "Type Library 작업"에서 설명한 대로 Type Library 에디터를 사용하면 이러한 타입 정보를 보거나 편집할 수 있습니다.

이 단원에서는 타입 라이브러리에 포함된 내용, 만드는 방법, 사용 시기 및 액세스 방법을 설명합니다. 이 단원은 여러 랭귀지 사이에서 인터페이스를 공유하고자 하는 개발자를 위해 타입 라이브러리 도구를 사용하는 것에 대한 제안으로 끝맺습니다.

타입 라이브러리의 내용

타입 라이브러리에는 어떤 COM 객체에 어떤 인터페이스가 있는지와 인터페이스 메소드에 대한 인수의 타입과 수를 나타내는 *타입 정보*가 포함되어 있습니다. 이러한 설명에는 Automation 인터페이스 메소드 및 속성에 대한 디스패치 식별자(dispatchID)뿐만 아니라 CoClass의 고유 식별자(CLSID) 및 인터페이스(IID)가 포함되어 있어 적절하게 액세스할 수 있습니다.

타입 라이브러리에는 다음과 같은 정보도 들어 있습니다.

- 사용자 지정 인터페이스와 관련된 사용자 지정 타입 정보에 대한 설명
- Automation 또는 ActiveX 서버에서 export되지만 인터페이스 메소드는 아닌 루틴
- 열거, 레코드(구조), 합집합, 알리아스 및 모듈 데이터 타입에 대한 정보
- 다른 타입 라이브러리의 타입 설명에 대한 참조

타입 라이브러리 생성

이전의 개발 도구를 사용하면 IDL(Interface Definition Language) 또는 ODL(Object Description Language)로 스크립트를 작성한 다음 컴파일러를 통해 해당 스크립트를 실행하여 타입 라이브러리를 만듭니다. 그러나 Delphi는 New Items 대화 상자에서 ActiveX 또는 Multitier 페이지의 마법사를 사용하여 ActiveX 컨트롤, Automation 객체, 원격 데이터 모듈 등을 포함한 COM 객체를 만들 때 자동으로 타입 라이브러리를 생성합니다. COM 객체 마법사를 사용할 때 타입 라이브러리를 만들지 않을 수도 있습니다. 또한 주 메뉴에서 File|New|Other를 선택하고 ActiveX 탭을 선택한 다음 Type Library를 선택하여 타입 라이브러리를 만들 수도 있습니다.

Delphi의 Type Library 에디터를 사용하여 타입 라이브러리를 볼 수 있습니다. Type Library 에디터를 사용하여 타입 라이브러리를 쉽게 편집할 수 있으며 Delphi는 타입 라이브러리를 저장할 때 해당 .tlb 파일(바이너리 타입 라이브러리 파일)을 자동으로 업데이트합니다. 또한 마법사를 사용하여 만든 인터페이스 및 CoClass의 모든 변경 내용에 대해 Type Library 에디터는 구현 파일을 업데이트합니다. Type Library 에디터를 사용한 인터페이스 및 CoClass 작성에 대한 자세한 내용은 34장 "Type Library 작업"을 참조하십시오.

타입 라이브러리 사용 시기

외부 사용자에게 노출된 각 객체 집합에 대해 타입 라이브러리를 만드는 것이 중요합니다. 예를 들면, 다음과 같습니다.

- ActiveX 컨트롤은 타입 라이브러리가 필요한데 타입 라이브러리는 ActiveX 컨트롤을 포함하는 DLL에 리소스로 포함되어야 합니다.
- vtable 참조는 컴파일 시 바인딩되기 때문에 사용자 지정 인터페이스의 vtable 바인딩을 지원하는 노출된 객체는 타입 라이브러리에서 설명해야 합니다. 클라이언트는 타입 라이브러리에서 인터페이스에 대한 정보를 import하고 그 정보를 사용하여 컴파일합니다. vtable 및 컴파일 시 바인딩에 대한 자세한 내용은 36-12 페이지의 "Automation 인터페이스"를 참조하십시오.

- Automation 서버를 구현하는 애플리케이션은 클라이언트가 우선 바인딩할 수 있도록 타입 라이브러리를 제공해야 합니다.
- VCL *TTypedComObject* 클래스의 모든 자손 같이 *IProvideClassInfo* 인터페이스를 지원하는 클래스에서 인스턴스화된 객체에는 타입 라이브러리가 있어야 합니다.
- 타입 라이브러리는 필수는 아니지만 OLE 드래그 앤 드롭과 함께 사용되는 객체를 식별하는 데 유용합니다.

애플리케이션 내에서 내부적으로 사용할 목적으로 인터페이스를 정의할 때는 타입 라이브러리를 만들지 않아도 됩니다.

타입 라이브러리 액세스

일반적으로 바이너리 타입 라이브러리는 리소스 파일(.res)이나 파일 이름 확장자가 .tlb인 독립 실행형 파일의 일부입니다. 리소스 파일에 포함되면 타입 라이브러리는 서버(.dll, .ocx 또는 .exe)로 바인딩될 수 있습니다.

일단 타입 라이브러리를 만들면 객체 브라우저, 컴파일러 및 유사한 도구들이 특수한 타입 인터페이스를 통해 타입 라이브러리를 액세스할 수 있습니다.

인터페이스	설명
<i>ITypeLib</i>	타입 설명에 대한 라이브러리를 액세스하는 메소드를 제공합니다.
<i>ITypeLib2</i>	설명서 문자열, 사용자 지정 데이터 및 타입 라이브러리에 대한 통계에 대한 지원을 포함하는 <i>ITypeLib</i> 가 증가합니다.
<i>ITypeInfo</i>	타입 라이브러리에 포함된 개별 객체에 대한 설명을 제공합니다. 예를 들어, 브라우저는 이 인터페이스를 사용하여 타입 라이브러리의 객체에 대한 정보를 추출합니다.
<i>ITypeInfo2</i>	사용자 지정 데이터 요소를 액세스하는 방법을 포함하여 기타 다른 타입 라이브러리 정보를 액세스하는 <i>ITypeInfo</i> 가 증가합니다.
<i>ITypeComp</i>	인터페이스로 바인딩할 때 컴파일러에 필요한 정보를 액세스하는 빠른 방법을 제공합니다.

Delphi는 Project|Import Type Library를 선택하여 다른 애플리케이션에서 타입 라이브러리를 import하여 사용할 수 있습니다. COM 애플리케이션에 사용되는 대부분의 VCL 클래스는 타입 라이브러리와 객체의 실행 중인 인스턴스에서 타입 정보를 검색하고 저장하는 데 사용되는 필수적인 인터페이스를 지원합니다. VCL 클래스 *TTypedComObject*는 타입 정보를 제공하는 인터페이스를 지원하고 ActiveX 객체 프레임워크의 기초로 사용됩니다.

타입 라이브러리 사용의 이점

애플리케이션에 타입 라이브러리가 필요하지 않은 경우에도 타입 라이브러리를 사용하면 다음과 같은 이점이 있습니다.

- 컴파일 시에 Type 확인을 수행할 수 있습니다.

- Automation과의 우선 바인딩을 사용할 수 있으며 vtables나 이중 인터페이스를 지원하지 않는 컨트롤러는 컴파일 시에 dispID를 인코딩하여 런타임 성능을 향상시킬 수 있습니다.
- 타입 브라우저는 라이브러리를 검색할 수 있으므로 클라이언트는 객체의 특징을 볼 수 있습니다.
- *RegisterTypeLib* 함수는 노출된 객체를 등록 데이터베이스에 등록하는 데 사용할 수 있습니다.
- *UnRegisterTypeLib* 함수는 애플리케이션의 타입 라이브러리를 시스템 레지스트리에서 완전히 설치 해제하는 데 사용할 수 있습니다.
- Automation이 타입 라이브러리의 정보를 사용하여 다른 프로세스의 객체로 전달된 매개변수를 패키지로 만들기 때문에 로컬 서버 액세스가 향상됩니다.

타입 라이브러리 도구 사용

타입 라이브러리 작업에 사용되는 도구는 다음과 같습니다.

- 기존의 타입 라이브러리를 가져와서 Delphi 인터페이스 파일(_TLB.pas 파일)을 작성하는 TLBIMP(Type Library Import) 도구는 Type Library 에디터로 통합됩니다. TLBIMP는 Type Library 에디터 내에서는 사용할 수 없는 추가 구성 옵션을 제공합니다.
- TregSvr은 서버와 타입 라이브러리를 등록하고 등록 해제하는 도구로 Delphi와 함께 제공됩니다. TregSvr의 소스는 Demos 디렉터리에서 예제로 사용할 수 있습니다.
- Microsoft IDL 컴파일러(MIDL)는 IDL 스크립트를 컴파일하여 타입 라이브러리를 만듭니다.
- RegSvr32.exe는 서버와 타입 라이브러리를 등록하고 등록 해제하는 도구로 표준 Windows 유틸리티입니다.
- OLEView는 타입 라이브러리 브라우저 도구로 Microsoft의 웹 사이트에 있습니다.

마법사로 COM 객체 구현

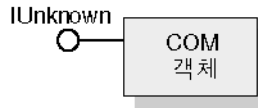
Delphi는 관련된 많은 정보를 처리하는 마법사를 제공하므로 쉽게 COM 서버 애플리케이션을 작성할 수 있습니다. Delphi에서 제공한 각각의 마법사로 다음과 같은 내용을 만들 수 있습니다.

- 간단한 COM 객체
- Automation 객체
- Active Server Object(Active Server 페이지에 포함된 경우)
- ActiveX 컨트롤
- ActiveX Form
- 트랜잭션 객체
- 속성 페이지
- 타입 라이브러리

- ActiveX 라이브러리

마법사는 COM 객체의 각 타입을 만드는 것과 관련된 많은 작업을 처리합니다. 마법사는 객체의 각 타입에 필요한 COM 인터페이스를 제공합니다. 그림 33.6에 표시된 대로 간단한 COM 객체를 사용하면 마법사는 필요한 COM 인터페이스, *IUnknown*를 구현하며 이 인터페이스는 객체에 대한 인터페이스 포인터를 제공합니다.

그림 33.6 간단한 COM 객체 인터페이스



또한 *IDispatch* 자손을 지원하는 객체를 만드는 것으로 지정한 경우 COM 객체 마법사는 *IDispatch*의 구현을 제공합니다.

그림 33.7에 표시된 대로 Automation 및 Active Server Object에 대해 마법사는 *IUnknown* 및 *IDispatch*를 구현하여 자동 마샬링을 제공합니다.

그림 33.7 Automation 객체 인터페이스

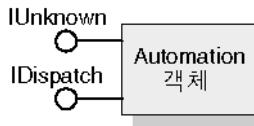


그림 33.8에 표시된 대로 ActiveX 컨트롤 객체 및 ActiveX 폼에 대해 마법사는 *IUnknown*, *IDispatch*, *IObject*, *IObjectControl* 등에서 필요한 모든 ActiveX 컨트롤 인터페이스를 구현합니다. 인터페이스 전체 목록에 대해서는 *TActiveXControl* 객체의 참조 페이지를 참조하십시오.

그림 33.8 ActiveX 객체 인터페이스

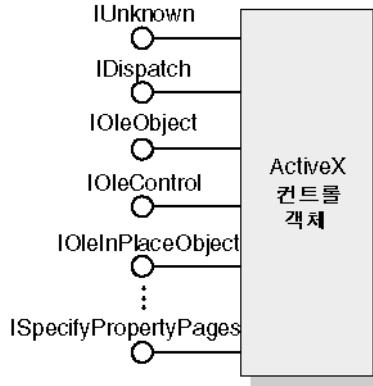


표 33.2에서는 다양한 마법사와 마법사가 구현하는 인터페이스를 나열합니다.

표 33.2 COM, Automation 및 ActiveX 객체를 구현하는 Delphi 마법사

마법사	구현된 인터페이스	마법사의 작업
COM Server	<i>IUnknown</i> 과 <i>IDispatch</i> 의 자손인 기본 인터페이스를 선택한 경우의 <i>IDispatch</i>	<p>서버 등록, 클래스 등록, 서버 로드 및 언로드와 객체 인스턴스화를 처리하는 루틴을 export 합니다.</p> <p>서버에 구현된 객체의 클래스 팩토리를 만들고 관리합니다.</p> <p>선택한 스투드 모델을 지정하는 객체에 레지스트리 항목을 제공합니다.</p> <p>선택한 인터페이스를 구현하는 메소드를 선언하여 사용자가 완료한 골격 구현을 제공합니다.</p> <p>요청한 경우 타입 라이브러리를 제공합니다.</p> <p>타입 라이브러리에 등록된 임의 인터페이스를 선택하여 구현할 수 있습니다. 이 작업을 하려면 타입 라이브러리를 사용해야 합니다.</p>
Automation Server	<i>IUnknown</i> , <i>IDispatch</i>	<p>위에서 설명한 COM 서버 마법사의 작업을 수행하고, 또 다음과 같은 작업을 수행합니다.</p> <p>dual 또는 dispatch로 지정한 인터페이스를 구현합니다. 요청한 경우 이벤트 생성에 서버측 지원을 제공합니다.</p> <p>타입 라이브러리를 자동으로 제공합니다.</p>
Active Server Object	<i>IUnknown</i> , <i>IDispatch</i> , (<i>IASPObjct</i>)	<p>위에서 설명한 Automation 객체 마법사의 작업을 수행하고 경우에 따라 웹 브라우저로 로드할 수 있는 .ASP 페이지를 생성합니다. 필요한 경우 객체의 속성 및 메소드를 수정할 수 있도록 Type Library 에디터에 그대로 남겨 둡니다.</p> <p>ASP 애플리케이션과 ASP 애플리케이션을 시작하는 HTTP 메시지에 대한 정보를 쉽게 얻을 수 있도록 ASP 내부를 속성으로 표면화합니다.</p>

표 33.2 COM, Automation 및 ActiveX 객체를 구현하는 Delphi 마법사 (계속)

마법사	구현된 인터페이스	마법사의 작업
ActiveX Control	<i>IUnknown, IDispatch, IPersistStreamInit, IOleInPlaceActiveObject, IPersistStorage, IViewObject, IOleObject, IViewObject2, IOleControl, IPerPropertyBrowsing, IOleInPlaceObject, ISpecifyPropertyPages</i>	위에서 설명한 Automation 서버 마법사의 작업을 수행하고, 또 다음과 같은 작업을 수행합니다. ActiveX 컨트롤이 기반으로 하고 모든 ActiveX 인터페이스를 구현하는 VCL 컨트롤에 해당하는 CoClass를 생성합니다. 구현 클래스를 수정할 수 있도록 소스 코드 에디터에 그대로 남겨 둡니다.
ActiveForm	ActiveX 컨트롤과 같은 인터페이스	ActiveX 컨트롤 마법사의 작업을 수행하고, 또 다음과 같은 작업을 수행합니다. ActiveX 컨트롤 마법사에 있는 기존의 VCL 클래스를 대신하는 <i>TActiveForm</i> 자손을 만듭니다. 이 새 클래스를 사용하면 Windows 애플리케이션에서 폼을 디자인하는 것과 같은 방법으로 활성 폼을 디자인할 수 있습니다.
Transactional Object	<i>IUnknown, IDispatch, IObjectControl</i>	MTS 또는 COM+ 객체 정의를 포함하고 있는 현재의 프로젝트에 새 유닛을 추가합니다. Delphi가 객체를 제대로 설치할 수 있도록 타입 라이브러리에 소유 GUID를 삽입하고, 객체가 클라이언트에 노출되는 인터페이스를 정의할 수 있도록 Type Library 에디터에 그대로 남겨 둡니다. 객체는 작성된 후에 별도로 설치해야 합니다.
Property Page	<i>IUnknown, IPropertyPage</i>	폼 디자이너에서 디자인할 수 있는 속성 페이지를 새로 만듭니다.
COM+Event Object	없음(기본값)	Type Library 에디터를 사용하여 정의할 수 있는 COM+ 이벤트 객체를 만듭니다. 다른 객체 마법사와 달리 COM+ 이벤트 객체 마법사는 클라이언트 싱크에서 제공되는 구현이 이벤트 객체에 없기 때문에 구현 유닛을 만들지 않습니다.
Type Library	없음(기본값)	새 타입 라이브러리를 만들어 활성 프로젝트와 연결합니다.
ActiveX Library	없음(기본값)	새로운 ActiveX 또는 Com 서버 DLL을 만들고 필요한 export 함수를 노출합니다.

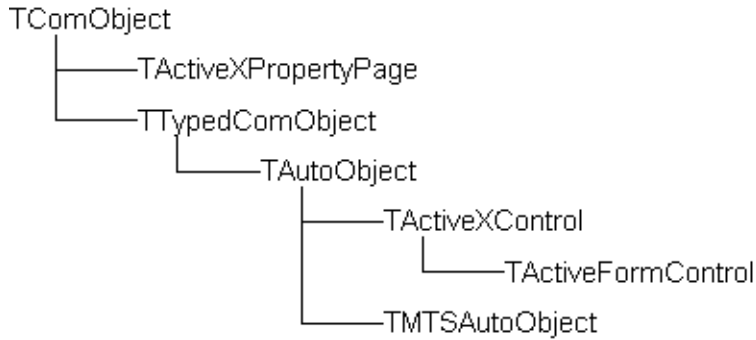
추가적인 COM 객체를 덧붙이거나 기존 구현을 다시 구현할 수 있습니다. 새 객체를 추가하려면 마법사를 두 번째에 사용하는 것이 가장 쉬운 방법인데 그 이유는 Type Library 에디터에서 변경한 내용이 자동으로 구현 객체에 적용되도록 마법사가 타입 라이브러리와 구현 클래스 사이에 연결을 설정하기 때문입니다.

마법사가 생성하는 코드

DAXDelphi의 마법사는 Delphi ActiveX 프레임워크(DAX)에서 파생되는 클래스를 생성합니다. 그 이름에도 불구하고 Delphi ActiveX 프레임워크는 ActiveX 컨트롤 뿐만

아니라 모든 타입의 COM 객체를 지원합니다. 이 프레임워크의 클래스는 마법사를 사용하여 만든 객체에 표준 COM 인터페이스의 기본 구현을 제공합니다. 그림 33.9에서 Delphi ActiveX 프레임워크의 객체를 보여 줍니다.

그림 33.9 Delphi ActiveX 프레임워크



각 마법사는 COM 서버 객체를 구현하는 구현 유닛을 생성합니다. COM 서버 객체(구현 객체)는 DAX에 있는 다음과 같은 클래스 중 하나의 자손입니다.

표 33.3 생성된 구현 클래스에 대한 DAX 기본 클래스

마법사	DAX의 기본 클래스	상속된 지원
COM Server	TTypedCOMObject	<i>IUnknown</i> 및 <i>ISupportErrorInfo</i> 인터페이스에 대한 지원 집계, OLE 예외 처리 및 이중 인터페이스의 safecall 호출 규칙에 대한 지원 타입 라이브러리 정보 읽기에 대한 지원
Automation Server Active Server Object	TAutoObject	다음에 포함하여 <i>TTypeCOMObject</i> 에서 제공하는 모든 지원 <i>IDispatch</i> 인터페이스에 대한 지원 자동 마살링 지원
ActiveX Control	TActiveXControl	다음에 포함하여 <i>TAutoObject</i> 에서 제공하는 모든 지원 컨테이너의 포함에 대한 지원 현재 위치에서 활성화에 대한 지원 속성 및 속성 페이지에 대한 지원 생성되어 창에서 실행되는 관련 컨트롤로 위임하는 기능
ActiveForm	TActiveFormControl	창에서 실행되는 다른 컨트롤 클래스가 아닌 <i>TActiveForm</i> 의 자손으로 작업하는 경우를 제외하고 <i>TAutoObject</i> 에서 제공하는 모든 지원

표 33.3 생성된 구현 클래스에 대한 DAX 기본 클래스 (계속)

마법사	DAX의 기본 클래스	상속된 지원
MTS Object	TMTSAutoObject	다음을 포함하여 <i>TAutoObject</i> 에서 제공하는 모든 지원 <i>IObjectControl</i> 인터페이스에 대한 지원
Property Page	TPropertyPage (내부적으로는 TActiveXPropertyPage 사용)	<i>IUnknown</i> 및 <i>ISupportErrorInfo</i> 인터페이스에 대한 지원 집계, OLE 예외 처리 및 이중 인터페이스의 safecall 호출 규칙에 대한 지원 <i>IPropertyPage</i> 인터페이스에 대한 지원

그림 33.9의 클래스에 해당하는 부분은 이러한 COM 객체 만들기를 처리하는 클래스 팩토리 객체의 계층 구조입니다. 마법사는 구현 클래스의 적절한 클래스 팩토리를 인스턴스화하는 구현 유닛의 초기화 구역에 코드를 추가합니다.

또한 마법사는 타입 라이브러리와 관련 유닛을 생성하며 Project1_TLB라는 폼 이름을 갖습니다. Project1_TLB 유닛에는 애플리케이션이 타입 라이브러리에 정의된 인터페이스와 타입 정의를 사용하는 데 필요한 정의가 포함됩니다. 이 파일의 내용에 대한 더 자세한 내용은 35-5 페이지의 "타입 라이브러리 정보를 import할 때 생성되는 코드"를 참조하십시오.

마법사가 생성한 인터페이스는 Type Library 에디터를 사용하여 수정할 수 있습니다. 이 작업을 수행하면 구현 클래스가 해당 변경 내용을 반영하도록 자동으로 업데이트됩니다. 구현을 완료하려면 생성된 메소드의 본문만 입력하면 됩니다.

Type Library 작업

이 장에서는 Delphi의 Type Library 에디터를 사용하여 타입 라이브러리를 만들고 편집하는 방법에 대해 설명합니다. 타입 라이브러리는 데이터 타입, 인터페이스, 멤버 함수 및 COM 객체에 의해 노출된 객체 클래스에 대한 정보를 포함하는 파일입니다. 타입 라이브러리는 서버에서 사용할 수 있는 객체 및 인터페이스의 타입을 식별하는 방법을 제공합니다. 타입 라이브러리를 사용하는 이유와 시기에 대한 자세한 내용은 33-15 페이지의 "타입 라이브러리"를 참조하십시오.

타입 라이브러리에는 다음과 같은 내용이 포함될 수 있습니다.

- 별칭 (Alias), 열거 (Enumeration), 구조 (Structure) 및 유니온 (Union)과 같은 사용자 지정 데이터 타입에 대한 정보
- interface, dispinterface 또는 CoClass와 같은 하나 이상의 COM 요소에 대한 설명 (이 설명들을 모두 *타입 정보*라고 합니다.)
- 외부 유닛에 정의된 상수 및 메소드에 대한 설명
- 다른 타입 라이브러리의 타입 설명에 대한 참조

COM 애플리케이션이나 ActiveX 라이브러리에 타입 라이브러리를 포함시키면 COM의 타입 라이브러리 도구와 인터페이스를 통해 다른 애플리케이션 및 프로그래밍 도구에서 애플리케이션의 객체에 대한 정보를 사용할 수 있습니다.

이전의 개발 도구를 사용하는 경우 IDL (Interface Definition Language)이나 ODL (Object Description Language)로 스크립트를 작성하여 타입 라이브러리를 만든 다음 컴파일러를 통해 해당 스크립트를 실행하십시오. Type Library 에디터는 이러한 프로세스의 일부를 자동화하여 사용자가 직접 타입 라이브러리를 만들고 수정하는 수고를 덜어 줍니다.

Delphi의 마법사를 사용하여 ActiveX 컨트롤, Automation 객체, 원격 데이터 모듈 등 모든 타입의 COM 서버를 만들 때 마법사는 자동으로 타입 라이브러리를 생성합니다 (COM 객체 마법사의 경우에는 옵션입니다.). 생성된 객체를 사용자 지정할 때 수행하는 대부분의 작업은 타입 라이브러리에서 시작됩니다. 타입 라이브러리에서 클라이언트에 노출되는 속성 및 메소드를 정의하고 Type Library 에디터를 사용하여 마법사에서 생성한

CoClass의 인터페이스를 변경하기 때문입니다. Type Library 에디터는 객체의 구현 유닛을 자동으로 업데이트하므로 사용자는 생성된 메소드의 본문만 입력하면 됩니다.

Delphi Type Library 에디터는 CORBA (Common Object Request Broker Architecture) 애플리케이션을 개발할 때 사용할 수도 있습니다. 이전의 CORBA 도구를 사용하는 경우에는 CORBA IDL (Interface Definition Language) 을 사용하여 애플리케이션에서 별도로 객체 인터페이스를 정의해야 합니다. 그런 다음 해당 정의에서 스텝 코드 및 골격 코드를 생성하는 유틸리티를 실행합니다. 그러나 Delphi는 스텝, 골격 및 IDL을 자동으로 생성합니다. Type Library 에디터를 사용하면 인터페이스를 쉽게 편집할 수 있으며 Delphi에서 적절한 소스 파일을 자동으로 업데이트합니다.

Type Library 에디터

개발자는 Type Library 에디터를 사용하여 COM 객체의 타입 정보를 조사하고 만들 수 있습니다. Type Library 에디터를 사용하면 인터페이스, CoClass 및 타입을 정의하고, 새 인터페이스의 GUID를 가져오고, CoClass와 인터페이스를 연결하고, 구현 유닛을 업데이트하는 등의 작업을 중앙으로 집중하여 COM 객체를 개발하는 작업이 매우 간단해질 수 있습니다.

참고 또한 Type Library 에디터는 CORBA 객체 또는 CORBA Data Module 마법사를 사용하는 프로젝트에서 CORBA 인터페이스를 정의하는 데 사용됩니다.

Type Library 에디터는 타입 라이브러리의 내용을 나타내는 두 가지 형식의 파일을 출력합니다.

표 34.1 Type Library 에디터 파일

파일	설명
.TLB 파일	바이너리 타입 라이브러리 파일입니다. 기본적으로 타입 라이브러리가 리소스로서 애플리케이션에 자동으로 컴파일되므로 이 파일을 사용하지 않아도 됩니다. 그러나 이 파일을 사용하면 타입 라이브러리를 다른 프로젝트에 명시적으로 컴파일하거나 .exe 또는 .ocx에서 별도로 타입 라이브러리를 배포할 수 있습니다. 자세한 내용은 34-19 페이지의 "기존 타입 라이브러리 열기" 및 34-26 페이지의 "타입 라이브러리 배포"를 참조하십시오. 참고: CORBA 인터페이스에 Type Library 에디터를 사용하면 Type Library 에디터는 .tlb 파일을 만들지 않습니다.
_TLB 유닛	이 유닛은 애플리케이션에서 사용되는 타입 라이브러리의 내용을 해석합니다. 애플리케이션이 타입 라이브러리에 정의된 요소를 사용해야 하는 모든 선언을 포함합니다. 코드 에디터에서 이 파일을 열 수 있지만 편집해서는 안 됩니다. 이 파일은 Type Library 에디터에서 관리하므로 모든 변경 내용은 Type Library 에디터에서 덮어쓰게 됩니다. 이 파일의 내용에 대한 자세한 내용은 35-5 페이지의 "타입 라이브러리 정보를 import할 때 생성되는 코드"를 참조하십시오. 참고: CORBA 인터페이스에 Type Library 에디터를 사용하면 이 유닛은 CORBA 애플리케이션에 필요한 스텝 및 골격 객체를 정의합니다.

Type Library 에디터의 각 부분

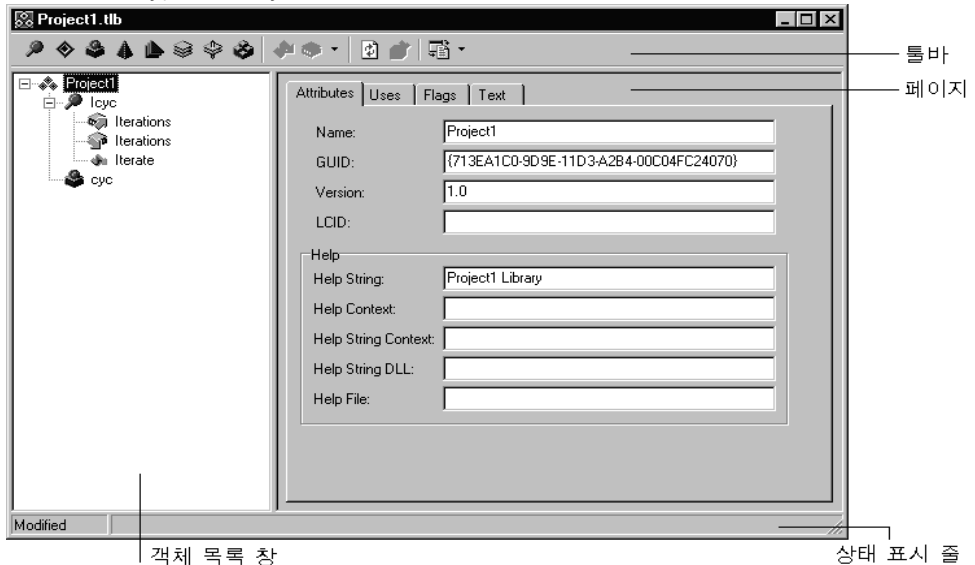
Type Library 에디터의 주요 부분은 다음 표 34.2에 설명되어 있습니다.

표 34.2 Type Library 에디터의 각 부분

부분	설명
툴바	새로운 타입, CoClass, 인터페이스 및 인터페이스 멤버 타입 라이브러리에 추가할 때 사용하는 버튼들이 들어 있습니다. 툴바에는 구현 유닛을 새로 고치고, 타입 라이브러리를 등록하며, 타입 라이브러리의 정보가 있는 IDL 파일을 저장하는 버튼도 들어 있습니다.
객체 목록 창	기존의 모든 요소를 타입 라이브러리에 표시합니다. 객체 목록 창의 항목을 클릭하면 해당 객체에 유효한 페이지가 표시됩니다.
상태 표시줄	타입 라이브러리에 잘못된 타입을 추가하려고 하면 구문 오류를 표시합니다.
페이지	선택한 객체에 대한 정보를 표시합니다. 여기서 표시되는 페이지는 선택한 객체의 타입에 따라 달라집니다.

이 부분들은 `cyc`라는 이름의 COM 객체 타입 정보를 표시하는 Type Library 에디터를 보여 주는 그림 34.1에 설명되어 있습니다.

그림 34.1 Type Library 에디터











툴바

Type Library 에디터의 툴바는 Type Library 에디터의 맨 위에 있으며 클릭하면 타입 라이브러리에 새 객체를 추가할 수 있는 버튼이 들어 있습니다.

첫 번째 버튼 그룹을 사용하면 타입 라이브러리에 요소를 추가할 수 있습니다. 툴바 버튼을 클릭하면 해당 요소의 아이콘이 객체 목록 창에 나타납니다. 그러면 오른쪽 창에서








그 속성을 사용자 지정할 수 있습니다. 선택한 아이콘의 타입에 따라 다른 페이지 정보가 오른쪽에 나타납니다.

다음 표는 타입 라이브러리에 추가할 수 있는 요소들을 나열한 것입니다.

아이콘	의미
	interface 설명
	dispinterface 설명 (CORBA 인터페이스 정의에는 사용되지 않음)
	CoClass
	열거
	별칭
	레코드
	유니온(Union)
	모듈

객체 목록 창에서 위에 나열된 요소 중 하나를 선택하면 두 번째 버튼 그룹에서 해당 요소에 유효한 멤버를 표시합니다. 예를 들어, Interface를 선택하면 인터페이스 정의에 메소드 및 속성을 추가할 수 있기 때문에 두 번째 상자의 메소드와 속성 아이콘이 활성화됩니다. Enum을 선택하면 두 번째 버튼 그룹이 변경되어 Enum 타입 정보에만 유효한 멤버인 Const 멤버를 표시합니다.

다음 표는 객체 목록 창의 요소에 추가할 수 있는 멤버를 나열한 것입니다.

아이콘	의미
	interface, dispinterface의 메소드 또는 모듈의 엔트리 포인트
	interface 또는 dispinterface의 속성
	쓰기 전용 속성(속성 버튼의 드롭다운 목록에서 사용 가능)
	읽기/쓰기 속성(속성 버튼의 드롭다운 목록에서 사용 가능)
	읽기 전용 속성(속성 버튼의 드롭다운 목록에서 사용 가능)
	레코드 또는 유니온의 필드
	열거 또는 모듈의 상수

세 번째 상자에서는 34-24 페이지의 "타입 라이브러리 정보의 저장 및 등록"에서 설명한 대로 타입 라이브러리를 새로 고치거나, 등록하거나, export하여 IDL 파일로 저장하도록 선택할 수 있습니다.

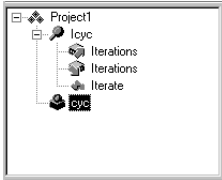
객체 목록 창

객체 목록 창은 현재 타입 라이브러리의 모든 요소를 트리 뷰로 표시합니다. 트리의 루트는 타입 라이브러리 자체를 나타내며 다음과 같은 아이콘으로 표시됩니다.



타입 라이브러리 노드의 자손은 타입 라이브러리의 요소입니다.

그림 34.2 객체 목록 창



타입 라이브러리 자체를 포함하여 이러한 요소들 중 어느 것을 선택하면 오른쪽 창의 타입 정보 페이지가 변경되어 해당 요소에 대한 관련 정보만 반영합니다. 이러한 페이지를 사용하면 선택한 요소의 정의 및 속성을 편집할 수 있습니다.

객체 목록 창에서 마우스 오른쪽 버튼을 클릭하여 객체 목록 창 컨텍스트 메뉴에서 요소를 처리할 수 있습니다. 이 메뉴에는 새로운 요소를 추가하거나 Type Library 에디터의 모양을 사용자 지정하는 명령뿐만 아니라 Windows 클립보드를 사용하여 기존 요소를 이동하거나 복사할 수 있는 명령이 포함됩니다.

상태 표시줄

타입 라이브러리를 편집하거나 저장할 때 구문, 변환 오류 및 경고가 상태 표시줄 창에 나열됩니다.

예를 들어, Type Library 에디터가 지원하지 않는 타입을 지정하면 구문 오류가 나타납니다. Type Library 에디터가 지원하는 타입의 전체 목록을 보려면 34-11 페이지의 "유효한 타입"을 참조하십시오.

타입 정보 페이지

객체 목록 창에서 요소를 선택하면 선택한 요소에 유효한 타입 정보 페이지가 Type Library 에디터에 나타납니다. 표시되는 페이지는 다음과 같이 객체 목록 창에서 선택한 요소에 따라 달라집니다.

표 34.3 타입 라이브러리 페이지

타입 정보 요소	페이지 타입 정보	페이지 내용
타입 라이브러리	속성	타입 라이브러리의 이름, 버전, GUID 및 타입 라이브러리를 도움말에 연결하는 정보.
	사용	이 페이지에서 이용하는 정의가 들어 있는 다른 타입 라이브러리의 목록.
	플래그	다른 애플리케이션에서 타입 라이브러리를 사용할 수 있는 방법을 나타내는 플래그.
	텍스트	타입 라이브러리 자체를 정의하는 선언 및 모든 정의(아래 설명 참조).
Interface	속성	인터페이스의 이름, 버전, GUID, 자손인 인터페이스의 이름 및 인터페이스를 도움말에 연결하는 정보.
	플래그	인터페이스가 숨김인지, 이중인지, Automation 호환인지, 확장 가능한지를 나타내는 플래그.
	텍스트	인터페이스에 대한 선언 및 정의(아래 설명 참조).
Dispinterface	속성	인터페이스의 이름, 버전 및 GUID와 도움말에 연결하는 정보.
	플래그	Dispinterface가 숨김인지, 이중인지, 확장 가능한지를 나타내는 플래그.
	텍스트	Dispinterface에 대한 선언 및 정의(아래 설명 참조).
CoClass	속성	CoClass의 이름, 버전, GUID 및 도움말에 연결하는 정보.
	구현	CoClass가 구현하는 인터페이스와 그 속성 목록.
	COM+	트랜잭션 모델, 호출 동기화, just-in-time 활성화, 객체 풀링 등과 같은 트랜잭션 객체의 속성. COM+ 이벤트 객체의 속성도 포함.
	플래그	클라이언트가 인스턴스를 만들고 사용하는 방법, 브라우저에서 사용자에게 표시되는지 여부, ActiveX 컨트롤인지 여부, 합성의 부분 역할을 하여 집계될 수 있는지 여부를 포함하여 CoClass의 다양한 속성을 나타내는 플래그.
열거	텍스트	CoClass에 대한 선언 및 정의(아래 설명 참조).
	속성	열거의 이름, 버전 및 GUID와 도움말에 연결하는 정보.
별칭	텍스트	열거 타입에 대한 선언 및 정의(아래 설명 참조).
	속성	열거의 이름, 버전 및 GUID, 별칭이 나타내는 타입, 도움말에 연결하는 정보.
레코드	텍스트	별칭에 대한 선언 및 정의(아래 설명 참조).
	속성	레코드의 이름, 버전 및 GUID와 도움말에 연결하는 정보.
유니온	텍스트	레코드에 대한 선언 및 정의(아래 설명 참조).
	속성	유니온의 이름, 버전 및 GUID와 도움말에 연결하는 정보.
	텍스트	유니온에 대한 선언 및 정의(아래 설명 참조).

표 34.3 타입 라이브러리 페이지 (계속)

타입 정보 요소	페이지 타입 정보	페이지 내용
모듈	속성	모듈의 이름, 버전, GUID 및 관련 DLL과 도움말에 연결하는 정보.
	텍스트	모듈에 대한 선언 및 정의(아래 설명 참조).
메소드	속성	메소드의 이름, 디스패치 ID 또는 DLL 엔트리 포인트 및 도움말에 연결하는 정보.
	매개변수	메소드 반환 타입 및 그 타입과 변경자를 가진 모든 매개변수 목록.
	플래그	클라이언트가 메소드를 보고 사용하는 방법, 인터페이스의 기본 메소드인지 여부와 대체 가능한지 여부를 나타내는 플래그.
속성	텍스트	메소드에 대한 선언 및 정의(아래 설명 참조).
	속성	속성 액세스 메소드 (getter 대 setter) 의 이름, 디스패치 ID, 타입 및 도움말에 연결하는 정보.
	매개변수	속성 액세스 메소드 반환 타입과 타입 및 변경자를 가진 모든 매개변수 목록.
	플래그	클라이언트가 속성을 보고 사용하는 방법, 인터페이스의 기본값인지 여부, 속성이 교체 가능한지, 바인딩 가능한지 여부 등을 나타내는 플래그.
Const	텍스트	속성 액세스 메소드에 대한 선언 및 정의(아래 설명 참조).
	속성	모듈 상수의 이름, 값, 타입 및 도움말에 연결하는 정보.
	플래그	클라이언트가 상수를 보고 사용하는 방법, 기본값을 나타내는지 여부, 상수가 바인딩 가능한지 여부 등을 나타내는 플래그.
필드	텍스트	상수에 대한 선언 및 정의(아래 설명 참조).
	속성	이름, 타입 및 도움말에 연결하는 정보.
	플래그	클라이언트가 필드를 보고 사용하는 방법, 기본값을 나타내는지 여부, 필드가 바인딩 가능한지 여부 등을 나타내는 플래그.
	텍스트	필드에 대한 선언 및 정의(아래 설명 참조).

참고 타입 정보 페이지에서 설정할 수 있는 다양한 옵션에 대한 자세한 내용은 Type Library 에디터의 온라인 도움말을 참조하십시오.

각 타입 정보 페이지를 사용하여 표시되는 값을 보거나 편집할 수 있습니다. 해당 선언에 대한 구문을 알 필요 없이 목록에서 값을 선택하거나 직접 입력할 수 있도록 대부분의 페이지는 정보를 컨트롤 집합으로 구성하고 있습니다. 이렇게 하면 제한된 집합에서 값을 지정할 때 오타와 같은 작은 실수를 방지할 수 있습니다. 그러나 선언에 직접 값을 입력하는 것이 더 빠를 수도 있습니다. 직접 값을 입력하려면 텍스트 페이지를 사용하십시오.

모든 타입 라이브러리 요소에는 해당 요소에 대한 구문을 표시하는 텍스트 페이지가 있습니다. 이 구문은 Microsoft Interface Definition Language의 IDL 서브셋 또는 오브젝트 파스칼에 나타납니다. 요소의 다른 페이지에서 변경한 모든 내용은 텍스트 페이지에 반영됩니다. 텍스트 페이지에 직접 코드를 추가하면 변경 내용은 Type Library 에디터의 다른 페이지에 반영됩니다.

현재 에디터에서 지원하지 않는 식별자를 추가하면 Type Library 에디터에서 구문 오류가 발생합니다. 현재 에디터는 Microsoft IDL 컴파일러에서 C++ 코드 생성이나 마샬링 지원에 사용하는 RPC 지원이나 구문이 아닌 타입 라이브러리 지원과 관련된 식별자만 지원합니다.

타입 라이브러리 요소

처음에는 Type Library 인터페이스가 매우 복잡해 보일 수 있습니다. 이것은 타입 라이브러리가 각각의 고유한 특징을 가진 많은 요소들에 대한 정보를 나타내기 때문입니다. 그러나 이러한 특징 중 많은 부분은 모든 요소에 공통적인 특징입니다. 예를 들어, 타입 라이브러리 자체를 포함한 모든 요소에는 다음과 같은 특징이 있습니다.

- 이름은 요소를 설명하는 데 사용되고 코드의 요소를 참조할 때 사용됩니다.
- GUID(Globally Unique Identifier)는 COM이 요소를 식별하는 데 사용하는 전역적으로 고유한 128비트 값입니다. 타입 라이브러리 자체와 CoClass 및 Interface에 대해서는 항상 GUID가 있어야 합니다. 그 외의 경우는 옵션입니다.
- 버전 번호는 요소의 여러 버전을 구분합니다. 버전 번호는 항상 옵션입니다. 그러나 CoClass와 Interface는 버전 번호가 없으면 일부 도구를 사용할 수 없기 때문에 반드시 있어야 합니다.
- 요소를 도움말 항목에 연결하는 정보입니다. 이 정보에는 도움말 문자열 및 도움말 컨텍스트 또는 도움말 문자열 컨텍스트 값이 포함됩니다. 도움말 컨텍스트는 타입 라이브러리가 독립 실행형 도움말 파일을 가지고 있는 종래의 Windows 도움말 시스템에 사용됩니다. 도움말 문자열 컨텍스트는 별도의 DLL로 도움말이 제공될 때 사용됩니다. 도움말 컨텍스트 또는 도움말 문자열 컨텍스트는 타입 라이브러리의 속성 페이지에 지정된 도움말 파일이나 DLL을 참조합니다. 이 정보는 항상 옵션입니다.

Interface

Interface는 가상 함수 테이블(Vtable)을 통해 액세스해야 하는 객체의 메소드와 'get' 및 'set' 함수로 표현되는 속성을 설명합니다. interface에 이중이라는 플래그가 설정되어 있으면 dispinterface도 포함되며 OLE Automation을 통해 액세스할 수 있습니다. 기본적으로 타입 라이브러리는 이중으로 추가한 모든 interface에 플래그를 설정합니다.

interface에는 메소드와 속성이라는 멤버를 할당할 수 있습니다. 이 멤버들은 interface 노드의 자식으로 객체 목록 창에 나타납니다. interface의 속성은 속성의 기본 데이터를 읽고 쓰는 데 사용되는 'get' 및 'set' 메소드로 표시되며 속성의 용도를 나타내는 특수한 아이콘을 사용하여 트리 뷰로 표시됩니다.

참고 속성을 참조별 작성으로 지정하면 값으로가 아닌 포인터로 속성이 전달된다는 의미입니다. Visual Basic과 같은 일부 애플리케이션에서는 참조별 작성이 있으면 성능을 최적화하는 데 사용합니다. 값이 아닌 참조별로만 속성을 전달하려면 속성 타입 *참조별 전용*을 사용합니다. 값뿐만 아니라 참조별로 속성을 전달하려면 Read|Write|Write By Ref를 선택합니다. 이 메뉴를 불러 오려면 툴바로 이동하여 속성 아이콘 옆의 화살표를 선택합니다.

일단 툴바 버튼이나 객체 목록 창 컨텍스트 메뉴를 사용하여 속성이나 메소드를 추가하고 나면 속성이나 메소드를 선택하고 타입 정보 페이지를 사용하여 해당 구문 및 속성을 설명합니다.

속성 페이지를 사용하면 IDispatch를 사용하여 호출할 수 있도록 속성이나 메소드에 이름 및 디스패치 ID를 제공할 수 있습니다. 속성에는 타입도 할당합니다. 매개변수 페이지를 사용하여 함수 시그니처를 만들며 이것으로 매개변수를 추가, 제거 및 재정렬하고, 그 타입과 변경자를 설정하며, 함수 반환 타입을 지정할 수 있습니다.

참고 예외를 발생시켜야 하는 interface의 멤버는 HRESULT를 반환하고 실제 반환 값에 대해 반환 값 매개변수(PARAM_RETURN)를 지정해야 합니다. **safecall** 호출 규칙을 사용하여 이 메소드를 선언하십시오.

interface에 속성 및 메소드를 할당할 때는 연결된 CoClass에 암시적으로 할당해야 합니다. 이런 이유에서 Type Library 에디터를 사용하여 CoClass에 속성 및 메소드를 직접 추가하지 않도록 합니다.

Dispinterface

객체의 속성 및 메소드를 설명할 때는 dispinterface보다 interface가 더 일반적으로 사용됩니다. dispinterface는 동적 바인딩을 통해서만 액세스할 수 있지만 interface는 vtable을 통해 정적 바인딩을 가질 수 있습니다.

interface에 메소드 및 속성을 추가하는 것과 같은 방법으로 dispinterface에 메소드 및 속성을 추가할 수 있습니다. 그러나 dispinterface의 속성을 만들 때는 함수 종류나 매개변수 타입을 지정할 수 없습니다.

CoClasses

CoClass는 하나 이상의 interface를 구현하는 유일한 COM 객체를 설명합니다. CoClass를 정의할 때는 구현되는 interface 중 객체의 기본값을 지정해야 하며 경우에 따라 이벤트의 기본 소스인 dispinterface를 지정해야 합니다. Type Library 에디터에서 CoClass에 속성이나 메소드를 추가해서는 안 됩니다. 속성 및 메소드는 interface에 의해 클라이언트에 노출되며 interface는 구현 페이지를 사용하여 CoClass와 연결됩니다.

타입 정의

열거, 별칭, 레코드 및 유니온은 모두 타입 라이브러리의 어느 곳에서나 선언한 다음 사용할 수 있는 타입을 선언합니다.

Enum은 상수 목록으로 구성되며 각 상수는 숫자여야 합니다. 일반적으로 숫자 입력은 10진수 또는 16진수 형식의 정수입니다. 기본적으로 기본값은 0입니다. 객체 목록 창에서 열거를 선택하고 툴바의 Const 버튼을 클릭하거나 객체 목록 창 컨텍스트 메뉴에서 New|Const 명령을 선택하여 열거에 상수를 추가할 수 있습니다.

참고 열거의 의미를 더 명확하게 하려면 열거에 도움말 문자열을 제공하는 것이 좋습니다. 다음은 마우스 버튼에 대한 열거 타입의 예제 항목이며 각 열거 요소에 대한 도움말 문자열이 포함됩니다.

```
mbLeft = 0 [helpstring 'mbLeft'];  
mbRight = 1 [helpstring 'mbRight'];  
mbMiddle = 3 [helpstring 'mbMiddle'];
```

별칭은 타입의 별칭(타입 정의)을 만듭니다. 별칭을 사용하면 레코드나 유니온과 같은 다른 타입 정보에 사용할 타입을 정의할 수 있습니다. 속성 페이지에서 타입 속성을 설정하여 기본 타입 정의에 별칭을 연결합니다.

레코드는 구조 멤버 또는 필드 목록으로 구성됩니다. 유니온은 가변 부분만 있는 레코드입니다. 레코드처럼 유니온도 구조 멤버 또는 필드 목록으로 구성됩니다. 그러나 레코드의 멤버와 달리 유니온의 각 멤버는 같은 실제 주소를 사용하므로 하나의 논리 값만 저장할 수 있습니다.

객체 목록 창에서 필드를 선택하고 툴바의 필드 버튼을 클릭하거나 마우스 오른쪽 버튼을 클릭하여 객체 목록 창 컨텍스트 메뉴에서 필드를 선택하여 레코드나 유니온에 필드를 추가합니다. 각 필드에는 필드를 선택하고 속성 페이지를 사용하여 값을 할당하는 이름과 타입이 있습니다. 레코드와 유니온을 옵션인 태그로 정의할 수 있습니다.

멤버는 기본 제공된 타입으로 구성되거나 레코드를 정의하기 전에 별칭을 사용하여 타입을 지정할 수 있습니다.

모듈

모듈은 일반적으로 DLL 엔트리 포인트의 집합인 함수의 그룹을 정의합니다. 다음과 같이 모듈을 정의합니다.

- 속성 페이지에서 모듈이 나타내는 DLL을 지정합니다.
- 툴바나 객체 목록 창 컨텍스트 메뉴를 사용하여 메소드 및 상수를 추가합니다. 그런 다음 객체 목록 창에서 메소드나 상수를 선택하고 속성 페이지에서 그 값을 설정하여 각 메소드나 상수에 대해 속성을 정의해야 합니다.

모듈 메소드의 경우에는 속성 페이지를 사용하여 이름과 DLL 엔트리 포인트를 할당해야 합니다. 매개변수 페이지를 사용하여 함수의 매개변수 및 반환 타입을 선언합니다.

모듈 상수의 경우에는 속성 페이지를 사용하여 이름, 타입 및 값을 지정합니다.

참고 Type Library 에디터가 모듈과 관련된 모든 선언이나 구현을 생성하지는 않습니다. 지정된 DLL은 별도의 프로젝트로 만들어야 합니다.

Type Library 에디터 사용

Type Library 에디터를 사용하면 새로운 타입 라이브러리를 만들거나 기존 타입 라이브러리를 편집할 수 있습니다. 일반적으로 애플리케이션 개발자는 마법사를 사용하여 타입 라이브러리에 나타나는 객체를 만들고 Delphi가 자동으로 타입 라이브러리를 생성할 수 있게 합니다. 그런 다음 interface를 정의하거나 수정하고 타입 정의를 추가하는 등의 작업을 수행할 수 있도록 자동으로 생성된 타입 라이브러리를 Type Library 에디터에서 엽니다.

그러나 객체를 정의하는 데 마법사를 사용하지 않는 경우에도 Type Library 에디터를 사용하면 새로운 타입 라이브러리를 정의할 수 있습니다. 이런 경우에는 마법사가 타입 라이브러리와 연결하지 않은 CoClass에 대한 코드를 Type Library 에디터가 생성하지 않기 때문에 모든 구현 클래스를 사용자가 직접 만들어야 합니다.

에디터는 아래에서 설명하는 대로 타입 라이브러리에 있는 유효한 타입의 서브셋을 지원합니다.

이 단원의 마지막 항목에서는 다음 작업들을 설명합니다.

- 새로운 타입 라이브러리 만들기
- 기존 타입 라이브러리 열기
- 타입 라이브러리에 interface 추가
- interface 수정
- 타입 라이브러리에 속성 및 메소드 추가
- 타입 라이브러리에 CoClass 추가
- CoClass에 interface 추가
- 타입 라이브러리에 열거 추가
- 타입 라이브러리에 별칭 추가
- 타입 라이브러리에 레코드나 유니온 추가
- 타입 라이브러리에 모듈 추가
- 타입 라이브러리 정보 저장 및 등록

유효한 타입

Type Library 에디터에서는 IDL에서 작업하는지 오브젝트 파스칼에서 작업하는지 여부에 따라 다른 타입 식별자를 사용합니다. Environment options 대화 상자에서 사용할 랭귀지를 지정합니다.

다음 타입들은 COM 개발용 타입 라이브러리에서 사용할 수 있습니다. Automation 호환 열은 Automation 또는 Dispinterface 플래그가 확인된 interface에서 타입을 사용할 수 있는지 여부를 지정합니다. 이러한 타입들은 COM이 타입 라이브러리를 통해 자동으로 마샬링할 수 있습니다.

표 34.4 유효한 타입

파스칼 타입	IDL 타입	가변 타입	Automation 호환	설명
Smallint	short	VT_I2	예	2바이트 부호 있는 정수
Integer	long	VT_I4	예	4바이트 부호 있는 정수
Single	single	VT_R4	예	4바이트 실수
Double	double	VT_R8	예	8바이트 실수
Currency	CURRENCY	VT_CY	예	통화
TDateTime	DATE	VT_DATE	예	날짜
WideString	BSTR	VT_BSTR	예	바이너리 문자열
IDispatch	IDispatch	VT_DISPATCH	예	IDispatch interface에 대한 포인터
SCODE	SCODE	VT_ERROR	예	Ole 오류 코드

표 34.4 유효한 타입 (계속)

파스칼 타입	IDL 타입	가변 타입	Automation 호환	설명
WordBool	VARIANT_BOOL	VT_BOOL	예	True = -1, False = 0
OleVariant	VARIANT	VT_VARIANT	예	Ole 가변
IUnknown	IUnknown	VT_UNKNOWN	예	IUnknown interface에 대한 포인터
Shortint	byte	VT_I1	아니오	1바이트 부호 있는 정수
Byte	unsigned char	VT_UI1	예	1바이트 부호 없는 정수
Word	unsigned short	VT_UI2	예*	2바이트 부호 없는 정수
LongWord	unsigned long	VT_UI4	예*	4바이트 부호 없는 정수
Int64	__int64	VT_I8	아니오	8바이트 부호 있는 정수
Largeuint	uint64	VT_UI8	아니오	8바이트 부호 없는 정수
SYSINT	int	VT_INT	예*	시스템 종속 정수 (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	예*	시스템 종속 부호 없는 정수
HResult	HRESULT	VT_HRESULT	아니오	32비트 오류 코드
Pointer		VT_PTR -> VT_VOID	아니오	타입이 지정되지 않은 포인터
SafeArray	SAFEARRAY	VT_SAFEARRAY	아니오	OLE 안전 배열
PChar	LPSTR	VT_LPSTR	아니오	Char에 대한 포인터
PWideChar	LPWSTR	VT_LPWSTR	아니오	WideChar에 대한 포인터

* Word, LongWord, SYSINT 및 SYSUINT는 대부분의 애플리케이션에서 Automation 호환이지만 이전 애플리케이션에는 호환되지 않을 수 있습니다.

참고 Byte (VT_UI1) 는 Automation 호환이지만 많은 Automation 서버에서 이 값을 제대로 처리하지 못하기 때문에 가변 또는 OleVariant에서 허용되지 않습니다.

이러한 IDL 타입뿐만 아니라 라이브러리에 정의되거나 참조된 라이브러리에 정의된 모든 인터페이스와 타입은 타입 라이브러리 정의에서 사용할 수 있습니다.

Type Library 에디터는 생성된 타입 라이브러리 (.TLB) 파일에 바이너리 형식으로 타입 정보를 저장합니다.

매개변수 타입을 Pointer 타입으로 지정하면 일반적으로 Type Library 에디터는 해당 타입을 가변 매개변수로 변환합니다. 타입 라이브러리를 저장하면 가변 매개변수와 연결된 ElemDesc의 IDL 플래그는 IDL_FIN 또는 IDL_FOUT로 표시됩니다.

중중 타입이 Pointer보다 우선할 때 ElemDesc IDL 플래그는 IDL_FIN이나 IDL_FOUT로 표시되지 않습니다. 또한 dispinterface의 경우 일반적으로 IDL 플래그가 사용되지 않습니다. 이런 경우에는 {IDL_None} 이나 {IDL_In} 같은 변수 식별자 옆에 설명이 있습니다. 이러한 설명은 타입 라이브러리를 저장할 때 IDL 플래그를 정확하게 표시하는 데 사용됩니다.

SafeArray

COM은 SafeArray 라는 특수한 데이터 타입을 통해 전달되는 배열을 필요로 합니다. 이 작업을 수행하기 위해 특수한 COM 함수를 호출하여 SafeArray 를 만들고 제거할 수 있으며 SafeArray 내 모든 요소는 유효한 Automation 호환 타입이어야 합니다. Delphi 컴파일러는 COM SafeArray 에 대한 기본 제공 (Built-in) 을 인식하며 자동으로 COM API를 호출하여 SafeArray 를 만들고, 복사하고, 제거합니다.

Type Library 에디터에서 *SafeArray* 는 요소의 타입을 지정해야 합니다. 예를 들어, 텍스트 페이지의 다음 줄은 매개변수가 Integer 라는 요소 타입을 가진 *SafeArray* 인 메소드를 선언합니다.

```
procedure HighLightLines(Lines: SafeArray of Integer);
```

참고 Type Library 에디터에서 *SafeArray* 타입을 선언할 때 요소 타입을 지정해야 하지만 생성된 _TLB 유틸의 선언은 요소 타입을 나타내지 않습니다.

오브젝트 파스칼 또는 IDL 구문 사용

Type Library 에디터의 텍스트 페이지는 다음 두 가지 방법 중 하나로 타입 정보를 표시합니다.

- 오브젝트 파스칼 구문의 확장자 사용
- Microsoft IDL 사용

참고 CORBA 객체에서 작업할 때는 텍스트 페이지에서 두 방법 모두 사용하지 않습니다. 그 대신 CORBA IDL을 사용해야 합니다.

Environment Options 대화 상자에서 설정을 변경하여 사용할 랭귀지를 선택할 수 있습니다. Tools|Environment Options를 선택하고 대화 상자의 Type Library 페이지에서 파스칼 또는 IDL을 랭귀지로 지정합니다.

참고 오브젝트 파스칼이나 IDL 구문을 선택하면 매개변수 속성 페이지의 선택에도 영향을 줍니다.

일반적으로 오브젝트 파스칼 애플리케이션처럼 타입 라이브러리의 식별자는 대소문자를 구분하지 않습니다. 최대 255자까지 사용할 수 있으며 문자나 밑줄(_)로 시작해야 합니다.

속성 사양

오브젝트 파스칼은 타입 라이브러리가 속성 사양을 포함할 수 있도록 확장되었습니다. 속성 사양은 대괄호로 둘러싸여 표시되고 쉼표로 구분됩니다. 각 속성 사양은 속성 이름과 경우에 따라 그 다음에 오는 값으로 구성됩니다.

다음 표는 속성 이름과 그에 해당하는 값을 나열한 것입니다.

표 34.5 속성 구문

속성 이름	예	적용 대상
aggregatable	[aggregatable]	typeinfo
appobject	[appobject]	CoClass typeinfo

표 34.5 속성 구문 (계속)

속성 이름	예	적용 대상
bindable	[bindable]	CoClass 멤버 이외의 멤버
control	[control]	타입 라이브러리, typeinfo
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	모두
default	[default]	CoClass 멤버
defaultbind	[defaultbind]	CoClass 멤버 이외의 멤버
defaultcollection	[defaultcollection]	CoClass 멤버 이외의 멤버
defaultvtbl	[defaultvtbl]	CoClass 멤버
dispid	[dispid]	CoClass 멤버 이외의 멤버
displaybind	[displaybind]	CoClass 멤버 이외의 멤버
dllname	[dllname 'Helper.dll']	모듈 typeinfo
dual	[dual]	인터페이스 typeinfo
helpfile	[helpfile 'c:\help\myhelp.hlp']	타입 라이브러리
helpstringdll	[helpstringdll 'c:\help\myhelp.dll']	타입 라이브러리
helpcontext	[helpcontext 2005]	CoClass 멤버와 매개변수 이외에 모두
helpstring	[helpstring 'payroll interface']	CoClass 멤버와 매개변수 이외에 모두
helpstringcontext	[helpstringcontext \$17]	CoClass 멤버와 매개변수 이외에 모두
hidden	[hidden]	매개변수 이외에 모두
immediatebind	[immediatebind]	CoClass 멤버 이외의 멤버
lcid	[lcid \$324]	타입 라이브러리
licensed	[licensed]	타입 라이브러리, CoClass typeinfo
nonbrowsable	[nonbrowsable]	CoClass 멤버 이외의 멤버
nonextensible	[nonextensible]	인터페이스 typeinfo
oleautomation	[oleautomation]	인터페이스 typeinfo
predeclid	[predeclid]	typeinfo
propget	[propget]	CoClass 멤버 이외의 멤버
propput	[propput]	CoClass 멤버 이외의 멤버
propputref	[propputref]	CoClass 멤버 이외의 멤버
public	[public]	별칭 typeinfo
readonly	[readonly]	CoClass 멤버 이외의 멤버
replaceable	[replaceable]	CoClass 멤버와 매개변수 이외에 모두
requestedit	[requestedit]	CoClass 멤버 이외의 멤버
restricted	[restricted]	매개변수 이외에 모두
source	[source]	모든 멤버
uidefault	[uidefault]	CoClass 멤버 이외의 멤버
usesgetlasterror	[usesgetlasterror]	CoClass 멤버 이외의 멤버

표 34.5 속성 구문 (계속)

속성 이름	예	적용 대상
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}']	타입 라이브러리, typeinfo(필수)
vararg	[vararg]	CoClass 멤버 이외의 멤버
version	[version 1.1]	타입 라이브러리, typeinfo

인터페이스 구문

인터페이스 타입 정보를 선언하는 오브젝트 파스칼 구문의 형식은 다음과 같습니다.

```
interfacename = interface[(baseinterface)] [attributes]
functionlist
[propertylist]
end;
```

예를 들어, 다음은 두 가지 메소드와 한 가지 속성이 있는 인터페이스를 선언하는 텍스트입니다.

```
Interface1 = interface (IDispatch)
[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
function Calculate(optional seed:Integer=0):Integer;
procedure Reset;
procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
function GetRange: Integer;[propget, dispid $00000005]; stdcall;
end;
```

Microsoft IDL의 해당 구문은 다음과 같습니다.

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',version 1.0]
interface Interface1 :IDispatch
{
    long Calculate([in, optional, defaultvalue(0)] long seed);
    void Reset(void);
    [propput, id(0x00000005)] void _stdcall PutRange([in] long Value);
    [propget, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

dispatch 인터페이스 구문

dispinterface 타입 정보를 선언하는 오브젝트 파스칼 구문의 형식은 다음과 같습니다.

```
dispinterfacename = dispinterface [attributes]
functionlist
[propertylist]
end;
```

예를 들어, 다음은 위의 인터페이스와 메소드 및 속성이 같은 dispinterface를 선언하는 텍스트입니다.

```
MyDispObj = dispinterface
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
version 1.0,
helpstring 'dispatch interface for MyObj']
```

```

    function Calculate(seed:Integer):Integer [dispid 1];
    procedure Reset [dispid 2];
    property Range:Integer [dispid 3];
end;

```

Microsoft IDL의 해당 구문은 다음과 같습니다.

```

[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
 version 1.0,
 helpstring "dispatch interface for MyObj"]
dispinterface Interfacel
{
    methods:
    [id(1)] int Calculate([in] int seed);
    [id(2)] void Reset(void);
    properties:
    [id(3)] int Value;
};

```

CoClass 구문

CoClass 타입 정보를 선언하는 오브젝트 파스칼 구문의 형식은 다음과 같습니다.

```

classname = coclass(interfacename[interfaceattributes], ...); [attributes];

```

예를 들어, 다음은 인터페이스 *IMyInt*와 *dispinterface DmyInt*의 *coclass*를 선언하는 텍스트입니다.

```

myapp = coclass(IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 version 1.0,
 helpstring 'A class',
 appobject]

```

Microsoft IDL의 해당 구문은 다음과 같습니다.

```

[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 version 1.0,
 helpstring 'A class',
 appobject]
coclass myapp
{
    methods:
    [source] interface IMyInt);
    dispinterface DMyInt;
};

```

Enum 구문

Enum 타입 정보를 선언하는 오브젝트 파스칼 구문의 형식은 다음과 같습니다.

```

enumname = ([attributes] enumlist);

```

예를 들어, 다음은 값이 세 개인 열거 타입을 선언하는 텍스트입니다.

```

location = ([uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 helpstring 'location of booth']

```

```

    Inside = 1 [helpstring 'Inside the pavillion'];
    Outside = 2 [helpstring 'Outside the pavillion'];
    Offsite = 3 [helpstring 'Not near the pavillion'];);

```

Microsoft IDL의 해당 구문은 다음과 같습니다.

```

[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 helpstring 'location of booth']
typedef enum
{
    [helpstring 'Inside the pavillion'] Inside = 1,
    [helpstring 'Outside the pavillion'] Outside = 2,
    [helpstring 'Not near the pavillion'] Offsite = 3
} location;

```

별칭 구문

별칭 타입 정보를 선언하는 오브젝트 파스칼 구문의 형식은 다음과 같습니다.

```
aliasname = basetype[attributes];
```

예를 들어, 다음은 정수의 별칭으로 DWORD를 선언하는 텍스트입니다.

```
DWORD = Integer [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'];
```

Microsoft IDL의 해당 구문은 다음과 같습니다.

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

레코드 구문

레코드 타입 정보를 선언하는 오브젝트 파스칼 구문의 형식은 다음과 같습니다.

```
recordname = record [attributes] fieldlist end;
```

예를 들어, 다음은 레코드를 선언하는 텍스트입니다.

```

Tasks = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 helpstring 'Task description']
    ID: Integer;
    StartDate: TDate;
    EndDate: TDate;
    Ownername: WideString;
    Subtasks: safearray of Integer;
end;

```

Microsoft IDL의 해당 구문은 다음과 같습니다.

```

[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 helpstring 'Task description']
typedef struct
{
    long ID;
    DATE StartDate;
    DATE EndDate;
    BSTR Ownername;
    SAFEARRAY (int) Subtasks;
} Tasks;

```

유니온 구문

유니온 타입 정보를 선언하는 오브젝트 파스칼 구문의 형식은 다음과 같습니다.

```
unionname = record [attributes]
case Integer of
  0: field1;
  1: field2;
  ...
end;
```

예를 들어, 다음은 유니온을 선언하는 텍스트입니다.

```
MyUnion = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  helpstring 'item description']
case Integer of
  0: (Name:WideString);
  1: (ID:Integer);
  3: (Value:Double);
end;
```

Microsoft IDL의 해당 구문은 다음과 같습니다.

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  helpstring 'item description']
typedef union
{
  BSTR Name;
  long ID;
  double Value;
} MyUnion;
```

모듈 구문

모듈 타입 정보를 선언하는 오브젝트 파스칼 구문의 형식은 다음과 같습니다.

```
modulename = module constants entrypoints end;
```

예를 들어, 다음은 모듈의 타입 정보를 선언하는 텍스트입니다.

```
MyModule = module [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  dllname 'circle.dll']
  PI: Double = 3.14159;
  function area(radius:Double):Double [ entry 1 ]; stdcall;
  function circumference(radius:Double):Double [ entry 2 ]; stdcall;
end;
```

Microsoft IDL의 해당 구문은 다음과 같습니다.

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  dllname("circle.dll")]
module MyModule
{
  double PI = 3.14159;
  [entry(1)] double stdcall area([in] double radius);
  [entry(2)] double stdcall circumference([in] double radius);
};
```

새로운 타입 라이브러리 생성

특정 COM 객체로부터 독립된 타입 라이브러리를 만들 수 있습니다. 예를 들어, 여러 개의 다른 타입 라이브러리에서 사용하는 타입 정의가 포함된 타입 라이브러리를 정의할 수 있습니다. 그런 다음 기본 정의로 타입 라이브러리를 만들고 다른 타입 라이브러리의 uses 페이지에 추가할 수 있습니다.

또한 아직 구현되지 않은 객체에 대한 타입 라이브러리도 만들 수 있습니다. 타입 라이브러리에 인터페이스 정의가 포함되면 COM 객체 마법사를 사용하여 CoClass 및 구현을 생성할 수 있습니다.

다음과 같은 방법으로 새로운 타입 라이브러리를 만듭니다.

- 1 File|New|Other를 선택하여 New Items 대화 상자를 엽니다.
- 2 ActiveX 페이지를 선택합니다.
- 3 Type Library 아이콘을 선택합니다.
- 4 OK를 선택합니다.

Type Library 에디터가 열리고 타입 라이브러리의 이름을 입력하라는 메시지가 표시됩니다.

- 5 타입 라이브러리의 이름을 입력합니다. 이어서 타입 라이브러리에 요소를 추가합니다.

기존 타입 라이브러리 열기

마법사를 사용하여 ActiveX 컨트롤, Automation 객체, 활성 폼, Active Server Page 객체, COM 객체, 트랜잭션 객체, 원격 데이터 모듈 또는 트랜잭션 데이터 모듈을 만들면 타입 라이브러리는 구현 유닛과 함께 자동으로 만들어집니다. 또한 사용자의 시스템에서 사용할 수 있는 다른 제품(서버)과 연결된 타입 라이브러리가 있을 수 있습니다.

다음과 같은 방법으로 현재 프로젝트의 일부가 아닌 타입 라이브러리를 엽니다.

- 1 IDE의 주 메뉴에서 File|Open을 선택합니다.
- 2 Open 대화 상자에서 File Type을 타입 라이브러리에 설정합니다.
- 3 원하는 타입 라이브러리 파일로 이동하여 Open을 선택합니다.

다음과 같은 방법으로 현재 프로젝트와 연결된 타입 라이브러리를 엽니다.

- 1 View|Type Library를 선택합니다.

참고 CORBA Object 마법사를 사용할 때는 View|Type Library를 선택하여 CORBA 객체 인터페이스를 편집할 수도 있습니다. 기술적으로 말해서 타입 라이브러리를 볼 수 있는 것은 아니지만 거의 같은 방법으로 타입 라이브러리를 사용할 수 있습니다.

이제 인터페이스, CoClass 및 열거, 속성, 메소드와 같은 타입 라이브러리의 다른 요소를 추가할 수 있습니다.

참고 Type Library 에디터를 사용하여 타입 라이브러리 정보에서 변경한 내용은 연결된 구현 클래스에 자동으로 반영될 수 있습니다. 변경 내용을 추가하기 전에 검토하려면 Apply Updates 대화 상자가 열려 있는지 확인하십시오. 이 대화 상자는 기본적으로 열려

있으며 Tools|Environment Options|Type Library 페이지에서 "Display updates before refreshing" 설정을 변경할 수 있습니다. 자세한 내용은 34-25 페이지의 "Apply Updates 대화 상자"를 참조하십시오.

- 팁** 클라이언트 애플리케이션을 작성할 때는 타입 라이브러리를 열지 않아도 됩니다. 타입 라이브러리 자체가 아니라 Type Library 에디터가 타입 라이브러리에서 만드는 *Project_TLB* 유닛만 필요합니다. 이 파일을 직접 클라이언트 프로젝트에 추가하거나 시스템에 타입 라이브러리를 등록한 경우에는 Import Type Library 대화 상자 (Project|Import Type Library)를 사용할 수 있습니다.

타입 라이브러리에 interface 추가

다음과 같은 방법으로 interface를 추가합니다.

- 1 툴바에서 interface 아이콘을 클릭합니다.

interface가 객체 목록 창에 추가되고 이름을 추가하라는 메시지가 표시됩니다.

- 2 interface의 이름을 입력합니다.

새 interface에는 필요에 따라 수정할 수 있는 기본 속성이 포함됩니다.

인터페이스의 용도에 맞게 getter/setter 함수로 표시되는 속성 및 메소드를 추가할 수 있습니다.

타입 라이브러리를 사용하여 interface 수정

interface나 dispinterface를 만든 후에 수정할 수 있는 몇 가지 방법이 있습니다.

- 변경하려는 정보가 포함된 타입 정보 페이지를 사용하여 interface의 속성을 변경할 수 있습니다. 객체 목록 창에서 interface를 선택한 다음 적절한 타입 정보 페이지에서 컨트롤을 사용합니다. 예를 들어, 속성 페이지를 사용하여 부모 인터페이스를 변경하거나 플래그 페이지를 사용하여 이중 인터페이스인지 여부를 변경할 수 있습니다.
- 객체 목록 창에서 인터페이스를 선택한 다음 텍스트 페이지에서 선언을 편집하면 인터페이스 선언을 직접 편집할 수 있습니다.
- 인터페이스에 속성 및 메소드를 추가할 수도 있습니다(다음 단원 참조).
- 속성 및 메소드의 타입 정보를 변경하면 인터페이스에서 미리 속성 및 메소드를 수정할 수 있습니다.
- 객체 목록 창에서 CoClass를 선택하고 구현 페이지에서 마우스 오른쪽 버튼을 클릭한 다음 Insert Interface를 선택하여 CoClass에 연결할 수 있습니다.

- 참고** 타입 라이브러리를 사용하여 CORBA 인터페이스를 추가할 때 속성 페이지에 있는 대부분의 정보는 관련이 없습니다. 또한 플래그 페이지도 필요하지 않습니다.

마법사에서 생성한 CoClass와 interface를 연결하면 툴바의 Refresh 버튼을 클릭하여 구현 파일에 변경 내용을 적용하도록 Type Library 에디터에 알려 줄 수 있습니다. Apply Updates 대화 상자를 활성화하면 Type Library 에디터는 소스를 업데이트하기

전에 알려 주고 발생할 수 있는 문제에 대해 경고합니다. 예를 들어, 실수로 이벤트 인터페이스의 이름을 바꾸면 소스 파일에 다음과 같은 메시지가 나타날 수 있습니다.

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name.As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

또한 바로 위에 있는 소스 파일에 TODO 설명이 삽입됩니다.

경고 이 경고와 TODO 설명을 무시하면 코드가 컴파일되지 않습니다.

interface나 dispinterface에 속성 및 메소드 추가

다음과 같은 방법으로 인터페이스나 dispinterface에 속성이나 메소드를 추가합니다.

- 1 인터페이스를 선택하고 툴바에서 속성 또는 메소드 아이콘을 선택합니다. 속성을 추가하려면 속성 아이콘을 직접 클릭하여 getter와 setter가 모두 있는 읽기/쓰기 속성을 만들거나 아래쪽 화살표를 클릭하여 속성 타입의 메뉴를 표시합니다.

속성 액세스 메소드 멤버 또는 메소드 멤버가 객체 목록 창에 추가되고 이름을 추가 하라는 메시지가 표시됩니다.

- 2 멤버의 이름을 입력합니다.

새 멤버에는 멤버에 맞게 수정할 수 있는 속성, 매개변수 및 플래그 페이지의 기본 설정이 포함됩니다. 예를 들어, 속성 페이지에서 속성에 타입을 할당하려고 할 수 있습니다. 메소드를 추가하려는 경우라면 매개변수 페이지에서 해당 매개변수를 지정할 수 있습니다.

다른 방법으로는 파스칼이나 IDL 구문을 사용하여 텍스트 페이지로 직접 입력하여 속성 및 메소드를 추가할 수 있습니다. 예를 들어, 파스칼 구문으로 작업하는 경우에는 인터페이스의 텍스트 페이지에 다음과 같은 속성 선언을 입력할 수 있습니다.

```
Interface1 = interface(IDispatch)
  [ uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
    version 1.0,
    dual,
    oleautomation ]
  function AutoSelect: Integer [propget, dispid $00000002]; safecall; // Add this
  function AutoSize: WordBool [propget, dispid $00000001]; safecall; // And this
  procedure AutoSize(Value: WordBool) [propput, dispid $00000001]; safecall; // And
  this
end;
```

IDL로 작업하는 경우에는 같은 선언을 다음과 같이 추가할 수 있습니다.

```
[
  uuid(5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
  version(1.0),
  dual,
  oleautomation
]
interface Interface1:IDispatch
{ // Add everything between the curly braces
  [propget, id(0x00000002)]
```

```

HRESULT _stdcall AutoSelect([out, retval] long Value );
[propget, id(0x00000003)]
HRESULT _stdcall AutoSize([out, retval] VARIANT_BOOL Value );
[propput, id(0x00000003)]
HRESULT _stdcall AutoSize([in] VARIANT_BOOL Value );
};

```

인터페이스 텍스트 페이지를 사용하여 interface에 멤버를 추가하면 멤버는 각각 고유한 속성, 플래그 및 매개변수 페이지가 있는 별도의 항목으로 객체 목록 창에 나타납니다. 객체 목록 창에서 속성이나 메소드를 선택하고 이 페이지를 사용하거나 텍스트 페이지에서 직접 편집하여 새로운 속성이나 메소드를 각각 수정할 수 있습니다.

인터페이스가 마법사에서 생성한 CoClass와 연결되면 툴바의 Refresh 버튼을 클릭하여 구현 파일에 변경 내용을 적용하도록 Type Library 에디터에 알려 줄 수 있습니다. Type Library 에디터는 새 메소드를 구현 클래스에 추가하여 새 멤버를 반영합니다. 그런 다음 구현 유닛의 소스 코드에서 새 메소드를 찾아 본문을 입력하여 구현을 완료합니다.

Apply Updates 대화 상자를 활성화하면 Type Library 에디터는 소스를 업데이트하기 전에 모든 변경 내용을 알려 주고 발생할 수 있는 문제에 대해 경고합니다.

타입 라이브러리에 CoClass 추가

프로젝트에 CoClass를 추가하는 가장 쉬운 방법은 IDE의 주 메뉴에서 File|New|Other를 선택하고 New Items 대화 상자의 ActiveX 또는 Multitier 페이지에서 적절한 마법사를 사용하는 것입니다. 이 방법의 장점은 CoClass와 그 인터페이스를 타입 라이브러리에 추가할 뿐만 아니라 마법사가 구현 유닛을 추가하고 프로젝트 파일을 업데이트하여 uses 절에 새 구현 유닛을 포함시킨다는 점입니다.

그러나 마법사를 사용하지 않는 경우에는 툴바의 CoClass 아이콘을 클릭한 다음 그 속성을 지정하여 CoClass를 만들 수 있습니다. 속성 페이지에서 새로운 CoClass에 이름을 지정하고 플래그 페이지를 사용하여 CoClass가 애플리케이션 객체인지 또 ActiveX 컨트롤을 나타내는지 여부 등과 같은 정보를 나타낼 수 있습니다.

참고 마법사 대신 툴바를 사용하여 타입 라이브러리에 CoClass를 추가하면 CoClass의 인터페이스 중 하나에 있는 요소를 변경할 때마다 CoClass 자체에 대한 구현을 생성하고 사용자가 직접 업데이트해야 합니다. CoClass에 멤버를 직접 추가할 수는 없습니다. 그 대신 CoClass에 인터페이스를 추가할 때 암시적으로 멤버를 추가합니다.

CoClass에 인터페이스 추가

CoClass는 클라이언트에 표시되는 인터페이스에 의해 정의됩니다. CoClass의 구현 클래스에 많은 속성 및 메소드를 추가해도 클라이언트는 CoClass와 연결된 인터페이스에 의해 노출되는 속성 및 메소드만 볼 수 있습니다.

인터페이스를 CoClass와 연결하려면 클래스의 구현 페이지를 마우스 오른쪽 버튼으로 클릭하고 Insert Interface를 선택하여 선택할 수 있는 인터페이스 목록을 표시합니다. 목록에는 현재의 타입 라이브러리에 정의되어 있는 인터페이스와 현재의 타입 라이브러리가 참조하는 모든 타입 라이브러리에 정의된 인터페이스가 들어 있습니다. 클래스가

구현하고자 하는 인터페이스를 선택합니다. GUID 및 다른 속성이 있는 인터페이스가 페이지에 추가됩니다.

마법사에서 CoClass를 생성하면 Type Library 에디터는 자동으로 구현 클래스를 업데이트하여 이 방법으로 추가한 모든 인터페이스의 메소드 (속성 액세스 메소드 포함)에 대한 골격 메소드를 포함시킵니다. Apply Updates 대화 상자를 활성화하면 Type Library 에디터는 소스를 업데이트하기 전에 알려 주고 발생할 수 있는 문제에 대해 경고합니다.

타입 라이브러리에 열거 추가

다음과 같은 방법으로 타입 라이브러리에 열거를 추가합니다.

- 1 툴바에서 enum 아이콘을 클릭합니다.

enum 타입이 객체 목록 창에 추가되고 이름을 추가하라는 메시지가 나타납니다.

- 2 열거의 이름을 입력합니다.

새 enum은 비어 있고 속성 페이지에는 수정할 수 있는 기본 속성이 포함됩니다.

New Const 버튼을 클릭하여 enum에 값을 추가합니다. 그런 다음 각 열거 값을 선택하고 속성 페이지를 사용하여 이름과 가능하면 값을 지정합니다.

열거를 추가하면 새 타입은 타입 라이브러리나 uses 페이지에서 타입 라이브러리를 참조하는 다른 모든 타입 라이브러리에서 사용할 수 있습니다. 예를 들어, 속성이나 매개변수의 타입으로 열거를 사용할 수 있습니다.

타입 라이브러리에 별칭 추가

다음과 같은 방법으로 타입 라이브러리에 별칭을 추가합니다.

- 1 툴바에서 별칭 아이콘을 클릭합니다.

별칭 타입이 객체 목록 창에 추가되고 이름을 추가하라는 메시지가 나타납니다.

- 2 별칭의 이름을 입력합니다.

기본적으로 새 별칭은 Integer 타입입니다. 속성 페이지를 사용하여 별칭이 표시할 타입으로 이 타입을 변경합니다.

별칭을 추가하면 새 타입은 타입 라이브러리 또는 uses 페이지에서 해당 타입 라이브러리를 참조하는 다른 모든 타입 라이브러리에서 사용할 수 있습니다. 예를 들어, 속성이나 매개변수의 타입으로 별칭을 사용할 수 있습니다.

타입 라이브러리에 레코드 또는 유니온 추가

다음과 같은 방법으로 타입 라이브러리에 레코드 또는 유니온을 추가합니다.

- 1 툴바에서 레코드 아이콘이나 유니온 아이콘을 클릭합니다.

선택한 타입 요소가 객체 목록 창에 추가되고 이름을 추가하라는 메시지가 나타납니다.

- 2 레코드 또는 유니온의 이름을 입력합니다.

이 시점에 새 레코드나 유니온에는 필드가 없습니다.

- 3 객체 목록 창에서 레코드 또는 유니온을 선택하고 툴바에서 필드 아이콘을 클릭합니다. 속성 페이지를 사용하여 필드 이름 및 타입을 지정합니다.
- 4 필요한 모든 필드에 3 단계를 반복합니다.

레코드 또는 유니온을 정의하면 새 타입은 타입 라이브러리 또는 uses 페이지에서 타입 라이브러리를 참조하는 다른 모든 타입 라이브러리에서 사용할 수 있습니다. 예를 들어, 속성이나 매개변수의 타입으로 레코드나 유니온을 사용할 수 있습니다.

타입 라이브러리에 모듈 추가

다음과 같은 방법으로 타입 라이브러리에 모듈을 추가합니다.

- 1 툴바에서 모듈 아이콘을 클릭합니다.
선택한 모듈이 객체 목록 창에 추가되고 이름을 추가하라는 메시지가 나타납니다.
- 2 모듈의 이름을 입력합니다.
- 3 속성 페이지에서 모듈이 엔트리 포인트를 나타내는 DLL의 이름을 지정합니다.
- 4 툴바에서 메소드 아이콘을 클릭하여 3 단계에서 지정한 DLL에서 모든 메소드를 추가하고 속성 페이지를 사용하여 메소드를 설명합니다.
- 5 툴바에서 Const 아이콘을 클릭하여 모듈에서 정의할 모든 상수를 추가합니다. 각 상수에 대해 이름, 타입 및 값을 지정합니다.

타입 라이브러리 정보의 저장 및 등록

타입 라이브러리를 수정하고 나면 그 타입 라이브러리 정보를 저장하고 등록해야 합니다.

타입 라이브러리를 저장하면 다음과 같은 항목이 자동으로 업데이트됩니다.

- 바이너리 타입 라이브러리 파일(.tlb 확장자)
- 그 내용을 나타내는 *Project_TLB* 유닛
- 마법사가 생성한 모든 CoClass의 구현 코드

참고 타입 라이브러리는 별도의 바이너리(.TLB) 파일로 저장되지만 서버(.EXE, DLL 또는 .OCX)에도 연결됩니다.

참고 CORBA 인터페이스에 Type Library 에디터를 사용하면 *Project_TLB.pas* 유닛은 CORBA 애플리케이션에 필요한 스텝 및 뼈대 객체를 정의합니다.

Type Library 에디터는 타입 라이브러리 정보를 저장하는 옵션을 제공합니다. 선택하는 방법은 타입 라이브러리를 구현하는 단계에 따라 달라집니다.

- Save는 .TLB와 *Project_TLB* 유닛을 모두 디스크에 저장합니다.
- Refresh는 메모리에서만 타입 라이브러리 유닛을 업데이트합니다.
- Register는 시스템의 Windows 레지스트리에 타입 라이브러리의 항목을 추가합니다. 이 작업은 .TLB가 연결된 서버가 자체적으로 등록되면 자동으로 수행됩니다.
- Export는 IDL 구문의 타입 및 인터페이스 정의가 포함된 .IDL 파일을 저장합니다.

위의 메소드는 모두 구문 검사를 수행합니다. 타입 라이브러리를 새로 고치거나, 등록하거나, 저장하면 Delphi는 마법사를 사용하여 작성된 모든 CoClass의 구현 유닛을 자동으로 업데이트합니다. Type Library 에디터 옵션, Apply Updates를 설정하면 업데이트가 실행되기 전에 업데이트를 검토할 수 있는 경우도 있습니다.

Apply Updates 대화 상자

Tools|Environment Options|Type Library 페이지(기본적으로 활성화되어 있음)에서 "Display updates before refreshing"을 선택하면 타입 라이브러리를 새로 고치거나, 등록하거나, 저장할 때 Apply Updates 대화 상자가 나타납니다.

이 옵션을 설정하지 않으면 Type Library 에디터는 에디터에서 변경할 때 연결된 객체의 소스를 자동으로 업데이트합니다. 이 옵션을 설정하면 타입 라이브러리를 새로 고치거나, 저장하거나, 등록할 때 변경 내용을 거부할 수 있습니다.

Apply Updates 대화 상자는 발생할 수 있는 오류에 대해 경고하고 소스 파일에 TODO 설명을 삽입합니다. 예를 들어, 실수로 이벤트 이름을 바꾸면 소스 파일에 다음과 같은 경고 메시지가 나타납니다.

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name.As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

또한 바로 위에 있는 소스 파일에 TODO 설명이 삽입됩니다.

참고 이 경고와 TODO 설명을 무시하면 코드가 컴파일되지 않습니다.

타입 라이브러리 저장

타입 라이브러리를 저장하면 다음 작업을 수행합니다.

- 구문 및 유효성 검사를 수행합니다.
- 정보를 .TLB 파일로 저장합니다.
- 정보를 *Project_TLB* 유닛으로 저장합니다.
- 타입 라이브러리가 마법사로 생성한 CoClass와 연결되어 있는 경우, IDE 모듈 관리자에서 구현 파일을 업데이트하도록 공지합니다.

타입 라이브러리를 저장하려면 Delphi의 주 메뉴에서 File|Save를 선택합니다.

타입 라이브러리 새로 고침

타입 라이브러리를 새로 고치면 다음 작업을 수행합니다.

- 구문 검사를 수행합니다.
- Delphi 타입 라이브러리 유닛을 메모리에만 다시 생성합니다. 모든 파일을 디스크로 저장하지는 않습니다.
- 타입 라이브러리가 마법사로 생성한 CoClass와 연결되어 있는 경우, IDE 모듈 관리자에서 구현 파일을 업데이트하도록 공지합니다.

타입 라이브러리를 새로 고치려면 Type Library 에디터 툴바에서 Refresh 아이콘을 선택합니다.

참고 타입 라이브러리에 있는 항목의 이름을 바꾼 경우 구현을 새로 고치면 중복 항목이 만들어질 수 있습니다. 이런 경우에는 코드를 올바른 항목으로 이동하고 모든 중복을 삭제해야 합니다. 마찬가지로 타입 라이브러리에서 항목을 삭제한 경우 단순히 클라이언트에게 표시되는 부분만 제거했다고 가정하고 구현을 새로 고치면 CoClass의 항목이 제거되지 않습니다. 이러한 항목이 더 이상 필요하지 않으면 구현 유닛에서 수동으로 삭제해야 합니다.

타입 라이브러리 등록

일반적으로 타입 라이브러리는 COM 서버 애플리케이션을 등록할 때 자동으로 등록되기 때문에 명시적으로 등록하지 않아도 됩니다(36-17 페이지의 "COM 객체 등록" 참조). 그러나 Type Library 마법사를 사용하여 타입 라이브러리를 만들 경우 서버 객체와 연결되지 않습니다. 이런 경우에는 툴바를 사용하여 직접 타입 라이브러리를 등록할 수 있습니다.

타입 라이브러리를 등록하면 다음 작업을 수행합니다.

- 구문 검사를 수행합니다.
- Windows 레지스트리에 타입 라이브러리에 대한 항목을 추가합니다.

타입 라이브러리를 등록하려면 Type Library 에디터 툴바에서 Register 아이콘을 선택합니다.

IDL 파일 export하기

타입 라이브러리를 export하면 다음 작업을 수행합니다.

- 구문 검사를 수행합니다.
- 타입 정보 선언이 포함된 IDL 파일을 만듭니다. 이 파일은 CORBA IDL이나 Microsoft IDL로 타입 정보를 설명합니다.

타입 라이브러리를 export하려면 Type Library 에디터 툴바에서 Export 아이콘을 선택합니다.

타입 라이브러리 배포

기본적으로 ActiveX 또는 Automation 서버 프로젝트의 일부로 만든 타입 라이브러리가 있으면 그 타입 라이브러리는 리소스인 .DLL, .OCX 또는 .EXE에 자동으로 연결됩니다.

그러나 원한다면 Delphi가 타입 라이브러리를 관리하는 것처럼 별도의 .TLB로 타입 라이브러리와 애플리케이션을 배포할 수 있습니다.

Automation 애플리케이션의 타입 라이브러리는 .TLB 확장자를 가진 별도의 파일로 저장됩니다. 이제 종래의 Automation 애플리케이션은 타입 라이브러리를 .OCX 또는

.EXE 파일로 직접 컴파일합니다. 운영 체제는 타입 라이브러리가 실행 (.DLL, .OCX, .EXE) 파일의 첫 번째 리소스가 되게 합니다.

애플리케이션 개발자가 사용 가능한 기본 프로젝트 타입 라이브러리가 아닌 타입 라이브러리를 만들 경우 그 타입 라이브러리는 다음과 같은 형식이 될 수 있습니다.

- 리소스. 이 리소스는 TYPelib 타입이고 정수 ID가 있어야 합니다. 리소스 컴파일러로 타입 라이브러리를 작성하려면 리소스 (.RC) 파일에서 다음과 같이 선언해야 합니다.

```
1 typelib mylib1.tlb
2 typelib mylib2.tlb
```

ActiveX 라이브러리에는 여러 개의 타입 라이브러리 리소스가 있을 수 있습니다. 애플리케이션 개발자는 리소스 컴파일러를 사용하여 .TLB 파일을 자체적인 ActiveX 라이브러리에 추가합니다.

- 독립 실행형 바이너리 파일. Type Library 에디터에서 출력한 .TLB 파일은 바이너리 파일입니다.

35

COM 클라이언트 생성

COM 클라이언트는 다른 애플리케이션 또는 라이브러리에 의해 구현되는 COM 객체를 사용하는 애플리케이션입니다. 가장 일반적인 타입은 Automation 서버 (Automation 컨트롤러)를 제어하는 애플리케이션과 ActiveX 컨트롤 (ActiveX 컨테이너)을 호스팅하는 애플리케이션입니다.

언뜻 보기에 이 두 가지 타입의 COM 클라이언트는 매우 다릅니다. 전형적인 Automation 컨트롤러는 외부 서버 EXE를 시작하고 명령을 발생시켜 서버가 대신하여 작업을 수행하도록 합니다. Automation 서버는 일반적으로 닐비주얼하고 out-of-process입니다. 반면 전형적인 ActiveX 클라이언트는 컴포넌트 팔레트에서 컨트롤을 사용하는 것과 같은 방법을 사용하여 비주얼 컨트롤을 호스팅합니다. ActiveX 서버는 항상 in-process 서버입니다.

그러나 이 두 가지 타입의 COM 클라이언트를 작성하는 작업은 매우 유사합니다. 클라이언트 애플리케이션은 서버 객체에 대한 인터페이스를 가지고 있으며 자신의 속성과 메소드를 사용합니다. Delphi는 클라이언트의 컴포넌트에서 서버 CoClass를 래핑할 수 있도록 하여 이를 특히 쉽게 만들며 컴포넌트 팔레트에 설치할 수도 있습니다. 그러한 컴포넌트 래퍼의 예제는 컴포넌트 팔레트의 두 페이지에 나타납니다. ActiveX 래퍼의 예제는 ActiveX 페이지에 나타나고 Automation 객체의 예제는 Servers 페이지에 나타납니다.

COM 클라이언트를 작성할 때 애플리케이션에서 사용하기 위해 컴포넌트 팔레트에 있는 컴포넌트의 속성과 메소드를 이해해야 하는 것처럼 서버가 클라이언트에 노출하는 인터페이스를 이해해야 합니다. 이러한 인터페이스 또는 인터페이스 집합은 서버 애플리케이션에 의해 결정되고 일반적으로 타입 라이브러리에 게시됩니다. 특정 서버 애플리케이션의 게시된 인터페이스에 대한 특정 정보에 대해서는 해당 애플리케이션의 설명서를 참고해야 합니다.

컴포넌트 래퍼에서 서버 객체를 래핑하는 것을 선택하지 않고 컴포넌트 팔레트에 이를 설치하지 않더라도 인터페이스 정의가 사용자의 애플리케이션에 사용 가능하도록 만들어야 합니다. 이렇게 하기 위해 서버의 타입 정보를 import할 수 있습니다.

참고 또한 COM API를 사용하여 타입 정보를 직접 쿼리할 수 있지만 Delphi는 이에 대한 특별한 지원을 제공하지 않습니다.

객체 연결 및 포함(OLE)과 같은 일부 이전의 COM 기술은 타입 라이브러리에 있는 타입 정보를 제공하지 않습니다. 그 대신 이미 정의된 인터페이스의 표준 집합에 의존합니다. 이는 35-16 페이지의 "타입 라이브러리가 없는 서버에 대한 클라이언트 생성"에 설명되어 있습니다.

타입 라이브러리 정보 Import하기

클라이언트 애플리케이션에서 사용 가능한 COM 서버에 대한 정보를 만들려면 서버의 타입 라이브러리에 저장된 서버에 대한 정보를 import해야 합니다. 그런 다음 애플리케이션은 결과로 생성된 클래스를 사용하여 서버 객체를 제어할 수 있습니다.

타입 라이브러리 정보를 import하는 방법은 두 가지가 있습니다.

- Import Type Library 대화 상자를 사용하여 서버 타입, 객체 및 인터페이스에 대한 모든 사용 가능한 정보를 import할 수 있습니다. 모든 타입 라이브러리로부터 정보를 import할 수 있으며 Hidden, Restricted 또는 PreDeclID로 플래그되지 않는 타입 라이브러리에 있는 생성 가능한 모든 CoClasses에 대한 컴포넌트 래퍼를 옵션으로 생성할 수 있기 때문에 이것은 가장 일반적인 메소드입니다.
- ActiveX 컨트롤의 타입 라이브러리로부터 import할 경우, Import ActiveX 대화 상자를 이용할 수 있습니다. 이는 동일한 타입 정보를 import하지만 ActiveX 컨트롤을 나타내는 CoClasses에 대한 컴포넌트 래퍼만 생성합니다.
- IDE 내에서 사용할 수 없는 추가적인 구성 옵션을 제공하는 명령줄 유틸리티 `tlibimp.exe`를 사용할 수 있습니다.
- 마법사를 사용하여 생성된 타입 라이브러리는 타입 라이브러리 메뉴 항목과 동일한 메커니즘을 사용하여 자동으로 import됩니다.

타입 라이브러리 정보를 import하기 위해 선택하는 메소드에 상관 없이 결과 대화 상자는 `TypeLibName`이 타입 라이브러리의 이름인 `TypeLibName_TLB`라는 이름을 가진 유닛을 생성합니다. 이 파일에는 타입 라이브러리에 정의된 클래스, 타입 및 인터페이스에 대한 선언이 들어 있습니다. 이를 사용자의 프로젝트에 포함시키면 그러한 정의를 애플리케이션에 사용할 수 있게 되어 사용자는 객체를 생성하고 인터페이스를 호출할 수 있습니다. 이 파일은 때때로 IDE에 의해 다시 만들어집니다. 결과적으로 파일을 수동으로 변경하지 않는 것이 좋습니다.

`TypeLibName_TLB` 유닛에 타입 정의를 추가하는 것 외에도 대화 상자에서 타입 라이브러리에 정의된 모든 CoClasses에 대해 VCL 클래스 래퍼를 생성할 수 있습니다. Import Type Library 대화 상자를 사용할 때 이러한 래퍼는 옵션입니다. Import ActiveX 대화 상자를 사용할 때 항상 컨트롤을 나타내는 모든 CoClasses에 대해 생성됩니다.

생성된 클래스 래퍼는 사용자의 애플리케이션에 CoClasses를 나타내며 인터페이스의 속성과 메소드를 노출시킵니다. CoClass가 이벤트 (`IConnectionPointContainer` 및 `IConnectionPoint`)를 생성하기 위한 인터페이스를 지원할 경우, VCL 클래스 래퍼는 다른 컴포넌트에 대해 할 수 있는 것처럼 간단하게 이벤트에 대해 이벤트 핸들러를 할당할 수 있도록 이벤트 싱크(sink)를 만듭니다. 생성된 VCL 클래스를 컴포넌트 팔레트에 설치하도록 대화 상자에 지시하면 Object Inspector를 사용하여 속성 값 및 이벤트 핸들러를 할당할 수 있습니다.

참고 Import Type Library 대화 상자는 COM+ 이벤트 객체에 대해 클래스 래퍼를 만들지 않습니다. COM+ 이벤트 객체에 의해 생성된 이벤트에 응답하는 클라이언트를 작성하려면 프로그램에서 이벤트 싱크를 만들어야 합니다. 이 프로세스는 35-15 페이지의 "COM+ 이벤트 처리"에서 다룹니다.

타입 라이브러리를 import할 때 생성된 코드에 대한 자세한 내용은 35-5 페이지의 "타입 라이브러리 정보를 import할 때 생성되는 코드"를 참조하십시오.

Import Type Library 대화 상자 사용

다음과 같은 방법으로 타입 라이브러리를 import합니다.

- 1 Project|Import Type Library를 선택합니다.
- 2 목록에서 타입 라이브러리를 선택합니다.

대화 상자는 현재 시스템에 등록된 모든 라이브러리를 나열합니다. 해당 타입 라이브러리가 목록에 없으면 Add 버튼을 클릭하고 해당 타입 라이브러리 파일을 찾아 선택한 후 OK를 클릭합니다. 이렇게 하면 타입 라이브러리가 등록되어 사용 가능해 집니다. 그런 다음 2 단계를 반복합니다. 타입 라이브러리는 독립형 타입 라이브러리 파일(.tlb, .olb)이거나 타입 라이브러리(.dll, .ocx, .exe)를 제공하는 서버일 수 있습니다.

- 3 타입 라이브러리에서 CoClass를 래핑하는 VCL 컴포넌트를 생성하려는 경우 Generate Component Wrapper를 선택 표시합니다. 컴포넌트를 생성하지 않는다면 여전히 *TypeLibName_TLB* 유닛의 정의를 사용하여 CoClass를 사용할 수 있습니다. 하지만 서버 객체를 생성하는 호출을 사용자가 직접 작성해야 하고 필요한 경우 이벤트 싱크를 설정해야 합니다.

Import Type Library 대화 상자는 CanCreate 플래그 집합은 가지고 Hidden, Restricted 또는 PreDeclID 플래그 집합은 가지지 않는 CoClass만 import합니다. 이러한 플래그는 명령줄 유틸리티 tlibimp.exe를 사용하여 오버라이드할 수 있습니다.

- 4 컴포넌트 팔레트에 생성된 컴포넌트 래퍼를 설치하지 않으려면 Create Unit을 선택합니다. 이는 *TypeLibName_TLB* 유닛을 생성하고 3 단계에서 Generate Component Wrapper를 선택 표시했으면 컴포넌트 래퍼의 선언을 추가합니다. 이렇게 하면 Import Type Library 대화 상자는 종료됩니다.
- 5 컴포넌트 팔레트에 생성된 컴포넌트 래퍼를 설치하려면 이 컴포넌트가 상주할 Palette 페이지를 선택한 다음 Install을 선택합니다. 이 작업으로 Create Unit 버튼처럼 *TypeLibName_TLB* unit이 생성된 후 표시되는 Install component 대화 상자에서 컴포넌트가 상주할 기존 패키지 또는 새로운 패키지를 지정할 수 있습니다. 타입 라이브러리에 대해 어떤 컴포넌트도 생성되지 않을 경우, 이 버튼은 비활성화됩니다.

Import Type Library 대화 상자를 종료하면 새 *TypeLibName_TLB* 유닛은 Unit dir name 컨트롤에서 지정한 디렉토리에 나타납니다. Generate Component Wrapper를 선택 표시했을 경우, 이 파일에는 생성된 컴포넌트 래퍼뿐만 아니라 타입 라이브러리에 정의된 요소에 대한 선언도 들어 있습니다.

또한 생성된 컴포넌트 랩퍼를 설치하면 타입 라이브러리가 설명한 서버 객체가 컴포넌트 팔레트에 있게 됩니다. Object Inspector를 사용하여 속성을 설정하거나 서버에 대한 이벤트 핸들러를 작성할 수 있습니다. 폼이나 데이터 모듈에 컴포넌트를 추가하면 디자인 시 이를 더블 클릭하여 속성 페이지를 볼 수 있습니다(지원하는 경우에 해당).

참고 컴포넌트 팔레트의 Servers 페이지는 사용자를 위해 이런 방식으로 import된 많은 예제 Automation 서버를 포함합니다.

Import ActiveX 대화 상자 사용

다음과 같은 방법으로 ActiveX 컨트롤을 import합니다.

- 1 Component|Import ActiveX Control을 선택합니다.
- 2 목록에서 타입 라이브러리를 선택합니다.

대화 상자는 ActiveX 컨트롤을 정의하는 등록된 모든 라이브러리를 나열합니다. (이는 Import Type Library 대화 상자에 나열된 라이브러리의 서브셋입니다.) 해당 타입 라이브러리가 목록에 없으면 Add 버튼을 클릭하고 해당 타입 라이브러리 파일을 찾아 선택한 후 OK를 클릭합니다. 이렇게 하면 타입 라이브러리가 등록되어 사용 가능해집니다. 그런 다음 2 단계를 반복합니다. 타입 라이브러리는 독립형 타입 라이브러리 파일(.tlb, .olb)이거나 ActiveX 서버(.dll, .ocx)일 수 있습니다.

- 3 컴포넌트 팔레트에 ActiveX 컨트롤을 설치하지 않으려면 Create Unit을 선택합니다. 이는 *TypeLibName_TLB* 유닛을 생성하고 컴포넌트 랩퍼의 선언을 추가합니다. 이렇게 하면 Import ActiveX 대화 상자가 종료됩니다.
- 4 컴포넌트 팔레트에 ActiveX 컨트롤을 설치하려면 이 컴포넌트가 상주할 Palette 페이지를 선택한 다음 Install을 선택합니다. 이 작업으로 Create Unit 버튼처럼 *TypeLibName_TLB* unit이 생성된 후 표시되는 Install component 대화 상자에서 컴포넌트가 상주할 기존 패키지 또는 새로운 패키지를 지정할 수 있습니다.

Import ActiveX 대화 상자를 종료하면 새 *TypeLibName_TLB* 유닛은 Unit dir name 컨트롤에서 지정한 디렉토리에 나타납니다. 이 파일에는 ActiveX 컨트롤에 대한 생성된 컴포넌트 랩퍼뿐만 아니라 타입 라이브러리에서 정의한 요소에 대한 선언도 들어 있습니다.

참고 옵션인 Import Type Library 대화 상자과 달리 Import ActiveX 대화 상자는 항상 컴포넌트 랩퍼를 생성합니다. 왜냐하면 ActiveX 컨트롤은 비주얼 컨트롤이므로 VCL 폼과 맞도록 컴포넌트 랩퍼의 추가적인 지원을 필요로 하기 때문입니다.

생성된 컴포넌트 랩퍼를 설치했으면 ActiveX 컨트롤은 컴포넌트 팔레트에 상주합니다. Object Inspector를 사용하여 속성을 설정하거나 이 컨트롤에 대한 이벤트 핸들러를 작성할 수 있습니다. 폼이나 데이터 모듈에 컨트롤을 추가하면 디자인 시 마우스 오른쪽 버튼을 클릭하여 속성 페이지를 볼 수 있습니다(지원하는 경우에 해당).

참고 컴포넌트 팔레트의 ActiveX 페이지는 사용자를 위해 이런 방식으로 import된 많은 예제 ActiveX 컨트롤을 포함합니다.

타입 라이브러리 정보를 import할 때 생성되는 코드

일단 타입 라이브러리를 import하면 생성된 *TypeLibName_TLB* 유닛을 볼 수 있습니다. 유닛의 상단부에서 다음을 찾을 수 있습니다.

- 타입 라이브러리, 해당 인터페이스 및 CoClasses의 GUIDS에 심볼릭 이름을 주는 상수 선언. 이러한 상수에 대한 이름은 다음과 같이 생성됩니다.
 - 타입 라이브러리의 이름이 *TypeLibName*이면 타입 라이브러리에 대한 GUID는 *LBID_TypeLibName*인 형식을 갖습니다.
 - 인터페이스의 이름이 *InterfaceName*이면 인터페이스에 대한 GUID는 *IID_InterfaceName*인 형식을 갖습니다.
 - dispinterface의 이름이 *InterfaceName*이면 dispinterface에 대한 GUID는 *DIID_InterfaceName*인 형식을 갖습니다.
 - CoClass의 이름이 *ClassName*이면 CoClass에 대한 GUID는 *CLASS_ClassName*인 형식을 갖습니다.
- 타입 라이브러리에 있는 CoClass에 대한 선언. 이들은 각 CoClass를 기본 인터페이스에 매핑합니다.
- 타입 라이브러리에 있는 인터페이스 및 dispinterface에 대한 선언.
- 기본 인터페이스가 Vtable 바인딩을 지원하는 각 CoClass의 생성자 클래스에 대한 선언. 생성자 클래스는 *Create* 및 *CreateRemote*의 두 가지 클래스 메소드를 가지며, 이들은 CoClass를 지역적으로 (*Create*) 또는 원격으로 (*CreateRemote*) 인스턴스화하는 데 사용할 수 있습니다. 이러한 메소드는 CoClass에 대한 기본 인터페이스를 반환합니다.

이러한 선언은 CoClass의 인스턴스를 생성하고 해당 인터페이스에 액세스하기 위해 필요한 것을 제공합니다. 필요한 모든 작업은 CoClass를 바인딩하고 해당 인터페이스를 호출하려는 유닛의 uses 절에 생성된 *TypeLibName_TLB.pas* 파일을 추가하는 것입니다.

참고 *TypeLibName_TLB* 유닛의 이 부분은 Type Library 에디터 또는 명령줄 유틸리티 TLBIMP를 사용할 때도 생성됩니다.

ActiveX 컨트롤을 사용하려면 위에서 설명한 선언 외에도 생성된 VCL 래퍼가 필요합니다. VCL 래퍼는 컨트롤에 대한 창 관리 문제를 처리합니다. Import Type Library 대화 상자에서 다른 CoClasses에 대한 VCL 래퍼를 생성했을 수도 있습니다. 이러한 VCL 래퍼는 서버 객체 생성 작업과 메소드 호출 작업을 단순화시킵니다. 클라이언트 애플리케이션이 이벤트에 응답하도록 하려는 경우에 특히 좋습니다.

생성된 VCL 래퍼에 대한 선언은 인터페이스 섹션의 하단부에 나타납니다. ActiveX 컨트롤에 대한 컴포넌트 래퍼는 *TOleControl*의 자손입니다. Automation 객체에 대한 컴포넌트 래퍼는 *TOleServer*의 자손입니다. 생성된 컴포넌트 래퍼는 CoClass의 인터페이스에 의해 노출되는 속성, 이벤트 및 메소드를 추가합니다. 다른 모든 VCL 컴포넌트처럼 이 컴포넌트를 사용할 수 있습니다.

경고 생성된 *TypeLibName_TLB* 유닛을 편집하지 않아야 합니다. 왜냐하면 타입 라이브러리는 새로 고칠 때마다 다시 생성되기 때문에 편집한 변경 사항을 덮어쓰게 되기 때문입니다.

참고 생성된 코드에 대한 최신 정보는 자동으로 생성된 *TypeLibName_TLB* 유닛의 주석을 참조하십시오.

Import된 객체 제어

타입 라이브러리 정보를 import한 다음 import된 객체를 프로그래밍할 준비가 되었습니다. 진행하는 방법은 일부분은 객체에 따라 다르고, 일부분은 컴포넌트 래퍼를 생성하기 위해 선택했는지의 여부에 따라 다릅니다.

컴포넌트 래퍼 사용

서버 객체에 대해 컴포넌트 래퍼를 생성했다면 사용자의 COM 클라이언트 애플리케이션을 작성하는 것은 VCL 컴포넌트를 포함하는 다른 애플리케이션을 작성하는 것과 크게 다르지 않습니다. 서버 객체의 속성, 메소드, 이벤트는 이미 VCL 컴포넌트에 캡슐화되어 있습니다. 사용자가 할 작업은 이벤트 핸들러를 할당하고, 속성 값을 설정하고, 메소드를 호출하는 것입니다.

서버 객체의 속성, 메소드, 이벤트를 사용하려면 해당 서버에 대한 설명서를 참조하십시오. 컴포넌트 래퍼는 가능한 경우 이중 인터페이스를 자동으로 제공합니다. Delphi는 타입 라이브러리에 있는 정보로부터 VTable 레이아웃을 결정합니다.

또한 사용자의 새로운 컴포넌트는 기초 클래스로부터 중요한 속성 및 메소드를 상속합니다.

ActiveX 래퍼

컴포넌트 래퍼는 컨트롤의 창을 VCL 프레임워크에 통합하기 때문에 ActiveX 컨트롤을 호스팅할 때는 항상 컴포넌트 래퍼를 사용해야 합니다.

ActiveX 컨트롤이 *TOleControl*에서 상속한 속성 및 메소드를 통해 기본이 되는 인터페이스에 액세스하거나 컨트롤에 대한 정보를 얻을 수 있습니다. 하지만 대부분의 애플리케이션은 이를 사용할 필요가 없습니다. 그 대신 다른 VCL 컨트롤을 사용하는 것과 동일한 방법으로 import된 컨트롤을 사용합니다.

일반적으로 ActiveX 컨트롤은 속성을 설정할 수 있는 속성 페이지를 제공합니다. 속성 페이지는 폼 디자이너에서 더블 클릭할 때 일부 컴포넌트가 표시하는 컴포넌트 편집기와 유사합니다. ActiveX 컨트롤의 속성 페이지를 표시하려면 마우스 오른쪽 버튼을 클릭하고 Properties를 선택합니다.

대부분의 import 된 ActiveX 컨트롤을 사용하는 방법은 서버 애플리케이션에 의해 결정됩니다. 하지만 ActiveX 컨트롤은 데이터베이스 필드의 데이터를 표시할 때 표준 공지(notification) 집합을 사용합니다. 그러한 ActiveX 컨트롤을 호스팅하는 방법에 대한 자세한 내용은 35-8 페이지의 "Data-aware ActiveX 컨트롤 사용"을 참조하십시오.

Automation 객체 래퍼(object wrappers)

Automation 객체에 대한 래퍼를 사용하면 사용자의 서버 객체에 연결을 구성하는 방법을 제어할 수 있습니다.

- *ConnectKind* 속성은 서버가 로컬인지 원격인지 여부와 이미 실행 중인 서버에 연결하려고 하는지 여부 또는 새 인스턴스가 시작되어야 하는지 여부를 나타냅니다. 원격 서버에 연결할 때 *RemoteMachineName* 속성을 사용하여 시스템 이름을 지정해야 합니다.
- 일단 *ConnectKind*를 지정했다면 사용자의 컴포넌트를 서버에 연결할 수 있는 세 가지 방법이 있습니다.
 - 컴포넌트의 *Connect* 메소드를 호출하여 서버에 명시적으로 연결할 수 있습니다.
 - *AutoConnect* 속성을 *True*로 설정하여 애플리케이션 시작 시 자동으로 연결하도록 컴포넌트에 지시할 수 있습니다.
 - 그러면 서버에 명시적으로 연결하지 않아도 됩니다. 컴포넌트는 컴포넌트를 사용하여 서버의 속성 또는 메소드 중의 하나를 사용할 때 자동으로 연결을 구성합니다.

메소드를 호출하거나 속성에 액세스하는 것은 다른 컴포넌트를 사용할 때와 동일합니다.

```
TServerComponent1.DoSomething;
```

Object Inspector를 사용하여 이벤트 핸들러를 작성하므로 이벤트를 처리하는 것은 쉽습니다. 하지만 사용자의 컴포넌트에 있는 이벤트 핸들러는 타입 라이브러리의 이벤트에 대해 정의된 것과는 약간 다른 매개변수를 갖습니다. 특히 포인터 타입 (var 매개변수 및 인터페이스 포인터)이 가변 타입으로 바뀝니다. 값을 할당하기 전에 var 매개변수를 기본이 되는 타입으로 명시적으로 변환해야 합니다. 인터페이스 포인터는 **as** 연산자를 사용하여 적절한 인터페이스 타입으로 타입 변환될 수 있습니다. 예를 들어, 다음 코드는 ExcelApplication 이벤트인 OnNewWorkbook에 대한 이벤트 핸들러를 보여줍니다. 이벤트 핸들러는 다른 CoClass(ExcelWorkbook)의 인터페이스를 제공하는 매개변수를 가집니다. 하지만 인터페이스는 ExcelWorkbook 인터페이스 포인터가 아니라 OleVariant로 전달됩니다.

```
procedure TForm1.XLappNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
  { Note how the OleVariant for the interface must be cast to the correct type }
  ExcelWorkbook1.ConnectTo((iUnknown(wb) as ExcelWorkbook));
end;
```

이 예제에서 이벤트 핸들러는 ExcelWorkbook 컴포넌트(ExcelWorkbook1)에 workbook 을 할당합니다. 이는 *ConnectTo* 메소드를 사용하여 컴포넌트 래퍼를 기존 인터페이스에 연결하는 방법을 보여 줍니다. *ConnectTo* 메소드는 컴포넌트 래퍼에 대해 생성된 코드에 추가됩니다.

애플리케이션 객체를 갖는 서버는 해당 객체에 있는 Quit 메소드를 노출시켜 클라이언트가 연결을 종료할 수 있도록 합니다. 일반적으로 Quit은 애플리케이션을 종료하기 위해 File 메뉴를 사용하는 것과 동일한 기능을 노출시킵니다. Quit 메소드를 호출하는 코드는 사용자 컴포넌트의 *Disconnect* 메소드에서 생성됩니다. 매개변수 없이 Quit 메소드를 호출할 수 있으면 컴포넌트 래퍼는 또한 *AutoQuit* 속성을 가집니다. 컴포넌트가 해제될 때 *AutoQuit*을 통해 컨트롤러가 Quit을 호출합니다. 다른 때에 연결을 해제하려고 하거나 Quit 메소드가 매개변수를 필요로 하는 경우에는 이를 명시적으로 호출해야 합니다. Quit은 생성된 컴포넌트에 public 메소드로 나타납니다.

Data-aware ActiveX 컨트롤 사용

Delphi 애플리케이션에서 data-aware ActiveX 컨트롤을 사용할 때 이것이 나타내는 데이터의 데이터베이스와 이를 연결해야 합니다. 이렇게 하려면 data-aware VCL 컨트롤에 대해 데이터 소스가 필요한 것처럼 데이터 소스 컴포넌트가 필요합니다.

data-aware ActiveX 컨트롤을 폼 디자이너에 놓은 다음 해당 *DataSource* 속성을 원하는 데이터셋을 나타내는 데이터 소스에 할당합니다. 일단 데이터 소스를 지정했으면 Data Bindings 에디터를 사용하여 컨트롤의 데이터 바인딩 속성을 데이터셋의 필드에 연결할 수 있습니다.

Data Bindings 에디터를 나타내려면 data-aware ActiveX 컨트롤을 마우스 오른쪽 버튼으로 클릭하여 옵션 목록을 나타냅니다. 기본적인 옵션 외에도 추가적인 Data Bindings 항목이 나타납니다. 데이터셋에 있는 필드의 이름을 나열하고 ActiveX 컨트롤의 바인딩 가능한 속성을 나열하는 Data Bindings 에디터를 보려면 이 항목을 선택합니다.

다음과 같은 방법으로 필드를 속성에 바인딩합니다.

- 1 ActiveX Data Bindings Editor 대화 상자에서 필드 및 속성 이름을 선택합니다.

Field Name 은 데이터베이스의 필드를 나열하고 Property Name 은 데이터베이스 필드에 바인딩될 수 있는 ActiveX 컨트롤 속성을 나열합니다. 예를 들어, Value (12) 와 같이 속성의 dispID는 괄호로 묶여 있습니다.

- 2 Bind를 클릭한 다음 OK를 클릭합니다.

참고 대화 상자에 아무 속성도 나타나지 않으면 ActiveX 컨트롤은 data-aware 속성을 포함하지 않습니다. ActiveX 컨트롤의 속성에 대해 간단한 데이터 바인딩을 활성화하려면 38-10 페이지의 "타입 라이브러리를 이용한 간단한 데이터 바인딩 활성화"에서 설명한 대로 타입 라이브러리를 사용합니다.

다음 예제는 Delphi 컨테이너에서 data-aware ActiveX 컨트롤을 사용하는 단계를 형식적으로 보여 주는 것입니다. 이 예제는 시스템에 Microsoft Office 97이 설치되어 있는 경우 사용 가능한 Microsoft Calendar Control을 사용합니다.

- 1 Delphi 메인 메뉴에서 Component|Import ActiveX Control을 선택합니다.

- 2 Microsoft Calendar control 8.0과 같은 data-aware ActiveX 컨트롤을 선택하고 해당 클래스 이름을 *TCalendarAXControl*로 변경한 다음 Install을 클릭합니다.
- 3 Install 대화 상자에서 컨트롤을 팔레트에서 사용 가능하게 만드는 기본 사용자 패키지에 컨트롤을 추가하려면 OK를 클릭합니다.
- 4 새로운 애플리케이션을 시작하려면 Close All 및 File|New|Application을 선택합니다.
- 5 ActiveX 탭에서 팔레트에 추가한 *TCalendarAXControl* 객체를 폼에 가져다 놓습니다.
- 6 Data Access 탭에서 *DataSource* 및 *Table* 객체를 폼에 가져다 놓습니다.
- 7 *DataSource* 객체를 선택하고 *DataSet* 속성을 *Table1*로 설정합니다.
- 8 *Table* 객체를 선택하고 다음을 수행합니다.
 - *DatabaseName* 속성을 DBDEMOS로 설정합니다.
 - *TableName* 속성을 EMPLOYEE.DB로 설정합니다.
 - *Active* 속성을 *True*로 설정합니다.
- 9 *TCalendarAXControl* 객체를 선택하고 *DataSource* 속성을 *DataSource1*로 설정합니다.
- 10 ActiveX Control Data Bindings Editor를 호출하려면 *TCalendarAXControl* 객체를 선택하고 마우스 오른쪽 단추를 클릭한 다음 Data Bindings를 선택합니다.
Field Name 은 활성 데이터베이스에 있는 모든 필드를 나열합니다. Property Name은 데이터베이스 필드에 바인딩할 수 있는 ActiveX 컨트롤의 속성을 나열합니다. 속성의 dispID는 괄호로 묶여 있습니다.
- 11 *HireDate* 필드 및 *Value* 속성 이름을 선택하고 Bind를 선택한 후 OK를 클릭합니다.
이제 필드 이름 및 속성이 바인딩됩니다.
- 12 Data Controls 탭에서 *DBGrid* 객체를 폼에 가져다 놓고 *DataSource* 속성을 *DataSource1*로 설정합니다.
- 13 Data Controls 탭에서 *DBNavigator* 객체를 폼에 가져다 놓고 *DataSource* 속성을 *DataSource1*로 설정합니다.
- 14 애플리케이션을 실행합니다.
- 15 다음과 같은 방법으로 애플리케이션을 테스트합니다.
HireDate 필드가 *DBGrid* 객체에 표시되어 있으면 Navigator 객체를 사용하여 데이터베이스를 탐색합니다. ActiveX 컨트롤의 날짜는 데이터베이스를 이동할 때 변경됩니다.

예제: Microsoft Word로 문서 인쇄

다음 단계는 Office 97의 Microsoft Word 8을 사용하여 문서를 인쇄하는 Automation 컨트롤러를 생성하는 방법을 보여 줍니다.

시작하기 전에 폼, 버튼, Open 대화 상자(*TOpenDialog*)로 구성된 새 프로젝트를 생성합니다. 이 컨트롤은 Automation 컨트롤러를 구성합니다.

1 단계: 이 예제를 위한 Delphi 준비

편의상 Delphi는 컴포넌트 팔레트에서 Word, Excel, PowerPoint와 같은 많은 일반적인 서버를 제공해 왔습니다. 서버를 import하는 방법을 보여 주기 위해 여기서는 Word를 사용합니다. 이는 이미 컴포넌트 팔레트에 있기 때문에 이 첫 번째 단계에서는 팔레트에 설치하는 방법을 볼 수 있도록 Word를 포함하는 패키지를 제거할 것을 요구합니다. 4 단계는 컴포넌트 팔레트를 일반적인 상태로 반환하는 방법을 설명합니다.

다음과 같은 방법으로 컴포넌트 팔레트에서 Word를 제거합니다.

- 1 Component|Install Packages를 선택합니다.
- 2 Borland Sample Automation Server 컴포넌트를 클릭하고 Remove를 선택합니다.
컴포넌트 팔레트의 Servers 페이지에는 더 이상 Delphi와 함께 제공된 서버가 없습니다. (다른 서버가 import되지 않으면 Servers 페이지도 사라집니다.)

2 단계: Word 타입 라이브러리 import하기

다음과 같은 방법으로 Word 타입 라이브러리를 import합니다.

- 1 Project|Import Type Library를 선택합니다.
- 2 Import Type Library 대화 상자에서 다음과 같이 합니다.
 - 1 Microsoft Office 8.0 Object Library를 선택합니다.
Word(Version 8)가 목록에 없으면 Add 버튼을 선택하고 Program Files\Microsoft Office\Office로 가서 Word 타입 라이브러리 파일인 MSWord8.olb를 선택한 다음 Add를 선택하고 목록에서 Word(Version 8)를 선택합니다.
 - 2 팔레트 페이지의 경우, Servers를 선택합니다.
 - 3 Install을 선택합니다.
Install 대화 상자가 나타납니다. 이런 타입 라이브러리가 포함된 새로운 패키지를 만들려면 Into New Packages 탭을 선택하고 WordExample을 입력합니다.
- 3 Servers Palette 페이지로 가서 WordApplication을 선택하여 이를 폼에 놓습니다.
- 4 다음 단계에서 설명하는 대로 버튼 객체에 대한 이벤트 핸들러를 작성합니다.

3 단계: Microsoft Word를 제어하기 위해 Vtable 또는 디스패치 인터페이스 객체 사용

Vtable이나 디스패치 객체 중 하나를 사용하여 Microsoft Word를 제어할 수도 있습니다.

VTable 인터페이스 객체 사용

WordApplication 객체의 인터페이스를 사용자의 폼에 가져다 놓으면 VTable 인터페이스 객체를 사용하여 컨트롤에 쉽게 액세스할 수 있습니다. 방금 작성했던 클래스의 메소드를 호출합니다. Word의 경우, 이는 *TWordApplication* 클래스입니다.

- 1 버튼을 선택하고 *OnClick* 이벤트 핸들러를 더블 클릭한 후 다음과 같은 이벤트 처리 코드를 제공합니다.

```

procedure TForm1.Button1Click(Sender:TObject);
var
    FileName:OleVariant;
begin
    if OpenFileDialog1.Execute then
        begin
            FileName := OpenFileDialog1.FileName;

            WordApplication1.Documents.Open(FileName,
                EmptyParam,EmptyParam,EmptyParam,
                EmptyParam,EmptyParam,EmptyParam,
                EmptyParam,EmptyParam,EmptyParam);

            WordApplication1.ActiveDocument.PrintOut(
                EmptyParam,EmptyParam,EmptyParam,
                EmptyParam, EmptyParam,EmptyParam,
                EmptyParam,EmptyParam,EmptyParam,
                EmptyParam,EmptyParam,EmptyParam,
                EmptyParam,EmptyParam);
            end;

        end;

```

- 2 프로그램을 빌드하고 실행합니다. 버튼을 클릭하면 인쇄할 파일을 선택하라는 메시지가 나타납니다.

디스패치 인터페이스 객체 사용

또 다른 방법으로 지연 바인딩 (late binding)을 위해 디스패치 인터페이스를 사용할 수 있습니다. 디스패치 인터페이스 객체를 사용하려면 다음과 같이 *_ApplicationDisp* 디스패치 랩퍼 클래스를 사용하여 애플리케이션 객체를 생성하고 초기화합니다. *dispinterface* 메소드는 VTable 인터페이스를 반환할 때 소스에 의해 "문서화"된다는 사실에 주의하십시오. 그러나 실제로는 이 메소드를 디스패치 인터페이스로 타입 변환해야 합니다.

- 1 버튼을 선택하고 *OnQuit* 이벤트 핸들러를 더블 클릭한 후 다음과 같은 이벤트 처리 코드를 제공합니다.

```

procedure TForm1.Button1Click(Sender:TObject);
var
    MyWord : _ApplicationDisp;
    FileName :OleVariant;
begin
    if OpenFileDialog1.Execute then
        begin

```

```

FileName := OpenDialog1.FileName;
MyWord := CoWordApplication.Create as
_ApplicationDisp;
(MyWord.Documents as DocumentsDisp).Open(FileName, EmptyParam,
EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam,
EmptyParam, EmptyParam, EmptyParam);
(MyWord.ActiveDocument as _DocumentDisp).PrintOut(EmptyParam,
EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam,
EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam,
EmptyParam, EmptyParam, EmptyParam);
MyWord.Quit(EmptyParam, EmptyParam, EmptyParam);
end;
end;

```

- 2 프로그램을 빌드하고 실행합니다. 버튼을 클릭하면 인쇄할 파일을 선택하라는 메시지가 나타납니다.

4 단계: 예제 클린업

이 예제를 완료한 다음 Delphi를 원래 폼으로 복원할 수 있습니다.

- 1 다음과 같은 방법으로 Servers 페이지에서 객체를 삭제합니다.
 - Component | Install Packages를 선택합니다.
 - 목록에서 WordExample Package를 선택하고 Remove를 클릭합니다.
 - 확인을 요청하는 메시지 상자에서 Yes를 클릭합니다.
 - OK를 클릭하여 Install Packages 대화 상자를 종료합니다.
- 2 Borland Sample Automation Server Components Package를 반환합니다.
 - Component | Install Packages를 선택합니다.
 - Add 버튼을 클릭합니다.
 - 결과 대화 상자에서 dclaxserver50.bpl을 선택합니다.
 - OK를 클릭하여 Install Packages 대화 상자를 종료합니다.

타입 라이브러리 정의를 기반으로 클라이언트 코드 작성

ActiveX 컨트롤을 호스트하기 위해 컴포넌트 랩퍼를 사용해야 하지만 *TypeLibName_* TLB 유닛에 나타나는 타입 라이브러리의 정의만 사용하여 Automation 컨트롤러를 작성할 수 있습니다. 이 프로세스는 특히 이벤트에 응답해야 할 경우 컴포넌트가 해당 작업을 하도록 하는 것과 좀더 관련이 있습니다.

서버에 연결

사용자의 컨트롤러 애플리케이션에서 Automation 서버를 사용할 수 있기 전에 지원하는 인터페이스에 대한 참조를 얻어야 합니다. 일반적으로 메인 인터페이스를 통해 서버에 연결합니다. 예를 들면, WordApplication 컴포넌트를 통해 Microsoft Word에 연결합니다.

메인 인터페이스가 이중 인터페이스이면 *TypeLibName_TLB.pas* 파일에서 생성자 객체를 사용할 수 있습니다. 생성자 클래스는 접두사 "Co"로 시작하는 CoClass와 같은 이름을 가집니다. *Create* 메소드를 호출하여 동일한 컴퓨터에 있는 서버에 연결하거나 *CreateRemote* 메소드를 사용하여 원격 기계에 있는 서버에 연결할 수 있습니다. *Create* 및 *CreateRemote*는 클래스 메소드이기 때문에 이를 호출하기 위한 생성자 클래스의 인스턴스가 필요하지 않습니다.

```
MyInterface := CoServerClassName.Create;
MyInterface := CoServerClassName.CreateRemote('Machine1');
```

Create 및 *CreateRemote*는 CoClass에 대한 기본 인터페이스를 반환합니다.

기본 인터페이스가 디스패치 인터페이스이면 CoClass에 대해 생성된 생성자 클래스가 없습니다. 그 대신 전역 *CreateOleObject* 함수를 호출하여 CoClass에 대한 GUID에 전달합니다(_TLB 유닛의 상단부에 이 GUID에 대한 상수가 정의되어 있습니다). *CreateOleObject*는 기본 인터페이스에 대한 IDispatch 포인터를 반환합니다.

이중 인터페이스를 사용하여 Automation 서버 제어

자동으로 생성된 생성자 클래스를 사용하여 서버에 연결한 다음, 인터페이스의 메소드를 호출합니다. 예를 들면, 다음과 같습니다.

```
var
  MyInterface: _Application;
begin
  MyInterface := CoWordApplication.Create;
  MyInterface.DoSomething;
```

인터페이스 및 생성자 클래스는 타입 라이브러리를 import 할 때 자동으로 생성된 *TypeLibName_TLB* 유닛에 정의됩니다.

이중 인터페이스에 대한 내용은 36-13 페이지의 "이중 인터페이스"를 참조하십시오.

디스패치 인터페이스를 사용하여 Automation 서버 제어

일반적으로 이중 인터페이스를 사용하여 위에서 설명한 대로 Automation 서버를 제어합니다. 하지만 이중 인터페이스를 사용할 수 없기 때문에 디스패치 인터페이스를 사용하여 Automation 서버를 제어해야 하는 경우도 있습니다.

다음과 같은 방법으로 디스패치 인터페이스의 메소드를 호출합니다.

- 1 전역 *CreateOleObject* 함수를 사용하여 서버에 연결합니다.
- 2 **as** 연산자를 사용하여 *CreateOleObject*에서 반환된 *IDispatch* 인터페이스를 CoClass에 대한 *dispinterface*로 타입 변환합니다. *dispinterface* 타입은 *TypeLibName_TLB* 유닛에서 선언됩니다.
- 3 *dispinterface*의 메소드를 호출하여 Automation 서버를 제어합니다.

디스패치 인터페이스를 사용하는 다른 방법은 *가변*에 할당하는 것입니다.

*CreateOleObject*에서 반환한 인터페이스를 *가변* 타입에 할당함으로써 인터페이스에 대한 *가변* 타입의 기본 제공 지원을 이용할 수 있습니다. 인터페이스의 메소드를 호출하면 *가변* 타입은 모든 *IDispatch* 호출을 자동으로 처리하며 디스패치 ID를 폐치하고

적당한 메소드를 호출합니다. 가변 타입은 **var**을 통해 디스패치 인터페이스를 호출하는 기본 제공 지원을 포함합니다.

```
V:Variant;
begin
  V:= CreateOleObject('TheServerObject');
  V.MethodName; { calls the specified method }
  ...
end;
```

가변 타입은 표준 *IDispatch* 메소드만 사용하여 서버를 호출하기 때문에 가변 타입 사용의 장점은 타입 라이브러리를 import 하지 않아도 된다는 것입니다. 런타임 시 동적 바인딩을 사용하기 때문에 가변 변수가 더 느리다는 것이 모순된 점입니다.

디스패치 인터페이스에 대한 자세한 내용은 36-12 페이지의 "Automation 인터페이스"를 참조하십시오.

Automation 컨트롤러의 이벤트 처리

사용자가 import한 타입 라이브러리의 객체에 대한 컴포넌트 래퍼를 생성할 때 생성된 컴포넌트에 추가된 이벤트를 사용하여 이벤트에 응답할 수 있습니다. 하지만 컴포넌트 래퍼를 사용하지 않을 경우 또는 서버가 COM+ 이벤트를 사용할 경우에는 이벤트 싱크 코드를 사용자가 직접 작성해야 합니다.

프로그램에서 Automation 이벤트 처리

이벤트를 처리하기 전에 반드시 이벤트 싱크를 정의해야 합니다. 이벤트 싱크는 서버의 타입 라이브러리에서 정의된 이벤트 디스패치 인터페이스를 구현하는 클래스입니다.

이벤트 싱크를 작성하려면 이벤트 디스패치 인터페이스를 구현하는 객체를 생성합니다.

```
TServerEventsSink = class(TObject, _TheServerEvents)
...{ declare the methods of _TheServerEvents here }
end;
```

일단 이벤트 싱크의 인스턴스를 가지면 서버가 이를 호출할 수 있도록 서버 객체에 존재 여부를 알려야 합니다. 이렇게 하려면 전역 *InterfaceConnect* 프로시저를 호출하여 다음을 전달합니다.

- 이벤트를 생성하는 서버에 대한 인터페이스
- 이벤트 싱크가 처리하는 이벤트 인터페이스에 대한 GUID
- 이벤트 싱크에 대한 IUnknown 인터페이스
- 서버와 사용자의 이벤트 싱크 간의 연결을 나타내는 Longint를 받는 변수

```
{MyInterface is the server interface you got when you connected to the server }
InterfaceConnect(MyInterface, DIID_TheServerEvents,
  MyEventSinkObject as IUnknown, cookievar);
```

*InterfaceConnect*를 호출한 다음, 사용자의 이벤트 싱크가 연결되고 이벤트가 발생하면 서버로부터 호출을 받습니다.

사용자의 이벤트 싱크를 해제하기 전에 반드시 연결을 종료해야 합니다. 이렇게 하려면 전역 *InterfaceDisconnect* 프로시저를 호출하여 사용자의 이벤트 싱크에 대한 인터페이스

이스를 제외한 모든 동일한 매개변수를 전달합니다(마지막 매개변수는 outgoing이라기 보다는 ingoing입니다.).

```
InterfaceDisconnect(MyInterface, DIID_TheServerEvents, cookievar);
```

참고 이를 메모리 해제하기 전에 서버가 사용자의 이벤트 싱크에 대한 연결을 해제했는지 확인해야 합니다. *InterfaceDisconnect*에 의해 초기화된 연결 해제 공지에 대해 서버가 어떻게 응답할지 모르기 때문에 호출 후 바로 사용자의 이벤트 싱크를 메모리 해제하면 경합 조건(race condition)이 발생할 수도 있습니다. 문제를 막는 가장 손쉬운 방법은 사용자의 이벤트 싱크가 서버가 이벤트 싱크의 인터페이스를 해제할 때까지 감소하지 않은 자신의 참조 카운트를 유지하도록 하는 것입니다.

COM+ 이벤트 처리

COM+에서 서버는 특정 인터페이스 집합(*IConnectionPointContainer* 및 *IConnectionPoint*)이 아니라 특별한 helper 객체를 사용하여 이벤트를 생성합니다. 따라서 *TeventDispatcher*의 자손인 이벤트 싱크를 사용할 수 없습니다. *TEventDispatcher*는 COM+ 이벤트 객체가 아니라 인터페이스를 사용하도록 디자인됩니다.

이벤트 싱크를 정의하는 대신 클라이언트 애플리케이션은 subscriber 객체를 정의합니다. 이벤트 싱크처럼 subscriber 객체도 이벤트 인터페이스의 구현을 제공합니다. 그러나 서버의 연결 포인트에 연결하는 것이 아니라 특정 이벤트 객체를 예약한다는 점에서 이벤트 싱크와 다릅니다.

subscriber 객체를 정의하려면 구현하고자 하는 이벤트 객체의 인터페이스를 선택하기 위해 COM Object 마법사를 사용합니다. 마법사는 이벤트 핸들러를 채워 넣을 수 있는 뼈대가 되는 메소드를 갖는 구현 유닛을 생성합니다. 기존 인터페이스를 구현하기 위해 COM Object 마법사를 사용하는 것에 대한 자세한 내용은 36-2 페이지의 "COM 객체 마법사 사용"을 참조하십시오.

참고 사용자가 구현할 수 있는 인터페이스의 목록에 나타나지 않을 경우 마법사를 사용하여 이벤트 객체의 인터페이스를 레지스트리에 추가해야 할 수도 있습니다.

일단 subscriber 객체를 생성하면 이벤트 객체의 인터페이스 또는 해당 인터페이스에 있는 개별 메소드(이벤트)를 반드시 예약해야 합니다. 사용자가 선택할 수 있는 세 가지 타입의 예약이 있습니다.

- **일시적 예약.** 전통적인 이벤트 싱크처럼 일시적인 예약은 객체 인스턴스의 수명과 관련이 있습니다. subscriber 객체가 해제되면 예약은 종료되고 COM+는 더 이상 이벤트를 진행하지 않습니다.
- **지속적 예약.** 이것은 특정 객체 인스턴스가 아닌 객체 클래스에 연결됩니다. 이벤트가 발생하면 COM은 subscriber 객체의 인스턴스를 찾거나 시작하고 이벤트 핸들러를 호출합니다. In-process 객체(DLL)는 이러한 타입의 예약을 사용합니다.
- **사용자별 예약.** 이러한 예약은 좀더 안전한 버전의 일시적 예약을 제공합니다. 이벤트를 발생시키는 subscriber 객체 및 서버 객체는 모두 동일한 시스템에 있는 동일한 사용자 계정에서 실행되어야 합니다.

참고 COM+ 이벤트를 예약한 객체는 반드시 COM+ 애플리케이션에 설치되어야 합니다.

타입 라이브러리가 없는 서버에 대한 클라이언트 생성

객체 연결 및 포함(OLE)과 같은 일부 이전 COM 기술은 타입 라이브러리에 있는 타입 정보를 제공하지 않습니다. 그 대신 이미 정의된 인터페이스의 표준 집합에 의존합니다. 그러한 객체를 호스트하는 클라이언트를 작성하려면 *TOleContainer* 컴포넌트를 사용할 수 있습니다. 이 컴포넌트는 컴포넌트 팔레트의 System 페이지에 있습니다.

*TOleContainer*는 Ole2 객체의 호스트 사이트로 작동합니다. *IOleClientSite* 인터페이스를 구현하고 옵션으로 *IOleDocument* 사이트를 구현합니다. 통신은 OLE 동사를 사용하여 처리됩니다.

*TOleContainer*를 사용하려면 다음과 같이 합니다.

- 1 폼에 *TOleContainer* 컴포넌트를 놓습니다.
- 2 Active document를 호스트하려면 *AllowActiveDoc* 속성을 *True*로 설정합니다.
- 3 호스트된 객체가 *TOleContainer*에 나타나야 하는지 또는 별도의 창에 나타나야 하는지 나타내기 위해 *AllowInPlace* 속성을 설정합니다.
- 4 객체가 활성화되거나, 비활성화되거나, 이동하거나 또는 크기가 조정될 때 응답하도록 이벤트 핸들러를 작성합니다.
- 5 디자인 타임 시 *TOleContainer* 객체를 바인딩하려면 Insert Object를 마우스 오른 쪽 버튼으로 클릭하고 선택합니다. Insert Object 대화 상자에서 호스트할 서버 객체를 선택합니다.
- 6 런타임 시 *TOleContainer* 객체를 바인딩하려면 서버 객체를 식별하는 방법에 따라 여러 가지 메소드 가운데에서 선택할 수 있습니다. 여기에는 프로그램 ID가 있는 *CreateObject*, 객체가 저장된 파일의 이름이 있는 *CreateObjectFromFile*, 객체 생성 방법에 관한 정보가 포함된 레코드가 있는 *CreateObjectFromInfo*, 객체가 저장되고 이를 포함하는 것이 아니라 연결하는 파일의 이름이 있는 *CreateLinkToFile*이 있습니다.
- 7 일단 객체가 바인딩되면 *OleObjectInterface* 속성을 사용하는 인터페이스에 액세스할 수 있습니다. 하지만 Ole2 객체와의 통신은 OLE 동사를 기반으로 하기 때문에 *DoVerb* 메소드를 사용하는 서버에 명령을 보낼 수 있습니다.
- 8 서버 객체를 해제하려면 *DestroyObject* 메소드를 호출합니다.

36

일반 COM 서버 생성

Delphi의 마법사를 이용하면 다양한 COM 객체를 만들 수 있습니다. 가장 일반적인 COM 객체는 클라이언트가 호출할 수 있는 기본 인터페이스를 통해 속성과 메소드(및 이벤트)를 노출하는 서버입니다.

참고 COM 서버와 Automation은 CLX 애플리케이션에서는 사용할 수 없습니다. 이 기술은 Windows에서만 사용할 수 있으며 크로스 플랫폼을 지원하지 않습니다.

특히 다음 두 가지 마법사를 이용하면 일반 COM 객체를 쉽게 만들 수 있습니다.

- COM 객체 마법사는 기본 인터페이스가 *IUnknown*에서 파생되거나 사용자의 시스템에 이미 등록된 인터페이스를 구현하는 경량급(lightweight) COM 객체를 만듭니다. 이 마법사는 사용자가 만들 수 있는 COM 객체의 타입에 유연성을 최대한 제공합니다.
- Automation 객체 마법사는 기본 인터페이스가 *IDispatch*에서 파생된 일반 Automation 객체를 만듭니다. *IDispatch*는 표준 마샬링(marshaling) 메커니즘을 도입하고 인터페이스 호출의 지연 바인딩(late binding)을 지원합니다.

참고 COM은 특정 상황을 처리하기 위한 많은 표준 인터페이스와 메커니즘을 정의합니다. Delphi 마법사는 대부분의 일반적인 작업을 자동화합니다. 하지만 사용자 지정 마샬링과 같은 일부 작업은 Delphi 마법사에서 지원하지 않습니다. 이에 관한 정보나 Delphi가 명시적으로 지원하지 않는 다른 기술들에 대한 내용은 Microsoft Developer's Network(MSDN) 설명서를 참조하십시오. 또한 Microsoft 웹 사이트에는 COM 지원에 대한 정보도 나와 있습니다.

COM 객체 생성에 대한 개요

Automation 객체 마법사를 사용하여 새 Automation 서버를 생성하든지 COM 객체 마법사를 사용하여 다른 타입의 COM 객체 마법사를 생성하든지 그 과정은 동일합니다. 다음과 같은 단계를 거칩니다.

- 1 COM 객체를 디자인합니다.

- 2 COM 객체 마법사 또는 Automation 객체 마법사를 사용하여 서버 객체를 만듭니다.
- 3 객체가 클라이언트에 노출하는 인터페이스를 정의합니다.
- 4 COM 객체를 등록합니다.
- 5 애플리케이션을 테스트하고 디버그합니다.

COM 객체 디자인

COM 객체를 디자인할 때 구현할 COM 인터페이스를 결정해야 합니다. 이미 정의되어 있는 인터페이스를 구현하는 COM 객체를 작성할 수도 있고 구현할 객체의 새로운 인터페이스를 정의할 수도 있습니다. 또한 사용자의 객체가 둘 이상의 인터페이스를 지원하게 할 수 있습니다. 지원할 표준 COM 인터페이스에 대한 자세한 내용은 MSDN 설명서를 참조하십시오.

- 기존 인터페이스를 구현하는 COM 객체를 만들려면 COM 객체 마법사를 사용하십시오.
- 새로 정의하는 인터페이스를 구현하는 COM 객체를 만들려면 COM 객체 마법사 또는 Automation 객체 마법사를 사용하십시오. COM 객체 마법사는 *IUnknown*에서 파생된 새 기본 인터페이스를 생성할 수 있고, Automation 객체 마법사는 객체에 *IDispatch*에서 파생된 기본 인터페이스를 제공합니다. 어느 마법사를 사용하든지 상관 없이 항상 Type Library 에디터를 사용하여 마법사가 생성하는 기본 인터페이스의 부모 인터페이스를 나중에 변경할 수 있습니다.

지원할 인터페이스를 결정하는 것 외에 COM 객체가 in-process 서버, out-of-process 서버, 원격 서버 중 어느 것인지 결정해야 합니다. in-process 서버와 타입 라이브러리를 사용하는 out-of-process 및 원격 서버의 경우 COM은 데이터를 마샬링합니다. 그렇지 않으면 데이터를 out-of-process 서버에 마샬링할 방법을 고려해야 합니다. 서버 타입에 대한 자세한 내용은 33-6 페이지의 "In-process, Out-of-process 및 원격 서버"를 참조하십시오.

COM 객체 마법사 사용

COM 객체 마법사는 다음과 같은 작업을 수행합니다.

- 새로운 유닛을 생성합니다.
- *TCOMObject*에서 파생된 새 클래스를 정의하고 클래스 팩토리 생성자를 설정합니다. 기존 클래스에 대한 자세한 내용은 33-21 페이지의 "마법사가 생성하는 코드"를 참조하십시오.
- 옵션으로 타입 라이브러리를 프로젝트에 추가하고 사용자의 객체와 해당 인터페이스를 타입 라이브러리에 추가하기도 합니다.

COM 객체를 만들기 전에 구현할 기능을 포함하는 애플리케이션에 대한 프로젝트를 작성하거나 여십시오. 사용자의 필요에 따라 프로젝트는 애플리케이션과 ActiveX 라이브러리 중 하나일 수 있습니다.

다음과 같은 방법으로 COM 객체 마법사를 불러 옵니다.

- 1 File|New|Other를 선택하여 New Items 대화 상자를 엽니다.
- 2 ActiveX 탭을 선택합니다.
- 3 COM 객체 아이콘을 더블 클릭합니다.

마법사에서 다음을 지정해야 합니다.

- **CoClass name:** 이것은 클라이언트에게 보이는 것과 동일한 객체 이름입니다. 객체를 구현하기 위해 만든 클래스는 이 이름을 가지며 앞에 'I'가 붙습니다. 기존 인터페이스를 구현하지 않을 경우 마법사는 앞에 'I'가 붙고 이 이름을 갖는 기본 인터페이스를 CoClass에게 제공합니다.
- **구현할 인터페이스:** 기본적으로 마법사는 *IUnknown*에서 파생된 기본 인터페이스를 객체에 제공합니다. 마법사를 종료하고 난 후 Type Library 에디터를 사용하여 속성과 메소드를 이 인터페이스에 추가할 수 있습니다. 하지만 구현할 객체에 대해 이미 정의된 인터페이스도 선택할 수 있습니다. COM 객체 마법사에서 List 버튼을 클릭하여 Interface Selection 마법사를 불러 옵니다. 여기서 시스템에 등록된 타입 라이브러리에 정의된 이중 또는 사용자 지정 인터페이스를 선택할 수 있습니다. 선택한 인터페이스가 새 CoClass의 기본 인터페이스가 됩니다. 마법사는 이 인터페이스의 모든 메소드를 생성된 구현 클래스에 추가하므로 구현 유닛에서 메소드의 몸체만 채우면 됩니다. 기존 인터페이스를 선택한 경우 그 인터페이스가 사용자 프로젝트의 타입 라이브러리에 추가되지 않는다는 것에 유의하십시오. 즉, 사용자의 객체를 배포할 때는 인터페이스를 정의하는 타입 라이브러리도 배포해야 한다는 것을 의미합니다.
- **인스턴스:** in-process 서버를 만들지 않을 경우, COM이 사용자의 COM 객체를 수행하는 애플리케이션을 실행하는 방법을 지정해야 합니다. 애플리케이션에서 COM 객체를 하나 이상 구현하는 경우 그 모든 객체에 대해 동일한 인스턴스를 지정해야 합니다. 다른 가능성에 대한 내용은 36-5 페이지의 "COM 객체 인스턴스 타입"을 참조하십시오.
- **스레드 모델:** 일반적으로 사용자의 객체에 대한 클라이언트 요청은 다른 실행 스레드에 입력됩니다. 사용자는 클라이언트가 사용자의 객체를 호출할 때 COM이 스레드를 일련화하는 방법을 지정할 수 있습니다. 어떤 스레드 모델을 선택했는지에 따라 객체 등록 방법이 달라집니다. 사용자가 선택한 모델에 대한 모든 스레드 지원은 사용자 스스로 해결해야 합니다. 다른 가능성에 대한 내용은 36-6 페이지의 "스레드 모델 선택"을 참조하십시오. 애플리케이션에 대한 스레드 지원 방법에 대한 내용은 9장 "다중 스레드 애플리케이션 작성"을 참조하십시오.
- **타입 라이브러리:** 객체의 타입 라이브러리를 포함할 것인지 여부를 선택할 수 있습니다. 다음 두 가지 이유에서 객체의 타입 라이브러리를 포함하는 것이 좋습니다. Type Library 에디터를 사용하여 인터페이스를 정의할 수 있으며 그에 따라 대부분의 구현을 업데이트하고 클라이언트는 객체와 인터페이스에 관한 정보를 쉽게 얻을 수 있습니다. 기존 인터페이스를 구현할 경우, Delphi에서는 사용자의 프로젝트가 타입 라이브러리를 사용할 것을 요구합니다. 이렇게 하는 것이 원래의 인터페이스 선언에 액세스할 수 있는 유일한 방법입니다. 타입 라이브러리에 대한 내용은 33-15 페이지의 "타입 라이브러리" 및 34장 "Type Library 작업"을 참조하십시오.

- **마샬링**: 타입 라이브러리를 만들려고 선택했고 Automation 호환 타입으로 제한할 경우 COM을 사용하면 in-process 서버를 생성하지 않을 때 마샬링을 처리할 수 있습니다. 타입 라이브러리에서 사용자 객체의 인터페이스를 OleAutomation으로 표시함으로써 COM으로 프록시와 스텝을 설정하고 프로세스 경계에 걸쳐 매개변수를 전달하는 핸들을 설정할 수 있습니다. 이 처리에 대한 자세한 내용은 33-8 페이지의 "마샬링 메커니즘 (Marshaling mechanism)"을 참조하십시오. 새 인터페이스를 생성할 경우 인터페이스가 Automation에 호환되는지 여부만 지정할 수 있습니다. 기존 인터페이스를 선택한 경우 해당 속성은 이미 타입 라이브러리에 지정됩니다. 사용자 객체의 인터페이스가 OleAutomation으로 표시되지 않는 경우 in-process 서버를 작성하거나 사용자 고유의 마샬링 코드를 작성해야 합니다.

COM 객체에 대한 설명을 선택적으로 추가할 수 있습니다. 이 설명을 새로 만들면 객체의 타입 라이브러리에 나타납니다.

Automation 객체 마법사 사용

Automation 객체 마법사는 다음과 같은 작업을 수행합니다.

- 새로운 유닛을 생성합니다.
- *TAutoObject*에서 파생되는 새 클래스를 정의하고 클래스 팩토리 생성자를 설정합니다. 기존 클래스에 대한 자세한 내용은 33-21 페이지의 "마법사가 생성하는 코드"를 참조하십시오.
- 사용자의 프로젝트에 타입 라이브러리를 추가하고 사용자의 객체와 해당 인터페이스를 타입 라이브러리에 추가합니다.

Automation 객체를 만들기 전에 노출하려는 기능을 포함하는 애플리케이션의 프로젝트를 작성하거나 여십시오. 사용자의 필요에 따라 프로젝트는 애플리케이션과 ActiveX 라이브러리 중 하나일 수 있습니다.

다음과 같은 방법으로 Automation 마법사를 표시합니다.

- 1 File|New|Other를 선택합니다.
- 2 ActiveX 탭을 선택합니다.
- 3 Automation 객체 아이콘을 더블 클릭합니다.

마법사 대화 상자에서 다음을 지정합니다.

- **CoClass name**: 이것은 클라이언트에게 보이는 것과 동일한 객체 이름입니다. CoClass 이름에 기반하여 앞에 'I'가 붙은 이름을 가진 객체의 기본 인터페이스가 만들어지고 객체를 구현하기 위해 만든 클래스는 앞에 'T'가 붙은 이름을 갖게 됩니다.
- **인스턴스**: in-process 서버를 만들지 않을 경우, COM이 사용자의 COM 객체를 담고 있는 애플리케이션을 실행하는 방법을 지정해야 합니다. 애플리케이션에서 COM 객체를 하나 이상 구현하는 경우 그 모든 객체에 대해 동일한 인스턴스를 지정해야 합니다. 다른 가능성에 대한 내용은 36-5 페이지의 "COM 객체 인스턴스 타입"을 참조하십시오.

- **스레드 모델:** 일반적으로 사용자의 객체에 대한 클라이언트 요청은 다른 실행 스레드에 입력됩니다. 사용자는 클라이언트가 사용자의 객체를 호출할 때 COM이 스레드를 일련화하는 방법을 지정할 수 있습니다. 어떤 스레드 모델을 선택했는지에 따라 객체 등록 방법이 달라집니다. 사용자가 선택한 모델에 대한 모든 스레드 지원은 사용자 스스로 해결해야 합니다. 다른 가능성에 대한 내용은 36-6 페이지의 "스레드 모델 선택"을 참조하십시오. 애플리케이션에 대한 스레드 지원 방법에 대한 내용은 9장 "다중 스레드 애플리케이션 작성"을 참조하십시오.
- **이벤트 지원:** 객체로 클라이언트가 응답할 수 있는 이벤트를 생성할 것인지 여부를 지정해야 합니다. 마법사는 이벤트를 생성하는 데 필요한 인터페이스 및 클라이언트 이벤트 핸들러에 대한 호출 디스패치에 대한 지원을 제공할 수 있습니다. 이벤트가 작업하는 방식과 이벤트 구현 시 사용자가 해야 할 일에 대한 내용은 36-10 페이지의 "클라이언트에 이벤트 노출"을 참조하십시오.

COM 객체에 대한 설명을 선택적으로 추가할 수 있습니다. 이 설명은 사용자 객체의 타입 라이브러리에 나타납니다.

Automation 객체는 **이중 인터페이스**를 구현하는데 VTable을 통한 우선 바인딩(컴파일 시)과 *IDispatch* 인터페이스를 통한 지연 바인딩(런타임 시)을 모두 지원합니다. 자세한 내용은 36-13 페이지의 "이중 인터페이스"를 참조하십시오.

COM 객체 인스턴스 타입

많은 COM 마법사는 객체의 인스턴스 모드를 지정하도록 합니다. 인스턴스는 하나의 실행 파일에서 만들 수 있는 객체 클라이언트 수를 결정합니다. 예를 들어, Single Instance 모델을 지정하고 나서 COM이 일단 클라이언트가 사용자의 객체를 인스턴스화한 경우라면 COM은 뷰에서 애플리케이션을 제거하므로 다른 클라이언트는 자신의 애플리케이션 인스턴스를 실행해야 합니다. 이는 전체적으로 사용자 애플리케이션의 가시성에 영향을 주기 때문에 인스턴스 모드는 클라이언트에 의해 인스턴스화될 수 있는 사용자 애플리케이션의 모든 객체에 걸쳐 일관되어야 합니다. 즉, 애플리케이션에서 Single Instance 모드를 사용하는 하나의 객체와 동일한 애플리케이션에서 Multiple Instance 모드를 사용하는 다른 객체를 만들지 않아야 합니다.

참고 사용자의 COM 객체가 in-process 서버로 사용될 때는 인스턴스가 무시됩니다.

마법사가 새 COM 객체를 만들면 다음 중 어느 인스턴스 타입이든지 가질 수 있습니다.

인스턴스	의미
Internal	객체를 내부적으로만 만들 수 있습니다. 사용자의 애플리케이션이 클라이언트에게 인터페이스를 전달하는 객체를 만들 수 있다고 해도 외부 애플리케이션은 객체의 인스턴스를 직접 만들 수 없습니다.
Single Instance	클라이언트가 각 실행 파일(애플리케이션)에 대해 객체의 단일 인스턴스만 만들도록 하므로 여러 인스턴스를 만들면 애플리케이션의 여러 인스턴스를 실행하게 됩니다. 각 클라이언트는 서버 애플리케이션의 전용 인스턴스를 가지고 있습니다. 이 옵션은 일반적으로 MDI(Multiple Document Interface) 애플리케이션에 사용됩니다.
Multiple Instances	여러 클라이언트가 동일한 프로세스 공간에서 객체의 인스턴스를 만들 수 있다고 지정합니다. 클라이언트가 서비스를 요청할 때 객체의 개별 인스턴스가 만들어집니다. (즉, 하나의 실행 파일에 여러 인스턴스가 있을 수 있습니다.)

스레드 모델 선택

마법사를 사용하여 객체를 만들 때는 사용자의 객체에서 지원하기로 한 스레드 모델을 선택합니다. 사용자의 COM 객체에 스레드 지원을 추가하면 성능을 향상시킬 수 있습니다. 그 이유는 여러 클라이언트가 동시에 애플리케이션을 액세스할 수 있기 때문입니다.

표 36.1은 지정할 수 있는 다른 스레드 모델을 나열한 것입니다.

표 36.1 COM 객체의 스레드 모델

스레드 모델	설명	pros 및 cons 구현
Single	서버는 어떤 스레드 지원도 하지 않습니다. COM은 애플리케이션이 요청을 한 번에 하나씩 받을 수 있도록 클라이언트 요청들을 일련화합니다.	클라이언트가 한 번에 하나씩 처리되므로 어떤 스레드 지원도 필요하지 않습니다. 성능 면에서 이점이 없습니다.
Single-threaded apartment	COM은 한 번에 하나의 클라이언트 스레드만 객체를 호출할 수 있습니다. 모든 클라이언트 호출은 객체가 만들어졌던 스레드를 사용합니다.	객체는 자신의 인스턴스 데이터를 안전하게 액세스할 수 있지만 전역 데이터는 임계 구역이나 일부 다른 일련화 형식을 사용하여 보호되어야 합니다. 스레드의 로컬 변수는 여러 호출에서 신뢰할만합니다. 일부 성능에 이점이 있습니다.
Free (multi-threaded apartment라고도 함)	객체는 언제든지 원하는 수의 스레드에 대한 호출을 받을 수 있습니다.	객체는 임계 구역이나 일부 다른 일련화 형식을 사용하여 모든 인스턴스와 전역 데이터를 보호해야 합니다. 스레드 로컬 변수는 여러 호출에서 신뢰할만하지 않습니다.

표 36.1 COM 객체의 스레드 모델 (계속)

스레드 모델	설명	pros 및 cons 구현
Both	이는 outgoing 호출(예를 들어, 콜백)이 동일한 스레드에서 실행되는 것을 제외하고는 Free 스레드 모델과 같습니다.	최대 성능과 유연성을 제공합니다. 애플리케이션은 outgoing 호출에 제공되는 매개변수에 대한 스레드 지원을 제공하지 않아도 됩니다.
Neutral	여러 클라이언트가 동시에 다른 스레드의 객체를 호출할 수 있지만 COM은 두 개 호출이 확실히 충돌하지 않게 합니다.	여러 메소드가 액세스하는 인스턴스 데이터와 전역 데이터와 관련된 스레드 충돌을 막아야 합니다. 이 모델은 사용자 인터페이스(비주얼 컨트롤)를 갖는 객체와 함께 사용해서는 안 됩니다. 이 모델은 COM+에서만 사용할 수 있습니다. COM에서는 Apartment 모델에 매핑됩니다.

참고 로컬 변수(콜백의 경우는 제외)는 스레드 모델에 상관 없이 항상 안전합니다. 그 이유는 로컬 변수는 스택에 저장되고 각 스레드는 자신의 스택을 가지기 때문입니다. 로컬 변수는 free 스레드 사용 시에는 콜백에서 안전하지 않을 수도 있습니다.

마법사에서 선택하는 스레드 모델에 따라 객체가 시스템 레지스트리에 등록되는 방식이 결정됩니다. 사용자의 객체 구현이 선택한 스레드 모델에 적합하지 확인해야 합니다. 스레드 안전 코드 작성에 대한 일반적인 내용은 9장 "다중 스레드 애플리케이션 작성"을 참조하십시오.

In-process 서버의 경우 마법사에서 스레드 모델을 설정하면 CLSID 레지스트리 항목에 스레드 모델 키가 설정됩니다.

Out-of-process 서버는 EXE로 등록되고 Delphi는 필요로 하는 최고의 스레드 모델에 대한 COM을 초기화합니다. 예를 들어, EXE가 free 스레드 객체를 포함하고 있으면 free 스레드에 맞게 초기화됩니다. 즉, EXE에 포함된 free 스레드나 apartment thread 대해 예상된 지원을 제공할 수 있다는 뜻입니다. EXE에서 스레드 동작을 수동으로 오버라이드하려면 온라인 도움말에 설명되어 있는 *CoInitFlags* 변수를 사용하십시오.

Free thread 모델을 지원하는 객체 작성

객체가 둘 이상의 스레드에서 액세스되어야 할 때마다 apartment 스레드보다는 free 스레드 모델을 사용하거나 두 가지를 함께 사용하십시오. 일반적인 예로는 원격 시스템의 객체에 연결된 클라이언트 애플리케이션이 있습니다. 원격 클라이언트가 그 객체에 대한 메소드를 호출하면 서버는 서버 시스템의 스레드 풀에서 스레드에 대한 호출을 받습니다. 이 수신 스레드는 호출을 실제 객체에 로컬로 만듭니다. 그리고 객체가 free 스레드 모델을 지원하기 때문에 스레드는 객체를 직접 호출할 수 있습니다.

그 대신 객체가 apartment thread 모델을 지원했으면 호출은 객체가 만들어진 스레드로 전송되고 결과는 클라이언트로 반환되기 전에 다시 수신 스레드로 다시 전송될 것입니다. 이 방식에는 추가 마샬링이 필요합니다.

Free 스레드를 지원하려면 각 메소드마다 인스턴스 데이터가 액세스될 수 있는 방법을 고려해야 합니다. 메소드가 인스턴스 데이터에 쓸 경우, 임계 구역이나 일부 다른 일련화 형식을 사용하여 인스턴스 데이터를 보호해야 합니다. 임계 호출 일련화의 오버헤드는 COM의 마샬링 코드를 실행하는 것보다 적습니다.

인스턴스 데이터가 읽기 전용이면 일련화가 필요하지 않습니다.

Free 스레드 in-process 서버는 free 스레드 마샬러를 이용하여 추상화에서 외부 객체로 작용하여 성능을 향상시킬 수 있습니다. free 스레드 마샬러는 free 스레드 DLL이 free 스레드가 아닌 호스트(클라이언트)에 의해 호출될 때 COM의 표준 스레드 처리를 위한 단축키를 제공합니다.

Free 스레드 마샬러를 추상화하려면 다음과 같이 해야 합니다.

- 결과적인 free 스레드 마샬러가 사용할 사용자 객체의 *IUnknown* 인터페이스를 전달하여 *CoCreateFreeThreadedMarshaler*를 호출합니다.

```
CoCreateFreeThreadedMarshaler(self as IUnknown, FMarshaler);
```

이 줄은 free 스레드 마샬러에 대한 인터페이스를 클래스 멤버인 *FMarshaler*에 할당합니다.

- Type Library 에디터를 사용하여 *IMarshal* 인터페이스를 CoClass가 구현하는 일련의 인터페이스에 추가합니다.
- 사용자 객체의 *QueryInterface* 메소드에서 *IDD_IMarshal*에 대한 호출을 free 스레드 마샬러(위의 *FMarshaler*로 저장됨)에 위임합니다.

경고 Free 스레드 마샬러는 추가 효율성을 제공하기 위해 COM 마샬링의 일반 규칙을 위반합니다. 그러므로 주의해서 사용해야 합니다. 특히 in-process 서버에서 free 스레드 객체로만 추상화되어야 하며(다른 스레드가 아닌) 그것을 사용하는 객체에 의해서만 인스턴스화되어야 합니다.

Apartment threading model을 지원하는 객체 작성

(Single-threaded) apartment threading model을 구현하려면 다음과 같은 몇 가지 규칙을 따라야 합니다.

- 애플리케이션에서 작성되는 첫 번째 스레드는 COM의 메인 스레드입니다. 이것은 대개 WinMain이 호출되었던 스레드입니다. 또한 COM을 초기화 해제하는 마지막 스레드여야 합니다.
- Apartment 스레드 모델에서 각 스레드에는 메시지 루프가 있어야 하며 메시지 대기열을 자주 확인해야 합니다.
- 스레드는 COM 인터페이스에 대한 포인터를 가지며 해당 포인터는 해당 스레드에서만 사용될 수 있습니다.

Single-threaded apartment 모델은 스레드 지원을 전혀 제공하지 않는 것과 free 스레드 모델의 완전한 다중 스레드 지원 사이의 중간에 해당됩니다. apartment 모델에 대한 서버의 위임은 서버가 모든 전역 데이터(예: 객체 카운트)에 대한 액세스를 일련화했다는 것을 보증합니다. 그 이유는 다른 객체들이 다른 스레드로부터 전역 데이터를 액세스

하려고 할 수 있기 때문입니다. 하지만 객체의 인스턴스 데이터는 안전합니다. 메소드가 항상 동일한 스레드에서 호출되기 때문입니다.

일반적으로 웹 브라우저에서 사용할 컨트롤은 apartment 스레드 모델을 사용합니다. 브라우저 애플리케이션이 스레드를 항상 apartment로 초기화하기 때문입니다.

Neutral threading model을 지원하는 객체 작성

COM+에서는 Free 스레드와 Apartment 스레드 사이에 있는 또 다른 스레드 모델인 Neutral 모델을 사용할 수 있습니다. free 스레드 모델처럼 이 모델을 사용하면 여러 스레드가 동시에 사용자의 객체에 액세스할 수 있습니다. 객체가 만들어졌던 스레드로 전송하는 추가 마살링은 없습니다. 하지만 객체는 충돌하는 어떤 호출도 받지 않습니다.

Neutral threading model을 사용하는 객체 작성은 객체의 인터페이스에 있는 다른 메소드들이 액세스할 수 있을 경우 스레드 충돌에 대해 인스턴스 데이터를 보호해야 한다는 것을 제외하고는 apartment 스레드 객체를 작성하는 규칙을 따르게 됩니다. 단일 인터페이스 메소드만 액세스하는 인스턴스 데이터는 자동적으로 스레드에 안전합니다.

COM 객체의 인터페이스 정의

마법사를 사용하여 COM 객체를 만들 때 (COM 객체 마법사에서 따로 지정하지 않는 한) 자동으로 타입 라이브러리를 생성합니다. 타입 라이브러리를 통해 호스트 애플리케이션은 객체가 무엇을 할 수 있는지 알아냅니다. 그리고 Type Library 에디터를 사용하여 사용자 객체의 인터페이스를 정의할 수도 있습니다. Type Library 에디터에서 정의하는 인터페이스는 사용자의 객체가 클라이언트에 노출하는 속성, 메소드 및 이벤트를 정의합니다.

참고 COM 객체 마법사에서 기존 인터페이스를 선택했으면 속성과 메소드를 추가할 필요가 없습니다. 인터페이스의 정의는 정의되었던 타입 라이브러리로부터 import됩니다. 그 대신 구현 유닛에서 import된 인터페이스의 메소드를 찾아 몸체를 채워야 합니다.

객체의 인터페이스에 속성 추가

Type Library 에디터를 사용하여 객체의 인터페이스에 속성을 추가할 경우 속성 값이나 메소드를 읽어 속성 값을 설정하는 메소드가 자동으로 추가됩니다. 그 다음에 Type Library 에디터는 이들 메소드를 구현 클래스에 추가하고 구현 유닛에서 완성하는 빈 메소드 구현을 만듭니다.

다음과 같은 방법으로 객체의 인터페이스에 속성을 추가합니다.

- 1 Type Library 에디터에서 객체의 기본 인터페이스를 선택합니다.

- 기본 인터페이스는 "I"로 시작하는 객체 이름이어야 합니다. 기본값을 결정하려면 Type Library 에디터에서 CoClass and Implements 탭을 선택한 후 구현 인터페이스 목록에서 "Default"라고 표시된 것을 확인합니다.
- 2 읽기/쓰기 속성을 노출하려면 툴바에서 Property 버튼을 클릭합니다. 그렇지 않으면 툴바의 Property 버튼 옆에 있는 화살표를 클릭한 다음 노출할 속성 타입을 클릭합니다.
 - 3 Attributes 창에서 속성의 이름과 타입을 지정합니다.
 - 4 툴바에서 Refresh 버튼을 클릭합니다.
속성 액세스 메소드를 위한 정의와 뼈대 구현이 객체의 구현 유닛에 삽입됩니다.
 - 5 구현 유닛에서 속성의 액세스 메소드를 찾습니다. 이들은 Get_PropertyName 과 Set_PropertyName 형식의 이름을 갖습니다. 객체의 속성 값을 구하거나 설정하는 코드를 추가합니다. 이러한 코드는 애플리케이션 내의 기존 함수를 호출하거나 객체 정의에 추가하는 데이터 멤버를 액세스하거나 그렇지 않으면 속성을 구현할 수도 있습니다.

객체의 인터페이스에 메소드 추가

Type Library 에디터를 사용하여 객체의 인터페이스에 메소드를 추가하면 Type Library 에디터는 구현 클래스에 메소드를 추가하고 구현 유닛에 완성할 구현을 만듭니다.

다음과 같은 방법으로 객체의 인터페이스를 통해 메소드를 노출합니다.

- 1 Type Library 에디터에서 객체의 기본 인터페이스를 선택합니다.
기본 인터페이스는 "I"로 시작하는 객체 이름이어야 합니다. 기본값을 결정하려면 Type Library 에디터에서 CoClass and Implements 탭을 선택한 후 구현 인터페이스 목록에서 "Default"라고 표시된 것을 선택 표시합니다.
- 2 Method 버튼을 클릭합니다.
- 3 Attributes 창에서 메소드의 이름을 지정합니다.
- 4 Parameters 창에서 메소드의 반환 타입을 지정하고 해당 매개변수를 추가합니다.
- 5 툴바에서 Refresh 버튼을 클릭합니다.
메소드의 정의와 뼈대 구현이 객체의 구현 유닛에 삽입됩니다.
- 6 구현 유닛에서 새로 삽입된 메소드 구현을 찾습니다. 메소드는 완전히 비어 있습니다. 메소드가 나타내는 작업마다 수행할 몸체를 채웁니다.

클라이언트에 이벤트 노출

COM 객체가 생성할 수 있는 이벤트에는 두 가지 타입, 종래의 이벤트와 COM+ 이벤트가 있습니다.

- COM+ 이벤트는 이벤트 객체 마법사를 사용하여 개별적인 이벤트 객체를 만들고 서버 객체에서 그 이벤트 객체를 호출하는 코드를 추가하도록 합니다. COM+ 이벤트

생성에 대한 자세한 내용은 39-18 페이지의 "COM+에서 이벤트 생성"을 참조하십시오.

- 마법사를 사용하면 종래의 이벤트를 생성할 때 많은 작업을 처리할 수 있습니다. 이 과정은 다음과 같습니다.

참고 COM 객체 마법사는 이벤트 지원 코드를 생성하지 않습니다. 객체가 종래의 이벤트를 생성할 수 있게 하려면 Automation 객체 마법사를 사용해야 합니다.

다음과 같은 방법으로 객체가 이벤트를 생성합니다.

- 1 Automation 마법사에서 Generate event support code 상자를 선택 표시합니다.
 마법사는 기본 인터페이스뿐만 아니라 Events 인터페이스를 포함하는 객체를 생성합니다. 이 Events 인터페이스는 *ICoClassnameEvents* 형식의 이름을 갖습니다. 이는 outgoing(소스) 인터페이스로서 객체가 구현하는 인터페이스가 아니라 오히려 클라이언트가 구현해야 하고 객체가 호출하는 인터페이스라는 의미입니다. (CoClass를 선택하고 Implements 페이지로 가서 Events 인터페이스의 Source 열이 *True*로 되어 있는지 확인하면 볼 수 있습니다.)
 Events 인터페이스 외에 마법사는 구현 클래스의 선언에 *IConnectionPointContainer* 인터페이스를 추가하고 이벤트를 처리할 수 있는 여러 클래스 멤버를 추가합니다. 이 새로운 클래스 멤버 중 가장 중요한 것은 *FConnectionPoint*와 *FConnectionPoints*로서 기본 제공 VCL 클래스를 사용하여 *IConnectionPoint*와 *IConnectionPointContainer* 인터페이스를 구현합니다. *FConnectionPoint*는 마법사가 추가하는 또 다른 메소드인 *EventSinkChanged*에 의해 유지됩니다.
- 2 Type Library 에디터에서 객체의 outgoing Events 인터페이스를 선택합니다. (*ICoClassNameEvents* 형식의 이름을 가지고 있습니다.)
- 3 Type Library 툴바에서 Method 버튼을 클릭합니다. Events 인터페이스에 추가하는 각 메소드는 클라이언트가 구현해야 하는 이벤트 핸들러를 나타냅니다.
- 4 Attributes 창에서 MyEvent 같은 이벤트 핸들러의 이름을 지정합니다.
- 5 툴바에서 Refresh 버튼을 클릭합니다.
 객체 구현은 이제 클라이언트 이벤트 싱크를 받아들이고 이벤트가 발생할 때 호출할 인터페이스의 목록을 유지하는 데 필요한 모든 것을 가졌습니다. 이러한 인터페이스를 호출하기 위해 클라이언트에 각 이벤트를 생성하는 메소드를 만들 수 있습니다.
- 6 코드 에디터에서 각 이벤트를 발생시키기 위해 객체에 메소드를 추가합니다. 예를 들면, 다음과 같습니다.

```

unit ev;
interface
uses
  ComObj, AxCtrls, ActiveX, Project1_TLB;
type
  TMyAutoObject = class (TAutoObject, IConnectionPointContainer, IMyAutoObject)
  private
    .
    .
    .
  public

```

```

procedure Initialize; override;
procedure Fire_MyEvent; { Add a method to fire the event}

```

- 7 마지막 단계에서 추가했던 메소드를 구현하여 객체의 *FConnectionPoint* 멤버가 유지하는 모든 이벤트 싱크에서 반복하도록 합니다.

```

procedure TMyAutoObject.Fire_MyEvent;
var
  I:Integer;
  EventSinkList:TList;
  EventSink: IMyAutoObjectEvents;
begin
  if FConnectionPoint <> nil then
  begin
    EventSinkList :=FConnectionPoint.SinkList; {get the list of client sinks }
    for I := 0 to EventSinkList.Count - 1 do
    begin
      EventSink := IUnknown(FEvents[I]) as IMyAutoObjectEvents;
      EventSink.MyEvent;
    end;
  end;
end;

```

- 8 이벤트를 발생시켜 클라이언트에게 이벤트가 발생했음을 알리도록 해야할 때마다 모든 이벤트 싱크에 이벤트를 디스패치하는 메소드를 호출합니다.

```

if EventOccurs then Fire_MyEvent; { Call method you created to fire events.}

```

Automation 객체의 이벤트 관리

서버가 종래의 COM 이벤트를 지원하는 경우, 클라이언트에 의해 구현되는 outgoing 인터페이스의 정의를 제공해야 합니다. 이 outgoing 인터페이스에는 클라이언트가 서버 이벤트에 대한 응답을 구현해야 하는 모든 이벤트 핸들러가 포함되어 있습니다.

클라이언트는 outgoing 이벤트 인터페이스를 구현했을 경우 서버의 *IConnectionPointContainer* 인터페이스에 쿼리하여 이벤트 공지 수신을 등록합니다. *IConnectionPointContainer* 인터페이스는 서버의 *IConnectionPoint* 인터페이스를 반환하는데, 그러면 클라이언트는 이것을 사용하여 이벤트 핸들러(싱크라고도 함)의 구현을 가리키는 포인터를 서버에 전달합니다.

서버는 위에서 설명한 대로 이벤트가 발생할 때 모든 클라이언트 싱크와 호출 메소드의 목록을 유지합니다.

Automation 인터페이스

Automation 객체 마법사는 기본적으로 이중 인터페이스를 구현하는데 이것은 Automation 객체가 다음 두 가지를 모두 지원한다는 것을 의미합니다.

- 런타임 시의 지연 바인딩으로서 *IDispatch* 인터페이스를 통해 수행됩니다. 이는 디스패치 인터페이스 또는 **dispinterface**로 구현됩니다.

- 컴파일 시 우선 바인딩으로서 객체의 가상 함수 테이블(VTable)의 멤버 함수 중 하나를 직접 호출하면 가능합니다. 이는 **사용자 지정 인터페이스**로 참조됩니다.

참고 *IDispatch*에서 파생되지 않고 COM 객체 마법사에 의해 생성된 인터페이스는 VTable 호출만 지원합니다.

이중 인터페이스

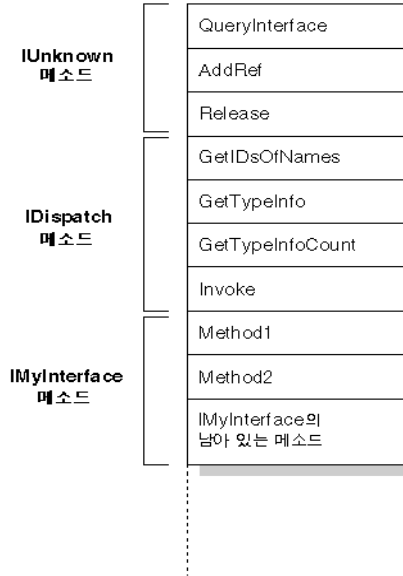
이중 인터페이스는 사용자 지정 인터페이스이자 *dispinterface*로서 *IDispatch*에서 파생하는 COM VTable 인터페이스로 구현됩니다. 런타임 시에만 객체를 액세스할 수 있는 컨트롤러의 경우 *dispinterface*를 사용할 수 있습니다. 컴파일 타임 바인딩을 이용할 수 있는 객체의 경우 더 효율적인 VTable 인터페이스가 사용됩니다.

이중 인터페이스는 다음과 같은 VTable interface와 *dispinterface*의 결합된 이점을 제공합니다.

- VTable interface의 경우 컴파일러는 타입 확인을 수행하고 정보를 더 많이 담고 있는 오류 메시지를 제공합니다.
- 타입 정보를 얻을 수 없는 Automation 컨트롤러의 경우 *dispinterface*는 객체에 런타임 액세스만 제공합니다.
- in-process 서버의 경우 VTable 인터페이스를 통해 빠른 액세스가 가능합니다.
- out-of-process 서버의 경우 VTable 인터페이스와 *dispinterface* 모두에 대해 COM은 데이터를 마샬링합니다. COM은 타입 라이브러리에 포함된 정보를 기반으로 인터페이스를 마샬링할 수 있는 일반 프록시/스텝 구현을 제공합니다. 마샬링에 대한 자세한 내용은 36-15 페이지의 "데이터 마샬링 (Marshaling)"을 참조하십시오.

다음 다이어그램은 이중 인터페이스인 *IMyInterface*를 지원하는 객체의 *IMyInterface* 인터페이스를 나타낸 것입니다. 이중 인터페이스에 대한 VTable의 처음 세 개 항목은 *IUnknown* 인터페이스를 가리키고 다음 네 개 항목은 *IDispatch* 인터페이스를 가리키며 나머지 항목은 사용자 지정 인터페이스의 멤버를 직접 액세스하기 위한 COM 항목입니다.

그림 36.1 이 중 인터페이스 VTable



디스패치 인터페이스

Automation 컨트롤러는 COM *IDispatch* 인터페이스를 사용하여 COM 서버 객체에 액세스하는 클라이언트입니다. 컨트롤러는 먼저 객체를 만든 다음 *IDispatch* 인터페이스에 대한 포인터를 위해 객체의 *IUnknown* 인터페이스를 쿼리합니다. *IDispatch*는 인터페이스 멤버의 고유한 ID인 디스패치 식별자 (dispID) 로 내부적으로 메소드와 속성을 추적합니다. *IDispatch*를 통해 컨트롤러는 디스패치 인터페이스에 대한 객체의 타입 정보를 가져와서 인터페이스 멤버 이름을 특정 dispID에 매핑합니다. 이 dispID는 런타임 시 사용할 수 있으며 컨트롤러는 *IDispatch* 메소드인 *GetIDsOfNames*를 호출하여 이를 얻습니다.

dispID를 가지면 컨트롤러는 *IDispatch* 메소드인 *Invoke*를 호출하여 해당 코드(속성 또는 메소드)를 실행하여 속성이나 메소드의 매개변수를 *Invoke* 매개변수 중 하나에 패키지할 수 있습니다. *Invoke*는 고정된 컴파일 타임 시그니처(signature)를 가지고 있어서 인터페이스 메소드 호출 시 임의의 수의 인수를 허용할 수 있습니다.

*Invoke*의 Automation 객체 구현은 매개변수를 패키지 해제하고 속성이나 메소드를 호출하고 발생하는 오류를 처리할 준비가 되어 있어야 합니다. 속성이나 메소드 반환 시 객체는 그 반환 값을 컨트롤러에 다시 전달합니다.

이를 지연 바인딩이라고 하는데 컨트롤러가 컴파일 타임보다는 런타임 시에 속성이나 메소드에 바인딩하기 때문입니다.

참고 타입 라이브러리를 import할 경우 Delphi는 코드를 생성할 때 dispID에 대해 쿼리하므로 생성된 랩퍼 클래스는 *GetIDsOfNames*를 호출하지 않고도 *Invoke*를 호출할 수 있습니다. 이를 통해 컨트롤러의 런타임 성능이 상당히 좋아질 수 있습니다.

사용자 지정 인터페이스

사용자 지정 인터페이스는 사용자가 정의한 인터페이스로서 클라이언트는 VTable에서의 순서 및 알고 있는 인수 타입에 기반하여 인터페이스 메소드를 호출할 수 있습니다. VTable은 지원하는 인터페이스의 멤버 함수를 포함해서 객체의 멤버인 모든 속성과 메소드의 주소를 나열합니다. 객체가 *IDispatch*를 지원하지 않으면 객체의 사용자 지정 인터페이스 멤버 항목은 즉시 *IUnknown*의 멤버를 따릅니다.

객체가 타입 라이브러리를 가지고 있다면 Type Library 에디터를 사용하여 얻을 수 있는 VTable 레이아웃을 통해 사용자 지정 인터페이스에 액세스할 수 있습니다. 객체가 타입 라이브러리를 가지고 있고 *IDispatch*도 지원한다면 클라이언트는 *IDispatch* 인터페이스의 dispID를 얻을 수 있고 VTable 오프셋에 직접 바인딩할 수 있습니다. Delphi의 Type Library importer (TLIBIMP)는 import 시 dispID를 가져오므로 dispinterface를 사용하는 클라이언트는 *GetIDsOfNames*에 대한 호출을 피할 수 있습니다. 이 정보는 이미 `_TLB` 유닛에 있습니다. 하지만 클라이언트는 여전히 *Invoke*를 호출해야 합니다.

데이터 마샬링(Marshaling)

out-of-process 및 원격 서버의 경우, COM이 현재 프로세스의 외부에서 데이터를 마샬링하는 방법을 고려해야 합니다. 다음과 같은 마샬링을 제공할 수 있습니다.

- 자동으로 *IDispatch* 인터페이스를 사용하여 마샬링합니다.
- 자동으로 서버로 타입 라이브러리를 만들고 OLE Automation 플러그로 인터페이스를 표시하여 마샬링합니다. COM은 타입 라이브러리에서 모든 **Automation 호환** 타입을 마샬링하는 방법을 알고 있으므로 프록시와 스텝을 설정할 수 있습니다. 일부 타입 제한이 적용되어 자동 마샬링이 가능합니다.
- 수동으로 *IMarshal* 인터페이스의 모든 메소드를 구현하여 마샬링합니다. 이를 **사용자 지정 마샬링**이라고 합니다.

참고 첫 번째 메소드 (*IDispatch* 사용)는 Automation 서버에서만 사용할 수 있습니다. 두 번째 메소드는 마법사에 의해 만들어지고 타입 라이브러리를 사용하는 모든 객체에서 자동으로 사용할 수 있습니다.

Automation 호환 타입

OLE Automation으로 표시한 이중 및 디스패치 인터페이스에 선언된 메소드의 함수 결과와 매개변수 타입은 *Automation 호환* 타입이어야 합니다. 다음 타입은 OLE Automation에 호환됩니다.

- *Smallint*, *Integer*, *Single*, *Double*, *WideString*과 같은 이미 정의된 유효한 타입. 전체 목록은 34-11 페이지의 "유효한 타입"을 참조하십시오.
- Type Library에 정의된 열거 타입. OLE Automation 호환 열거 타입은 32비트 값으로 저장되며 매개변수 전달의 목적으로 정수 타입 값으로 취급됩니다.

- OLE Automation에 안전한, 즉 *IDispatch*에서 파생되고 OLE Automation 호환 타입만 포함하는 타입 라이브러리에 정의된 인터페이스 타입.
- 타입 라이브러리에 정의된 *Dispinterface* 타입.
- 타입 라이브러리 내에 정의된 사용자 지정 레코드 타입.
- *IFont*, *IStrings*, *IPicture*. Helper 객체는 인스턴스화되고 다음과 같이 매핑되어야 합니다.
 - *IFont*는 *TFont*로 매핑되어야 합니다.
 - *IStrings*는 *TStrings*로 매핑되어야 합니다.
 - *IPicture*는 *TPicture*로 매핑되어야 합니다.

ActiveX 컨트롤과 ActiveForm 마법사는 필요할 때 자동적으로 이러한 helper 객체를 만듭니다. helper 객체를 사용하려면 전역 루틴인 *GetOleFont*, *GetOleStrings*, *GetOlePicture*를 각각 호출합니다.

자동 마샬링을 위한 타입 제한 사항

인터페이스가 자동 마샬링 (Automation 마샬링 또는 타입 라이브러리 마샬링이라고도 함)을 지원하는 경우 다음과 같은 제한 사항이 적용됩니다. Type Library 에디터를 사용하여 객체를 편집할 때 에디터는 다음과 같은 제한 사항을 철저히 적용합니다.

- 타입이 크로스 플랫폼 통신에 적합해야 합니다. 예를 들어, 데이터 구조(다른 속성 객체 구현 이외에), 부호 없는 인수, *AnsiString* 등을 사용할 수 없습니다.
- 문자열 데이터 타입은 와이드 문자열(BSTR)로 전송되어야 합니다. *PChar*와 *AnsiString*은 안전하게 마샬링될 수 없습니다.
- 이중 인터페이스의 모든 멤버는 함수의 결과 값으로 *HRESULT*를 전달합니다. 메소드가 *safecall* 호출 규칙을 사용하여 선언되었으면 이 조건은 출력 매개변수로 변환된 선언된 반환 타입으로 자동적으로 부여됩니다.
- 다른 값을 반환해야 할 이중 인터페이스의 멤버는 함수의 값을 반환하는 출력 매개변수를 나타내는 이러한 매개변수를 **var** 또는 **out**으로 지정해야 합니다.

참고 Automation 타입 제한 사항을 무시하는 하나의 방법은 개별적인 *IDispatch* 인터페이스와 사용자 지정 인터페이스를 구현하는 것입니다. 그렇게 하면 전체 범위의 가능한 인수 타입을 사용할 수 있습니다. 즉, COM 클라이언트가 Automation 컨트롤러가 여전히 액세스할 수 있는 사용자 지정 인터페이스를 사용하는 옵션을 가지고 있다는 것을 의미합니다. 그렇지만 이 경우에는 수동으로 마샬링 코드를 구현해야 합니다.

사용자 지정 마샬링

일반적으로 COM 작업이 더 쉽다는 이유에서 out-of-process 및 원격 서버에서 자동 마샬링을 사용합니다. 하지만 마샬링 성능을 향상시킬 수 있다고 생각한다면 사용자 지정 마샬링을 제공할 수도 있습니다. 사용자 고유의 사용자 지정 마샬링을 구현할 때는 *IMarshal* 인터페이스를 지원해야 합니다. 이 방법에 대한 자세한 내용은 Microsoft 설명서를 참조하십시오.

COM 객체 등록

서버 객체를 in-process 또는 out-of-process 서버로 등록할 수 있습니다. 서버 타입에 대한 자세한 내용은 33-6 페이지의 "In-process, Out-of-process 및 원격 서버"를 참조하십시오.

참고 시스템에서 COM 객체를 제거하기 전에 등록을 해제해야 합니다.

In-process 서버 등록

In-process 서버(DLL 또는 OCX)를 등록하려면

- Run|Register ActiveX Server를 선택합니다.

In-process 서버를 등록 해제하려면

- Run|Unregister ActiveX Server를 선택합니다.

Out-of-process 서버 등록

Out-of-process 서버를 등록하려면

- **/regserver** 명령줄 옵션으로 서버를 실행합니다.

Run|Parameters 대화 상자로 명령줄 옵션을 설정할 수 있습니다.

해당 서버를 실행하여 등록할 수도 있습니다.

Out-of-process 서버를 등록 해제하려면

- **/unregserver** 명령줄 옵션으로 서버를 실행합니다.

대안으로 명령줄에서 **tregsvr** 명령을 사용하거나 운영 체제에서 regsvr32.exe를 실행할 수 있습니다.

참고 COM 서버가 COM+에서 사용하도록 만들어졌다면 등록하기 전에 COM+ 애플리케이션에 설치해야 합니다. (COM+ 애플리케이션에 객체를 설치하면 자동으로 등록이 됩니다.) COM+ 애플리케이션에 객체를 설치하는 방법에 대한 내용은 39-22 페이지의 "트랜잭션 객체 설치"를 참조하십시오.

애플리케이션 테스트 및 디버깅

다음과 같은 방법으로 COM 서버 애플리케이션을 테스트하고 디버깅합니다.

- 1 필요한 경우, Project|Options 대화 상자의 Compiler 탭에서 Debugging Information 을 선택합니다. 또한 Tools|Debugger Options 대화 상자에서 Integrated Debugging 을 선택합니다.
- 2 in-process 서버의 경우, Run|Parameters를 선택하고 Host Application 상자에서 Automation 컨트롤러의 이름을 입력하고 OK를 선택합니다.
- 3 Run|Run을 선택합니다.
- 4 Automation 서버에서 브레이크포인트를 설정합니다.
- 5 Automation 컨트롤러를 사용하여 Automation 서버와 상호 작용합니다.

Automation 서버는 브레이크포인트에 도달하면 멈춥니다.

참고 다른 방법으로 Automation 컨트롤러도 작성하고 있다면 COM cross-process 지원을 사용하여 in-process 서버로 디버깅할 수 있습니다. Tools|Debugger Options 대화 상자의 General 페이지를 사용하여 cross-process 지원을 사용합니다.

Active Server Page 생성

Microsoft Internet Information Server(IIS)를 웹 서버로 사용하고 있다면 ASP(Active Server Page)를 사용하여 동적 웹 기반 클라이언트-서버 애플리케이션을 만들 수 있습니다. Active Server Page는 서버가 웹 페이지를 로드할 때마다 호출되는 스크립트를 작성할 수 있게 해줍니다. 이 스크립트는 생성된 HTML 페이지에 포함된 정보를 얻기 위해서 차례로 Automation 객체를 호출할 수 있습니다. 예를 들어, 비트맵을 생성하거나 데이터베이스에 연결하는 서버와 같은 Delphi Automation 서버를 작성할 수 있고, 이 컨트롤을 사용하여 서버가 웹 페이지를 로드할 때마다 업데이트되는 데이터를 액세스할 수 있습니다.

클라이언트 측에서 ASP는 표준 HTML 문서처럼 동작하고 사용자에게 의해 웹 브라우저를 사용하여 플랫폼에서 보여질 수 있습니다.

ASP 애플리케이션은 Delphi의 웹 브로커 기술을 사용하여 작성하는 애플리케이션과 유사합니다. 웹 브로커 기술에 대한 자세한 내용은 27장 "인터넷 애플리케이션 생성"을 참조하십시오. 하지만 ASP는 비즈니스 룰 또는 복잡한 애플리케이션 로직의 구현으로부터 UI 디자인을 분리시키는 방법이 다릅니다.

- UI 디자인은 Active Server Page에서 관리됩니다. 이는 근본적으로는 HTML 문서이지만 Active Server 객체가 비즈니스 룰 또는 애플리케이션 로직을 반영하는 콘텐츠를 제공하도록 요청하는 내장 스크립트를 포함할 수 있습니다.
- 애플리케이션 로직은 Active Server Page에 간단한 메소드를 노출시키는 Active Server 객체에 의해 캡슐화되어 필요한 콘텐츠를 제공합니다.

참고 ASP가 UI 디자인을 애플리케이션 로직에서 분리시키는 이점을 제공하더라도 그 수행은 규모에 있어서 제한됩니다. 아주 많은 수의 클라이언트에 응답하는 웹 사이트의 경우, 웹 브로커 기술에 기반한 접근 방법을 사용하는 것이 좋습니다.

Active Server Page의 스크립트와 사용자가 포함시킨 활성 서버 페이지의 Automation 객체는 ASP intrinsics(현재 애플리케이션에 대한 정보를 제공하는 기본 제공 객체, 브라우저의 HTTP 메시지 등)를 사용할 수 있습니다.

이 장에서는 Delphi Active Server Object 마법사를 사용하여 Active Server Object 를 만드는 방법에 대해 설명합니다. 이 특별한 Automation 컨트롤은 Active Server Page에 의해 호출되어 콘텐츠를 제공할 수 있습니다.

Active Server Object를 만드는 단계는 다음과 같습니다.

- 애플리케이션에 대한 Active Server Object를 생성합니다.
- Active Server Object의 인터페이스를 정의합니다.
- Active Server Object를 등록합니다.
- 애플리케이션을 테스트하고 디버그합니다.

Active Server Object 생성

Active Server Object는 전체 ASP 애플리케이션에 대한 정보 및 브라우저와 통신하는 데 사용하는 HTTP 메시지에 대한 정보에 액세스할 수 있는 Automation 객체입니다. 이 객체는 (*TAutoObject*의 자손인) *TASPObject* 또는 *TASPMTSObject*의 자손이고 다른 애플리케이션(또는 Active Server 페이지의 스크립트)에서 사용하도록 자신을 노출시키는 Automation 프로토콜을 지원합니다. Active Server Object 마법사를 사용하여 Active Server Object를 만듭니다.

Active Server Object 프로젝트는 필요에 따라 실행 파일(exe) 또는 라이브러리(dll)가 될 수 있습니다. 하지만 out-of-process 서버 사용의 단점에 대해 알고 있어야 합니다. 이러한 단점은 37-7 페이지의 "In-process 또는 Out-of-process 서버용 ASP 생성"에서 설명됩니다.

다음과 같은 방법으로 Active Server Object 마법사를 나타냅니다.

- 1 File|New|Other를 선택합니다.
- 2 ActiveX 탭을 선택합니다.
- 3 Active Server Object 아이콘을 더블 클릭합니다.

마법사에서 새 Active Server Object를 명명하고 지원하고자 하는 인스턴스 및 스레드 모델을 지정합니다. 이러한 세부 사항은 사용자의 객체가 호출되는 방식에 영향을 줍니다. 모델에 따른(예를 들어, 스레드 충돌을 피하는) 구현을 작성해야 합니다. 인스턴스 및 스레드 모델은 다른 COM 객체에 대해 사용자가 만든 선택과 동일하게 적용됩니다. 자세한 내용은 36-5 페이지의 "COM 객체 인스턴스 타입" 및 36-6 페이지의 "스레드 모델 선택"을 참조하십시오.

Active Server Object의 고유한 점은 ASP 애플리케이션에 대한 정보 및 Active Server 페이지와 클라이언트 웹 브라우저 간에 전달하는 HTTP 메시지에 대한 정보를 액세스할 수 있다는 것입니다. ASP intrinsics를 사용하여 이러한 정보에 액세스할 수 있습니다. 마법사에서 다음과 같이 Active Server Type을 설정하여 사용자의 객체가 이에 액세스하는 방법을 지정할 수 있습니다.

- IIS 3 또는 IIS 4를 사용할 경우 Page Level Event Methods를 사용합니다. 이 모델에서 사용자의 객체는 Active Server 페이지가 로드되거나 언로드될 때 호출되는 *OnStartPage* 및 *OnEndPage* 메소드를 구현합니다. 사용자의 객체가 로드될 때

ASP intrinsics에 액세스하는 데 사용되는 *IScriptingContext* 인터페이스를 자동으로 얻습니다. 이 인터페이스는 기본 클래스 (*TASPObject*)에서 상속된 속성으로 나타납니다.

- IIS5 또는 이후 버전을 사용할 경우 Object Context 타입을 사용합니다. 이 모델에서 사용자의 객체는 ASP intrinsics에 액세스하는 데 사용하는 *IObjectContext* 인터페이스를 폐치합니다. 또한 이러한 인터페이스는 상속된 기본 클래스 (*TASPMTSObject*)에서 속성으로 나타납니다. 후자쪽 접근 방법의 한 가지 이점은 사용자의 객체가 *IObjectContext*를 통해 사용할 수 있는 모든 다른 서비스에 액세스할 수 있다는 것입니다. *IObjectContext* 인터페이스에 액세스하려면 다음과 같이 mtz 유닛에서 정의된 *GetObjectContext*를 호출합니다.

```
ObjectContext := GetObjectContext;
```

*IObjectContext*를 통해 사용할 수 있는 서비스에 대한 자세한 내용은 39장 "MTS 또는 COM+ 객체 생성"을 참조하십시오.

새로운 Active Server Object를 호스트하기 위해서 마법사에서 간단한 ASP 페이지를 생성하게 할 수 있습니다. 생성된 페이지는 ProgID에 기초하여 Active Server Object를 만드는 VBScript로 작성된 최소한의 스크립트를 제공하고 메소드를 호출할 수 있는 곳을 지시합니다. 이 스크립트는 **Server.CreateObject**를 호출하여 Active Server Object를 시작합니다.

참고 생성된 테스트 스크립트가 VBScript를 사용하더라도 Active Server Page는 Jscript를 사용하여 작성될 수도 있습니다.

마법사를 종료할 때 Active Server Object의 정의를 포함하는 현재 프로젝트에 새로운 유닛이 추가됩니다. 또한 마법사는 타입 라이브러리를 추가하고 Type Library 에디터를 엽니다. 36-9 페이지의 "COM 객체의 인터페이스 정의"에 설명된 대로 타입 라이브러리를 통해 인터페이스의 속성과 메소드를 노출시킬 수 있습니다. 사용자 객체의 속성과 메소드의 구현을 작성할 때 아래에서 설명하는 ASP intrinsics를 사용하여 ASP 애플리케이션에 대한 정보 및 브라우저와 통신하는 데 사용하는 HTTP 메시지에 대한 정보를 얻을 수 있습니다.

다른 Automation 객체처럼 Active Server Object는 *Vtable*을 통한 우선 바인딩(컴파일 타임 시) 및 *IDispatch* 인터페이스를 통한 지연 바인딩(런타임 시) 모두를 지원하는 **이중 인터페이스**를 구현합니다. 이중 인터페이스에 대한 내용은 36-13 페이지의 "이중 인터페이스"를 참조하십시오.

ASP intrinsics 사용

ASP intrinsics는 Active Server Page에서 실행하는 객체에 대해 ASP가 제공하는 COM 객체의 집합입니다. 이를 통해 Active Server Object는 동일한 ASP 애플리케이션에 속하는 Active Server Objects 간에 공유된 정보를 저장하는 장소뿐만 아니라 애플리케이션과 웹 브라우저 간에 전달하는 메시지를 반영하는 정보에 액세스할 수 있습니다.

이러한 객체가 쉽게 액세스하도록 하기 위해 Active Server Object에 대한 기본 클래스는 객체들을 속성으로 나타냅니다. 이러한 객체에 대한 완전한 이해를 위해서는 Microsoft 설명서를 참조하십시오. 하지만 다음의 항목에서 간략한 개요를 제공합니다.

애플리케이션

애플리케이션 객체는 *IApplicationObject* 인터페이스를 통해 액세스됩니다. 애플리케이션 객체는 가상의 디렉토리 및 그 하위 디렉토리에 있는 모든 .asp 파일들의 집합으로 정의된 전체 ASP 애플리케이션을 나타냅니다. 애플리케이션 객체는 여러 클라이언트에 의해 공유될 수 있으므로 스레드 충돌을 방지하는 데 사용해야 하는 잠금 지원을 포함합니다.

*ApplicationObject*에는 다음과 같은 내용이 포함됩니다.

표 37.1 IApplicationObject 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
Contents 속성	스크립트 명령을 사용하여 애플리케이션에 추가된 모든 객체를 나열합니다. 이 인터페이스에는 목록의 객체 중 하나 또는 모두를 삭제하는 데 사용할 수 있는 두 가지 메소드, <i>Remove</i> 및 <i>RemoveAll</i> 이 있습니다.
StaticObjects 속성	<OBJECT> 태그를 가지고 애플리케이션에 추가된 모든 객체를 나열합니다.
Lock 메소드	Unlock 메소드를 호출할 때까지 다른 클라이언트가 Application 객체를 잠글 수 없도록 합니다. 모든 클라이언트는 속성처럼 공유된 메모리에 액세스하기 전에 Lock 메소드를 호출해야 합니다.
Unlock 메소드	Lock 메소드를 사용하여 설정되었던 잠금을 해제합니다.
Application_OnEnd 이벤트	Session_OnEnd 이벤트 후 애플리케이션이 중지될 때 발생합니다. 사용 가능한 유일한 intrinsics는 애플리케이션과 서버입니다. 이벤트 핸들러는 VBScript 또는 JScript로 작성해야 합니다.
Application_OnStart 이벤트	새로운 세션이 만들어지기 전 (Session_OnStart 전) 에 발생합니다. 사용 가능한 유일한 intrinsics는 애플리케이션과 서버입니다. 이벤트 핸들러는 VBScript 또는 JScript로 작성해야 합니다.

Request

Request 객체는 *IRequest* 인터페이스를 통해 액세스됩니다. Active Server Page를 여는 HTTP 요청 메시지에 대한 정보를 제공합니다.

*IRequest*에는 다음과 같은 내용이 포함됩니다.

표 37.2 IRequest 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
ClientCertificate 속성	HTTP 메시지와 함께 보내진 클라이언트 인증의 모든 필드 값을 나타냅니다.
Cookies 속성	HTTP 메시지의 모든 Cookie 헤더 값을 나타냅니다.
Form 속성	HTTP 몸체의 폼 요소 값을 나타냅니다. 이는 이름으로 액세스될 수 있습니다.
QueryString 속성	HTTP 헤더의 쿼리 문자열에 있는 모든 변수 값을 나타냅니다.
ServerVariables 속성	다양한 환경 변수 값들을 나타냅니다. 이러한 변수들은 대부분의 일반 HTTP 헤더 변수를 나타냅니다.

표 37.2 IRequest 인터페이스 멤버 (계속)

속성, 메소드 또는 이벤트 의미	
TotalBytes 속성	요청 몸체의 바이트 수를 나타냅니다. 이는 BinaryRead 메소드에 의해 반환되는 바이트 수에 대한 상위 제한입니다.
BinaryRead 메소드	Post 메시지의 콘텐츠를 가져옵니다. 메소드를 호출하여 읽을 수 있는 최대 바이트 수를 지정합니다. 결과 콘텐츠는 가변 배열의 바이트로 반환됩니다. BinaryRead 호출 후 Form 속성을 사용할 수 없습니다.

Response

Request 객체는 *IResponse* 인터페이스를 통해 액세스됩니다. 이를 통해 클라이언트 브라우저에 반환되는 HTTP 응답 메시지에 대한 정보를 지정할 수 있습니다.

*IResponse*에는 다음과 같은 내용이 포함됩니다.

표 37.3 IResponse 인터페이스 멤버

속성, 메소드 또는 이벤트 의미	
Cookies 속성	HTTP 메시지의 모든 Cookie 헤더 값을 결정합니다.
Buffer 속성	페이지 출력이 버퍼링될 때 페이지 출력이 버퍼링되는지 여부를 나타냅니다. 현재 페이지의 서버 스크립트가 처리될 때까지 서버는 클라이언트에 응답을 보내지 않습니다.
CacheControl 속성	프록시 서버가 응답으로 출력을 캐시로 저장할 수 있는지 여부를 결정합니다.
Charset 속성	문자 집합의 이름을 콘텐츠 타입 헤더에 추가합니다.
ContentType 속성	응답 메시지 몸체의 HTTP 콘텐츠 타입을 지정합니다.
Expires 속성	종료되기 전에 응답이 브라우저에 의해 캐시로 저장될 수 있는 기간을 지정합니다.
ExpiresAbsolute 속성	응답이 종료될 때 날짜와 시간을 지정합니다.
IsClientConnected 속성	클라이언트가 서버로부터 연결 해제되었는지 여부를 나타냅니다.
Pics 속성	응답 헤더의 픽스(pics)-레이블 필드에 대한 값을 설정합니다.
Status 속성	응답의 상태를 나타냅니다. 이는 HTTP 상태 헤더의 값입니다.
AddHeader 메소드	지정된 이름과 값을 가진 HTTP 헤더를 추가합니다.
AppendToLog 메소드	이 요청에 대한 웹 서버 로그 항목의 끝에 문자열을 추가합니다.
BinaryWrite 메소드	응답 메시지의 몸체에 해석되지 않은 원래 정보를 씁니다.
Clear 메소드	버퍼링된 HTML 출력을 지웁니다.
End 메소드	.asp 파일의 처리를 중지하고 현재 결과를 반환합니다.
Flush 메소드	버퍼링된 출력을 즉시 보냅니다.
Redirect 메소드	클라이언트 브라우저를 다른 URL로 재지정하는 리디렉션 응답 메시지를 보냅니다.
Write 메소드	현재 HTTP 출력에 대한 변수를 문자열로 작성합니다.

Session

Session 객체는 *ISessionObject* 인터페이스를 통해 액세스됩니다. 이를 통해 ASP 애플리케이션과 클라이언트의 상호 작용 기간 동안 지속되는 변수들을 저장할 수 있습니다. 즉, 클라이언트가 ASP 애플리케이션 내의 페이지 사이에서 이동할 때는 이러한 변수들이 해제되어 있지 않지만 클라이언트가 애플리케이션을 종료할 때는 해제되어 있습니다.

*ISessionObject*에는 다음과 같은 내용이 포함됩니다.

표 37.4 ISessionObject 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
Contents 속성	<OBJECT> 태그를 사용하여 세션에 추가된 모든 객체를 나열합니다. 이름으로 목록의 변수들에 액세스하거나 Contents 객체의 <i>Remove</i> 또는 <i>RemoveAll</i> 메소드를 호출하여 값을 삭제할 수 있습니다.
StaticObjects 속성	<OBJECT> 태그를 가지고 세션에 추가된 모든 객체를 나열합니다.
CodePage 속성	기호 매핑에 사용하는 코드 페이지를 지정합니다. 각기 다른 로케일은 각기 다른 코드를 사용할 수도 있습니다.
LCID 속성	문자열 콘텐츠를 해석하는 데 사용할 로케일 식별자를 지정합니다.
SessionID 속성	현재 클라이언트에 대한 세션 식별자를 나타냅니다.
Timeout 속성	애플리케이션이 종료될 때까지 클라이언트로부터의 요청 (또는 새로 고침) 없이 세션이 지속되는 시간을 분 단위로 지정합니다.
Abandon 메소드	세션을 소멸시키고 해당 리소스를 해제합니다.
Session_OnEnd 이벤트	세션이 버려지거나 시간이 경과될 때 발생합니다. 사용 가능한 유일한 intrinsics는 Application, Server 및 Session입니다. 이벤트 핸들러는 VBScript 또는 JScript로 작성해야 합니다.
Session_OnStart 이벤트	서버가 새로운 세션을 생성할 때 (Application_OnStart 후 Active Server Page의 스크립트를 실행하기 전에) 발생합니다. 모든 intrinsics는 사용 가능합니다. 이벤트 핸들러는 VBScript 또는 JScript로 작성해야 합니다.

Server

Server 객체는 *IServer* 인터페이스를 통해 액세스됩니다. ASP 애플리케이션 작성을 위한 다양한 유틸리티를 제공합니다.

*IServer*에는 다음과 같은 내용이 포함됩니다.

표 37.5 IServer 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
ScriptTimeout 속성	Session 객체의 Timeout 속성과 동일합니다.
CreateObject 메소드	지정된 Active Server Object를 인스턴스화합니다.
Execute 메소드	지정된 .asp 파일의 스크립트를 실행합니다.
GetLastError 메소드	오류 조건을 설명하는 ASPError 객체를 반환합니다.

표 37.5 IServer 인터페이스 멤버 (계속)

속성, 메소드 또는 이벤트	의미
HTMLEncode 메소드	HTML 헤더에서의 사용을 위해 문자열을 인코딩하여 적절한 기호 상수로 예약 문자를 바꿉니다.
MapPath 메소드	지정된 가상 경로 (현재 서버의 절대 경로 또는 현재 페이지에 상대적인 경로)를 실제 경로로 매핑합니다.
Transfer 메소드	모든 현재 상태 정보를 처리하기 위해 다른 Active Server Page 로 보냅니다.
URLEncode 메소드	이스케이프 문자를 포함한 URL 인코딩 규칙을 지정된 문자열에 적용합니다.

In-process 또는 Out-of-process 서버용 ASP 생성

ASP 페이지의 **Server.CreateObject**를 사용하여 사용자의 요구에 따라 In-process 또는 Out-of-process 서버를 시작할 수 있습니다. 하지만 In-process 서버의 시작이 더 일반적입니다.

대부분의 In-process 서버와 달리 In-process 서버의 Active Server Object는 클라이언트의 프로세스 공간에서 실행되지 않습니다. 그 대신 IIS 프로세스 공간에서 실행됩니다. 이는 (예를 들어, ActiveX 객체를 사용할 때 클라이언트가 애플리케이션을 다운로드하는 것처럼) 클라이언트가 사용자의 애플리케이션을 다운로드할 필요가 없다는 것을 의미합니다. In-process 컴포넌트 DLL은 out-of-process 서버보다 빠르고 안전해서 서버측 사용에 더 적합합니다.

Out-of-process 서버는 덜 안전하기 때문에 out-of-process 실행 파일을 허용하지 않도록 IIS를 구성하는 것이 일반적입니다. 이 경우에 Active Server Object에 대한 Out-of-process 서버를 생성하면 다음과 유사한 오류를 나타냅니다.

```
Server object error 'ASP 0196'
Cannot launch out of process component
/path/outofprocess_exe.asp, line 11
```

또한 out-of-process 컴포넌트는 경우에 따라 각 객체 인스턴스에 대한 개별 서버 프로세스를 생성하므로 CGI 애플리케이션에 비해 속도가 더 느립니다. 컴포넌트 DLL과 더불어 규모는 상관하지 않습니다.

성능과 확장성이 사용자측에 대해 우선하는 경우에는 In-process 서버가 더 좋습니다. 하지만 낮은 트래픽에 대한 중재를 받는 인트라넷측은 사이트의 전체적인 성능에 나쁜 영향을 미치지 않으면서 Out-of process 컴포넌트를 사용할 수도 있습니다.

In-process 와 Out-of-process 서버에 대한 일반적인 내용은 33-6 페이지의 "In-process, Out-of-process 및 원격 서버"를 참조하십시오.

Active Server Object 등록

Active Server Page를 In-process 또는 Out-of-process 서버로서 등록할 수 있습니다. 하지만 In-process 서버가 더 일반적입니다.

참고 사용자의 시스템에서 Active Server Page 객체를 제거하려면 먼저 등록을 해제하고 Windows 레지스트리에서 항목을 삭제합니다.

In-process 서버 등록

In-process 서버(DLL 또는 OCX)를 등록하려면

- Run|Register ActiveX Server를 선택합니다.

In-process 서버를 등록 해제하려면

- Run|Unregister ActiveX Server를 선택합니다.

Out-of-process 서버 등록

Out-of-process 서버를 등록하려면

- /regserver 명령줄 옵션으로 서버를 실행합니다. (Run|Parameters 대화 상자로 명령줄 옵션을 설정할 수 있습니다.)

해당 서버를 실행하여 등록할 수도 있습니다.

Out-of-process 서버를 등록 해제하려면

- /unregserver 명령줄 옵션으로 서버를 실행합니다.

Active Server Page 애플리케이션의 테스트 및 디버깅

Active Server Object와 같은 In-process 서버를 디버깅하는 것은 DLL을 디버깅하는 것과 매우 유사합니다. DLL을 로드하는 호스트 애플리케이션을 선택하고 평소처럼 디버그합니다. Active Server Object를 테스트하고 디버그하려면 다음과 같이 합니다.

- 1 필요한 경우, Project|Options 대화 상자의 Compiler 탭에서 Debugging Information을 선택합니다. 또한 Tools|Debugger Options 대화 상자에서 Integrated Debugging을 선택합니다.
- 2 Run|Parameters를 선택하고 Host Application 상자에 사용자 웹 서버의 이름을 입력한 후 OK를 선택합니다.
- 3 Run|Run을 선택합니다.
- 4 Active Server Object 구현의 브레이크포인트를 설정합니다.
- 5 웹 브라우저를 사용하여 Active Server Page와 상호 작용합니다. 브레이크포인트에 도달하면 디버거는 정지됩니다.

38

ActiveX 컨트롤 생성

ActiveX 컨트롤은 C++Builder, Delphi, Visual Basic, Internet Explorer, (플러그인을 사용하는) Netscape Navigator와 같은 ActiveX 컨트롤을 지원하는 모든 호스트 애플리케이션의 기능을 통합하고 확장하는 소프트웨어 컴포넌트입니다. ActiveX 컨트롤은 이러한 통합을 허용하는 특정 인터페이스 집합을 구현합니다.

예를 들어 Delphi에는 차트, 스프레드시트, 그래픽 컨트롤을 비롯한 여러 ActiveX 컨트롤이 있습니다. IDE의 컴포넌트 팔레트에 이러한 컨트롤을 추가한 다음 표준 VCL 컴포넌트처럼 사용하여 폼에 가져다 놓고 Object Inspector를 사용하여 속성을 설정할 수 있습니다.

또한 ActiveX 컨트롤은 웹에 배포되어 HTML 문서에서 참조할 수 있으며 ActiveX 활성 웹 브라우저로 볼 수 있습니다.

Delphi는 두 가지 타입의 ActiveX 컨트롤을 생성할 수 있게 하는 마법사를 제공합니다.

- **VCL 클래스를 래핑하는 ActiveX 컨트롤.** VCL 클래스를 래핑하여 기존의 컴포넌트를 ActiveX 컨트롤로 변환하거나 새로운 컴포넌트를 만들어 부분적으로 테스트한 다음 ActiveX 컨트롤로 변환할 수 있습니다. 일반적으로 ActiveX 컨트롤은 더 큰 호스트 애플리케이션에 포함됩니다.
- **Active 폼.** Active 폼에서는 폼 디자이너를 사용하여 대화 상자 또는 완전한 애플리케이션처럼 동작하는 더 정교한 컨트롤을 만들 수 있습니다. 일반적인 Delphi 애플리케이션을 개발한 것과 같은 방법으로 Active 폼을 개발합니다. 일반적으로 Active 폼은 웹에 배포됩니다.

이 장에서는 Delphi 환경에서 ActiveX 컨트롤을 만드는 방법에 관한 개요를 제공합니다. 마법사를 사용하지 않고 ActiveX 컨트롤을 작성하는 완전한 구현 세부 사항을 제공하지는 않습니다. 해당 정보는 Microsoft Developer의 Network(MSDN) 문서를 참조하거나 Microsoft 웹 사이트에서 ActiveX 정보를 검색하십시오.

ActiveX 컨트롤 생성에 대한 개요

Delphi를 사용하여 ActiveX 컨트롤을 만드는 것은 일반적인 컨트롤 또는 폼을 만드는 것과 매우 유사합니다. 이는 객체의 인터페이스를 정의한 다음 구현을 완료하는 다른 COM 객체 생성과는 현저히 다릅니다. Active 폼이 아닌 ActiveX 컨트롤을 생성하려면 순서를 반대로 하여 VCL 컨트롤을 구현한 다음 일단 컨트롤이 작성되면 인터페이스 및 타입 라이브러리를 생성합니다. Active 폼을 생성할 때 인터페이스 및 타입 라이브러리가 사용자의 폼과 동시에 생성되고 폼 디자이너를 사용하여 폼을 구현합니다.

완료된 ActiveX 컨트롤은 기본 구현, VCL 컨트롤을 래핑하는 COM 객체와 COM 객체의 속성, 메소드, 이벤트를 나열하는 타입 라이브러리 등을 제공하는 VCL 컨트롤로 구성됩니다.

Active 폼이 아닌 ActiveX 컨트롤을 생성하려면 다음 단계를 수행합니다.

- 1 ActiveX 컨트롤의 기본을 만드는 사용자 지정 VCL 컨트롤을 디자인하고 생성합니다.
- 2 ActiveX 컨트롤 마법사를 사용하여 1단계에서 생성한 VCL 컨트롤에서 ActiveX 컨트롤을 생성합니다.
- 3 ActiveX 속성 페이지 마법사를 사용하여 컨트롤에 대해 하나 이상의 속성 페이지를 생성합니다(옵션).
- 4 속성 페이지를 ActiveX 컨트롤에 연결합니다(옵션).
- 5 컨트롤을 등록합니다.
- 6 모든 가능성 있는 대상 애플리케이션으로 컨트롤을 테스트합니다.
- 7 웹에 ActiveX 컨트롤을 배포합니다(옵션).

새로운 Active 폼을 생성하려면 다음 단계를 실행합니다.

- 1 ActiveForm 마법사를 사용하여 IDE에 빈 폼으로 나타나는 Active 폼과 해당 폼에 대해 연결된 ActiveX 래퍼를 생성합니다.
- 2 폼 디자이너를 사용하여 Active 폼에 컴포넌트를 추가하고, 폼 디자이너를 사용하여 일반 폼을 생성하고 구현하는 것과 같은 방법으로 동작을 구현합니다.
- 3 Active 폼에 속성 페이지를 주어 이를 등록해서 웹에 배포하려면 위의 3-7단계를 따릅니다.

ActiveX 컨트롤의 요소

ActiveX 컨트롤에는 각각이 특정 기능을 수행하는 여러 요소를 포함합니다. 요소에는 VCL 컨트롤, 속성, 메소드, 이벤트를 노출하는 해당 COM 객체 래퍼 및 하나 이상의 연결된 타입 라이브러리 등이 있습니다.

VCL 컨트롤

Delphi에 있는 ActiveX 컨트롤의 기본 구현은 VCL 컨트롤입니다. ActiveX 컨트롤을 생성할 때 ActiveX 컨트롤을 만들 VCL 컨트롤을 먼저 디자인하거나 선택해야 합니다.

기본 VCL 컨트롤은 호스트 애플리케이션에 의해 만들어질 수 있는 창이 있어야 하기 때문에 반드시 *TWinControl*의 자손이어야 합니다. Active 폼을 생성할 때 이 객체는 *TActiveForm*의 자손입니다.

참고 ActiveX 컨트롤 마법사는 선택하여 ActiveX 컨트롤을 만들 수 있는 사용 가능한 *TWinControl* 자손을 나열합니다. 하지만 이 목록은 모든 *TWinControl* 자손을 포함하지는 않습니다. *THeaderControl*과 같은 일부 컨트롤은 *RegisterNonActiveX* 프로시저를 사용하여 ActiveX 비호환으로 등록되어 목록에 나타나지 않습니다.

ActiveX 래퍼

실제 COM 객체는 VCL 컨트롤에 대한 ActiveX 래퍼 객체입니다. Active 폼의 경우 이 클래스는 항상 *TActiveFormControl*입니다. 다른 ActiveX 컨트롤의 경우 *TVCLClassX* 형식의 이름을 가지며 여기서 *TVCLClass*는 VCL 컨트롤 클래스의 이름입니다. 예를 들어, *TButton*에 대한 ActiveX 래퍼는 *TButtonX*라는 이름으로 지정해야 합니다.

래퍼 클래스는 ActiveX 인터페이스에 대한 지원을 제공하는 *TActiveXControl*의 자손입니다. ActiveX 래퍼는 이 지원을 상속하며 이를 사용하면 Windows 메시지를 VCL 컨트롤에 전달하고 호스트 애플리케이션에서 해당 창을 만들 수 있습니다.

ActiveX 래퍼는 기본 인터페이스를 통해 VCL 컨트롤의 속성 및 메소드를 클라이언트에 노출합니다. 마법사는 래퍼 클래스의 속성 및 메소드의 대부분을 자동으로 구현하여 메소드 호출을 기본 VCL 컨트롤에 위임합니다. 또한 마법사는 클라이언트에서 VCL 컨트롤의 이벤트를 발생시키는 메소드를 래퍼 클래스에 제공하고 이 메소드를 VCL 컨트롤의 이벤트 핸들러로 할당합니다.

타입 라이브러리

ActiveX 컨트롤 마법사는 래퍼 클래스에 대한 타입 정의, 기본 인터페이스 및 필요한 모든 타입 정의가 포함된 타입 라이브러리를 자동으로 생성합니다. 이러한 타입 정보는 컨트롤이 호스트 애플리케이션에 서비스를 통지하는 방법을 제공합니다. Type Library 에디터를 사용하여 이 정보를 보고 편집할 수 있습니다. 이러한 정보는 확장자가 .TLB인 별도의 바이너리 타입 라이브러리 파일에 저장되지만, 또한 ActiveX 컨트롤 DLL에 리소스로서 자동으로 컴파일됩니다.

속성 페이지

옵션으로 사용자의 ActiveX 컨트롤에 속성 페이지를 줄 수 있습니다. 속성 페이지를 통해 호스트(클라이언트) 애플리케이션의 사용자는 해당 컨트롤의 속성을 보고 편집할 수 있습니다. 한 페이지에서 여러 속성을 그룹화하거나, 한 페이지를 사용하여 속성에 대해 대화 상자와 같은 인터페이스를 제공할 수 있습니다. 속성 페이지 생성 방법에 관한 내용은 38-12 페이지의 "ActiveX 컨트롤에 대한 속성 페이지 생성"을 참조하십시오.

ActiveX 컨트롤 디자인

ActiveX 컨트롤을 디자인할 때 사용자 지정 VCL 컨트롤을 생성하는 것부터 시작합니다. 이것은 ActiveX 컨트롤의 기초를 형성합니다. 사용자 지정 컨트롤 생성에 대한 내용은 5부 "사용자 지정 컴포넌트 생성"을 참조하십시오.

VCL 컨트롤을 디자인할 때 다른 애플리케이션에 포함된다는 점을 기억해야 합니다. 컨트롤 자체가 애플리케이션이 아니기 때문입니다. 이러한 이유로 정교한 대화 상자 또는 다른 주요 사용자 인터페이스 컴포넌트를 사용하지 않을 수도 있습니다. 일반적으로는 내부에서 작동하고 메인 애플리케이션의 규칙을 따르는 간단한 컨트롤을 만드는 것이 목적입니다.

또한 ActiveX 컨트롤의 인터페이스는 *IDispatch*를 지원해야 하기 때문에 객체를 클라이언트에 노출시키는 모든 속성 및 메소드에 대한 타입은 Automation 호환이어야 합니다. 마법사는 Automation 호환이 아닌 매개변수를 갖는 랩퍼 클래스의 인터페이스에는 메소드를 추가하지 않습니다. Automation 호환 타입의 목록을 보려면 34-11 페이지의 "유효한 타입"을 참조하십시오.

마법사는 COM 랩퍼 클래스 사용 시 요구되는 필요한 ActiveX 인터페이스를 모두 구현합니다. 또한 랩퍼 클래스의 기본 인터페이스를 통해 모든 Automation 호환 속성, 메소드, 이벤트를 표면화합니다. 일단 마법사가 COM 랩퍼 클래스 및 해당 인터페이스를 생성했다면 Type Library 에디터를 사용하여 추가 인터페이스 구현을 통해 기본 인터페이스를 수정하거나 랩퍼 클래스를 증가시킬 수 있습니다.

VCL 컨트롤로부터 ActiveX 컨트롤 생성

VCL 컨트롤로부터 ActiveX 컨트롤을 생성하려면 ActiveX 컨트롤 마법사를 사용합니다. VCL 컨트롤의 속성, 메소드, 이벤트는 ActiveX 컨트롤의 속성, 메소드, 이벤트가 됩니다.

ActiveX 컨트롤 마법사를 사용하기 전에 어떠한 VCL 컨트롤로부터 생성된 ActiveX 컨트롤의 기본 구현을 제공할지 결정해야 합니다.

다음과 같은 방법으로 ActiveX 컨트롤 마법사를 불러 옵니다.

- 1 File|New를 선택하여 New Items 대화 상자를 엽니다.
- 2 ActiveX 탭을 선택합니다.
- 3 ActiveX Control 아이콘을 더블 클릭합니다.

마법사에서 새로운 ActiveX 컨트롤이 래핑할 VCL 컨트롤의 이름을 선택합니다. 대화 상자는 *RegisterNonActiveX* 프로시저를 사용하여 ActiveX 비호환으로 등록되지 않은 *TWinControl*의 자손인 모든 사용 가능한 컨트롤을 나열합니다.

팁 드롭다운 목록에서 원하는 컨트롤을 볼 수 없으면 IDE에 설치했는지 또는 프로젝트에 해당 유닛을 추가했는지 확인합니다.

일단 VCL 컨트롤을 선택했다면 마법사는 CoClass의 이름, ActiveX 랩퍼의 구현 유닛, ActiveX 라이브러리 프로젝트를 자동으로 생성합니다. (현재 ActiveX 라이브러리 프로

젝트가 열려 있고 COM+ 이벤트 객체가 포함되어 있지 않으면 현재 프로젝트가 자동으로 사용됩니다.) ActiveX 라이브러리 프로젝트가 열려 있지 않아서 프로젝트 이름을 편집할 수 없는 경우에는 마법사에서 이를 변경할 수 있습니다.

마법사는 항상 스투드 모델로 Apartment를 지정합니다. 일반적으로 ActiveX 프로젝트가 단일 컨트롤만 포함할 경우에는 문제가 되지 않습니다. 하지만 프로젝트에 객체를 추가하면 스투드 지원을 제공해야 합니다.

또한 마법사를 통해 ActiveX 컨트롤에 다음과 같은 다양한 옵션을 구성할 수 있습니다.

- **라이선스 사용 가능:** 컨트롤의 사용자가 컨트롤에 대한 라이선스 키를 가지고 있지 않으면 디자인 목적을 위해서나 런타임 시에 이를 열 수 없도록 컨트롤에 라이선싱을 허용합니다.
- **버전 정보 포함:** ActiveX 컨트롤에 저작권 또는 파일 설명과 같은 버전 정보를 포함시킬 수 있습니다. 이 정보는 브라우저에서 볼 수 있습니다. Visual Basic 4.0 같은 일부 호스트 클라이언트는 버전 정보를 필요로 하거나 ActiveX 컨트롤을 호스트하지 않습니다. Project|Options를 선택하고 Version Info 페이지를 선택하여 버전 정보를 지정합니다.
- **About 상자 포함:** 컨트롤에 대한 About 상자를 구현하는 별도의 폼을 생성하도록 마법사에 지시할 수 있습니다. 호스트 애플리케이션의 사용자는 개발 환경에서 이 About 상자를 표시할 수 있습니다. 기본적으로 About 상자는 ActiveX 컨트롤의 이름, 이미지, 저작권 정보 및 OK 버튼 등을 포함합니다. 마법사가 프로젝트에 추가한 이 기본 폼을 수정할 수 있습니다.

마법사를 종료하면 다음과 같은 내용이 생성됩니다.

- ActiveX 컨트롤을 시작하는 데 필요한 코드가 들어 있는 ActiveX 라이브러리 프로젝트 파일. 일반적으로 이 파일은 변경하지 않습니다.
- 정의하는 타입 라이브러리, 컨트롤의 CoClass, 클라이언트에 노출되는 인터페이스 및 이 모든 것에 필요한 타입 정의. 타입 라이브러리에 대한 자세한 내용은 34 장 "Type Library 작업"을 참조하십시오.
- ActiveX 컨트롤을 정의하고 구현하는 ActiveX 구현 유닛은 *TActiveXControl*의 자손입니다. 이 ActiveX 컨트롤은 사용자가 추가 작업을 할 필요가 없는 완전한 기능을 갖춘 구현입니다. 하지만 ActiveX 컨트롤이 클라이언트에 노출하는 속성, 메소드, 이벤트를 사용자 지정하려는 경우 이 클래스를 수정할 수 있습니다.
- 요청한 About 상자 폼 및 유닛.
- 라이선싱을 허용한 .LIC 파일.

VCL 폼을 기반으로 ActiveX 컨트롤 생성

다른 ActiveX 컨트롤과 달리 Active 폼은 먼저 디자인하고 나서 ActiveX 랩퍼 클래스에 의해 래핑되지 않습니다. 그 대신 ActiveForm 마법사는 마법사가 폼 디자이너에 남겨둘 때 나중에 디자인하는 빈 폼을 생성합니다.

ActiveForm이 웹에 배포되면 Delphi는 HTML 페이지를 생성하여 ActiveForm에 대한 참조를 포함시키고 페이지에 해당 위치를 지정합니다. 그러면 ActiveForm은 표시될 수 있고 웹 브라우저에서 실행할 수 있습니다. 브라우저에서 폼은 독립형 Delphi 폼처럼 동작합니다. 폼은 사용자 지정 VCL 컨트롤을 비롯하여 VCL 컴포넌트 또는 ActiveX 컨트롤을 포함할 수 있습니다.

다음과 같은 방법으로 ActiveForm 마법사를 시작합니다.

- 1 File|New를 선택하여 New Items 대화 상자를 엽니다.
- 2 ActiveX 탭을 선택합니다.
- 3 ActiveForm 아이콘을 더블 클릭합니다.

래핑할 VCL 클래스의 이름을 지정할 수 없다는 것만 제외하면 Active Form 마법사는 ActiveX 컨트롤 마법사와 같습니다. 이는 Active 폼이 항상 *TActiveForm*을 기반으로 하기 때문입니다.

ActiveX 컨트롤 마법사에서처럼 CoClass, 구현 유닛, ActiveX 라이브러리 프로젝트 등의 기본 이름을 변경할 수 있습니다. 마찬가지로 이 마법사를 통해 Active Form이 라이선스를 필요로 하는지 여부, 버전 정보를 포함해야 하는지 여부, About 상자 폼이 필요한지 여부 등을 나타낼 수 있습니다.

마법사를 종료하면 다음이 생성됩니다.

- ActiveX 컨트롤을 시작하는 데 필요한 코드가 들어 있는 ActiveX 라이브러리 프로젝트 파일. 일반적으로 이 파일은 변경하지 않습니다.
- 정의하는 타입 라이브러리, 컨트롤의 CoClass, 클라이언트에 노출되는 인터페이스 및 이 모든 것에 필요한 타입 정의. 타입 라이브러리에 대한 자세한 내용은 34 장 "Type Library 작업"을 참조하십시오.
- *TActiveForm*의 자손인 폼. 이 폼은 폼 디자이너에 나타나며 여기서 클라이언트에 나타나는 Active Form을 비주얼하게 디자인할 수 있습니다. 해당 구현은 생성된 구현 유닛에 나타납니다. 구현 유닛의 초기화 섹션에서 이 폼에 대한 ActiveX 래퍼로 *TActiveFormControl*을 설정하여 클래스 팩토리를 생성합니다.
- 요청한 About 상자 폼 및 유닛.
- 라이선스를 허용한 .LIC 파일.

여기서 컨트롤을 추가할 수 있고 원하는 대로 폼을 디자인할 수 있습니다.

ActiveForm 프로젝트를 디자인하고 확장자가 OCX인 ActiveX 라이브러리로 컴파일한 후 프로젝트를 웹 서버에 배포할 수 있으며 Delphi는 ActiveForm에 대한 참조를 갖는 테스트 HTML 페이지를 생성합니다.

ActiveX 컨트롤 라이선싱

ActiveX 컨트롤 라이선싱은 디자인 타임 시 라이선스 키를 제공하는 것과 런타임 시 생성된 컨트롤에 대해 동적으로 라이선스의 생성을 지원하는 것으로 구성됩니다.

디자인 타임 라이선스를 제공하기 위해 ActiveX 마법사는 컨트롤에 대한 키를 생성하여 LIC 확장자를 갖는 프로젝트와 동일한 이름을 가진 파일에 저장합니다. 이 .LIC 파일은 프로젝트에 추가됩니다. 컨트롤의 사용자는 .LIC 파일의 복사본이 있어야만 개발 환경에서 컨트롤을 열 수 있습니다. Make Control Licensed를 선택 표시한 프로젝트에 있는 각 컨트롤은 .LIC 파일에 별도의 키 항목을 갖습니다.

런타임 라이선스를 지원하기 위해 래퍼 클래스는 *GetLicenseString* 및 *GetLicenseFilename*의 두 가지 메소드를 구현합니다. 이들은 각각 컨트롤에 대한 라이선스 문자열과 .LIC 파일의 이름을 반환합니다. 호스트 애플리케이션이 ActiveX 컨트롤을 만들 때 컨트롤에 대한 클래스 팩토리는 이러한 메소드를 호출하고 *GetLicenseString*에 의해 반환된 문자열과 .LIC 파일에 저장된 문자열을 비교합니다.

사용자가 모든 웹 페이지의 HTML 소스 코드를 볼 수 있고 ActiveX 컨트롤이 표시되기 전에 사용자의 컴퓨터에 복사되기 때문에 Internet Explorer에 대한 런타임 라이선스는 별도 수준의 간접적인 조치를 필요로 합니다. Internet Explorer에서 사용되는 컨트롤을 위한 런타임 라이선스를 생성하려면 우선 라이선스 패키지 파일(LPK 파일)을 생성하여 컨트롤을 포함하는 HTML 페이지에 이 파일을 포함시켜야 합니다. LPK 파일은 본래 ActiveX 컨트롤 CLSID 및 라이선스 키의 배열입니다.

참고 LPK 파일을 생성하려면 Microsoft 웹 사이트(www.microsoft.com)에서 다운로드할 수 있는 유틸리티인 LPK_TOOL.EXE를 사용합니다.

웹 페이지에 LPK 파일을 포함시키려면 다음과 같이 HTML 객체인 <OBJECT> 및 <PARAM>을 사용합니다.

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
  <PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

CLSID는 라이선스 패키지로 객체를 식별하고 PARAM은 HTML 페이지에 대하여 라이선스 패키지 파일의 상대적인 위치를 지정합니다.

Internet Explorer가 컨트롤이 포함된 웹 페이지를 표시할 때 이는 LPK 파일을 구문 분석하고 라이선스 키를 추출하며 라이선스 키가 *GetLicenseString*에서 반환된 컨트롤의 라이선스와 일치하면 페이지에 컨트롤을 나타냅니다. 두 개 이상의 LPK가 웹 페이지에 포함되는 경우, Internet Explorer는 첫 번째 것을 제외한 모든 것을 무시합니다.

자세한 내용은 Microsoft 웹 사이트에 있는 Licensing ActiveX Controls를 참조하십시오.

ActiveX 컨트롤의 인터페이스 사용자 지정

ActiveX Control 및 ActiveForm 마법사는 ActiveX 래퍼 클래스에 대한 기본 인터페이스를 생성합니다. 이러한 기본 인터페이스는 다음과 같은 예외를 제외하고 원래의 VCL 컨트롤 또는 폼의 속성, 메소드, 이벤트를 노출합니다.

- Data-aware 속성은 나타내지 않습니다. ActiveX 컨트롤은 컨트롤을 data-aware로 만드는 데 VCL 컨트롤이 아닌 다른 메커니즘을 가지기 때문에 마법사는 데이터와 관련된 속성을 변환하지 않습니다. ActiveX 컨트롤을 data-aware로 만드는 방법에 관한 내용은 38-10 페이지의 "타입 라이브러리를 이용한 간단한 데이터 바인딩 활성화"를 참조하십시오.
- Automation 호환이 아닌 타입의 속성, 메소드, 이벤트는 나타내지 않습니다. 마법사를 완료하면 ActiveX 컨트롤의 인터페이스에 이를 추가할 수 있습니다.

타입 라이브러리를 편집하여 ActiveX 컨트롤의 속성, 메소드, 이벤트를 추가, 편집, 제거할 수 있습니다. 34장 "Type Library 작업"에서 설명하듯이 Type Library 에디터를 사용할 수 있습니다. 이벤트 추가 시 ActiveX 컨트롤의 기본 인터페이스가 아닌 Events 인터페이스에 추가해야 한다는 점에 유의하십시오.

참고 ActiveX 컨트롤의 인터페이스에 published가 아닌 속성을 추가할 수 있습니다. 이런 속성들은 런타임 시 설정할 수 있으며 개발 환경에 나타나지만 변경 내용이 유지되지 않습니다. 즉, 컨트롤의 사용자가 디자인 타임 시 속성의 값을 변경하면 컨트롤이 실행될 때 변경 내용이 반영되지 않습니다. 소스가 VCL 객체이고 속성이 published가 아닌 경우, VCL 객체의 자손을 생성하고 자손에 속성을 published로 만들어 속성을 영구적으로 만들 수 있습니다.

또한 VCL 컨트롤의 속성, 메소드, 이벤트를 모두 호스트 애플리케이션에 노출하지 않도록 선택할 수도 있습니다. Type Library 에디터를 사용하여 마법사가 생성한 인터페이스에서 이들을 제거할 수 있습니다. Type Library 에디터를 사용하여 인터페이스에서 속성 및 메소드를 제거할 때 Type Library 에디터는 해당 구현 클래스에서 이들을 제거하지는 않습니다. Type Library 에디터에서 인터페이스를 변경했던 후에 이들을 제거하려면 구현 유닛에서 ActiveX 랩퍼 클래스를 편집합니다.

경고 원래의 VCL 컨트롤 또는 폼에서 ActiveX 컨트롤을 다시 생성하면 타입 라이브러리에 대한 변경 내용을 잃어버립니다.

팁 마법사가 사용자의 ActiveX 랩퍼 클래스에 추가한 메소드를 확인하는 것이 좋습니다. 이렇게 하면 마법사가 Automation 호환이 아닌 data-aware 속성 또는 메소드를 생략한 위치를 참고할 수 있을 뿐만 아니라 마법사가 구현을 생성할 수 없는 메소드를 알아낼 수 있습니다. 이러한 메소드는 문제를 나타내는 구현에 설명과 함께 나타납니다.

다른 속성, 메소드, 이벤트 추가

Type Library 에디터를 사용하여 컨트롤에 다른 속성, 메소드, 이벤트들을 추가할 수 있습니다. 선언은 컨트롤의 구현 유닛, 타입 라이브러리(TLB) 파일, 타입 라이브러리 유닛에 자동으로 추가됩니다. Delphi가 제공하는 세부적인 사항들은 속성이나 메소드를 추가했는지 또는 이벤트를 추가했는지에 따라 달라집니다.

속성 및 메소드 추가

ActiveX 랩퍼 클래스는 read 및 write 액세스 메소드를 사용하여 해당 인터페이스에서 속성을 구현합니다. 즉, 랩퍼 클래스는 getter 및 setter 메소드로 인터페이스에 나타나는 COM 속성을 갖습니다. VCL 속성과 달리 COM 속성의 경우 인터페이스에서 "속성"

선언을 볼 수 없습니다. 더 정확히 말하면 속성 액세스 메소드로 플래그된 메소드가 보입니다. 인터페이스에 메소드를 추가할 때 래퍼 클래스는 사용자가 구현할 해당 메소드를 얻는 것처럼 ActiveX 컨트롤의 기본 인터페이스에 속성을 추가할 때 (Type Library 에디터에 의해 업데이트되는 _TLB 유닛에 나타나는) 래퍼 클래스 정의는 구현해야 할 하나 또는 두 개의 새로운 메소드 (getter 및 setter)를 얻습니다. 따라서 래퍼 클래스의 인터페이스에 속성을 추가하는 것은 본질적으로 메소드를 추가하는 것과 같습니다. 래퍼 클래스 정의는 사용자가 완성하는 새로운 뼈대 메소드 구현을 얻습니다.

참고 생성된 _TLB 유닛에 무엇이 나타나는지에 대한 자세한 내용은 35-5 페이지의 "타입 라이브러리 정보를 import할 때 생성되는 코드"를 참조하십시오.

예를 들어, 기본이 되는 VCL 객체에 있는 *TCaption* 타입의 *Caption* 속성을 고려해 봅니다. 객체의 인터페이스에 이 속성을 추가하려면 Type Library 에디터를 통해 인터페이스에 속성을 추가할 때 다음을 입력합니다.

```
property Caption: TCaption read Get_Caption write Set_Caption;
```

Delphi는 래퍼 클래스에 다음과 같은 선언을 추가합니다.

```
function Get_Caption: WideString; safecall;  
procedure Set_Caption(const Value: WideString); safecall;
```

또한 사용자가 완성하는 뼈대가 되는 메소드 구현을 추가합니다.

```
function TButtonX.Get_Caption:WideString;  
begin  
end;  
  
procedure TButtonX.Set_Caption(Value:WideString);  
begin  
end;
```

일반적으로 래퍼 클래스의 *FDelphiControl* 멤버를 사용하여 액세스할 수 있는 연결된 VCL 컨트롤에 위임하여 이러한 메소드를 구현할 수 있습니다.

```
function TButtonX.Get_Caption:WideString;  
begin  
    Result := WideString(FDelphiControl.Caption);  
end;  
  
procedure TButtonX.Set_Caption(const Value:WideString);  
begin  
    FDelphiControl.Caption := TCaption(Value);  
end;
```

경우에 따라 COM 데이터 타입을 원시 오브젝트 파스칼 타입으로 변환하기 위한 코드를 추가해야 합니다. 앞의 예제는 타입 변환으로 이를 관리합니다.

참고 Automation 인터페이스 메소드는 **safecall**로 선언되기 때문에 이러한 메소드에 대한 COM 예외 코드를 구현하지 않아도 됩니다. Delphi 컴파일러는 Delphi 예외를 잡아내고 COM 오류 info 구조로 변환하고 코드를 반환하기 위해 **safecall** 메소드의 몸체 주위에 코드를 생성하여 이를 처리합니다.

이벤트 추가

Automation 객체가 클라이언트에 이벤트를 발생시키는 것과 같은 방식으로 ActiveX 컨트롤은 자신의 컨테이너에 이벤트를 발생시킬 수 있습니다. 이 메커니즘은 36-10 페이지의 "클라이언트에 이벤트 노출"에 설명되어 있습니다.

ActiveX 컨트롤의 기초로 사용하는 VCL 컨트롤에 published 이벤트가 있으면 마법사는 ActiveX 랩퍼 클래스에 대한 클라이언트 이벤트 싱크의 목록을 관리하는 데 필요한 지원을 자동으로 추가하고 클라이언트가 이벤트에 응답하기 위해 구현해야 하는 발신 dispinterface를 자동으로 정의합니다.

발신 dispinterface에 이벤트를 추가합니다. Type Library 에디터에서 이벤트를 추가하려면 이벤트 인터페이스를 선택하고 메소드 아이콘을 클릭합니다. 매개변수 페이지를 사용하여 포함시킬 매개변수의 목록을 수동으로 추가합니다.

그런 다음 기본이 되는 VCL 컨트롤의 이벤트에 대한 이벤트 핸들러와 동일한 타입인 랩퍼 클래스의 메소드를 선언해야 합니다. Delphi는 이벤트 핸들러가 사용 중인지 모르기 때문에 이는 자동으로 생성되지 않습니다.

```
procedure KeyPressEvent(Sender:TObject; var Key:Char);
```

랩퍼 클래스의 *FEvents* 멤버에 저장된 호스트 애플리케이션의 이벤트 싱크를 사용하여 이 메소드를 구현합니다.

```
procedure TButtonX.KeyPressEvent(Sender:TObject; var Key:Char);
var
  TempKey:Smallint;
begin
  TempKey := Smallint(Key); {cast to an OleAutomation compatible type }
  if FEvents <> nil then
    FEvents.OnKeyPress(TempKey)
  Key := Char(TempKey);
end;
```

참고 ActiveX 컨트롤에서 이벤트를 발생시킬 때 컨트롤은 단일 호스트 애플리케이션만 가지므로 이벤트 싱크의 목록을 통해 반복하지 않아도 됩니다. 이는 대부분의 Automation 서버에 대한 프로세스보다 더 간단합니다.

마지막으로 이벤트가 발생할 때 호출될 수 있도록 이 이벤트 핸들러를 기본이 되는 VCL 컨트롤에 할당해야 합니다. *InitializeControl* 메소드에 할당합니다.

```
procedure TButtonX.InitializeControl;
begin
  FDelphiControl := Control as TButton;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;
```

타입 라이브러리를 이용한 간단한 데이터 바인딩 활성화

간단한 데이터 바인딩을 사용하여 ActiveX 컨트롤의 속성을 데이터베이스의 필드에 바인딩할 수 있습니다. 이렇게 하려면 ActiveX 컨트롤은 필드 데이터를 나타내는 값과 그

변경 시기에 대해 호스트 애플리케이션과 통신해야 합니다. Type Library 에디터를 사용하여 속성의 바인딩 플래그를 설정하여 통신할 수 있습니다.

속성을 바인딩할 수 있게 표시함으로써 사용자가 데이터베이스의 필드와 같은 속성을 수정할 때 컨트롤은 값이 변경되어 데이터베이스 레코드를 업데이트해야 함을 컨테이너(클라이언트 호스트 애플리케이션)에 알려 줍니다. 컨테이너는 데이터베이스와 상호 작용한 다음 레코드 업데이트의 성공 또는 실패 여부를 컨트롤에 알려 줍니다.

참고 사용자의 ActiveX 컨트롤을 호스트하는 컨테이너 애플리케이션은 타입 라이브러리에서 할 수 있는 data-aware 속성을 데이터베이스에 연결하기만 하면 됩니다. Delphi를 사용하여 이러한 컨테이너를 작성하는 방법에 대한 자세한 내용은 35-8 페이지의 "Data-aware ActiveX 컨트롤 사용"을 참조하십시오.

타입 라이브러리를 사용하여 간단한 데이터 바인딩을 활성화합니다.

- 1 툴바에서 바인딩할 속성을 클릭합니다.
- 2 플래그 페이지를 선택합니다.
- 3 다음과 같은 바인딩 속성을 선택합니다.

바인딩 속성	설명
Bindable	속성이 데이터 바인딩을 지원함을 나타냅니다. bindable이 표시되어 있으면 속성은 속성 값이 변경될 때 컨테이너에 알려 줍니다.
Request Edit	속성이 OnRequestEdit 공지를 지원함을 나타냅니다. 이렇게 하면 컨트롤은 사용자가 값을 편집할 수 있는지 컨테이너에 묻습니다.
Display Bindable	컨테이너가 이 속성이 바인딩 가능한지 사용자에게 보여 줄 수 있는지 나타냅니다.
Default Bindable	객체를 가장 잘 나타내는 단일 바인딩 가능 속성을 나타냅니다. 기본 바인딩 속성을 갖는 속성은 바인딩 가능 속성도 가져야 합니다. dispinterface에서 두 개 이상의 속성에 지정할 수 없습니다.
Immediate Bindable	폼의 개별 바인딩 가능 속성이 이 동작을 지정하도록 해줍니다. 이 비트가 설정되면 모든 변경 내용이 공지됩니다. 이 새로운 비트가 영향을 미치기 위해서는 Bindable 및 Request Edit 속성 비트를 설정해야 합니다.

- 4 타입 라이브러리를 업데이트하려면 툴바의 Refresh 버튼을 클릭합니다.

데이터 바인딩 컨트롤을 테스트하기 위해서 먼저 이를 등록해야 합니다.

예를 들어, *TEdit* 컨트롤을 data-bound ActiveX 컨트롤로 변환하려면 *TEdit*에서 ActiveX 컨트롤을 생성한 다음 Text 속성 플래그를 Bindable, Display Bindable, Default Bindable, Immediate Bindable로 변경합니다. 컨트롤이 등록되고 import된 후에는 데이터를 표시하는 데 사용될 수 있습니다.

ActiveX 컨트롤에 대한 속성 페이지 생성

속성 페이지는 사용자가 ActiveX 컨트롤의 속성을 변경할 수 있는 Delphi Object Inspector와 유사한 대화 상자입니다. 속성 페이지 대화 상자는 컨트롤에 대한 많은 속성을 그룹화하여 한꺼번에 편집할 수 있게 해줍니다. 또는 더 복잡한 속성에 대한 대화 상자를 제공할 수 있습니다.

일반적으로 사용자는 ActiveX 컨트롤을 마우스 오른쪽 버튼으로 클릭하고 Properties를 선택하여 속성 페이지에 액세스합니다.

속성 페이지 생성 프로세스는 폼 생성 프로세스와 유사합니다.

- 1 새 속성 페이지를 생성합니다.
- 2 속성 페이지에 컨트롤을 추가합니다.
- 3 속성 페이지의 컨트롤을 ActiveX 컨트롤의 속성에 연결합니다.
- 4 속성 페이지를 ActiveX 컨트롤에 연결합니다.

참고 ActiveX 컨트롤 또는 ActiveForm에 속성을 추가할 때 계속 유지할 속성을 published로 만들어야 합니다. 이 속성들이 기본이 되는 VCL 컨트롤에서 published가 아니면 published로 속성을 다시 선언하는 VCL 컨트롤의 사용자 지정 자손을 만든 다음 ActiveX 컨트롤 마법사를 사용하여 자손 클래스로부터 ActiveX 컨트롤을 생성해야 합니다.

새 속성 페이지 생성

Property Page 마법사를 사용하여 새 속성 페이지를 생성합니다.

다음과 같은 방법으로 새 속성 페이지를 생성합니다.

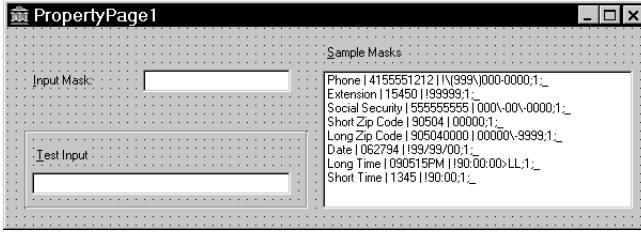
- 1 File|New를 선택합니다.
- 2 ActiveX 탭을 선택합니다.
- 3 Property Page 아이콘을 더블 클릭합니다.

마법사는 속성 페이지에 대한 새 폼 및 구현 유닛을 생성합니다. 폼은 속성을 편집하는 ActiveX 컨트롤과 폼을 연결할 수 있는 *TPropertyPage*의 자손입니다.

속성 페이지에 컨트롤 추가

사용자가 액세스할 ActiveX 컨트롤의 각 속성에 대한 속성 페이지에 컨트롤을 추가해야 합니다.

예를 들어, 다음 그림 예제는 ActiveX 컨트롤의 MaskEdit 속성을 설정하는 속성 페이지를 보여 줍니다.

그림 38.1 디자인 모드에서의 Mask Edit 속성 페이지

리스트 박스를 통해 사용자는 예제 마스크의 목록에서 선택할 수 있습니다. 편집 컨트롤을 통해 ActiveX 컨트롤에 적용하기 전에 사용자는 마스크를 테스트할 수 있습니다. 폼에 대한 속성 페이지와 동일한 속성 페이지에 컨트롤을 추가합니다.

속성 페이지 컨트롤을 ActiveX 컨트롤 속성에 연결

속성 페이지에 필요한 컨트롤을 추가한 다음 각 컨트롤을 해당 속성에 연결해야 합니다. 속성 페이지의 *UpdatePropertyPage* 및 *UpdateObject* 메소드에 코드를 추가하여 연결합니다.

속성 페이지 업데이트

ActiveX 컨트롤의 속성이 변경될 때 속성 페이지의 컨트롤을 업데이트하려면 *UpdatePropertyPage* 메소드에 코드를 추가합니다. ActiveX 컨트롤에 있는 속성의 현재 값을 가지고 속성 페이지를 업데이트하려면 *UpdatePropertyPage* 메소드에 코드를 추가해야 합니다.

ActiveX 컨트롤의 인터페이스가 포함된 *OleVariant*인 속성 페이지의 *OleObject* 속성을 사용하여 ActiveX 컨트롤에 액세스할 수 있습니다.

예를 들어, 다음 코드는 ActiveX 컨트롤의 *EditMask* 속성의 현재 값이 있는 속성 페이지의 편집 컨트롤(InputMask)을 업데이트합니다.

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
  { Update your controls from OleObject }
  InputMask.Text := OleObject.EditMask;
end;
```

참고 또한 두 개 이상의 ActiveX 컨트롤을 나타내는 속성 페이지를 작성할 수 있습니다. 이러한 경우 *OleObject* 속성을 사용하지 않습니다. 그 대신 *OleObjects* 속성에 의해 유지되는 인터페이스의 목록을 통해 반복해야 합니다.

객체 업데이트

사용자가 속성 페이지의 컨트롤을 변경할 때 속성을 업데이트하려면 *UpdateObject* 메소드에 코드를 추가합니다. ActiveX 컨트롤의 속성을 새로운 값으로 설정하려면 *UpdateObject* 메소드에 코드를 추가해야 합니다.

OleObject 속성을 사용하여 ActiveX 컨트롤에 액세스합니다.

예를 들어, 다음 코드는 속성 페이지의 편집 상자 컨트롤(InputMask)을 사용하여 ActiveX 컨트롤의 *EditMask* 속성을 설정합니다.

```
procedure TPropertyPage1.UpdateObject;
begin
  {Update OleObject from your control }
  OleObject.EditMask := InputMask.Text;
end;
```

속성 페이지를 ActiveX 컨트롤에 연결

다음과 같은 방법으로 속성 페이지를 ActiveX 컨트롤에 연결합니다.

- 1 유닛에 대한 컨트롤의 구현 내에 있는 *DefinePropertyPages* 메소드 구현에 매개변수로 속성 페이지의 GUID 상수가 있는 *DefinePropertyPage*를 추가합니다. 예를 들면, 다음과 같습니다.

```
procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_PropertyPage1);
end;
```

속성 페이지의 GUID 상수인 *Class_PropertyPage1*은 속성 페이지 유닛에서 찾을 수 있습니다.

GUID는 속성 페이지의 구현 유닛에 정의됩니다. 즉, Property Page 마법사에 의해 자동으로 생성됩니다.

- 2 컨트롤 구현 유닛의 **uses** 절에 속성 페이지 유닛을 추가합니다.

ActiveX 컨트롤 등록

ActiveX 컨트롤을 생성한 다음 다른 애플리케이션이 찾아서 사용할 수 있도록 반드시 등록해야 합니다.

다음과 같은 방법으로 ActiveX 컨트롤을 등록합니다.

- Run|Register ActiveX Server를 선택합니다.

참고 사용자의 시스템에서 ActiveX 컨트롤을 제거하기 전에 등록을 해제해야 합니다.

다음과 같은 방법으로 ActiveX 컨트롤의 등록을 해제합니다.

- Run|Unregister ActiveX Server를 선택합니다.

또 다른 방법으로 명령줄에서 **tregrsvr** 명령을 사용하거나 운영 체제의 regsvr32.exe를 실행할 수 있습니다.

ActiveX 컨트롤 테스트

사용자의 컨트롤을 테스트하려면 패키지에 이를 추가하여 ActiveX 컨트롤로 import 합니다. 이 프로시저는 Delphi 컴포넌트 팔레트에 ActiveX 컨트롤을 추가합니다. 컨트롤을 폼에 가져다 놓고 필요할 때 테스트할 수 있습니다.

또한 사용자의 컨트롤은 해당 컨트롤을 사용하는 모든 대상 애플리케이션에서 테스트해야 합니다.

ActiveX 컨트롤을 디버깅하려면 Run|Parameters를 선택하고 Host Application 편집 상자에서 클라이언트 이름을 입력합니다.

그러면 매개변수는 호스트 애플리케이션에 적용됩니다. Run|Run을 선택하면 호스트 또는 클라이언트 애플리케이션을 실행하여 컨트롤에서 브레이크포인트를 설정할 수 있습니다.

웹에 ActiveX 컨트롤 배포

사용자가 만든 ActiveX 컨트롤이 웹 클라이언트에 의해 사용될 수 있게 하려면 웹 서버에 배포해야 합니다. ActiveX 컨트롤을 변경할 때마다 클라이언트 애플리케이션이 변경 내용을 볼 수 있도록 이를 다시 컴파일하고 다시 배포해야 합니다.

ActiveX 컨트롤을 배포하려면 클라이언트 메시지에 응답하는 웹 서버가 있어야 합니다.

ActiveX 컨트롤을 배포하는 단계는 다음과 같습니다.

- 1 Project|Web Deployment Options를 선택합니다.
- 2 Project 페이지에서 웹 서버에서의 경로로 Target Dir을 ActiveX 컨트롤 DLL의 위치로 설정합니다. 이는 C:\INETPUB\wwwroot와 같이 로컬 경로 또는 UNC 경로로 일 수 있습니다.
- 3 Target URL을 http://mymachine.inprise.com/ 같이 웹 서버에서 파일 이름이 없는 ActiveX 컨트롤 DLL의 Uniform Resource Locators(URL)로 설정합니다. 이 방법에 대한 자세한 내용은 웹 서버에 대한 문서를 참조하십시오.
- 4 HTML Dir을 C:\INETPUB\wwwroot와 같이 ActiveX 컨트롤에 대한 참조가 포함된 HTML 파일을 둘 위치로 설정합니다. 이 경로는 표준 경로 이름 또는 UNC 경로일 수 있습니다.
- 5 38-16 페이지의 "옵션 설정"에서 설명한 대로 원하는 웹 배포 옵션을 선택합니다.
- 6 OK를 선택합니다.
- 7 Project|Web Deploy를 선택합니다.

이렇게 하면 확장자가 OCX인 ActiveX 라이브러리에 ActiveX 컨트롤을 포함하는 배포 코드 기본을 생성합니다. 사용자가 지정한 옵션에 따라 이 배포 코드 기본은 확장자가 CAB인 캐비닛 또는 확장자가 INF인 정보를 포함할 수도 있습니다.

ActiveX 라이브러리는 2단계에서 지정한 Target Directory에 둡니다. HTML 파일은 프로젝트 파일과 이름은 같지만 확장자는 HTM입니다. 이는 4단계에서 지정한

HTML Directory 에서 생성됩니다. HTML 파일은 3 단계에서 지정한 위치에서 ActiveX 라이브러리에 대한 URL 참조를 포함합니다.

참고 이러한 파일을 웹 서버에 두려면 ftp와 같은 외부 유틸리티를 사용합니다.

8 ActiveX 활성 웹 브라우저를 호출하고 생성된 HTML 페이지를 봅니다.

이 HTML 페이지를 웹 브라우저에서 볼 때 해당 폼 또는 컨트롤이 표시되어 브라우저 내에서 포함된 애플리케이션처럼 실행합니다. 즉, 라이브러리는 브라우저 애플리케이션과 동일한 프로세스를 실행합니다.

옵션 설정

ActiveX 컨트롤을 배포하기 전에 ActiveX 라이브러리를 만들 때 따라야 하는 웹 배포 옵션을 지정합니다.

웹 배포 옵션은 설정을 포함하고 있어 다음을 설정할 수 있습니다.

- **추가 파일 포함:** ActiveX 컨트롤이 패키지 또는 다른 추가 파일에 따라 다르다면 이들은 프로젝트와 함께 배포되어야 함을 나타낼 수 있습니다. 기본적으로 이러한 파일은 전체 프로젝트에 대해 지정한 것과 동일한 옵션을 사용하지만 Packages 또는 Additional 파일 탭을 사용하여 이러한 설정을 오버라이드할 수 있습니다. 패키지 또는 추가 파일을 포함시킬 때 Delphi는 INFormation에 대해 확장자가 .INF인 파일을 생성합니다. 이 파일은 ActiveX 라이브러리를 실행하기 위해 다운로드하고 설정해야 하는 다양한 파일을 지정합니다. INF 파일의 구문을 통해 URL은 다운로드할 패키지 또는 추가 파일을 가리킵니다.
- **CAB 파일 압축:** 캐비닛은 일반적으로 파일 확장자가 CAB인 하나의 파일이며 파일 라이브러리에 압축 파일을 저장합니다. 캐비닛 압축은 파일 다운로드 시간을 최대 70%까지 줄일 수 있습니다. 브라우저는 설치 중에 캐비닛에 저장된 파일의 압축을 풀어 사용자의 시스템에 복사합니다. 배포한 각 파일은 압축된 CAB 파일일 수 있습니다. ActiveX 라이브러리가 Web Deployment 옵션 대화 상자의 Project 탭에서 CAB 파일 압축을 사용하도록 지정할 수 있습니다.
- **버전 정보:** 버전 정보가 ActiveX 컨트롤에 포함되도록 지정할 수 있습니다. 이 정보는 Project Options 대화 상자의 VersionInfo 페이지에서 설정합니다. 이 정보의 부분은 ActiveX 컨트롤을 배포할 때마다 자동으로 업데이트할 수 있는 릴리스 번호입니다. 추가 패키지 또는 파일을 포함시킬 경우, 해당 버전 정보 리소스를 INF 파일에 추가할 수 있습니다.

추가 파일을 포함하는지 여부와 CAB 파일 압축을 사용하는지 여부에 따라 결과 ActiveX 라이브러리는 OCX 파일, OCX 파일을 포함한 CAB 파일 또는 INF 파일이 될 수 있습니다. 다음 표는 다른 조합을 선택한 결과를 요약한 것입니다.

패키지 및 추가 파일	CAB 파일 압축	결과
아니오	아니오	ActiveX 라이브러리 (OCX) 파일.
아니오	예	ActiveX 라이브러리 파일을 포함하는 CAB 파일.
예	아니오	INF 파일, ActiveX 라이브러리 파일, 모든 추가 파일 및 패키지.
예	예	INF 파일, ActiveX 라이브러리를 포함하는 CAB 파일, 모든 추가 파일 및 패키지용 CAB 파일.

39

MTS 또는 COM+ 객체 생성

Delphi는 Windows 2000 이전 버전인 경우 MTS(Microsoft Transaction Server)에서 제공하거나 Windows 2000 이상 버전인 경우에는 COM+에서 제공하는 트랜잭션 서비스, 보안 및 리소스 관리를 사용하는 객체를 가리킬 때 트랜잭션 객체라는 용어를 사용합니다. 이 객체들은 대규모 분산 환경에서 작동하도록 디자인되었습니다. 또한 이 객체들은 Windows 관련 기술에 의존하기 때문에 크로스 플랫폼 애플리케이션에서 사용할 수 없습니다.

Delphi는 COM+ 속성이나 MTS 환경의 이점을 활용할 수 있도록 트랜잭션 객체를 만드는 마법사를 제공합니다. 이 기능을 사용하면 COM 클라이언트 및 서버, 특히 원격 서버를 만드는 작업을 더 쉽게 구현할 수 있습니다.

참고 데이터베이스 애플리케이션에 대해 Delphi는 트랜잭션 데이터 모듈도 제공합니다. 자세한 내용은 25장 "다계층(multi-tiered) 애플리케이션 생성"을 참조하십시오.

트랜잭션 객체는 다음과 같은 여러 가지 저수준 서비스를 이용합니다.

- 서버 애플리케이션이 많은 동시 사용자를 처리할 수 있도록 프로세스, 스레드 및 데이터베이스 연결을 포함하는 시스템 리소스 관리
- 애플리케이션이 안정적하도록 자동으로 트랜잭션 초기화 및 제어
- 필요할 때 서버 컴포넌트 생성, 실행 및 삭제
- 승인된 사용자만 애플리케이션을 액세스할 수 있도록 역할 기반 보안(role-based security) 제공
- 클라이언트가 서버에서 발생하는 상황에 응답할 수 있도록 이벤트 관리(COM+ 전용)

MTS 또는 COM+에서 이러한 기본 서비스를 제공하게 하면 사용자는 특정 분산 애플리케이션의 세부 항목을 개발하는 데 집중할 수 있습니다. 선택하는 기술(MTS 또는 COM+)은 애플리케이션을 실행하도록 선택한 서버에 따라 달라집니다. 클라이언트가 특수한 인터페이스를 통해 트랜잭션 서비스를 명시적으로 처리하지만 않는다면 클라이언트에서 MTS와 COM+의 차이(또는 실제로 서버 객체가 이 서비스를 모두 이용한다는 사실)는 분명합니다.

트랜잭션 객체에 대한 이해

일반적으로 트랜잭션 객체는 작으며 각기 다른 비즈니스 기능에 사용됩니다. 트랜잭션 객체는 애플리케이션의 비즈니스 룰을 구현하고 애플리케이션 상태의 뷰와 변환을 제공할 수 있습니다. 예를 들어, 의학 관련 애플리케이션의 경우를 가정해 보십시오. 다양한 데이터베이스에 저장된 진료 기록은 환자의 건강 기록과 같은 지속적인 애플리케이션 상태를 나타냅니다. 트랜잭션 객체는 그 상태를 업데이트하여 새로운 환자, 검사 결과 및 X선 파일과 같은 변경 내용을 반영합니다.

트랜잭션 객체는 분산 컴퓨팅 환경에서 발생하는 문제를 처리하는 데 MTS 또는 COM+에서 제공하는 속성 집합을 사용한다는 점에서 다른 COM 객체와 구분됩니다. 이 속성 중 일부는 *ObjectContext* 인터페이스를 구현할 때 트랜잭션 객체를 사용합니다. *ObjectContext*은 객체가 활성화되거나 비활성화될 때 호출되는 메소드를 정의하고, 사용자는 데이터베이스 연결과 같은 리소스를 관리할 수 있습니다. *ObjectContext*은 객체 풀링(pooling)에도 필요하며 39-8 페이지의 "객체 풀링"에 설명되어 있습니다.

참고 MTS를 사용하는 경우 트랜잭션 객체는 *ObjectContext*을 구현해야 합니다. COM+에서는 *ObjectContext*이 꼭 필요하지는 않지만 적극 권장합니다. Transactional Object 마법사가 *ObjectContext*에서 파생되는 객체를 제공하기 때문입니다.

트랜잭션 객체의 클라이언트는 **기본 클라이언트**라고 합니다. 기본 클라이언트의 관점에서 보면 트랜잭션 객체는 다른 COM 객체와 비슷해 보입니다.

MTS에서 트랜잭션 객체는 라이브러리(DLL)에서 기본 제공되어야 하며 MTS 런타임 환경(MTS 실행 파일, *mtxex.exe*)에 설치됩니다. 즉, 서버 객체는 MTS 런타임 프로세스 공간에서 실행됩니다. MTS 실행 파일은 기본 클라이언트와 같은 프로세스로 실행되거나, 기본 클라이언트와 같은 시스템에서 별도의 프로세스로 실행되거나, 별도의 시스템에 있는 원격 서버 프로세스로 실행될 수 있습니다.

COM+에서 서버 애플리케이션은 in-process 서버일 필요가 없습니다. 다양한 서비스가 COM 라이브러리로 통합되기 때문에 별도의 MTS 프로세스로 서버에 대한 호출을 인터셉트할 필요가 없습니다. 그 대신 COM 자체 또는 COM+는 리소스 관리, 트랜잭션 지원 등을 제공합니다. 그러나 서버 애플리케이션은 여전히 COM+ 애플리케이션에 설치되어야 합니다.

기본 클라이언트와 트랜잭션 객체 사이의 연결은 out-of-process 서버에서처럼 서버의 스텝(stub)과 클라이언트에 있는 프록시에서 처리합니다. 연결 정보는 프록시에서 유지 관리합니다. 클라이언트가 서버에 대한 연결을 필요로 하면 기본 클라이언트와 프록시 사이의 연결이 계속 열려 있으므로 클라이언트에서는 서버에 계속 액세스하는 것처럼 보입니다. 그러나 실제로는 프록시가 객체를 비활성화하고 재활성화할 수 있으며 다른 클라이언트가 연결을 사용할 수 있도록 리소스를 유지할 수 있습니다. 활성화 및 비활성화에 대한 자세한 내용은 39-4 페이지의 "Just-in-time 활성화"를 참조하십시오.

트랜잭션 객체에 대한 요구 사항

트랜잭션 객체는 COM 요구 사항 외에도 다음과 같은 요구 사항을 만족해야 합니다.

- 객체에는 표준 클래스 팩토리가 있어야 하며 객체를 만들 때 마법사에서 자동으로 만들어집니다.
- 서버는 표준 *DllGetClassObject* 메소드를 내보내 클래스 객체를 노출해야 합니다. 이 작업을 수행하는 코드는 마법사에서 제공합니다.
- 모든 객체 인터페이스와 CoClass는 마법사가 자동으로 만드는 타입 라이브러리에서 설명되어야 합니다. Type Library 에디터를 사용하면 타입 라이브러리의 인터페이스에 메소드 및 속성을 추가할 수 있습니다. MTS Explorer나 COM+ Component Manager는 타입 라이브러리의 정보를 사용하여 런타임 시에 설치된 컴포넌트에 대한 정보를 추출합니다.
- 서버는 표준 COM 마샬링 (marshaling)을 사용하는 인터페이스를 내보내야만 합니다. 이것은 Transactional Object 마법사에서 자동으로 제공합니다. Delphi의 트랜잭션 객체에 대한 지원은 사용자 지정 인터페이스에 대한 수동 마샬링을 허용하지 않습니다. 모든 인터페이스는 COM의 자동 마샬링 지원을 사용하는 이중 인터페이스로 구현되어야 합니다.
- 서버는 *DllRegisterServer* 함수를 내보내고 이 루틴의 CLSID, ProgID, 인터페이스 및 타입 라이브러리에 대한 자체 등록을 수행해야 합니다. 이것은 Transactional Object 마법사에서 제공합니다.

COM+가 아닌 MTS를 사용할 때는 다음과 같은 조건도 적용됩니다.

- MTS에서는 서버가 동적 연결 라이브러리(DLL)여야 합니다. 실행 파일(.EXE 파일)로 구현되는 서버는 MTS 런타임 환경에서 실행할 수 없습니다.
- 객체는 *IObjectControl* 인터페이스를 구현해야 합니다. 이 인터페이스에 대한 지원은 Transactional Object 마법사에서 자동으로 추가됩니다.
- MTS 프로세스 공간에서 실행되는 서버는 MTS에서 실행되지 않는 COM 객체와 집계할 수 없습니다.

리소스 관리

애플리케이션에서 사용하는 리소스를 더 잘 관리하도록 트랜잭션 객체를 관리할 수 있습니다. 이 리소스에는 객체 인스턴스 자체에 대한 메모리로부터 데이터베이스 연결과 같이 객체 인스턴스에서 사용하는 모든 리소스에 이르는 모든 것이 포함됩니다.

일반적으로 객체를 설치하고 구성하는 방법으로 애플리케이션이 리소스를 관리하는 방법을 구성합니다. 트랜잭션 객체가 다음을 이용하도록 설정합니다.

- Just-in-time 활성화
- 리소스 풀링
- 객체 풀링 (COM+ 전용)

그러나 객체가 이러한 서비스를 최대한 이용하게 하려면 *IObjectContext* 인터페이스를 사용하여 리소스가 안전하게 해제되는 시기를 나타내야 합니다.

객체 컨텍스트 액세스

모든 COM 객체처럼 트랜잭션 객체는 사용하기 전에 만들어야 합니다. COM 클라이언트는 COM 라이브러리 함수, *CoCreateInstance*를 호출하여 객체를 만듭니다.

각 트랜잭션 객체에는 해당 컨텍스트 객체가 있어야 합니다. 이 컨텍스트 객체는 MTS 나 COM+에서 자동으로 구현되고 트랜잭션 객체를 관리하는 데 사용됩니다. 컨텍스트 객체의 인터페이스는 *IObjectContext*입니다. 객체 컨텍스트의 메소드 대부분을 액세스하려면 *TMtsAutoObject* 객체의 *ObjectContext* 속성을 사용할 수 있습니다. 예를 들어, 다음과 같이 *ObjectContext* 속성을 사용할 수 있습니다.

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

객체 컨텍스트를 액세스하는 또 다른 방법은 다음과 같이 *TMtsAutoObject* 객체의 메소드를 사용하는 것입니다.

```
if IsCallerInRole ('Manager') ...
```

위의 메소드 중 어느 것이든 사용할 수 있습니다. 그러나 애플리케이션을 테스트할 때에는 *TMtsAutoObject* 메소드를 사용하는 것이 *ObjectContext* 속성을 참조하는 것보다 좋습니다. 그 차이점에 대해서는 39-21 페이지의 "트랜잭션 객체 디버깅 및 테스트"를 참조하십시오.

Just-in-time 활성화

클라이언트가 객체에 대한 참조를 유지하면서 객체를 비활성화하고 재활성화하는 기능을 **just-in-time 활성화**라고 합니다. 클라이언트의 관점에서 보면 클라이언트가 객체를 만드는 시기에서 마지막으로 해제하는 시기까지 객체의 인스턴스는 한 번만 있습니다. 실제로 객체는 여러 번 비활성화했다가 다시 활성화할 수 있습니다. 객체를 비활성화하면 클라이언트는 시스템 리소스에 영향을 주지 않고 더 오랫동안 객체에 대한 참조를 유지할 수 있습니다. 객체를 비활성화하면 모든 리소스도 해제할 수 있습니다. 예를 들어, 객체를 비활성화하면 데이터베이스 연결을 해제하여 다른 객체가 이를 사용할 수 있습니다.

비활성화된 상태에서 트랜잭션 객체가 만들어지고 클라이언트 요청을 받으면 활성화됩니다. 트랜잭션 객체가 만들어지면 해당 컨텍스트 객체도 만들어집니다. 이 컨텍스트 객체는 트랜잭션 객체의 전체 수명, 하나 이상의 재활성화 주기 동안 존재합니다. *IObjectContext* 인터페이스로 액세스한 컨텍스트 객체는 비활성화 중에 객체를 추적하고 트랜잭션을 조정합니다.

트랜잭션 객체는 안전하다고 판단되는 즉시 비활성화됩니다. 이것은 **as-soon-as-possible 비활성화**라고 합니다. 트랜잭션 객체는 다음과 같은 경우에 비활성화됩니다.

- 객체는 *SetComplete* 또는 *SetAbort*로 비활성화를 요청합니다. 작업을 성공적으로 완료한 후 다음에 클라이언트에서 호출하기 위해 내부 객체 상태를 저장할 필요가 없을 때 객체가 *IObjectContext SetComplete* 메소드를 호출합니다. 객체는 *SetAbort*를

호출하여 객체가 작업을 성공적으로 완료할 수 없고 객체 상태를 저장할 필요가 없음을 나타냅니다. 즉, 객체의 상태는 현재의 트랜잭션 이전 상태로 롤백됩니다. 종종 객체는 **상태 없음(stateless)**이 되도록 디자인될 수 있는데 이것은 모든 메소드에서 반환되는 즉시 객체가 비활성화된다는 의미입니다.

- **트랜잭션이 커밋되거나 중지됩니다.** 객체의 트랜잭션이 커밋되거나 중지되면 객체는 비활성화됩니다. 이렇게 비활성화된 객체 중에서 계속 존재하는 유일한 객체는 트랜잭션 외부의 클라이언트로부터 참조를 가지는 객체입니다. 이러한 객체를 다음에 호출하면 객체가 재활성화되어 새로운 트랜잭션으로 실행되게 합니다.
- **마지막 클라이언트는 객체를 해제합니다.** 물론 클라이언트가 객체를 해제하면 객체는 비활성화되고 객체 컨텍스트도 해제됩니다.

참고 IDE의 COM+에 트랜잭션 객체를 설치하면 Type Library 에디터의 COM+ 페이지를 사용하여 객체가 just-in-time 활성화를 지원하는지 여부를 지정할 수 있습니다. Type Library 에디터에서 객체(CoClass)를 선택하고 COM+ 페이지로 이동한 다음 Just In Time Activation 상자를 선택하거나 선택을 해제하면 됩니다. 그렇지 않은 경우, 시스템 관리자가 COM+ Component Manager 또는 MTS Explorer를 사용하여 이 속성을 지정합니다. 또한 시스템 관리자는 Type Library 에디터를 사용하여 설정한 모든 설정을 오버라이드할 수도 있습니다.

리소스 풀링

비활성화 중에는 유휴 시스템 리소스가 해제되므로 해제된 리소스는 다른 서버 객체에 사용할 수 있습니다. 예를 들어, 서버 객체에서 더 이상 사용하지 않는 데이터베이스 연결은 다른 클라이언트에서 다시 사용할 수 있습니다. 이것을 **리소스 풀링(pooling)**이라고 합니다. 풀링된 리소스는 리소스 디스펜서에서 관리합니다.

리소스 디스펜서는 리소스를 캐시하므로 함께 설치된 트랜잭션 객체는 리소스를 공유할 수 있습니다. 또한 리소스 디스펜서는 지속적인지 않은 공유 상태 정보를 관리합니다. 이런 점에서 리소스 디스펜서는 SQL Server와 같은 리소스 관리자와 유사하지만 지속성을 보장하지는 않습니다.

트랜잭션 객체를 작성할 때는 이미 제공된 두 가지 종류의 리소스 디스펜서를 사용할 수 있습니다.

- 데이터베이스 리소스 디스펜서
- Shared Property Manager

다른 객체가 풀링된 리소스를 사용할 수 있으려면 먼저 리소스를 해제해야 합니다.

데이터베이스 리소스 디스펜서

데이터베이스에 대한 연결을 열고 닫는 데는 시간이 걸릴 수 있습니다. 리소스 디스펜서를 사용하여 데이터베이스 연결을 풀링하면 객체는 새 연결을 만들지 않고 기존 데이터베이스 연결을 다시 사용할 수 있습니다. 예를 들어, 고객 유지 관리 애플리케이션에서 실행되는 데이터베이스 조회 및 데이터베이스 업데이트 컴포넌트가 있으면 해당 컴포넌트를 함께 설치한 다음 데이터베이스 연결을 공유할 수 있습니다. 따라서 애플리케이션

선에는 많은 연결이 필요하지 않으며 새 객체 인스턴스는 이미 열려 있지만 사용되지 않는 연결을 사용하여 더 빨리 데이터에 액세스할 수 있습니다.

- BDE 컴포넌트를 사용하여 데이터에 연결하면 리소스 디스펜서는 BDE(Borland Database Engine)가 됩니다. 이 리소스 디스펜서는 트랜잭션 객체가 MTS와 함께 설치될 때만 사용할 수 있습니다. 리소스 디스펜서를 사용하려면 BDE 관리자를 사용하여 구성의 System/Init 영역에서 MTS POOLING을 선택합니다.
- ADO 데이터베이스 컴포넌트를 사용하여 데이터에 연결하면 리소스 디스펜서는 ADO에서 제공합니다.

참고 데이터베이스 액세스에 InterbaseExpress 컴포넌트를 사용하는 경우에는 기본 제공된 리소스 풀링이 없습니다.

원격 트랜잭션 데이터 모듈의 경우 연결은 객체의 트랜잭션에 자동으로 포함되므로 리소스 디스펜서가 연결을 자동으로 수정하여 다시 사용할 수 있습니다.

Shared Property Manager

Shared Property Manager는 서버 프로세스 내의 여러 객체 사이에서 상태를 공유하는 데 사용할 수 있는 리소스 디스펜서입니다. Shared Property Manager를 사용하면 공유 데이터를 관리하기 위해 애플리케이션에 많은 코드를 추가할 필요가 없습니다. Shared Property Manager는 동시 액세스로부터 공유 속성을 보호하기 위해 잠금과 세마포(semaphores)를 구현하여 처리합니다. Shared Property Manager는 포함된 공유 속성에 고유한 이름 공간을 설정하는 **공유 속성 그룹**을 제공하여 이름 충돌을 제거합니다.

Shared Property Manager 리소스를 사용하려면 먼저 *CreateSharedPropertyGroup* helper 함수를 사용하여 공유 속성 그룹을 만듭니다. 그런 다음 해당 그룹에 대한 모든 속성을 작성하고 그 그룹에서 모든 속성을 읽어 올 수 있습니다. 공유 속성 그룹을 사용하면 상태 정보는 트랜잭션 객체의 모든 비활성화에 저장됩니다. 또한 상태 정보는 같은 MTS 패키지 또는 COM+ 애플리케이션에 설치된 모든 트랜잭션 객체 사이에서 공유할 수 있습니다. 39-22 페이지의 "트랜잭션 객체 설치"에서 설명한 대로 트랜잭션 객체를 패키지로 설치할 수 있습니다.

객체가 상태를 공유하려면 모두 같은 프로세스로 실행되어야 합니다. 다른 컴포넌트의 인스턴스가 속성을 공유하게 하려면 같은 MTS 패키지나 COM+ 애플리케이션에 컴포넌트를 설치해야 합니다. 관리자가 패키지 사이에서 컴포넌트를 이동할 수 있는 위험이 있으므로 공유 속성 그룹의 사용은 같은 DLL이나 EXE에 정의된 객체의 인스턴스로 제한하는 것이 가장 안전합니다.

속성을 공유하는 객체에는 같은 활성화 속성이 있어야 합니다. 같은 패키지 내의 두 컴포넌트가 다른 활성화 속성을 가지고 있으면 일반적으로 두 컴포넌트는 속성을 공유할 수 없습니다. 예를 들어, 한 컴포넌트는 클라이언트의 프로세스로 실행되도록 구성되고 다른 컴포넌트는 서버 프로세스로 실행되도록 구성되면 같은 MTS 패키지 또는 COM+ 애플리케이션에 있는 경우에도 일반적으로 그 객체는 서로 다른 프로세스로 실행됩니다.

다음 예제는 트랜잭션 객체의 Shared Property Manager를 지원하도록 코드를 추가하는 방법을 보여 줍니다.

예제: 트랜잭션 객체 인스턴스 사이에서 속성 공유

다음 예제에서는 객체와 객체 인스턴스 사이에서 공유할 속성이 포함된 MyGroup이라는 속성 그룹을 만듭니다. 이 때 공유되는 속성은 Counter 속성입니다. 또한 예제에서는 *CreateSharedPropertyGroup* helper 함수를 사용하여 속성 그룹 관리자와 속성 그룹을 만든 다음 Group 객체의 *CreateProperty* 메소드를 사용하여 Counter라는 속성을 만듭니다.

속성 값을 구하려면 아래에 표시된 대로 Group 객체의 *PropertyByName* 메소드를 사용합니다. 또한 *PropertyByPosition* 메소드를 사용할 수도 있습니다.

```

unit Unit1;
interface
uses
    MtsObj, Mtx, ComObj, Project2_TLB;
type
    Tfoobar = class(TMtsAutoObject, Ifoobar)
    private
        Group: ISharedPropertyGroup;
    protected
        procedure OnActivate; override;
        procedure OnDeactivate; override;
        procedure IncCounter;
    end;
implementation
uses ComServ;
{ Tfoobar }
procedure Tfoobar.OnActivate;
var
    Exists: WordBool;
    Counter: ISharedProperty;
begin
    Group := CreateSharedPropertyGroup('MyGroup');
    Counter := Group.CreateProperty('Counter', Exists);
end;
procedure Tfoobar.IncCounter;
var
    Counter: ISharedProperty;
begin
    Counter := Group.PropertyByName['Counter'];
    Counter.Value := Counter.Value + 1;
end;
procedure Tfoobar.OnDeactivate;
begin
    Group := nil;
end;
initialization
    TAutoObjectFactory.Create(ComServer, Tfoobar, Class_foobar, ciMultiInstance,
tmApartment);
end.

```

리소스 해제

객체의 리소스 해제를 담당합니다. 일반적으로 클라이언트 요청을 처리한 후 *IObjectContext* 메소드 *SetComplete*와 *SetAbort*를 호출하여 이 작업을 수행합니다. 이러한 메소드는 리소스 디스펜서가 할당한 리소스를 해제합니다.

이 때 트랜잭션 객체 및 컨텍스트 객체를 포함하는 다른 객체에 대한 참조를 포함하여 다른 모든 리소스에 대한 참조와 컴포넌트를 해제하는 컴포넌트의 인스턴스가 보유한 메모리를 해제해야 합니다.

이러한 호출을 포함하지 않는 유일한 경우는 클라이언트 호출 사이에서 상태를 유지 관리하려는 경우입니다. 자세한 내용은 39-11 페이지의 "상태 있는 (stateful) 객체 및 상태 없는 (stateless) 객체"를 참조하십시오.

객체 풀링

리소스를 풀링할 수 있는 것처럼 COM+에서 객체도 풀링할 수도 있습니다. 객체가 비활성화되면 COM+는 *IObjectContext* 인터페이스 메소드, *CanBePooled*를 호출하는데 이것은 객체를 풀링하여 다시 사용할 수 있음을 나타냅니다. *CanBePooled*가 *True*를 반환하면 객체는 비활성화에서 소멸하지 않고 객체 풀로 이동합니다. 객체는 지정된 시간 초과, 즉 객체를 요청하는 모든 클라이언트에 사용할 수 있는 시간 동안 객체 풀에 남아 있습니다. 객체 풀이 비어 있을 때만 객체의 새 인스턴스가 만들어집니다. *False*를 반환하는 객체 또는 *IObjectContext* 인터페이스를 지원하지 않는 객체는 비활성화될 때 소멸합니다.

MTS에서는 객체 풀링을 사용할 수 없습니다. MTS는 설명한 대로 *CanBePooled*를 호출하지만 풀링이 발생하지는 않습니다. 객체가 COM+에서만 실행되는 경우 객체 풀링을 허용하려면 객체의 *Pooled* 속성을 *True*로 설정합니다.

객체의 *CanBePooled* 메소드가 *True*를 반환하는 경우에도 COM+가 객체를 객체 풀로 이동하지 않도록 구성할 수 있습니다. IDE의 COM+에서 트랜잭션 객체를 설치하면 COM+가 Type Library 에디터의 COM+ 페이지를 사용하여 객체를 풀링할지 지정할 수 있습니다. Type Library 에디터에서 객체 (CoClass)를 선택하고 COM+ 페이지로 이동한 다음 Object Pooling 상자를 선택하거나 선택을 해제하면 됩니다. 그렇지 않은 경우, 시스템 관리자가 COM+ Component Manager 또는 MTS Explorer를 사용하여 이 속성을 지정합니다.

마찬가지로 비활성화된 객체가 해제되기 전에 객체 풀에 남아 있는 시간을 구성할 수 있습니다. IDE에서 설치하면 Type Library 에디터의 COM+ 페이지에서 Creation Timeout 설정을 사용하여 이 기간을 지정할 수 있습니다. 그렇지 않은 경우에는 시스템 관리자가 COM+ Component Manager를 사용하여 이 속성을 지정합니다.

MTS 및 COM+ 트랜잭션 지원

트랜잭션 객체에 그 이름을 제공하는 트랜잭션 지원을 사용하면 작업을 트랜잭션으로 그룹화할 수 있습니다. 예를 들어, 의학 관련 레코드 애플리케이션에서 의사들 간에 레코드를 전송하는 Transfer 컴포넌트가 있으면 같은 트랜잭션에 Add 및 Delete 메소드

를 포함할 수 있습니다. 따라서 전체 Transfer가 작동하거나 이전 상태로 롤백될 수 있습니다. 트랜잭션은 여러 데이터베이스를 액세스해야 하는 애플리케이션의 오류 복구를 간단하게 합니다.

트랜잭션은 다음 사항을 보증합니다.

- 단일 트랜잭션의 모든 업데이트가 커밋되거나 중지되고 이전 상태로 롤백됩니다. 이것을 **원자성 (atomicity)**이라고 합니다.
- 트랜잭션은 상태 불변 값을 보존하는 시스템 상태의 합당한 변환입니다. 이것을 **일관성 (consistency)**이라고 합니다.
- 병행 트랜잭션은 서로의 부분적인 결과 또는 커밋되지 않은 결과를 모르므로 애플리케이션 상태에서 비일관성을 만들 수 있습니다. 이것을 **분리 (isolation)**라고 합니다. 리소스 관리자는 트랜잭션 기반 동기화 프로토콜을 사용하여 활성 트랜잭션의 커밋되지 않은 작업을 분리합니다.
- 데이터베이스 레코드처럼 관리되는 리소스로 커밋된 업데이트에는 통신 오류, 프로세스 오류 및 서버 시스템 오류를 포함하는 오류가 남아 있습니다. 이것을 **지속성 (durability)**이라고 합니다. 디스크 매체 실패 후 트랜잭션 로그를 사용하여 지속 상태를 복구합니다.

객체와 연결된 컨텍스트 객체는 객체가 트랜잭션 내에서 실행되는지 나타내며 실행되는 경우에는 트랜잭션의 ID를 나타냅니다. 객체가 트랜잭션의 일부이면 리소스 관리자와 리소스 디스펜서가 대신 수행하는 서비스도 트랜잭션에서 실행됩니다. 리소스 디스펜서는 컨텍스트 객체를 사용하여 트랜잭션 기반 서비스를 제공합니다. 예를 들어, 트랜잭션 내에서 실행되는 객체가 ADO 또는 BDE 리소스 디스펜서를 사용하여 데이터베이스 연결을 할당하면 연결은 자동으로 트랜잭션에 참여합니다. 이러한 연결을 사용하는 모든 데이터베이스 업데이트는 트랜잭션의 일부가 되어 커밋되거나 중지됩니다.

여러 객체의 작업을 단일 트랜잭션으로 구성할 수 있습니다. 객체가 고유한 트랜잭션에 있거나 단일 트랜잭션에 속하는 더 큰 객체 그룹의 일부가 되게 하는 것은 MTS 또는 COM+의 주요 이점입니다. 객체가 다양한 방식으로 사용할 수 있게 하여 애플리케이션 개발자는 애플리케이션 로직을 다시 작성하지 않고 다른 애플리케이션에서 애플리케이션 코드를 다시 사용할 수 있습니다. 실제로 개발자는 트랜잭션 객체를 설치할 때 트랜잭션에서 객체가 사용되는 방법을 결정할 수 있습니다. 개발자는 다른 MTS 패키지 또는 COM+ 애플리케이션에 객체를 추가하기만 하면 트랜잭션 동작을 변경할 수 있습니다. 트랜잭션 객체 설치에 대한 자세한 내용은 39-22 페이지의 "트랜잭션 객체 설치"를 참조하십시오.

트랜잭션 속성

모든 트랜잭션 객체에는 MTS 카탈로그에 기록되거나 COM+와 함께 등록된 트랜잭션 속성이 있습니다.

Delphi를 사용하면 Transactional Object 마법사나 Type Library 에디터를 사용하여 디자인 타임에 트랜잭션 속성을 설정할 수 있습니다.

각 트랜잭션 속성은 다음과 같이 설정할 수 있습니다.

- 트랜잭션 요청** 객체는 *트랜잭션의 범위 내에서* 실행되어야 합니다. 새 객체를 만들면 객체의 컨텍스트는 클라이언트의 컨텍스트로부터 트랜잭션을 상속합니다. 클라이언트에 트랜잭션 컨텍스트가 없으면 새 트랜잭션 컨텍스트가 자동으로 만들어집니다.
- 새 트랜잭션 요청** 객체는 *고유 트랜잭션 내에서* 실행되어야 합니다. 새 객체를 만들면 클라이언트에 트랜잭션이 있는지 여부에 관계 없이 그 객체에 대한 새 트랜잭션이 자동으로 만들어집니다. 객체는 클라이언트의 트랜잭션 범위 내에서 실행되지 않습니다. 그 대신 시스템은 항상 새 객체에 대해 독립된 트랜잭션을 만듭니다.
- 트랜잭션 지원** 객체는 *클라이언트의 트랜잭션 범위 내에서* 실행될 수 있습니다. 새 객체를 만들면 객체의 컨텍스트는 클라이언트의 컨텍스트로부터 트랜잭션을 상속합니다. 이렇게 하면 여러 객체를 단일 트랜잭션으로 구성할 수 있습니다. 클라이언트에 트랜잭션이 없으면 새 컨텍스트도 트랜잭션 없이 만들어집니다.
- 트랜잭션 무시** 객체는 *트랜잭션의 범위 내에서 실행되지 않습니다*. 새 객체를 만들면 클라이언트에 트랜잭션이 있는지 여부에 관계 없이 객체 컨텍스트는 트랜잭션 없이 만들어집니다. 이 설정은 COM+에서만 사용할 수 있습니다.
- 트랜잭션 지원
하지 않음** 객체를 MTS에서 설치하는지 COM+에서 설치하는지 여부에 따라 이 설정의 의미는 다양합니다. MTS에서 이 설정은 COM+의 트랜잭션 무시와 같은 의미입니다. COM+에서는 트랜잭션 없이 만들어지는 객체 컨텍스트일 뿐만 아니라 클라이언트에 트랜잭션이 있는 경우 이 설정은 객체가 활성화되지 않게 합니다.

트랜잭션 속성 설정

Transactional Object 마법사를 사용하여 처음으로 트랜잭션 객체를 만들 때 트랜잭션 속성을 설정할 수 있습니다.

또한 Type Library 에디터를 사용하여 트랜잭션 속성을 설정하거나 변경할 수 있습니다. 다음과 같은 방법으로 Type Library 에디터에서 트랜잭션 속성을 변경합니다.

- 1 View|Type Library를 선택하여 Type Library 에디터를 엽니다.
- 2 트랜잭션 객체에 해당하는 클래스를 선택합니다.
- 3 COM+ 탭을 클릭하고 원하는 트랜잭션 속성을 선택합니다.

경고 트랜잭션 속성을 설정하면 Delphi는 지정한 속성의 특수 GUID를 타입 라이브러리의 사용자 지정 데이터로 삽입합니다. 이 값은 Delphi 외부에서는 인식되지 않습니다. 따라서 이 값은 IDE에서 트랜잭션 객체를 설치한 경우에만 효과가 있습니다. 그렇지 않으면 시스템 관리자는 MTS Explorer나 COM+ Component Manager를 사용하여 이 값을 설정해야 합니다.

참고 트랜잭션 객체가 이미 설치되어 있으면 먼저 객체를 제거하고 트랜잭션 속성을 변경할 때 다시 설치해야 합니다. 이렇게 하려면 Run|Install MTS objects 또는 Run|Install COM+ objects를 사용합니다.

상태 있는(stateful) 객체 및 상태 없는(stateless) 객체

COM 객체처럼 트랜잭션 객체는 클라이언트와의 여러 상호 작용에서 내부 상태를 유지 관리할 수 있습니다. 예를 들어, 클라이언트는 한 번 호출로 속성 값을 설정하고 다음에 호출할 때 그 속성 값이 변경되지 않고 유지하도록 합니다. 이러한 객체를 **상태 있음(stateful)**이라고 합니다. 트랜잭션 객체는 **상태 없음(stateless)**이 될 수도 있는데 이것은 객체가 클라이언트의 다음 호출을 기다리는 동안 중간 상태를 유지하지 않는다는 의미입니다.

트랜잭션이 커밋되거나 중지되면 트랜잭션과 관련된 모든 객체는 비활성화되고 트랜잭션 중에 얻은 상태를 잃게 됩니다. 이것은 트랜잭션 분리와 데이터베이스 일관성을 보장 하며 다른 트랜잭션에서 사용하도록 서버 리소스를 해제합니다. 트랜잭션을 완료하면 객체에서 보유한 리소스는 객체가 비활성화될 때 수정됩니다. 객체의 상태를 해제하는 시기를 제어하는 방법에 대해서는 다음 단원을 참조하십시오.

객체의 상태를 유지 관리하려면 객체를 활성화 상태로 유지하고 데이터베이스 연결처럼 잠재적으로 가치있는 리소스를 보유해야 합니다.

트랜잭션의 완료 방법에 대한 영향

트랜잭션 객체는 다음 표에 표시된 대로 *IObjectContext* 메소드를 사용하여 트랜잭션이 완료되는 방법에 영향을 줍니다. 객체의 트랜잭션 속성과 함께 이 메소드를 사용하면 하나 이상의 객체를 단일 트랜잭션에 참여시킬 수 있습니다.

표 39.1 트랜잭션 지원을 위한 IObjectContext 메소드

메소드	설명
SetComplete	객체가 트랜잭션에 대한 작업을 성공적으로 완료했는지 나타냅니다. 객체는 컨텍스트를 처음 입력한 메소드의 반환 시 비활성화됩니다. 객체는 객체 실행을 요구하는 그 다음 호출 시 재활성화됩니다.
SetAbort	객체의 작업이 커밋될 수 없는지와 트랜잭션이 롤백되어야 하는지 나타냅니다. 객체는 컨텍스트를 처음 입력한 메소드의 반환 시 비활성화됩니다. 객체는 객체 실행을 요구하는 그 다음 호출 시 재활성화됩니다.

표 39.1 트랜잭션 지원을 위한 IObjectContext 메소드 (계속)

메소드	설명
EnableCommit	<p>객체의 작업이 반드시 완료되지는 않지만 트랜잭션 업데이트는 현재의 형식으로 커밋될 수 있음을 나타냅니다. 이 메소드를 사용하여 클라이언트로부터 여러 호출 사이에서 상태를 유지하고 트랜잭션을 완료할 수 있습니다. 객체는 SetComplete 또는 SetAbort를 호출할 때까지 비활성화되지 않습니다.</p> <p>EnableCommit은 객체가 활성화될 때의 기본 상태입니다. 이것은 클라이언트에서 다음에 호출할 때까지 객체가 내부 상태를 유지하지 않게 하려는 경우 <i>메소드에서 반환되기 전에 항상 SetComplete 또는 SetAbort를 호출하는 이유</i>입니다.</p>
DisableCommit	<p>객체의 작업이 일관되지 않고 클라이언트로부터 메소드 호출을 더 받을 때까지 작업을 완료할 수 없음을 나타냅니다. 클라이언트에 컨트롤을 반환하여 현재의 트랜잭션이 활성화되어 있는 동안 여러 클라이언트 호출 사이에서 상태를 유지하려면 이 메소드를 호출합니다.</p> <p>DisableCommit은 객체가 비활성화되고 메소드 호출에 대한 반환으로 리소스를 해제하지 않도록 합니다. 객체에서 DisableCommit을 호출한 경우 객체가 EnableCommit 또는 SetComplete를 호출하기 전에 클라이언트가 트랜잭션을 커밋하려고 하면 트랜잭션이 중지됩니다.</p>

트랜잭션 초기화

다음 세 가지 방법으로 트랜잭션을 제어할 수 있습니다.

- 클라이언트에서 제어할 수 있습니다.

클라이언트는 *ITransactionContext* 인터페이스를 사용하는 트랜잭션 컨텍스트 객체를 사용하여 트랜잭션을 직접 제어할 수 있습니다.

- 서버에서 제어할 수 있습니다.

서버는 객체 컨텍스트를 만들어 명시적으로 트랜잭션을 제어할 수 있습니다. 서버가 이런 방법으로 객체를 만들면 만들어진 객체는 현재의 트랜잭션에 자동으로 참여합니다.

- 트랜잭션은 객체의 트랜잭션 속성의 결과로 자동 발생할 수 있습니다.

객체가 만들어진 방법에 관계 없이 그 객체가 항상 트랜잭션 내에서 실행되도록 트랜잭션 객체를 선언할 수 있습니다. 따라서 객체는 트랜잭션을 처리하기 위한 모든 로직을 포함할 필요가 없습니다. 또한 이 기능은 클라이언트 애플리케이션의 부담을 줄여 줍니다. 클라이언트가 사용하는 컴포넌트에 트랜잭션이 필요하기 때문에 클라이언트는 트랜잭션을 초기화할 필요가 없습니다.

클라이언트측에 트랜잭션 객체 설치

클라이언트 기반 애플리케이션은 *ITransactionContextEx* 인터페이스를 통해 트랜잭션 컨텍스트를 제어할 수 있습니다. 다음 코드 예제는 클라이언트 애플리케이션이 *CreateTransactionContextEx*를 사용하여 트랜잭션 컨텍스트를 만드는 방법을 보여줍니다. 이 메소드는 이 객체로 인터페이스를 반환합니다.

다음 예제에서는 *IObjectContext*의 메소드가 Windows에서 직접 노출되어 **safecall**로 선언되지 않기 때문에 반드시 필요한 OleCheck에 대한 호출에서 트랜잭션 컨텍스트에 대한 호출을 사용합니다.

```

procedure TForm1.MoveMoneyClick(Sender:TObject);
begin
    Transfer(CLASS_AccountA, CLASS_AccountB, 100);
end;
procedure TForm1.Transfer(DebitAccountId, CreditAccountId: TGuid; Amount:Currency);
var
    TransactionContextEx: ITransactionContextEx;
    CreditAccountIntf, DebitAccountIntf:IAccount;
begin
    TransactionContextEx := CreateTransactionContextEx;
    try
        OleCheck(TransactionContextEx.CreateInstance(DebitAccountId,
            IAccount, DebitAccountIntf));
        OleCheck(TransactionContextEx.CreateInstance(CreditAccountId,
            IAccount, CreditAccountIntf));
        DebitAccountIntf.Debit(Amount);
        CreditAccountIntf.Credit(Amount);
    except
        TransactionContextEx.Abort;
        raise;
    end;
    TransactionContextEx.Commit;
end;

```

서버측에 트랜잭션 객체 설치

서버측에서 트랜잭션 컨텍스트를 제어하려면 *ObjectContext*의 인스턴스를 만듭니다. 다음 예제에서 Transfer 메소드는 트랜잭션 객체에 있습니다. 이런 방법으로 ObjectContext를 사용할 때 만들어지는 객체의 인스턴스는 그 인스턴스를 만드는 객체의 모든 트랜잭션 속성을 상속하게 됩니다. *IObjectContext*의 메소드가 Windows에서 직접 노출되어 **safecall**로 선언되지 않기 때문에 OleCheck에 대한 호출을 사용합니다.

```

procedure TAccountTransfer.Transfer(DebitAccountId, CreditAccountId: TGuid;
    AmountCurrency);
var
    CreditAccountIntf, DebitAccountIntf:IAccount;
begin
    try
        OleCheck(ObjectContext.CreateInstance(DebitAccountId,
            IAccount, DebitAccountIntf));
        OleCheck(ObjectContext.CreateInstance(CreditAccountId,
            IAccount, CreditAccountIntf));
        DebitAccountIntf.Debit(Amount);
        CreditAccountIntf.Credit(Amount);
    except
        DisableCommit;
        raise;
    end;
    EnableCommit;
end;

```

트랜잭션 시간 초과

트랜잭션 시간 초과는 트랜잭션이 활성화 상태로 유지되는 시간(초 단위)을 설정합니다. 시스템은 시간 초과가 경과한 후에도 활성화되어 있는 트랜잭션을 자동으로 중지합니다. 시간 초과 기본값은 60초입니다. 값을 0으로 지정하면 트랜잭션 시간 초과를 사용할 수 없게 하므로 트랜잭션 객체를 디버깅할 때 유용합니다.

다음과 같은 방법으로 컴퓨터에서 시간 초과 값을 설정합니다.

- 1 MTS Explorer 또는 COM+ Component Manager에서 Computer, My Computer를 선택합니다.

기본적으로 My Computer는 로컬 컴퓨터에 해당합니다.

- 2 Properties를 마우스 오른쪽 버튼으로 클릭하여 선택한 다음 Options 탭을 선택합니다.

Options 탭은 컴퓨터의 트랜잭션 시간 초과 속성을 설정할 때 사용됩니다.

- 3 트랜잭션 시간 초과를 사용할 수 없게 하려면 시간 초과 값을 0으로 변경합니다.

- 4 OK를 클릭하여 설정을 저장합니다.

MTS 애플리케이션 디버깅에 대한 자세한 내용은 39-21 페이지의 "트랜잭션 객체 디버깅 및 테스트"를 참조하십시오.

역할 기반 보안(Role-based security)

MTS 및 COM+는 역할 기반 보안을 제공하고 여기서 사용자의 논리 그룹에 역할을 할당합니다. 예를 들어, 의학 정보 애플리케이션은 의사, X선 전문가 및 환자에 대한 역할을 정의할 수 있습니다.

역할을 할당하여 각 객체 및 인터페이스에 대한 승인을 정의합니다. 예를 들어, 의사의 의학 애플리케이션에서는 의사만 모든 진료 기록을 볼 수 있도록 승인받을 수 있으며 X선 전문가는 X선만 보고 환자는 자신의 진료 기록만 볼 수 있습니다.

일반적으로 애플리케이션을 개발하는 동안 역할을 정의하고 각 MTS 패키지나 COM+ 애플리케이션에 역할을 할당합니다. 그런 다음 애플리케이션을 배포할 때 이 역할을 특정 사용자에게 할당합니다. 관리자는 MTS Explorer나 COM+ Component Manager를 사용하여 역할을 구성할 수 있습니다.

전체 객체가 아닌 코드 블록에 대한 액세스를 제어하려면 *IObjectContext* 메소드, *IsCallerInRole*을 사용하여 더 섬세한 보안을 제공할 수 있습니다. 보안을 사용하면 이 메소드만 작동하고 *IObjectContext* 메소드 *IsSecurityEnabled*를 호출하여 확인할 수 있습니다. 이 메소드는 자동으로 트랜잭션 객체에 메소드로 추가됩니다. 예를 들면 다음과 같습니다.

```
if IsSecurityEnabled then {check if security is enabled }
begin
  if IsCallerInRole(@Physician1) then { check caller's role }
  begin
    { execute the call normally }
```

```

end
else
  { not a physician, do something appropriate }
end
end
else
  { no security enabled, do something appropriate }
end;

```

참고 더 강력한 보안이 필요한 애플리케이션을 위해 컨텍스트 객체는 *ISecurityProperty* 인터페이스를 구현하며 그 메소드를 사용하면 객체의 직접 호출자 및 작성자의 Windows 보안 식별자(SID)뿐만 아니라 객체를 사용하는 클라이언트의 SID를 검색할 수 있습니다.

트랜잭션 객체 생성에 대한 개요

트랜잭션 객체를 만드는 프로세스는 다음과 같습니다.

- 1 Transactional Object 마법사를 사용하여 트랜잭션 객체를 만듭니다.
- 2 Type Library 에디터를 사용하여 객체의 인터페이스에 메소드 및 속성을 추가합니다. Type Library 에디터를 사용하여 메소드 및 속성을 추가하는 것에 대한 자세한 내용은 34장 "Type Library 작업"을 참조하십시오.
- 3 객체의 메소드를 구현할 때 *IObjectContext* 인터페이스를 사용하면 트랜잭션, 영구적 상태 및 보안을 관리할 수 있습니다. 또한 객체 참조를 전달하는 경우에는 제대로 처리되도록 특별히 주의해야 합니다(39-20 페이지의 "객체 참조 전달" 참조).
- 4 트랜잭션 객체를 디버깅하고 테스트합니다.
- 5 트랜잭션 객체를 MTS 패키지나 COM+ 애플리케이션으로 설치합니다.
- 6 MTS Explorer나 COM+ Component Manager를 사용하여 객체를 관리합니다.

Transactional Object 마법사 사용

Transactional Object 마법사를 사용하여 MTS나 COM+에서 제공하는 리소스 관리, 트랜잭션 처리 및 역할 기반 보안을 이용할 수 있는 COM 객체를 만듭니다.

다음과 같은 방법으로 Transactional Object 마법사를 시작합니다.

- 1 File|New|Other를 선택합니다.
- 2 Multitier 탭을 선택합니다.
- 3 Transactional Object 아이콘을 더블 클릭합니다.

마법사에서 다음을 지정해야 합니다.

- 클라이언트 애플리케이션이 객체의 인터페이스를 호출할 수 있는 방법을 나타내는 스레드 모델. 스레드 모델은 객체가 등록되는 방법을 결정합니다. 객체 구현이 선택한 모델을 사용하는지 확인해야 합니다. 스레드 모델에 대한 자세한 내용은 39-17 페이지의 "트랜잭션 객체용 스레드 모델 선택"을 참조하십시오.

- 트랜잭션 모델
- 객체가 클라이언트에 이벤트를 알려 주는지 여부를 나타냅니다. 이벤트 지원은 COM+ 이벤트가 아닌 전통적인 이벤트에만 제공됩니다.

이 프로시저를 완료하면 트랜잭션 객체에 대한 정의를 포함하는 새로운 유닛이 현재 프로젝트에 추가됩니다. 또한 마법사는 타입 라이브러리를 프로젝트에 추가하고 Type Library 에디터로 엽니다. 이제 타입 라이브러리를 통해 인터페이스의 속성 및 메소드를 노출할 수 있습니다. 36-9 페이지의 "COM 객체의 인터페이스 정의"에서 설명한 바와 같이 모든 COM 객체를 정의할 때처럼 인터페이스를 정의합니다.

트랜잭션 객체는 *vtable*을 통한 우선 바인딩(컴파일 시)과 *IDispatch* 인터페이스를 통한 지연 바인딩(런타임 시)을 모두 지원하는 **이중 인터페이스**를 구현합니다.

생성된 트랜잭션 객체는 *IObjectControl* 인터페이스 메소드, *Activate*, *Deactivate* 및 *CanBePooled*를 구현합니다.

Transactional Object 마법사를 반드시 사용할 필요는 없습니다. Type Library 에디터의 COM+ 페이지를 사용한 다음 객체를 MTS 패키지나 COM+ 애플리케이션에 설치하면 모든 Automation 객체를 COM+ 트랜잭션 객체로 변환하고 모든 in-process Automation 객체는 MTS 트랜잭션 객체로 변환할 수 있습니다. 그러나 Transactional Object 마법사는 다음과 같은 몇 가지 이점을 제공합니다.

- 마법사는 자동으로 *IObjectControl* 인터페이스를 구현하고 객체가 활성화되거나 비활성화될 때 응답하는 이벤트 핸들러를 만들 수 있도록 *OnActivate*와 *OnDeactivate* 이벤트를 객체에 추가합니다.
- 마법사는 객체가 *IObjectContext* 메소드에 액세스하여 활성화와 트랜잭션을 쉽게 제어할 수 있도록 자동으로 *ObjectContext* 속성을 생성합니다.

트랜잭션 객체용 스레드 모델 선택

MTS 런타임 환경이나 COM+는 스레드를 관리합니다. 트랜잭션 객체는 스레드를 만들어서는 안 됩니다. 또한 DLL로 호출되는 스레드를 종료해서는 안 됩니다.

Transactional Object 마법사를 사용하여 스레드 모델을 지정할 때는 메소드 실행용 스레드에 객체를 할당하는 방법을 지정합니다.

표 39.2 트랜잭션 객체용 스레드 모델

스레드 모델	설명	pros 및 cons 구현
Single	스레드를 지원하지 않습니다. 클라이언트 요청은 호출 메커니즘으로 serialize됩니다. 단일 스레드 컴포넌트의 모든 객체는 주 스레드에서 실행됩니다. Threading Model Registry 속성이 없는 컴포넌트나 재입력할 수 없는 COM 컴포넌트에 사용되는 기본 COM 스레드 모델과 호환됩니다. 메소드 실행은 컴포넌트의 모든 객체와 한 프로세스의 모든 컴포넌트에서 연속적으로 이루어집니다.	컴포넌트가 재입력할 수 없는 라이브러리를 사용할 수 있게 합니다. 확장성이 매우 제한됩니다. 단일 스레드, 상태 있는 (stateful) 컴포넌트는 교착 상태에 빠지기 쉽습니다. 상태 없는 (stateless) 객체를 사용하고 메소드에서 반환되기 전에 SetComplete를 호출하여 이러한 문제를 해결할 수 있습니다.
Apartment (또는 Single-Threaded apartment)	각 객체는 그 객체의 수명 기간 동안 지속되는 독립 스레드에 할당되지만 여러 객체에는 여러 스레드를 사용할 수 있습니다. 이것은 표준 COM 동시성 모델입니다. 각 구분은 특정 스레드에 연결되고 Windows 메시지 펌프가 있습니다.	단일 스레드 모델에서 중요한 동시성을 향상시킵니다. 두 객체가 같은 활동에 있지 않으면 동시에 실행할 수 있습니다. 여러 프로세스에 객체를 배포할 수 있다는 점을 제외하면 COM 구분과 유사합니다.

참고 이 스레드 모델은 COM 객체에서 정의한 모델과 유사합니다. 그러나 MTS와 COM+가 더 우선적으로 스레드를 지원하기 때문에 여기서 각 스레드 모델의 의미는 달라집니다. 또한 Free 스레드 모델은 작업에 대한 기본 제공 지원으로 인해 트랜잭션 객체에 적용되지 않습니다.

활동(Activities)

스레드 모델뿐만 아니라 트랜잭션 객체도 **활동**을 통해 동시성을 얻습니다. 활동은 객체의 컨텍스트로 기록되고 객체와 활동 사이의 연결은 변경할 수 없습니다. 활동에는 객체와 그 자손으로 만든 트랜잭션 객체뿐만 아니라 기본 클라이언트로 만든 트랜잭션 객체가 포함됩니다. 이 객체는 하나 이상의 프로세스에 배포할 수 있으며 하나 이상의 컴퓨터에서 실행됩니다.

예를 들어, 의사의 의학 애플리케이션에는 다양한 의학 데이터베이스에 업데이트를 추가하고 레코드를 제거하는 트랜잭션 객체가 있을 수 있으며 각각은 다른 객체로 표시됩니다. 이 레코드 객체는 트랜잭션을 기록하는 수신 객체와 같은 다른 객체도 사용할 수

있습니다. 따라서 기본 클라이언트가 직접적으로나 간접적으로 제어하는 여러 트랜잭션 객체가 됩니다. 이 객체는 모두 같은 활동에 속합니다.

MTS나 COM+는 각 활동을 통해 실행의 흐름을 추적하여 무의식적인 병렬 처리로 애플리케이션 상태가 손상되지 않게 합니다. 이 기능으로 인해 잠재적으로 배포된 객체 컬렉션 전체에 실행의 단일 논리 스레드가 발생합니다. 하나의 논리 스레드가 있으면 애플리케이션은 작성하기가 매우 쉽습니다.

기존 컨텍스트에서 트랜잭션 객체를 만들면 트랜잭션 컨텍스트 객체나 객체 컨텍스트를 사용하여 새 객체는 같은 활동의 멤버가 됩니다. 즉, 새 컨텍스트는 컨텍스트를 만드는 데 사용된 컨텍스트의 활동 식별자를 상속합니다.

실행의 단일 논리 스레드 하나만 한 활동 내에서 허용됩니다. 객체를 여러 프로세스에 배포할 수 있다는 점을 제외하면 이것은 COM 독립 스레드 모델에 대한 동작과 유사합니다. 기본 클라이언트가 활동으로 호출되면 다른 클라이언트 스레드에서처럼 활동에 대한 다른 모든 작업 요청은 실행의 초기 스레드가 클라이언트로 다시 돌아올 때까지 차단됩니다.

MTS에서는 모든 트랜잭션 객체가 하나의 활동에 속합니다. COM+에서는 **호출 동기화 (call synchronization)**를 설정하여 객체가 활동에 참여하는 방법을 구성할 수 있습니다. 다음과 같은 옵션을 사용할 수 있습니다.

표 39.3 호출 동기화 옵션

옵션	의미
사용 불가능	COM+는 객체에 활동을 할당하지 않지만 호출자의 컨텍스트와 함께 상속할 수 있습니다. 호출자에 트랜잭션이나 객체 컨텍스트가 없으면 객체는 활동에 할당되지 않습니다. 결과는 객체가 COM+ 애플리케이션에 설치되는 않은 경우와 같습니다. 이 옵션은 애플리케이션의 객체가 리소스 관리자를 사용하거나 객체가 트랜잭션이나 just-in-time 활성을 지원하는 경우에는 사용할 수 없습니다.
지원하지 않음	COM+는 호출자의 상태에 관계 없이 활동에 객체를 할당하지 않습니다. 이 옵션은 애플리케이션의 객체가 리소스 관리자를 사용하거나 객체가 트랜잭션이나 just-in-time 활성을 지원하는 경우에는 사용할 수 없습니다.
지원함	COM+는 호출자와 같은 활동에 객체를 할당합니다. 호출자가 활동에 속하지 않으면 객체도 활동에 속하지 않습니다. 이 옵션은 애플리케이션의 객체가 리소스 관리자를 사용하거나 객체가 트랜잭션이나 just-in-time 활성을 지원하는 경우에는 사용할 수 없습니다.
필수	COM+는 항상 활동에 객체를 할당하며 필요하면 만듭니다. 이 옵션은 트랜잭션 속성이 Supported 또는 Required인 경우에 사용해야 합니다.
새로 필요	COM+는 항상 호출자의 활동과 구분되는 새로운 활동에 객체를 할당합니다.

COM+에서 이벤트 생성

COM+ 이전에 Automation 서버는 이벤트를 생성하는 특수한 인터페이스 집합을 사용했습니다. 그러나 COM+는 이벤트를 관리하는 새로운 시스템을 도입했습니다. 이벤트를 관리하는 서버 객체 대신 이벤트가 발생할 때 인터페이스를 호출하고 알려 줄 클라이언트를 추적하며 기본 시스템(COM+)에서 이 프로세스를 관리합니다.

참고 COM+에 설치된 트랜잭션 객체는 이벤트를 관리하는 데 여전히 이전 시스템을 사용합니다. 그러나 COM+를 사용하여 프로세스를 처리하면 융통성이 향상됩니다. 예를 들어, COM+가 이벤트를 관리하면 클라이언트는 이벤트가 발생할 때 COM+에서 시작하는 in-process 서버가 될 수 있습니다.

COM+ 객체가 이벤트를 생성할 때는 직접 생성하지 않고 이벤트를 생성하도록 특수하게 만들어진 관련 이벤트 객체를 사용합니다. COM+ 객체는 이벤트를 발생하려고 할 때 그 이벤트 객체를 호출합니다. 이런 경우 COM+는 특정 이벤트 객체에 등록된 모든 클라이언트를 호출합니다.

Event Object 마법사 사용

Event Object 마법사를 사용하여 이벤트 객체를 만들 수 있습니다. COM+ 이벤트 객체를 포함하는 프로젝트에는 구현이 포함되지 않기 때문에 마법사는 먼저 현재 프로젝트에 구현 코드가 포함되어 있는지 확인합니다. 이벤트 객체 정의만 포함할 수 있습니다. 그러나 단일 프로젝트에 여러 COM+ 이벤트 객체를 포함시킬 수 있습니다.

다음과 같은 방법으로 Event Object 마법사를 실행합니다.

- 1 File|New를 선택합니다.
- 2 ActiveX 탭을 선택합니다.
- 3 COM+ Event Object 아이콘을 더블 클릭합니다.

Event Object 마법사에서 이벤트 객체의 이름, 이벤트 핸들러를 정의하는 인터페이스 이름 및 경우에 따라 이벤트에 대한 간단한 설명을 지정합니다.

마법사를 종료하면 이벤트 객체와 그 인터페이스를 정의하는 타입 라이브러리가 포함된 프로젝트가 만들어집니다. Type Library 에디터를 사용하여 해당 인터페이스의 메소드를 정의합니다. 이 메소드는 클라이언트가 이벤트에 응답할 때 구현하는 이벤트 핸들러입니다.

이벤트 객체 프로젝트에는 프로젝트 파일, ATL 템플릿 클래스를 가져오는 _ATL 유닛 및 타입 라이브러리 정보를 정의하는 _TLB 유닛이 포함됩니다. 그러나 COM+ 이벤트 객체에 구현이 없기 때문에 구현 유닛은 포함하지 않습니다. 인터페이스의 구현은 클라이언트가 담당합니다. 서버 객체가 COM+ 이벤트 객체를 호출하면 COM+는 호출을 인터셉트하여 등록된 클라이언트로 디스패치합니다. COM+ 이벤트 객체에는 구현 객체가 필요하지 않기 때문에 Type Library 에디터에서 객체의 인터페이스를 정의한 후에는 프로젝트를 컴파일하고 COM+와 함께 설치하면 됩니다.

COM+는 이벤트 객체의 인터페이스에 몇 가지 제한을 둡니다. Type Library 에디터에서 이벤트 객체에 대해 정의한 인터페이스는 다음과 같은 규칙을 준수해야 합니다.

- 이벤트 객체의 인터페이스는 IDispatch에서 파생되어야 합니다.
- 모든 메소드 이름은 이벤트 객체의 모든 인터페이스에서 고유해야 합니다.
- 이벤트 객체의 인터페이스에 있는 모든 메소드는 HRESULT 값을 반환해야 합니다.
- 메소드의 모든 매개변수에 대한 변경자는 공백이어야 합니다.

COM+ 이벤트 객체를 사용하여 이벤트 발생

이벤트가 발생하면 COM+ 객체는 이벤트 객체를 호출하고 등록된 클라이언트에서 이벤트를 발생하도록 알려 주어야 합니다. 이벤트 객체의 인스턴스를 만들고 이벤트에 해당하는 메소드를 호출하여 이 작업을 수행합니다.

참고 이벤트 객체 자체처럼 COM+ 이벤트를 발생시키는 객체는 COM+ 애플리케이션에 설치되어야 합니다.

객체 참조 전달

참고 객체 참조를 전달할 때의 정보는 COM+가 아닌 MTS에만 적용됩니다. MTS에서 실행되는 객체에 대한 모든 포인터는 인터셉터를 통해 라우트되어야 하기 때문에 이 메커니즘은 MTS에서 필요합니다. COM+에서는 인터셉터가 기본 제공되기 때문에 객체 참조를 전달하지 않아도 됩니다.

예를 들어, 콜백으로 사용하는 경우처럼 MTS에서는 다음과 같은 방법만으로 객체 참조를 전달할 수 있습니다.

- *CoCreateInstance* 또는 그와 동등한 인터페이스, *ITransactionContext.CreateInstance* 또는 *IObjectContext.CreateInstance*처럼 객체 생성 인터페이스에서의 반환을 통해 전달합니다.
- *QueryInterface*에 대한 호출을 통해 전달합니다.
- 객체 참조를 얻기 위해 *SafeRef*를 호출한 메소드를 통해 전달합니다.

위의 방법으로 얻은 객체 참조는 **safe reference**라고 합니다. safe reference를 사용하여 호출된 메소드는 올바른 컨텍스트 내에서 실행되도록 보장됩니다.

MTS 런타임 환경에서는 컨텍스트 전환을 관리하고 트랜잭션 객체가 클라이언트 참조와 독립된 수명을 가질 수 있도록 safe reference를 사용하기 위한 호출을 해야 합니다. COM+에서는 safe reference가 반드시 필요하지는 않습니다.

SafeRef 메소드 사용

객체는 *SafeRef* 함수를 사용하여 컨텍스트 밖으로 전달하기에 안전한, 객체 자체에 대한 참조를 얻을 수 있습니다. *SafeRef* 함수를 정의하는 유닛은 Mtx입니다.

입력으로 받은 *SafeRef*입니다.

- 현재의 객체가 다른 객체나 클라이언트로 전달하려는 인터페이스의 인터페이스 ID(RIID)에 대한 참조입니다.
- 현재 객체의 IUnknown 인터페이스에 대한 참조입니다.

*SafeRef*는 현재 객체의 컨텍스트 밖으로 전달하기에 안전한 RIID 매개변수에 지정된 인터페이스에 대한 포인터를 반환합니다. 객체가 그 자체가 아닌 다른 객체에서 safe reference를 요청하거나 RIID 매개변수에 요청된 인터페이스가 구현되지 않으면 **nil**을 반환합니다.

예를 들어, 콜백으로 사용하는 경우처럼 MTS 객체가 클라이언트나 다른 객체로 자기 참조를 전달하려고 할 때는 항상 *SafeRef*를 먼저 호출한 다음 이 호출에서 반환하는 참조를 전달해야 합니다. 객체는 **self** 포인터 또는 *QueryInterface*에 대한 내부 호출을 통해 얻은 자기 참조를 클라이언트나 다른 모든 객체로 전달해서는 안 됩니다. 이런 참조를 객체의 컨텍스트 밖으로 전달하면 더 이상 유효한 참조가 아닙니다.

인터페이스의 참조 카운트가 증가하는 경우를 제외하고 이미 safe인 참조에서 *SafeRef*를 호출하면 변경되지 않은 safe reference를 반환합니다.

클라이언트가 safe한 참조에서 *QueryInterface*를 호출하면 클라이언트로 반환되는 참조 또한 safe reference입니다.

safe reference를 얻는 객체는 완료할 때 safe reference를 해제해야 합니다.

*SafeRef*에 대한 자세한 내용은 Microsoft 설명서의 *SafeRef* 항목을 참조하십시오.

콜백

객체는 클라이언트와 다른 트랜잭션 객체로 콜백할 수 있습니다. 예를 들어, 다른 객체를 만드는 객체가 있을 수 있습니다. 객체를 만들면 만들어진 객체로 자체에 대한 참조를 전달할 수 있으며 만들어진 객체는 이 참조를 사용하여 만드는 객체를 호출할 수 있습니다.

콜백을 사용하려는 경우에는 다음 제한을 참고하십시오.

- 기본 클라이언트나 다른 패키지로 콜백하면 클라이언트에 액세스 수준의 보안이 필요합니다. 또한 클라이언트는 DCOM 서버여야 합니다.
- 방화벽을 사용하면 클라이언트로 콜백을 차단할 수 있습니다.
- 콜백에서 수행된 작업은 객체가 호출되는 환경에서 실행됩니다. 이것은 같은 트랜잭션의 일부이거나, 다른 트랜잭션이거나, 트랜잭션이 없을 수 있습니다.
- MTS에서는 객체 자체로 콜백하려면 만드는 객체는 *SafeRef*를 호출하고 반환된 참조를 만들어진 객체로 전달해야 합니다.

트랜잭션 객체 디버깅 및 테스트

로컬 및 원격 트랜잭션 객체를 디버그할 수 있습니다. 트랜잭션 객체를 디버깅할 때는 트랜잭션 시간 초과를 선택하지 않아야 합니다.

트랜잭션 시간 초과는 트랜잭션이 활성화 상태로 유지되는 시간(초 단위)을 설정합니다. 시스템은 시간 초과가 경과한 후에도 활성화되어 있는 트랜잭션을 자동으로 중지합니다. 시간 초과 기본값은 60초입니다. 값을 0으로 지정하면 트랜잭션 시간 초과를 사용할 수 없게 하므로 디버깅할 때 유용합니다.

원격 디버깅에 대한 자세한 내용은 온라인 도움말의 "원격 디버깅" 항목을 참조하십시오.

MTS에서 실행할 트랜잭션 객체를 테스트할 때는 먼저 MTS 환경 외부에서 객체를 테스트하여 테스트 환경을 단순하게 해야 합니다.

서버를 개발하는 동안 서버가 여전히 메모리에 남아 있으면 서버를 다시 빌드할 수 없습니다. "Cannot write to DLL while executable is loaded"라는 컴파일러 오류가 발생할 수 있습니다. 이러한 문제를 해결하려면 서버가 유휴 상태일 때는 종료되도록 MTS 패키지나 COM+ 애플리케이션 속성을 설정할 수 있습니다.

다음과 같은 방법으로 유휴 상태일 때 서버를 종료합니다.

- 1 MTS Explorer나 COM+ Component Manager에서 트랜잭션 객체가 설치되어 있는 MTS 패키지나 COM+ 애플리케이션을 마우스 오른쪽 버튼으로 클릭하고 Properties를 선택합니다.
- 2 Advanced 탭을 선택합니다.
Advanced 탭은 패키지와 연결된 서버 프로세스가 항상 실행되는지 특정 시간이 지나면 종료하는지를 결정합니다.
- 3 클라이언트에서 더 이상 서비스를 제공할 수 없게 되면 서버를 종료하도록 시간 초과 값을 0으로 변경합니다.
- 4 OK를 클릭하여 설정을 저장합니다.

참고 MTS 환경 외부에서 테스트할 때는 *TMtsObject*의 *ObjectProperty*를 직접 참조하지 않습니다. *TMtsObject*는 객체 컨텍스트가 **nil**일 때 호출하기에 안전한 *SetComplete* 및 *SetAbort*와 같은 메소드를 구현합니다.

트랜잭션 객체 설치

MTS 애플리케이션은 MTS 실행 파일(EXE)의 단일 인스턴스에서 실행되는 in-process MTS 객체 그룹으로 구성됩니다. 모두 같은 프로세스에서 실행되는 COM 객체 그룹을 **패키지**라고 합니다. 한 시스템에서 여러 개의 다른 패키지를 실행할 수 있으며 각 패키지는 별도의 MTS EXE 내에서 실행됩니다.

COM+에서는 COM+ 애플리케이션이라는 유사한 그룹으로 작업합니다. **COM+ 애플리케이션**에서 객체는 in-process일 필요가 없으며 별도의 런타임 환경이 없습니다.

애플리케이션 컴포넌트를 단일 MTS 패키지나 COM+ 애플리케이션으로 그룹화하여 단일 프로세스로 관리할 수 있습니다. 컴포넌트를 다른 MTS 패키지나 COM+ 애플리케이션으로 배포하여 여러 프로세스나 시스템 사이에서 분할할 수 있습니다.

다음과 같은 방법으로 트랜잭션 객체를 MTS 패키지나 COM+ 애플리케이션으로 설치합니다.

- 1 시스템이 COM+를 지원하면 Run|Install COM+ objects를 선택합니다. 시스템이 COM+를 지원하지 않지만 MTS가 설치되어 있으면 Run|Install MTS objects를 선택합니다. 시스템이 MTS나 COM+를 모두 지원하지 않으면 트랜잭션 객체를 설치할 수 있는 메뉴 항목이 나타나지 않습니다.
- 2 Install Object 대화 상자에서 설치할 객체를 확인합니다.

3 MTS 객체를 설치하려면 Package 버튼을 클릭하여 시스템에 있는 MTS 패키지 목록을 가져옵니다. COM+ 객체를 설치하려면 Application 버튼을 클릭합니다. 객체를 설치할 MTS 패키지나 COM+ 애플리케이션을 나타냅니다. Into New Package 또는 Into New Application을 선택하여 객체를 설치할 MTS 패키지나 COM+ 애플리케이션을 새로 만들 수 있습니다. Into Existing Package 또는 Into Existing Application을 선택하여 나열된 기존의 MTS 패키지나 COM+ 애플리케이션에 객체를 설치할 수 있습니다.

4 OK를 선택하여 카탈로그를 새로 고치고 런타임에 객체를 사용할 수 있게 합니다.

MTS 패키지에는 여러 DLL의 컴포넌트가 포함될 수 있으며 단일 DLL의 컴포넌트는 다른 패키지에 설치될 수 있습니다. 그러나 단일 컴포넌트는 여러 패키지에 배포할 수 없습니다.

마찬가지로 COM+ 애플리케이션에는 여러 실행 파일의 컴포넌트가 포함될 수 있으며 단일 실행 파일의 여러 컴포넌트는 다른 COM+ 애플리케이션에 설치할 수 있습니다.

참고 COM+ Component Manager나 MTS Explorer를 사용하여 트랜잭션 객체를 설치할 수도 있습니다. 이 도구 중 하나를 사용하여 객체를 설치할 때는 Type Library 에디터의 COM+ 페이지에 나타나는 객체의 설정을 적용해야 합니다. IDE에서 설치하지 않으면 이 설정은 자동으로 적용되지 않습니다.

트랜잭션 객체 관리

트랜잭션 객체를 설치하면 MTS Explorer(MTS 패키지로 설치한 경우)나 COM+ Component Manager(COM+ 애플리케이션으로 설치한 경우)를 사용하여 이 런타임 객체를 관리할 수 있습니다. MTS Explorer는 MTS 런타임 환경에서 작동하고 COM+ Component Manager는 COM+ 객체에서 작동한다는 점을 제외하면 두 도구는 동일합니다.

COM+ Component Manager와 MTS Explorer에는 트랜잭션 객체를 배포하고 관리하는 그래픽 사용자 인터페이스가 있습니다. 이 도구 중 하나를 사용하면 다음 작업을 수행할 수 있습니다.

- 트랜잭션 객체, MTS 패키지 또는 COM+ 애플리케이션 및 역할을 구성합니다.
- 패키지나 COM+ 애플리케이션에 있는 컴포넌트의 속성을 보고 컴퓨터에 설치된 MTS 패키지나 COM+ 애플리케이션을 봅니다.
- 트랜잭션을 구성하는 객체의 트랜잭션을 모니터링하고 관리합니다.
- 컴퓨터 간에 MTS 패키지나 COM+ 애플리케이션을 이동합니다.
- 원격 트랜잭션 객체를 로컬 클라이언트에서 사용할 수 있게 합니다.

이 도구에 대한 자세한 내용은 Microsoft에서 제공하는 관련된 *Administrator's Guide*를 참조하십시오.



사용자 지정 컴포넌트 생성

"사용자 지정 컴포넌트 생성"에 포함된 장에서는 Delphi 사용자 지정 컴포넌트를 디자인하고 구현하는 데 필요한 개념을 설명합니다.

40

컴포넌트 생성 개요

이 장에서는 Delphi 애플리케이션을 위한 컴포넌트 디자인 및 컴포넌트 작성 프로세스에 대한 개요를 제공합니다. 여기서는 사용자가 Delphi 및 표준 컴포넌트에 익숙하다고 가정합니다.

- VCL 및 CLX
- 컴포넌트 및 클래스
- 컴포넌트 생성 방법
- 컴포넌트 작성 시 고려 사항
- 새 컴포넌트 생성
- 설치되지 않은 컴포넌트 테스트
- 설치된 컴포넌트 테스트

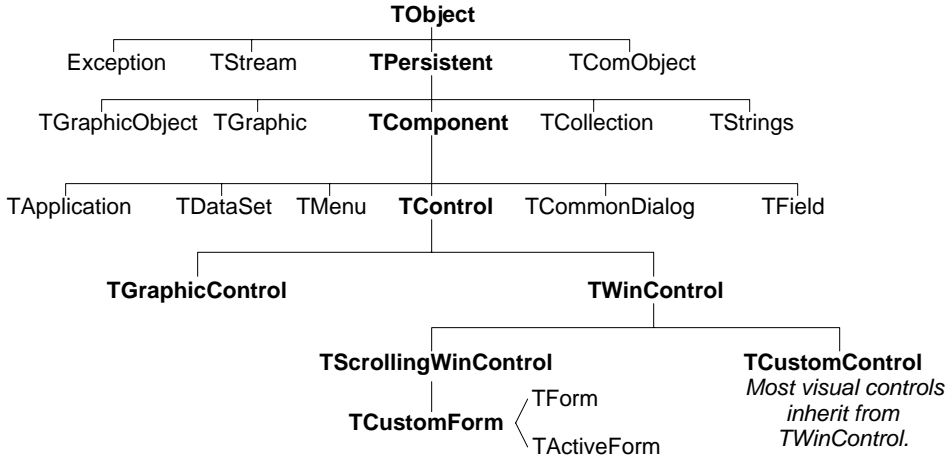
새 컴포넌트 설치에 대한 자세한 내용은 11-5 페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

VCL 및 CLX

Delphi의 컴포넌트는 VCL(Visual Component Library)과 CLX(Component Library for Cross Platform)라고 하는 두 클래스 계층 구조에 상주합니다. 그림 40.1은 VCL을 구성하는 선택된 클래스의 관계를 보여 줍니다. CLX 계층 구조는 VCL과 유사하지만 Windows 컨트롤은 widget이라 하고 (따라서 예를 들어, *TWinControl*은 *TWidgetControl*이라고 함) 그 밖의 다른 차이점이 있습니다. 클래스 계층 구조와 클래스 간의 상속 관계에 대한 자세한 설명은 41장 "컴포넌트 작성자를 위한 객체 지향 프로그래밍"을 참조하십시오. CLX와 VCL의 차이점에 대한 개요는 10-5 페이지의 "CLX와 VCL 비교"를 참조하시고 컴포넌트에 대한 세부 사항은 CLX 온라인 참조를 보십시오.

TComponent 클래스는 VCL과 CLX에 있는 모든 컴포넌트가 공유하는 조상입니다. *TComponent*는 Delphi에서 컴포넌트가 작동하는 데 필요한 최소한의 속성 및 이벤트를 제공합니다. 라이브러리의 다양한 분기는 보다 전문화된 다른 기능을 제공합니다.

그림 40.1 Visual Component Library 클래스 계층 구조



컴포넌트를 만들 때는 계층 구조에 있는 기존 클래스 타입 중 하나에서 새 클래스를 파생시켜 VCL 또는 CLX에 추가합니다.

컴포넌트 및 클래스

컴포넌트가 클래스이기 때문에 컴포넌트 작성자는 애플리케이션 개발자와는 다른 수준에서 객체를 다룹니다. 새 컴포넌트를 생성하려면 새 클래스를 파생시켜야 합니다.

간단히 말해서 컴포넌트를 생성하는 것과 애플리케이션에서 컴포넌트를 사용하는 것 사이에는 두 가지 중요한 차이가 있습니다. 컴포넌트를 생성할 때는 다음을 수행합니다.

- 애플리케이션 프로그래머가 액세스할 수 없는 클래스의 부분에 액세스합니다.
- 속성과 같은 새로운 부분을 컴포넌트에 추가합니다.

이러한 차이점 때문에 더 많은 규칙을 알아야 하고 애플리케이션 개발자가 컴포넌트를 사용하는 방법에 대해 고려해야 합니다.

컴포넌트 생성 방법

컴포넌트는 사용자가 디자인 타임에 조작하고자 하는 거의 모든 프로그램 요소가 될 수 있습니다. 컴포넌트 생성은 기존 클래스에서 새 클래스를 파생시키는 것을 의미합니다. 모든 기존 컴포넌트에서 새 컴포넌트를 파생시킬 수 있지만 일반적으로 다음과 같은 방법으로 컴포넌트를 생성합니다.

- 기존 컨트롤 수정
- 창 있는 컨트롤 생성
- 그래픽 컨트롤 생성
- Windows 컨트롤 하위 분류

- 년비주얼 컴포넌트 생성

표 40.1에는 각 작업에 대해 기본적으로 사용할 각기 다른 종류의 컴포넌트 및 클래스가 요약되어 있습니다.

표 40.1 컴포넌트 생성 기본 작업

수행할 작업	사용할 타입
기본 컴포넌트 수정	<i>TButton</i> 또는 <i>TListBox</i> 와 같은 기존 컴포넌트나 <i>TCustomListBox</i> 와 같은 추상 컴포넌트 타입
창 있는(또는 CLX의 widget 기반) 컨트롤 생성	<i>TWinControl</i> (CLX의 <i>TWidgetControl</i>)
그래픽 컨트롤 생성	<i>TGraphicControl</i>
컨트롤 하위 분류	Windows(VCL) 또는 widget 기반(CLX) 컨트롤
년비주얼 컴포넌트 생성	<i>TComponent</i>

컴포넌트가 아니고 폼에서 조작할 수 없는 클래스를 파생시킬 수도 있습니다. Delphi에는 *TRegIniFile* 및 *TFont* 등 많은 클래스가 들어 있습니다.

기존 컨트롤 수정

컴포넌트를 만드는 가장 간단한 방법은 기존 컴포넌트를 사용자 지정하는 것입니다. Delphi와 함께 제공되는 모든 컴포넌트에서 새로운 컴포넌트를 파생할 수 있습니다.

리스트 박스나 그리드와 같은 일부 컨트롤은 기본 테마에서 여러 가지로 변형됩니다. 이 경우 VCL 및 CLX는 사용자 지정 버전을 파생시키는 *TCustomGrid*와 같이 "custom"이라는 단어가 있는 추상 클래스를 포함합니다.

예를 들어, 표준 *TListBox* 클래스의 일부 속성을 갖지 않는 특수 리스트 박스를 만들 수도 있습니다. 조상 클래스에서 상속된 속성은 제거하거나 숨길 수 없기 때문에 계층에서 *TListBox* 위에 있는 클래스에서 컴포넌트를 파생시켜야 합니다. VCL 또는 CLX는 사용자가 추상 *TWinControl*(또는 CLX의 *TWidgetControl*) 클래스에서 시작하거나 모든 리스트 박스 기능을 재고안하도록 강제하기보다는 리스트 박스의 속성을 구현하지만 속성 모두를 게시하지는 않는 *TCustomListBox*를 제공합니다. 따라서 *TCustomListBox*와 같은 추상 클래스에서 컴포넌트를 파생시키는 경우, 컴포넌트에서 사용 가능하게 하려는 속성만 `published`로 만들고 나머지 속성은 `protected`로 그대로 둡니다.

42장 "속성 생성"에서는 상속된 속성 게시에 대해 설명합니다. 48장 "기존 컴포넌트 수정" 및 50장 "그리드 사용자 지정"에는 기존 컨트롤을 수정하는 예제가 나와 있습니다.

창 있는 컨트롤 생성

VCL 및 CLX의 창 있는 컨트롤은 런타임 시 나타나는 사용자가 상호 작용할 수 있는 객체입니다. 각각의 창 있는 컨트롤은 *Handle* 속성을 통해 액세스되고 운영 체제가 컨트롤을 식별하여 컨트롤에서 동작하도록 하는 핸들을 가집니다. VCL 컨트롤을 사용할 경우, 핸들은 컨트롤이 입력 포커스를 받도록 할 수 있고 Windows API 함수에 전달될 수 있습니다. CLX에서 이러한 컨트롤은 widget 기반 컨트롤입니다. 각 widget 기반

컨트롤에는 *Handle* 속성을 통해 액세스되고 원본으로 사용한 widget을 식별하는 핸들이 있습니다.

모든 창 있는 컨트롤은 *TWinControl*(CLX의 *TWidgetControl*) 클래스의 자손입니다. 누름 버튼, 리스트 박스, 편집 상자와 같은 대부분의 창 있는 표준 컨트롤이 여기에 포함됩니다. *TWinControl*(CLX의 *TWidgetControl*)에서 직접 원래의 컨트롤(기존 컨트롤과 관련 없는 컨트롤)을 파생시키는 동안 Delphi는 이를 위해 *TCustomControl* 컴포넌트를 제공합니다. *TCustomControl*은 복잡한 비주얼 이미지를 더 쉽게 그릴 수 있도록 하는 특화된 창 있는 컨트롤입니다.

50장 "그리드 사용자 지정"에는 창 있는 컨트롤을 생성하는 예제가 나와 있습니다.

그래픽 컨트롤 생성

입력 포커스를 받을 필요가 없는 컨트롤인 경우, 사용자는 컨트롤을 그래픽 컨트롤로 만들 수 있습니다. 그래픽 컨트롤은 창 있는 컨트롤과 유사하지만 창 핸들이 없기 때문에 시스템 리소스를 거의 소모하지 않습니다. 입력 포커스를 받지 않는 *TLabel*과 같은 컴포넌트는 그래픽 컨트롤입니다. 그래픽 컨트롤은 포커스를 받을 수는 없지만 시스템 이벤트에 반응하도록 디자인할 수 있습니다.

Delphi는 *TGraphicControl* 컴포넌트를 통해 사용자 지정 컨트롤의 생성을 지원합니다. *TGraphicControl*은 *TControl*에서 파생된 추상 클래스입니다. *TControl*에서 직접 컨트롤을 파생시킬 수 있더라도 Windows에서 그릴 수 있는 캔버스를 제공하고 *WM_PAINT* 메시지를 처리하는 *TGraphicControl*로부터 시작하는 것이 좋습니다. *Paint* 메소드를 오버라이드하기만 하면 됩니다.

49장 "그래픽 컴포넌트 생성"에는 그래픽 컨트롤을 생성하는 예제가 나와 있습니다.

Windows 컨트롤 하위 분류

일반적인 Windows 프로그래밍에서는 새 *window* 클래스를 정의하고 Windows에 등록하여 사용자 지정 컨트롤을 만듭니다. 객체 지향 프로그래밍에서의 *객체* 또는 *클래스*와 유사한 *window* 클래스는 동일한 종류의 컨트롤의 인스턴스 간에 공유된 정보를 포함하므로 *subclassing*이라고 하는 기존 클래스를 기준으로 새 *window* 클래스를 만들 수 있습니다. 그런 다음 표준 Windows 컨트롤과 매우 유사한 동적 연결 라이브러리(DLL)에 컨트롤을 두고 인터페이스를 제공합니다.

Delphi를 사용하면 기존 *window* 클래스의 주위에 컴포넌트 "랩퍼"를 만들 수 있습니다. 따라서 Delphi 애플리케이션에서 사용하고자 하는 사용자 지정 컨트롤의 라이브러리를 이미 갖고 있는 경우, 사용자의 컨트롤처럼 작동하는 Delphi 컴포넌트를 만들 수 있고 다른 컴포넌트의 경우와 마찬가지로 새 컨트롤을 파생시킬 수 있습니다.

Windows 컨트롤을 하위 분류하는 데 사용되는 기법에 대한 예제를 보려면 *TEdit*와 같은 표준 Window 컨트롤을 나타내는 *StdCtrls* 유닛의 컴포넌트를 참조하십시오. CLX 예제는 *QStdCtrls*를 참조하십시오.

년비주얼 컴포넌트 생성

년비주얼 컴포넌트는 데이터베이스나 시스템 시계 (*TTimer*)와 같은 요소 (*TDataSet* 또는 *TSQLConnection*)를 위한 인터페이스 및 대화 상자 (*TCommonDialog(VCL)* 또는 *TDialog(CLX)* 및 그 자손)를 위한 위치 표시자로서 사용됩니다. 사용자가 작성하는 컴포넌트의 대부분은 비주얼 컨트롤일 것입니다. 년비주얼 컴포넌트는 모든 컴포넌트의 추상 기본 클래스인 *TComponent*에서 직접 파생될 수 있습니다.

컴포넌트 작성 시 고려 사항

Delphi 환경에서 컴포넌트를 안정적으로 만들려면 디자인에 대한 특정 규칙을 따라야 합니다. 이 단원에서는 다음과 같은 항목을 다룹니다.

- 종속성 제거
- 속성, 메소드 및 이벤트
- 그래픽 캡슐화
- 등록

종속성 제거

컴포넌트를 사용 가능하게 하는 특성 중 하나는 코드의 어떤 부분에서도 컴포넌트가 수행할 수 있도록 작업에 대한 제한을 두지 않는 것입니다. 이러한 특성 때문에 컴포넌트는 다양한 조합, 순서 및 컨텍스트로 애플리케이션에 통합됩니다. 사용자는 전제 조건 없이 어떤 상황에서도 기능을 수행하는 컴포넌트를 디자인해야 합니다.

종속성 제거에 대한 좋은 예제는 *TWinControl*의 *Handle* 속성입니다. *CreateWindow* API 함수를 호출하여 Windows 애플리케이션을 작성해 본 적이 있다면 창이나 컨트롤을 다 만들 때까지 액세스하지 않도록 주의하는 것이 프로그램을 실행시키는 데 있어서 가장 어렵고 오류가 발생하기 쉬운 측면 중의 하나임을 알 수 있습니다. Delphi의 창 있는 컨트롤은 필요 시 유효한 창 핸들을 항상 사용할 수 있게 함으로써 사용자는 이에 대해 신경 쓸 필요가 없습니다. 창 핸들을 나타내는 속성을 사용하여 컨트롤은 창이 생성되었는지 여부를 확인할 수 있습니다. 핸들이 유효하지 않은 경우, 컨트롤은 창을 생성하고 핸들을 반환합니다. 따라서 애플리케이션 코드는 *Handle* 속성을 액세스할 때마다 유효한 핸들을 확실하게 얻게 됩니다.

Delphi 컴포넌트는 창 생성과 같은 백그라운드 작업을 제거하여 개발자가 진정으로 원하는 작업에 전념할 수 있게 해줍니다. API 함수에 창 핸들을 전달하기 전에 핸들이 존재하는지 검증하고 창을 생성할 필요가 없습니다. 따라서 응용 프로그램 개발자는 문제가 발생하는지 계속 확인하는 대신 모든 것이 정상적으로 작동한다고 가정할 수 있습니다.

종속성이 없는 컴포넌트를 작성하려면 많은 시간이 소요될 수 있지만 대체로 시간을 투자할 만한 가치가 있습니다. 이는 애플리케이션 개발자의 반복되는 과중한 업무를 줄여 주는 것은 물론 설명서와 지원 부담을 덜어 주기 때문입니다.

속성, 메소드 및 이벤트

폼 디자이너에서 처리하는 시각적 이미지 이외에 컴포넌트의 가장 분명한 특성은 속성, 이벤트 및 메소드입니다. 이 안내서에서의 각 장에서 이러한 특성을 개별적으로 다루고 있지만 여기서는 각 특성을 사용하는 이유에 대해 간단하게 설명합니다.

속성

속성은 애플리케이션 개발자에게 변수 값의 설정이나 읽기에 대한 오해를 제공하는 동시에 컴포넌트 작성자가 원본으로 사용한 데이터 구조를 숨기거나 값이 액세스될 때 특별한 처리를 구현할 수 있도록 합니다.

속성을 사용하면 다음과 같은 여러 가지 이점이 있습니다.

- 디자인 타임에 속성을 사용할 수 있습니다. 애플리케이션 개발자는 코드를 작성하지 않고 속성의 초기 값을 설정하거나 변경할 수 있습니다.
- 속성은 애플리케이션 개발자가 할당하는 값이나 형식을 확인할 수 있습니다. 디자인 타임에 입력을 검증하면 오류가 방지됩니다.
- 컴포넌트는 요청 시 적절한 값을 생성할 수 있습니다. 프로그래머가 범하는 가장 일반적인 오류는 초기화되지 않은 변수를 참조하는 것입니다. 데이터를 속성과 함께 나타내면 필요에 따라 항상 값을 사용할 수 있습니다.
- 속성을 사용하면 사용자는 간단하고 일관된 인터페이스에서 데이터를 숨길 수 있습니다. 변경 내용을 애플리케이션 개발자가 볼 수 있도록 설정하지 않아도 속성에 정보를 구성하는 방법을 바꿀 수 있습니다.

42장 "속성 생성"에는 컴포넌트에 속성을 추가하는 방법이 설명되어 있습니다.

이벤트

이벤트는 런타임에 입력이나 다른 활동에 응답하여 코드를 호출하는 특별한 속성입니다. 이벤트는 특정 코드 블록을 마우스 동작, 키 입력 등과 같은 특정 런타임 발생 항목에 연결할 수 있는 방법을 애플리케이션 개발자에게 제공합니다. 이벤트가 발생할 때 실행되는 코드를 *이벤트 핸들러*라고 합니다.

이벤트를 사용하면 애플리케이션 개발자는 새 컴포넌트를 정의하지 않아도 각기 다른 종류의 입력에 대한 응답을 지정할 수 있습니다.

43장 "이벤트 생성"에는 표준 이벤트를 구현하는 방법과 새 이벤트를 정의하는 방법이 설명되어 있습니다.

메소드

클래스 메소드는 클래스의 특정 인스턴스가 아니라 클래스에서 동작하는 프로시저 및 함수입니다. 예를 들어, 모든 컴포넌트의 생성자(*Create* 메소드)는 클래스 메소드입니다. 컴포넌트 메소드는 컴포넌트 인스턴스 자체에서 동작하는 프로시저 및 함수입니다. 애플리케이션 개발자는 메소드를 사용하여 컴포넌트에게 특정 작업을 수행하거나 다른 속성이 포함하지 않는 값을 반환하도록 지시합니다.

메소드에는 코드 실행이 필요하기 때문에 런타임 시에만 메소드를 호출할 수 있습니다. 메소드가 유용한 몇 가지 이유는 다음과 같습니다.

- 메소드는 데이터가 상주하는 동일한 객체 내에서 컴포넌트 기능을 캡슐화합니다.
- 메소드는 간단하고 일관된 인터페이스에 복잡한 프로시저를 숨길 수 있습니다. 애플리케이션 개발자는 특정 컴포넌트의 *AlignControls* 메소드가 작동하는 방법이나 다른 컴포넌트에 있는 *AlignControls* 메소드와의 차이점을 몰라도 그 메소드를 호출할 수 있습니다.
- 메소드를 사용하면 단일 호출로 여러 속성을 업데이트할 수 있습니다.

44장 "메소드 생성"에는 컴포넌트에 메소드를 추가하는 방법이 설명되어 있습니다.

그래픽 캡슐화

Delphi는 다양한 그래픽 툴을 캔버스에 캡슐화하여 Windows 그래픽을 단순화합니다. 캔버스는 창이나 컨트롤의 그리기 표면을 나타내며 펜, 브러시, 글꼴 등과 같은 다른 클래스를 포함합니다. 캔버스는 Windows 장치 컨텍스트와 같지만 사용자를 위한 모든 부기를 다룹니다.

그래픽 Windows 애플리케이션을 작성해 본 적이 있다면 사용자는 Window 그래픽 장치 인터페이스(GDI)에 의해 부여되는 요구 사항에 익숙할 것입니다. 예를 들어, GDI는 사용 가능한 장치 컨텍스트의 수를 제한하고 소멸하기 전에 사용자가 그래픽 객체를 초기 상태로 복구하도록 요구합니다.

Delphi를 사용하면 이러한 것들에 대해 염려할 필요가 없습니다. 폼이나 다른 컴포넌트에 그리려면 컴포넌트의 *Canvas* 속성을 액세스합니다. 펜이나 브러시를 사용자 지정하려면 색 또는 스타일을 설정합니다. 설정이 끝나면 Delphi는 리소스를 처분합니다. 또한 Delphi는 애플리케이션이 동일한 종류의 리소스를 자주 사용할 경우 리소스를 캐시로 저장하므로 다시 만들 필요가 없습니다.

사용자는 여전히 Windows GDI에 완벽하게 액세스할 수 있지만 Delphi 컴포넌트에 내장된 캔버스를 사용할 경우 코드가 더 단순하며 더 빠르게 실행된다는 것을 자주 깨닫게 될 것입니다. 그래픽 기능에 대한 자세한 내용은 45장 "컴포넌트에서 그래픽 사용"에 설명되어 있습니다.

CLX 그래픽 캡슐화는 다르게 작동합니다. 캔버스는 페인터 대신입니다. 폼이나 다른 컴포넌트에 그리려면 컴포넌트의 *Canvas* 속성을 액세스합니다. *Canvas*는 속성인 동시에 *TCanvas*라고 하는 객체입니다. *TCanvas*는 *Handle* 속성을 통해 액세스할 수 있는 Qt 페인터를 둘러싸는 래퍼입니다. 핸들을 사용하면 하위 수준의 Qt 그래픽 라이브러리 함수에 액세스할 수 있습니다.

펜이나 브러시를 사용자 지정하려면 색 또는 스타일을 설정합니다. 설정이 끝나면 Kylix는 리소스를 처리합니다. CLX 또한 리소스를 캐시로 저장합니다.

CLX 컴포넌트에 기본 제공된 캔버스를 사용할 수도 있습니다. 그래픽 이미지가 컴포넌트에서 작동하는 방법은 컴포넌트가 파생된 객체의 캔버스에 따라 달라집니다.

등록

Delphi IDE에 컴포넌트를 설치할 수 있으려면 우선 컴포넌트를 등록해야 합니다. 등록을 하면 컴포넌트 팔레트에 컴포넌트를 놓는 위치를 Delphi에 알려 줍니다. 또한 Delphi가 컴포넌트를 폼 파일에 저장하는 방법을 사용자 지정할 수도 있습니다. 컴포넌트 등록에 대한 내용은 47장 "디자인 타임 시 컴포넌트 사용"을 참조하십시오.

새 컴포넌트 생성

다음 두 가지 방법으로 새 컴포넌트를 만들 수 있습니다.

- Component 마법사 사용
- 수동으로 컴포넌트 생성

이러한 방법 중 하나를 사용하여 컴포넌트 팔레트에 설치할 수 있는 최소한의 기능을 가진 컴포넌트를 만들 수 있습니다. 설치 후에는 새 컴포넌트를 폼에 추가하고 디자인 타임 및 런타임 시 새 컴포넌트를 테스트할 수 있습니다. 그런 다음 더 많은 기능을 컴포넌트에 추가하고, 컴포넌트 팔레트를 업데이트하며, 테스트를 계속할 수 있습니다.

새 컴포넌트를 만들 때마다 수행되는 몇 가지 기본 단계가 있습니다. 이러한 단계는 아래에 설명되어 있습니다. 이 문서의 다른 예제에서는 사용자가 이러한 단계를 수행하는 방법을 알고 있다고 가정합니다.

- 1 새 컴포넌트의 유닛을 만듭니다.
- 2 기존 컴포넌트 타입에서 컴포넌트를 파생시킵니다.
- 3 속성, 메소드 및 이벤트를 추가합니다.
- 4 Delphi에 컴포넌트를 등록합니다.
- 5 컴포넌트와 그 속성, 메소드 및 이벤트에 대한 도움말 파일을 만듭니다.
- 6 패키지(특정 동적 연결 라이브러리)를 생성하여 Delphi IDE에 컴포넌트를 설치할 수 있습니다.

이를 완료하면 완성된 컴포넌트에 다음과 같은 파일이 포함됩니다.

- 패키지(.BPL) 또는 패키지 모음(.DPC) 파일
- 컴파일된 패키지(.DCP) 파일
- 컴파일된 유닛(.DCU) 파일
- 팔레트 비트맵(.DCR) 파일
- 도움말(.HLP) 파일

컴포넌트 사용자에게 컴포넌트 사용법을 알려 주는 도움말 파일의 생성은 선택 사항입니다.

5부의 나머지 장에서는 컴포넌트 구축에 대한 모든 측면을 설명하고 다른 종류의 컴포넌트를 작성하는 몇 가지 예제를 제공합니다.

Component 마법사 사용

Component 마법사는 컴포넌트 생성의 초기 단계를 단순화합니다. Component 마법사를 사용하는 경우, 다음을 지정해야 합니다.

- 파생된 클래스
- 새 컴포넌트의 클래스 이름
- 새 컴포넌트를 나타내고자 하는 컴포넌트 팔레트 페이지
- 컴포넌트가 만들어지는 유닛의 이름
- 유닛이 검색되는 검색 경로
- 컴포넌트를 두고자 하는 패키지의 이름

다음과 같이 Component 마법사는 컴포넌트를 수동으로 만들 때와 동일한 작업을 수행합니다.

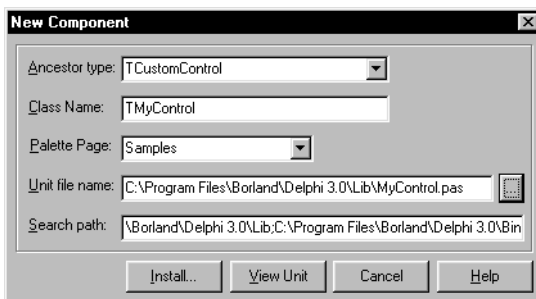
- 유닛 생성
- 컴포넌트 파생
- 컴포넌트 등록

Component 마법사는 기존 유닛에 컴포넌트를 추가할 수 없습니다. 따라서 수동으로 기존 유닛에 컴포넌트를 추가해야 합니다.

Component 마법사를 시작하려면 다음 두 가지 방법 중 하나를 선택합니다.

- Component | New Component를 선택합니다
- File | New | Other를 선택하고 Component를 더블 클릭합니다

그림 40.2 Component 마법사



다음과 같이 Component 마법사의 필드에 입력합니다.

- 1 Ancestor Type 필드에서 새 컴포넌트를 파생시킬 클래스를 지정합니다.

참고

드롭다운 목록에 있는 많은 컴포넌트들은 각기 다른 유닛 이름으로 두 번 나열되는데 즉, VCL에 대한 이름과 CLX에 대한 이름입니다. CLX 특정 유닛은 Graphics 대신 Qgraphics처럼 Q로 시작합니다. 올바른 컴포넌트의 자손인지 확인합니다.

- 2 Class Name 필드에서 새 컴포넌트 클래스의 이름을 지정합니다.
- 3 Palette Page 필드에 새 컴포넌트를 설치하고자 하는 컴포넌트 팔레트의 페이지를 지정합니다.

- 4 Unit file name 필드에서 컴포넌트 클래스를 선언하고자 하는 유닛의 이름을 지정합니다.
- 5 유닛이 검색 경로에 없는 경우 필요에 따라 Search Path 필드에서 검색 경로를 편집합니다.

컴포넌트를 새 패키지나 기존 패키지에 두려면 Component|Install을 클릭한 다음 나타나는 대화 상자를 사용하여 패키지를 지정합니다.

경고 *TCustomControl*과 같이 이름이 "custom"으로 시작하는 VCL 또는 CLX 클래스에서 컴포넌트를 파생시킬 경우, 원래 컴포넌트의 모든 추상 메소드를 오버라이드할 때까지 새 컴포넌트를 폼에 두려고 하면 안 됩니다. Delphi는 추상 메소드나 속성을 갖는 클래스의 인스턴스 객체를 만들 수 없습니다.

유닛의 소스 코드를 보려면 View Unit을 클릭합니다. Component 마법사를 이미 닫은 경우 File|Open을 선택하여 코드 에디터에서 유닛 파일을 엽니다. Delphi는 클래스 선언 및 *Register* 프로시저를 포함한 새 유닛을 만들고 모든 표준 Delphi 유닛을 포함하는 **uses** 절을 추가합니다.

Controls 유닛의 *TCustomControl* 자손인 경우, 유닛은 다음과 같이 나타납니다.

```

unit MyControl;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TMyControl = class(TCustomControl)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyControl]);
end;

end.

```

Qcontrols 유닛의 *TCustomControl* 자손인 경우, 유일한 차이는 다음과 같이 나타나는 **uses** 절입니다.

```

uses
  Windows, Messages, SysUtils, Classes, QControls;

```

CLX가 별도의 유닛을 사용하는 곳에 Q라는 접두어가 붙은 동일한 이름의 유닛으로 변경됩니다. 즉, Controls는 Qcontrols로 변경됩니다.

수동으로 컴포넌트 생성

새 컴포넌트를 만드는 가장 쉬운 방법은 Component 마법사를 사용하는 것입니다. 그러나 같은 단계를 수동으로 수행할 수도 있습니다.

컴포넌트를 수동으로 생성하려면 다음 단계를 따릅니다.

- 1 유닛 파일 생성
- 2 컴포넌트 파생
- 3 컴포넌트 등록

유닛 파일 생성

유닛은 별도로 컴파일된 오브젝트 파스칼 코드의 모듈입니다. Delphi는 유닛을 여러 용도로 사용합니다. 모든 폼에는 고유한 유닛이 있고 대부분의 컴포넌트나 관련 컴포넌트의 그룹에도 고유한 유닛이 있습니다.

컴포넌트를 만들 때 컴포넌트의 새 유닛을 만들거나 기존 유닛에 새 컴포넌트를 추가합니다.

유닛을 만들려면 File|New를 선택하고 Unit을 더블 클릭합니다. 그러면 Delphi는 새 유닛 파일을 만들고 코드 에디터에서 그 파일을 엽니다.

기존 유닛을 열려면 File|Open을 선택한 다음 컴포넌트를 추가할 소스 코드 유닛을 선택합니다.

참고 기존 유닛에 컴포넌트를 추가할 경우, 유닛에 컴포넌트 코드만 포함되어 있는지 확인합니다. 예를 들어, 폼을 포함하는 유닛에 컴포넌트 코드를 추가하면 컴포넌트 팔레트에서 오류가 발생합니다.

컴포넌트에 대한 새 유닛 또는 기존 유닛을 갖게 되면 컴포넌트 클래스를 파생시킬 수 있습니다.

컴포넌트 파생

모든 컴포넌트는 *TComponent*에서 파생되거나, *TControl*이나 *TGraphicControl*과 같이 좀더 특화된 자손 중 하나에서 파생되거나, 기존 컴포넌트 클래스에서 파생된 클래스입니다. 40-2 페이지의 "컴포넌트 생성 방법"에는 각기 다른 종류의 컴포넌트를 파생시키는 클래스에 대한 설명이 제공됩니다.

클래스 파생에 대한 자세한 내용은 41-1 페이지의 "새 클래스 정의" 단원에 설명되어 있습니다.

컴포넌트를 파생시키려면 컴포넌트를 포함할 유닛의 **interface** 부분에 객체 타입 선언을 추가합니다.

일반 컴포넌트 클래스는 *TComponent*에서 직접 파생된 논비주얼 컴포넌트입니다.

일반 컴포넌트 클래스를 만들려면 다음과 같은 클래스 선언을 사용자 컴포넌트 유닛의 **interface** 부분에 추가합니다.

```
type
  TNewComponent = class(TComponent)
  end;
```

이제까지의 작업을 보면 새 컴포넌트는 *TComponent*와 다르지 않습니다. 새 컴포넌트 구축에 대한 프레임워크가 만들어졌습니다.

컴포넌트 등록

등록은 Delphi에게 컴포넌트 라이브러리에 추가할 컴포넌트와 컴포넌트를 나타내야 할 컴포넌트 팔레트의 페이지를 알려 주는 간단한 과정입니다. 등록 과정에 대한 자세한 설명을 보려면 47장 "디자인 타임 시 컴포넌트 사용"을 참조하십시오.

다음과 같은 방법으로 컴포넌트를 등록합니다.

- 1 *Register*라는 프로시저를 컴포넌트 유닛의 **interface** 부분에 추가합니다. *Register*는 매개변수를 갖지 않기 때문에 다음과 같이 매우 간단하게 선언할 수 있습니다.

```
procedure Register;
```

이미 컴포넌트가 포함된 유닛에 컴포넌트를 추가하는 경우에 선언된 *Register* 프로시저를 유닛이 갖고 있으므로 선언을 변경할 필요가 없습니다.

- 2 *Register* 프로시저를 유닛의 **implementation** 부분에 작성하고 등록하고자 하는 각 컴포넌트에 대해 *RegisterComponents*를 호출합니다. *RegisterComponents*는 두 가지 매개변수, 즉 컴포넌트 팔레트 페이지의 이름과 컴포넌트 타입 집합을 갖는 프로시저입니다. 기존 등록에 컴포넌트를 추가하는 경우, 기존 문장의 집합에 새 컴포넌트를 추가하거나 *RegisterComponents*를 호출하는 새 문장을 추가할 수 있습니다.

*TMyControl*이라는 컴포넌트를 등록하여 팔레트의 Samples 페이지에 놓으려면 *Register* 프로시저를 *TMyControl*의 선언이 들어 있는 유닛에 추가합니다.

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewControl]);
end;
```

이 *Register* 프로시저는 컴포넌트 팔레트의 Samples 페이지에 *TMyControl*을 둡니다.

일단 컴포넌트를 등록하면 패키지로 컴파일하고(47장 "디자인 타임 시 컴포넌트 사용" 참조) 컴포넌트 팔레트에 설치할 수 있습니다.

설치되지 않은 컴포넌트 테스트

컴포넌트 팔레트에 설치하기 전에는 컴포넌트의 런타임 작동을 테스트할 수 있습니다. 이 작업은 새로 만든 컴포넌트를 디버깅하는 데 특히 유용하며 컴포넌트 팔레트에 있는지 여부에 관계 없이 모든 컴포넌트에서 잘 작동합니다. 이미 설치된 컴포넌트를 테스트하는 것에 대한 자세한 내용은 40-14 페이지의 "설치된 컴포넌트 테스트"를 참조하십시오.

팔레트에서 컴포넌트를 선택하여 폼에 놓을 때 Delphi가 수행하는 작업을 에뮬레이트 하여 설치되지 않는 컴포넌트를 테스트합니다.

다음과 같은 방법으로 설치되지 않은 컴포넌트를 테스트합니다.

- 1 컴포넌트 유닛의 이름을 폼 유닛의 **uses** 절에 추가합니다.
- 2 객체 필드를 폼에 추가하여 컴포넌트를 나타냅니다.

이 작업은 사용자가 컴포넌트를 추가하는 방법과 Delphi가 컴포넌트를 추가하는 방법 간의 중요한 차이점 중의 하나입니다. 사용자는 폼의 타입 선언의 하단부에 있는 `public` 부분에 객체 필드를 추가합니다. Delphi는 자신이 관리하는 타입 선언 부분에서 위의 객체 필드를 추가합니다.

Delphi가 관리하는 폼 타입의 선언 부분에 필드를 추가해서는 안 됩니다. 타입 선언의 이 부분에 있는 항목은 폼 파일에 저장된 항목에 해당합니다. 폼에 존재하지 않는 컴포넌트의 이름을 추가할 경우 폼 파일을 유효하지 않게 만들 수 있습니다.

- 3 폼의 `OnCreate` 이벤트에 핸들러를 첨부합니다.
- 4 폼의 `OnCreate` 핸들러에서 컴포넌트를 생성합니다.

컴포넌트의 생성자를 호출한 경우, 컴포넌트의 소유자(시간이 되었을 때 컴포넌트를 소멸시키는 작업을 책임지는 컴포넌트)를 지정하는 매개변수를 전달해야 합니다. 거의 항상 `Self`를 소유자로 전달합니다. 메소드에서 `Self`는 메소드를 포함한 객체에 대한 참조가 됩니다. 이 경우에는 폼의 `OnCreate` 핸들러에서 `Self`가 폼을 참조합니다.

- 5 `Parent` 속성을 할당합니다.

`Parent` 속성의 설정은 항상 컨트롤을 생성한 후 가장 먼저 해야 할 작업입니다. 부모는 사용자의 컨트롤을 시각적으로 포함하는 컴포넌트이며 대개 컨트롤이 표시되는 폼이지만 그룹 상자나 패널이 될 수도 있습니다. 보통 `Parent`를 `Self`, 즉 폼으로 설정합니다. 컨트롤의 다른 속성을 설정하기 전에 항상 `Parent`를 먼저 설정합니다.

경고

컴포넌트가 컨트롤이 아닌 경우, 다시 말해 `TControl`이 컴포넌트의 조상 중 하나가 아닌 경우에는 이 단계를 건너뛸 수 있습니다. 실수로 컴포넌트 대신 폼의 `Parent` 속성을 `Self`로 설정하면 운영 체제에 문제가 발생할 수 있습니다.

- 6 다른 모든 컴포넌트 속성을 원하는 대로 설정합니다.

예를 들어, `MyControl`이라는 유닛에서 `TMyControl` 타입의 새 컴포넌트를 테스트하려면 새 프로젝트를 만든 후 단계에 따라 다음과 같이 나타나는 폼 유닛을 만듭니다.

```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, MyControl;                                { 1. Add NewTest to uses clause }
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender:TObject);                    { 3. Attach a handler to OnCreate }
  private
    { Private declarations }
  public
    { Public Declarations }
    MyControl1:TMyControl1;                                 { 2. Add an object field }
```

```

    end;

var
    Form1:TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender:TObject);
begin
    MyControl1 := TMyControl.Create(Self);           { 4. Construct the component }
    MyControl1.Parent := Self;                       { 5. Set Parent property if component is a control }
    MyControl1.Left := 12;                           { 6. Set other properties... }
    :
    :
end;
end.

```

설치된 컴포넌트 테스트

컴포넌트 팔레트에 설치한 후 컴포넌트의 디자인 타임 동작을 테스트할 수 있습니다. 이 작업은 새로 만든 컴포넌트를 디버깅하는 데 특히 유용하며 컴포넌트 팔레트에 있는지 여부에 관계 없이 모든 컴포넌트에서 잘 작동합니다. 아직 설치하지 않은 컴포넌트를 테스트하는 것에 대한 자세한 내용은 40-12 페이지의 "설치되지 않은 컴포넌트 테스트"를 참조하십시오.

설치 후 컴포넌트를 테스트하면 폼에 가져다 놓을 때 디자인 타임 예외만을 생성하는 컴포넌트를 디버깅할 수 있습니다.

Delphi의 두 번째 실행 인스턴스를 사용하여 설치된 컴포넌트를 테스트합니다.

- 1 Delphi IDE 메뉴에서 Project|Options를 선택하고 Directories/Conditional 페이지에서 Debug Source Path를 컴포넌트 소스 파일로 설정합니다.
- 2 그런 다음 Tools|Debugger Options를 선택합니다. Language Exceptions에서 추적하고자 하는 예외를 활성화합니다.
- 3 컴포넌트 소스 파일을 열고 브레이크포인트를 설정합니다.
- 4 Run|Parameters를 선택하고 Host Application 필드를 Delphi 실행 파일의 이름과 위치로 설정합니다.
- 5 Run Parameters 대화 상자에서 Load 버튼을 클릭해서 Delphi의 두 번째 인스턴스를 시작합니다.
- 6 그런 다음 테스트할 컴포넌트를 폼에 가져다 놓습니다. 이 작업은 소스의 브레이크포인트에서 중단되어야 합니다.

41

컴포넌트 작성자를 위한 객체 지향 프로그래밍

Delphi로 애플리케이션을 작성해 본 경험이 있다면 사용자는 클래스에 데이터와 코드가 모두 들어 있다는 사실과 디자인 타임과 런타임 시 클래스를 처리할 수 있음을 알 수 있습니다. 그러면 능숙하게 컴포넌트를 사용할 수 있다는 것을 의미합니다.

사용자가 새 컴포넌트를 만들 때는 애플리케이션 개발자가 작업을 할 필요가 없도록 클래스를 처리합니다. 또한 사용자는 컴포넌트를 사용할 개발자가 볼 수 없도록 컴포넌트의 내부 작업을 숨기려고 합니다. 컴포넌트를 위한 적절한 조상을 선택하고, 개발자에게 필요한 속성과 메소드만 표시하는 인터페이스를 디자인하며, 이 장에서 제공하는 기타 지침에 따르면 다양한 기능의 재사용 가능한 컴포넌트를 만들 수 있습니다.

컴포넌트를 만들려면 우선 객체 지향 프로그래밍(OOP)과 관련된 다음 내용에 익숙해져야 합니다.

- 새 클래스 정의
- 조상, 자손 및 클래스 계층 구조
- 액세스 제어
- 메소드 디스패칭
- 추상 클래스 멤버
- 클래스 및 포인터

새 클래스 정의

컴포넌트 작성자와 애플리케이션 개발자 간의 차이는 컴포넌트 작성자가 새 클래스를 만드는 반면 애플리케이션 개발자는 클래스의 인스턴스를 조작한다는 점입니다.

클래스는 기본적으로 하나의 타입입니다. 프로그래머라면 항상 타입과 인스턴스를 다루게 되며 이러한 용어를 사용하지 않는 경우에도 마찬가지입니다. 예를 들면, *Integer*와 같은 타입의 변수를 만들 수 있습니다. 대개 클래스는 일반 데이터 타입보다 복잡하

지만 동작 방식은 동일합니다. 즉, 같은 타입의 인스턴스에 다른 값을 할당하면 다른 작업을 수행할 수 있습니다.

예를 들어, 하나는 OK로, 다른 하나는 Cancel로 레이블이 지정된 버튼 두 개를 포함한 폼을 만드는 경우를 흔히 볼 수 있습니다. 여기서 각 버튼은 `TButton` 클래스의 인스턴스지만 `Caption` 속성에 다른 값을 할당하고 `OnClick` 이벤트에 다른 핸들러를 할당하여 다르게 동작하는 두 개의 인스턴스를 만듭니다.

새 클래스 파생

새 클래스를 파생시키는 이유는 다음 두 가지입니다.

- 반복을 피하기 위한 클래스 기본값 변경
- 클래스에 새 기능 추가

각 경우에서의 목표는 재사용할 수 있는 객체를 만드는 것입니다. 재사용을 염두에 두면서 컴포넌트를 디자인하는 경우, 나중에 수행할 작업을 줄일 수 있습니다. 클래스에 사용 가능한 기본값을 제공하되 기본값을 사용자 지정할 수 있도록 설정합니다.

반복을 피하기 위한 클래스 기본값 변경

대부분의 프로그래머는 반복을 피하려고 합니다. 따라서 사용자가 같은 코드 줄을 반복하여 재작성하는 것을 깨닫는 경우, 하위 루틴 또는 함수에 코드를 두거나 여러 프로그램에서 사용할 수 있는 루틴 라이브러리를 만듭니다. 컴포넌트에도 같은 상황이 적용됩니다. 즉, 동일한 속성을 변경하거나 동일한 메소드를 호출하는 대신 이러한 작업을 기본적으로 수행하는 새 컴포넌트를 만들 수 있습니다.

예를 들어, 애플리케이션을 만들 때마다 대화 상자를 추가하여 특정 작업을 수행한다고 가정합니다. 매번 대화 상자를 다시 만드는 것이 어렵지는 않지만 불필요한 일이기도 합니다. 그 대신 대화 상자를 한 번만 디자인하고 속성을 설정한 다음 대화 상자에 연결된 래퍼(wrapper) 컴포넌트를 컴포넌트 팔레트에 설치할 수 있습니다. 이렇게 대화 상자를 재사용할 수 있는 컴포넌트로 만들면 반복적인 작업을 없앨 수 있을 뿐만 아니라 표준화를 개선하고 대화 상자를 다시 만들 때마다 발생할 수 있는 오류 가능성을 줄일 수 있습니다.

48장 "기존 컴포넌트 수정"에는 컴포넌트의 기본 속성을 변경하는 예제가 나와 있습니다.

참고 기존 컴포넌트의 `published`인 속성만 수정하거나 컴포넌트 또는 컴포넌트 그룹의 특정 이벤트 핸들러를 저장하려는 경우 `컴포넌트 템플릿`을 만들면 필요한 작업을 더 쉽게 수행할 수 있습니다.

클래스에 새 기능 추가

새 컴포넌트를 만드는 일반적인 이유는 기존 컴포넌트에 없는 기능을 추가하기 위해서입니다. 이러한 작업을 하는 경우에는 `TComponent` 또는 `TControl`과 같은 추상 기본 클래스나 기존 컴포넌트에서 새 컴포넌트를 파생시킵니다.

새 컴포넌트를 파생시킬 때는 원하는 기능과 가장 가까운 기능의 서브셋이 포함된 클래스에서 파생시킵니다. 클래스에 기능을 추가할 수 있지만 제거할 수는 없습니다. 따라서 기존 컴포넌트 클래스에 포함하고 싶지 않은 속성이 들어 있는 경우에는 해당 컴포넌트의 조상에서 파생시켜야 합니다.

예를 들어, 리스트 박스에 기능을 추가하려는 경우 *TListBox*에서 컴포넌트를 파생시킬 수 있습니다. 그러나 새 기능을 추가하되 표준 리스트 박스의 몇 가지 기능을 제외하려는 경우에는 *TListBox*의 조상인 *TCustomListBox*에서 컴포넌트를 파생시켜야 합니다. 그 다음 원하는 리스트 박스의 기능만 다시 만들거나 보이도록 하고 새 기능을 추가할 수 있습니다.

50장 "그리드 사용자 지정"에는 추상 컴포넌트 클래스를 사용자 지정하는 예제가 나와 있습니다.

새 컴포넌트 클래스 선언

Delphi는 표준 컴포넌트 외에 새 컴포넌트를 파생시키기 위한 기초로서의 많은 추상 클래스를 제공합니다. 40-3 페이지의 표 40.1에서는 사용자 고유의 컴포넌트를 만들 때부터 시작할 수 있는 클래스를 보여 줍니다.

새 컴포넌트 클래스를 선언하려면 컴포넌트의 유닛 파일에 클래스 선언을 추가합니다.

다음은 간단한 그래픽 컴포넌트를 선언한 것입니다.

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

완성된 컴포넌트 선언에는 **end** 앞에 속성, 이벤트 및 메소드 선언이 일반적으로 포함됩니다. 그러나 위와 같은 선언도 유효하며 컴포넌트 기능 추가를 위한 출발점을 제공합니다.

조상, 자손 및 클래스 계층 구조

애플리케이션 개발자는 모든 컨트롤이 폼에서의 위치를 결정하는 *Top* 및 *Left*라고 하는 속성을 가지는 것을 당연하게 생각합니다. 애플리케이션 개발자에게 있어서 모든 컨트롤이 공통 조상인 *TControl*로부터 이러한 속성을 상속한다는 것은 중요한 문제가 아닙니다. 그러나 컴포넌트를 만들 경우에는 컴포넌트가 적절한 기능을 상속하도록 컴포넌트가 파생되는 클래스를 알아야 합니다. 또한 상속된 기능을 다시 만들지 않고 활용할 수 있도록 컨트롤이 상속하는 모든 기능을 알아야 합니다.

컴포넌트를 파생시키는 클래스를 컴포넌트의 *직계 조상*이라고 합니다. 각 컴포넌트는 자신의 직계 조상은 물론 직계 조상의 직계 조상으로부터도 기능을 상속합니다. 컴포넌트는 기능을 상속하는 모든 클래스를 *조상*이라고 부르며 결국 컴포넌트가 조상의 *자손*이 됩니다.

애플리케이션에서 조상과 자손의 모든 관계는 클래스의 계층 구조를 구성합니다. 특정 클래스가 조상으로부터 모든 것을 상속하기 때문에 계층 구조에 있는 각 클래스는 조상

보다 더 많은 것을 포함하고, 따라서 새 속성과 메소드를 추가하거나 기존의 속성과 메소드를 다시 정의합니다.

사용자가 직계 조상을 지정하지 않은 경우 Delphi는 기본 조상인 *TObject*로부터 사용자의 컴포넌트를 파생시킵니다. *TObject*는 객체 계층 구조에 있는 모든 클래스의 궁극적인 조상입니다.

파생시킬 객체를 선택하기 위한 일반 규칙은 다음과 같이 간단합니다. 즉, 새 객체에 포함할 기능이 많이 있으면서 포함시키지 않으려는 기능은 가지지 않은 객체를 선택합니다. 객체에 기능을 추가할 수 있지만 제거할 수는 없습니다.

액세스 제어

속성, 메소드 및 필드에는 *가시성*이라고 하는 다섯 가지 레벨의 *액세스 제어*가 있습니다. 가시성은 클래스의 특정 부분에 액세스할 수 있는 코드를 결정합니다. 가시성을 지정하여 컴포넌트에 대한 *인터페이스*를 정의합니다.

표 41.1에는 액세스가 가장 제한적인 것부터 가장 허용적인 것까지 가시성의 수준이 나와 있습니다.

표 41.1 객체 내의 가시성 레벨

가시성	의미	용도
private	클래스가 정의된 유닛의 코드에서만 액세스 가능	구현 세부 사항 숨기기.
protected	클래스 및 자손이 정의된 유닛의 코드에서 액세스 가능	컴포넌트 작성자의 인터페이스 정의.
public	모든 코드에서 액세스 가능	런타임 인터페이스 정의.
automated	모든 코드에서 액세스 가능 Automation 타입 정보가 생성됩니다.	OLE automation에만 사용.
published	모든 코드와 Object Inspector에서 액세스 가능	디자인 타임 인터페이스 정의.

멤버가 정의된 클래스에서만 사용할 수 있도록 하려면 멤버를 **private**으로 선언하고, 클래스와 자손에서만 사용할 수 있도록 하려면 멤버를 **protected**로 선언합니다. 유닛 파일 내의 위치에 상관 없이 멤버를 사용할 수 있는 경우 파일의 *모든* 곳에서 멤버를 사용할 수 있다는 것을 기억합니다. 따라서 같은 유닛에서 두 개의 클래스를 정의할 경우 각 클래스는 다른 클래스의 **private** 메소드를 액세스할 수 있습니다. 또한 조상으로부터 다른 유닛의 클래스를 파생시킬 경우 새 유닛의 모든 클래스는 조상의 **protected** 메소드를 액세스할 수 있습니다.

구현 세부 사항 숨기기

클래스 부분을 **private**으로 선언하면 클래스 유닛 파일의 외부에 있는 코드로 해당 부분을 볼 수 없게 됩니다. 선언이 포함된 유닛 내에서는 코드가 **public**인 것처럼 이 부분을 액세스할 수 있습니다.

다음의 예에서는 필드를 애플리케이션 개발자로부터 숨기는 **private**으로 어떻게 선언하는지 보여 줍니다. 목록에서는 두 개의 폼 유닛을 보여 줍니다. 각 폼에는 private 필드에 값을 할당하는 *OnCreate* 이벤트에 대한 핸들러가 있습니다. 컴파일러는 자신이 선언된 폼에서만 필드에 대한 할당을 허용합니다.

```

unit HideInfo;
interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,
Dialogs;

type
  TSecretForm = class(TForm)                                { declare new form }
    procedure FormCreate(Sender:TObject);
    private                                                { declare private part }
      FSecretCode:Integer;                                  { declare a private field }
    end;

var
  SecretForm:TSecretForm;

implementation
procedure TSecretForm.FormCreate(Sender:TObject);
begin
  FSecretCode := 42;                                       { this compiles correctly }
end;
end.                                                       { end of unit }

unit TestHide;                                           { this is the main form file }

interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,
Dialogs,
  HideInfo;                                               { use the unit with TSecretForm }

type
  TTestForm = class(TForm)
    procedure FormCreate(Sender:TObject);
    end;
var
  TestForm:TTestForm;

implementation
procedure TTestForm.FormCreate(Sender:TObject);
begin
  SecretForm.FSecretCode := 13;    { compiler stops with "Field identifier expected" }
end;
end.                                                       { end of unit }

```

HideInfo 유닛을 사용하는 프로그램은 *TSecretForm* 타입의 객체를 사용할 수 있지만 이러한 객체에 있는 *FSecretCode* 필드는 액세스할 수 없습니다.

컴포넌트 작성자의 인터페이스 정의

클래스 부분을 **protected**로 선언하면 클래스 및 그 자손과 유닛 파일을 공유하는 다른 클래스에서만 해당 부분을 볼 수 있습니다.

protected 선언을 사용하면 클래스에 대한 *컴포넌트 작성자의 인터페이스*를 정의할 수 있습니다. 애플리케이션 유닛은 **protected** 부분을 액세스할 수 없지만 파생된 클래스는 이 부분을 액세스할 수 있습니다. 이는 컴포넌트 작성자가 애플리케이션 개발자가 세부 사항을 볼 수 있도록 만들지 않아도 클래스의 작동 방법을 변경할 수 있다는 것을 의미합니다.

참고 공통적인 실수 중 하나가 이벤트 핸들러에서 **protected** 메소드에 액세스하려는 것입니다. 이벤트 핸들러는 이벤트를 받는 컴포넌트가 아니라 폼의 메소드입니다. 결과적으로 이벤트 핸들러는 컴포넌트가 동일한 유닛에서 폼으로 선언되지 않는 경우에 컴포넌트의 **protected** 메소드에 액세스할 수 없습니다.

런타임 인터페이스 정의

클래스 부분을 **public**으로 선언하면 클래스를 전체적으로 액세스할 수 있는 모든 코드에서 해당 부분을 볼 수 있습니다.

public 부분은 런타임 시 모든 코드에서 사용할 수 있으므로 클래스의 **public** 부분은 *런타임 인터페이스*를 정의합니다. 런타임 인터페이스는 런타임 입력에 따라 달라지는 속성이나 읽기 전용 속성과 같이 디자인 타임에 무의미하거나 적절하지 않은 항목에 대해 사용할 수 있습니다. 또한 애플리케이션 개발자가 호출하도록 만드는 메소드도 **public**이어야 합니다.

다음 예제는 컴포넌트의 런타임 인터페이스 부분으로 선언된 두 개의 읽기 전용 속성을 보여 줍니다.

```

type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius:Integer;           { implementation details are private }
    function GetTempFahrenheit:Integer;
  public
    property TempCelsius:Integer read FTempCelsius;           { properties are public }
    property TempFahrenheit:Integer read GetTempFahrenheit;
  end;
  :
  function TSampleComponent.GetTempFahrenheit:Integer;
  begin
    Result := FTempCelsius * 9 div 5 + 32;
  end;

```

디자인 타임 인터페이스 정의

클래스 부분을 **published**로 선언하면 해당 부분이 **public**이 되고 런타임 타입 정보를 생성합니다. 무엇보다도 런타임 타입 정보는 Object Inspector가 속성 및 이벤트에 액세스할 수 있도록 합니다.

published 부분은 Object Inspector에서 나타나기 때문에 해당 클래스의 *디자인 타임 인터페이스*를 정의합니다. 디자인 타임 인터페이스는 애플리케이션 개발자가 디자인

타입에 사용자 지정하려는 클래스의 모든 측면을 포함해야 하지만 런타임 환경에 대한 특정 정보에 의존하는 속성을 포함해서는 안됩니다.

애플리케이션 개발자가 값을 읽기 전용 속성에 직접 할당할 수 없기 때문에 읽기 전용 속성은 디자인 타임 인터페이스의 부분이 될 수 없습니다. 그러므로 읽기 전용 속성은 `published`보다는 `public`이어야 합니다.

다음 예제는 `Temperature`라고 하는 `published` 속성을 보여 줍니다. 이 속성은 `published`이기 때문에 디자인 타임에 Object Inspector에 나타납니다.

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature:Integer;           { implementation details are private }
  published
    property Temperature:Integer read FTemperature write FTemperature; { writable! }
  end;
```

메소드 디스패칭

*디스패칭*은 메소드 호출이 발생했을 때 프로그램이 메소드를 호출해야 하는 위치를 결정하는 방법을 나타냅니다. 메소드를 호출하는 코드는 다른 모든 프로시저나 함수 호출과 같지만 클래스는 메소드를 디스패칭하는 각기 다른 방법을 가집니다.

메소드 디스패치에는 다음과 같은 세 가지 타입이 있습니다.

- 정적(Static)
- 가상(Virtual)
- 동적(Dynamic)

정적 메소드

메소드 선언 시기를 지정하지 않은 경우 모든 메소드는 정적입니다. 정적 메소드는 일반 프로시저나 함수와 같은 방식으로 작동합니다. 컴파일러는 메소드의 정확한 주소를 확인하고 컴파일 타임에 해당 메소드를 연결합니다.

정적 메소드의 가장 큰 이점은 메소드를 매우 빠르게 디스패치한다는 것입니다. 컴파일러는 메소드의 정확한 주소를 결정하기 때문에 메소드를 직접 연결합니다. 반대로 가상 메소드 및 동적 메소드는 런타임에 간접적인 방법으로 메소드 주소를 찾기 때문에 시간이 다소 오래 걸립니다.

정적 메소드는 자손 클래스가 상속할 때 변경되지 않습니다. 정적 메소드가 포함된 클래스를 선언한 다음 여기서 새 클래스를 파생시킬 경우, 파생된 클래스는 정확하게 동일한 주소에서 동일한 메소드를 공유합니다. 이는 정적 메소드를 오버라이드할 수 없으며 정적 메소드는 자신이 호출된 클래스에 상관 없이 항상 동일한 작업을 정확히 수행한다는 것을 의미합니다. 이름이 같은 파생된 클래스에 있는 메소드를 조상 클래스의 정적 메소드로 선언할 경우, 새 메소드는 단지 파생된 클래스의 상속된 메소드를 대체합니다.

정적 메소드의 예

다음 코드에서 첫 번째 컴포넌트가 두 개의 정적 메소드를 선언합니다. 두 번째 컴포넌트는 첫 번째 컴포넌트에 상속된 메소드를 대체하는 동일한 이름의 정적 메소드 두 개를 선언합니다.

```

type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move; { different from the inherited method, despite same declaration }
    function Flash(HowOften:Integer):Integer; { this is also different }
  end;

```

가상 메소드

가상 메소드는 정적 메소드보다 복잡하고 유연성 있는 디스패치 메커니즘을 사용합니다. 가상 메소드는 자손 클래스에서 재정의할 수 있지만 호출은 여전히 조상 클래스에서 할 수 있습니다. 가상 메소드의 주소는 컴파일 타임에 결정되지 않습니다. 그 대신 메소드가 정의된 객체가 런타임에 주소를 찾습니다.

가상 메소드를 만들려면 메소드 선언 뒤에 지시어 **virtual**을 추가합니다. **virtual** 지시어는 모든 가상 메소드 주소를 객체 타입으로 유지하는 객체의 *가상 메소드 테이블*(VMT) 또는 VMT에 항목을 만듭니다.

기존 클래스에서 새 클래스를 파생시키면 새 클래스는 조상의 VMT에 있는 모든 항목과 새 클래스에서 추가로 선언된 가상 메소드를 포함하는 고유한 VMT를 갖게 됩니다.

메소드 오버라이드

메소드 *오버라이드*는 메소드를 대체하는 것이 아니라 확장 또는 개량하는 것을 의미합니다. 자손 클래스는 상속된 모든 가상 메소드를 오버라이드할 수 있습니다.

자손 클래스에서 메소드를 오버라이드하려면 지시어 **override**를 메소드 선언의 끝 부분에 추가합니다.

다음과 같은 경우 메소드를 오버라이드하면 컴파일 오류가 발생합니다.

- 조상 클래스에 메소드가 존재하지 않는 경우
- 해당 이름의 조상 메소드가 정적인 경우
- 선언이 동일하지 않은 경우(인수 매개변수의 개수와 타입이 다른 경우)

다음 코드는 두 개의 일반 컴포넌트의 선언을 보여 줍니다. 첫 번째 컴포넌트는 각기 다른 종류의 디스패칭을 가진 메소드 세 개를 선언합니다. 첫 번째 컴포넌트에서 파생된 두 번째 컴포넌트는 정적 메소드를 대체하고 가상 메소드를 오버라이드합니다.

```

type
  TFirstComponent = class(TCustomControl)
    procedure Move; { static method }

```

```

procedure Flash; virtual;           { virtual method }
procedure Beep; dynamic;           { dynamic virtual method }
end;

TSecondComponent = class(TFirstComponent)
procedure Move;                       { declares new method }
procedure Flash; override;          { overrides inherited method }
procedure Beep; override;           { overrides inherited method }
end;

```

동적 메소드

동적 메소드는 약간 다른 디스패치 메커니즘을 갖는 가상 메소드입니다. 동적 메소드는 객체의 가상 메소드 테이블에 엔트리를 가지고 있지 않으므로 객체가 소모하는 메모리의 양을 감소시킬 수 있습니다. 하지만 동적 메소드 디스패칭은 일반적인 가상 메소드 디스패칭에 비해 약간 느립니다. 메소드가 자주 호출되거나 메소드 실행 시간이 중요한 관건이라면 사용자는 동적 메소드보다는 가상 메소드로 선언해야 합니다.

객체는 동적 메소드의 주소를 저장해야 합니다. 그러나 이 경우 가상 메소드 테이블에서 항목을 받는 대신 동적 메소드가 별도로 나열됩니다. 동적 메소드 목록은 특정 클래스가 사용하거나 오버라이드한 메소드에 대한 항목만 포함됩니다. 반대로 가상 메소드 테이블은 상속 및 도입된 객체의 모든 가상 메소드를 포함합니다. 상속 트리에서 뒤로 움직이면서 각 조상의 동적 메소드 목록을 검색하여 상속된 동적 메소드를 디스패치합니다.

동적 메소드를 만들려면 지시어 **dynamic**을 메소드 선언 뒤에 추가합니다.

추상 클래스 멤버

조상 클래스에서 메소드가 **abstract**로 선언된 경우 애플리케이션에서 새 컴포넌트를 사용할 수 있으려면 우선 해당 클래스를 다시 선언하고 구현하여 자손 컴포넌트에서 표면화해야 합니다. Delphi는 추상 멤버를 포함하는 클래스의 인스턴스를 만들 수 없습니다. 클래스의 상속된 부분을 표면화하는 것에 대한 자세한 내용은 42장 "속성 생성" 및 44장 "메소드 생성"을 참조하십시오.

클래스 및 포인터

모든 클래스와 컴포넌트는 사실상 포인터입니다. 컴파일러는 클래스 포인터를 자동으로 역참조하기 때문에 대개 사용자는 이에 대해 신경 쓸 필요가 없습니다. 포인터로서의 클래스 상태는 클래스를 매개변수로 전달할 때 중요해집니다. 클래스를 전달할 때는 참조가 아닌 값으로 전달해야 합니다. 클래스가 이미 참조인 포인터이기 때문에 클래스를 참조로 전달하면 참조를 참조에 전달하는 결과가 됩니다.

42

속성 생성

속성은 컴포넌트에서 가장 시각적인 부분입니다. 애플리케이션 개발자는 디자인 타임에 속성을 확인 및 조작하고 폼 디자이너에서 컴포넌트 반응으로 즉각적인 피드백을 얻을 수 있습니다. 속성을 잘 디자인하면 다른 사람이 컴포넌트를 사용하기도 쉽고 자신이 유지 관리하기도 쉽습니다.

컴포넌트의 속성을 가장 적절하게 사용하려면 다음 사항을 이해해야 합니다.

- 속성을 생성하는 이유
- 속성 타입
- 상속된 속성 게시
- 속성 정의
- 배열 속성 생성
- 속성 저장 및 로드

속성을 생성하는 이유

애플리케이션 개발자의 관점에서 보면 속성은 변수와 같아 보입니다. 개발자는 속성 값을 필드인 것처럼 설정하거나 읽을 수 있습니다. 변수를 사용하면 할 수 있지만 속성을 사용해서 할 수 없는 유일한 작업은 속성을 **var** 매개변수로 전달하는 것입니다.

다음과 같은 이유에서 속성은 일반 필드보다 강력한 기능을 제공합니다.

- 애플리케이션 개발자는 디자인 타임에 속성을 설정할 수 있습니다. 런타임에만 사용할 수 있는 메소드와는 달리 속성은 애플리케이션을 실행하기 전에 개발자가 컴포넌트를 사용자 지정할 수 있도록 합니다. 속성은 프로그래머의 작업을 단순화하는 Object Inspector에 표시할 수 있습니다. 즉, 객체를 생성하기 위해 여러 매개변수를 처리하는 대신 Delphi에서 Object Inspector의 값을 읽을 수 있습니다. 또한 Object Inspector는 속성 할당이 이루어진 후 즉시 유효성을 검사합니다.
- 속성은 구현 세부 사항을 숨길 수 있습니다. 예를 들면, 암호화된 형식으로 내부적으로 저장된 데이터가 속성의 값으로 암호가 풀려진 채 나타날 수 있습니다. 즉, 값이

간단한 숫자이더라도 컴포넌트는 데이터베이스에서 해당 값을 조회하거나 복잡한 계산을 수행하여 값에 도달합니다. 속성은 외견상 간단한 할당에 복잡한 효과를 덧붙일 수 있도록 해줍니다. 즉 필드에 대한 할당으로 보이는 것이 복잡한 처리를 구현하는 메소드에 대한 호출이 될 수 있습니다.

- 속성은 가상(virtual)일 수 있습니다. 따라서 애플리케이션 개발자에게 하나의 속성처럼 보이는 것이 다른 컴포넌트에서 다르게 구현될 수도 있습니다.

간단한 예가 모든 컨트롤의 *Top* 속성입니다. *Top*에 새 값을 할당하면 단순히 저장된 값을 변경하는 것이 아니라 컨트롤이 재배치되고 다시 그려집니다. 또한 속성 설정의 효과를 개별 컴포넌트로 제한할 필요가 없습니다. 예를 들어, 스피드 버튼의 *Down* 속성을 *True*로 설정하면 해당 그룹에 있는 다른 모든 스피드 버튼의 *Down* 속성은 *False*로 설정됩니다.

속성 타입

속성은 어떤 타입이라도 될 수 있습니다. 다른 타입의 속성은 Object Inspector에서 다르게 표시되는데 이것으로 디자인 타임 시 속성이 만들어진 것과 같이 속성이 할당되었는지 확인합니다.

표 42.1 Object Inspector에서 속성이 표시되는 방법

속성 타입	Object Inspector의 처리
일반(Simple)	숫자, 문자 및 문자열 속성은 숫자, 문자 및 문자열로 표시됩니다. 애플리케이션 개발자는 속성 값을 직접 편집할 수 있습니다.
열거	부울을 포함한 열거 타입의 속성은 편집 가능한 문자열로 표시됩니다. 또한 개발자는 값 열을 더블 클릭하여 가능한 값을 볼 수 있습니다. 가능한 모든 값을 보여 주는 드롭다운 목록이 있습니다.
집합	집합 타입의 속성은 집합으로 나타납니다. 개발자는 속성을 더블 클릭하여 집합을 확장하고 각 요소를 부울 값(집합에 포함된 경우 true)으로 처리할 수 있습니다.
객체	자기 자신이 클래스인 속성으로 종종 컴포넌트의 등록 프로시저에 지정된 고유한 속성 편집기를 가집니다. 속성이 보유한 클래스에 자기 자신의 published 속성이 있는 경우, 개발자는 Object Inspector를 사용하여 더블 클릭으로 목록을 확장하여 이러한 속성을 포함시키고 개별적으로 편집합니다. 객체 속성은 <i>TPersistent</i> 의 자손이어야 합니다.
인터페이스	해당 값이 <i>TComponent</i> 의 자손인 컴포넌트에 의해 구현된 인터페이스이면 인터페이스인 속성은 Object Inspector에 표시될 수 있습니다. 인터페이스 속성은 종종 고유한 속성 편집기를 가집니다.
배열	Object Inspector에서 배열 속성의 편집 지원을 기본 제공하지 않으므로 배열 속성은 자기 자신의 속성 편집기를 가져야 합니다. 컴포넌트를 등록할 때 속성 편집기를 지정할 수 있습니다.

상속된 속성 게시

모든 컴포넌트는 조상 클래스로부터 속성을 상속합니다. 기존 컴포넌트에서 새 컴포넌트를 파생시키면 새 컴포넌트는 직계 조상의 모든 속성을 상속합니다. 추상 클래스에서 컴포넌트를 파생시킬 경우, 대부분의 상속된 속성은 `published`가 아니라 `protected` 또는 `public`입니다.

Object Inspector에서 디자인 타임에 `protected` 또는 `public` 속성을 사용할 수 있도록 하려면 해당 속성을 `published`로 재선언해야 합니다. 재선언은 상속된 속성에 대한 선언을 자손 클래스의 선언에 추가하는 것을 의미합니다.

예를 들어, VCL 컴포넌트를 `TWinControl`로부터 파생시키면 VCL 컴포넌트는 `protected DockSite` 속성을 상속합니다. 이 때 새 컴포넌트에서 `DockSite`를 재선언하여 `public` 또는 `published`로 보호 수준을 변경할 수 있습니다.

다음 코드는 `DockSite`를 `published`로 재선언하여 디자인 타임 시 사용할 수 있도록 만듭니다.

```
type
  TSampleComponent = class(TWinControl)
    published
      property DockSite;
    end;
```

속성을 재선언할 때는 속성 이름만 지정하고 아래의 "속성 정의"에서 설명하는 타입 및 기타 다른 정보는 지정하지 않습니다. 또한 새 기본값을 선언하고 속성 저장 여부를 지정할 수 있습니다.

재선언은 속성을 덜 제한할 수는 있지만 더 많이 제한할 수는 없습니다. 따라서 `protected` 속성을 `public`으로 만들 수는 있지만 속성을 `protected`로 재선언하여 `public` 속성을 숨길 수는 없습니다.

속성 정의

이 단원에서는 새 속성을 선언하는 방법과 표준 컴포넌트에서 따라야 할 몇 가지 규칙을 설명합니다. 다음과 같은 항목이 들어 있습니다.

- 속성 선언
- 내부 데이터 저장소
- 직접 액세스
- 액세스 메소드
- 기본 속성 값

속성 선언

속성은 컴포넌트 클래스의 선언에서 선언됩니다. 속성을 선언하기 위해서는 다음 세 가지 사항을 지정해야 합니다.

- 속성의 이름.
- 속성의 타입.
- 속성의 값을 읽고 쓰는 데 사용되는 메소드. 쓰기 메소드가 선언되어 있지 않은 경우 속성은 읽기 전용입니다.

컴포넌트 클래스 선언의 **published** 섹션에 선언된 속성은 디자인 타임에 Object Inspector에서 편집할 수 있습니다. **published** 속성의 값은 컴포넌트와 함께 폼 파일에 저장됩니다. **public** 섹션에 선언된 속성은 런타임에 사용할 수 있고 프로그램 코드에서 읽거나 설정할 수 있습니다.

다음 예제는 *Count*라는 속성에 대한 전형적인 선언의 예입니다.

```
type
  TYourComponent = class(TComponent)
  private
    FCount:Integer;           { used for internal storage }
    procedure SetCount (Value:Integer); { write method }
  public
    property Count:Integer read FCount write SetCount;
  end;
```

내부 데이터 저장소

속성 데이터를 저장하는 방법에는 제한이 없습니다. 하지만 일반적으로 Delphi 컴포넌트는 다음 규칙을 따릅니다.

- 속성 데이터는 클래스 필드에 저장됩니다.
- 속성 데이터를 저장하는 데 사용되는 필드는 **private**이므로 컴포넌트 내부에서만 액세스해야 합니다. 파생된 컴포넌트는 상속된 속성을 사용해야 하기 때문에 속성의 내부 데이터 저장소에 직접 액세스할 필요가 없습니다.
- 이러한 필드에 대한 식별자는 *F*로 시작하여 속성의 이름을 뒤에 붙여 구성합니다. 예를 들어, *TControl*에 정의된 *Width* 속성의 원시 데이터는 *FWidth*라고 하는 필드에 저장됩니다.

이러한 규칙의 토대가 되는 원칙은 특정 속성에 대한 구현 메소드만 관련 데이터에 액세스할 수 있다는 것입니다. 만일 메소드나 다른 속성이 특정 속성과 관련된 데이터를 변경할 필요가 있는 경우, 해당 속성을 사용해야 하며 저장된 데이터에 직접 액세스하면 안 됩니다. 이렇게 하면 파생된 컴포넌트를 무효로 만들지 않고 상속된 속성의 구현을 변경할 수 있습니다.

직접 액세스

직접 액세스는 속성 데이터를 사용할 수 있게 하는 가장 간단한 방법입니다. 즉, 속성 선언의 **read** 및 **write** 부분은 액세스 메소드를 호출하지 않고 속성 값의 할당 또는 읽기가 내부 저장소 필드에서 직접 이루어지도록 지정합니다. 직접 액세스는 Object Inspector에서 속성을 사용할 수 있게 만들 때 유용하지만 속성 값에 대한 변경 내용은 즉시 처리되지 않습니다.

일반적으로 속성 선언의 **read** 부분에서는 직접 액세스를, **write** 부분에서는 액세스 메소드를 사용합니다. 이렇게 하면 속성 값이 바뀔 때 컴포넌트 상태가 업데이트됩니다.

다음과 같은 컴포넌트 타입의 선언은 **read** 및 **write** 부분 모두에서 직접 액세스를 사용하는 속성을 보여 줍니다.

```
type
  TSampleComponent = class(TComponent)
  private
    FMyProperty: Boolean;           { internal storage is private }
  published
    { declare field to hold property value }
    { make property available at design time }
    property MyProperty: Boolean read FMyProperty write FMyProperty;
  end;
```

액세스 메소드

속성 선언의 **read** 및 **write** 부분에서 필드 대신 액세스 메소드를 지정할 수 있습니다. 액세스 메소드는 protected여야 하고 대개 **virtual**로 선언되기 때문에 자손 컴포넌트는 속성의 구현을 오버라이드할 수 있습니다.

액세스 메소드를 public으로 만들어서는 안 됩니다. 액세스 메소드를 protected로 유지하면 애플리케이션 개발자가 실수로 이러한 메소드 중 하나를 호출하여 속성을 수정하는 것을 방지할 수 있습니다.

다음 클래스는 모든 속성이 동일한 read 및 write 액세스 메소드를 갖도록 하는 인덱스 지정자를 사용하여 세 개의 속성을 선언합니다.

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index: Integer): Integer; { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  :
  ;
```

날짜의 각 요소(연/월/일)가 정수이고 각 설정에 날짜 인코딩이 필요하기 때문에 이 코드에서는 세 개의 속성 모두에 대해 read 및 write 메소드를 공유하여 중복을 방지합니다. 따라서 날짜 요소를 읽고 쓰는 데에는 오직 하나의 메소드만 필요합니다.

다음은 날짜 요소를 얻는 read 메소드입니다.

```
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);           { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
```

```

end;
end;

```

다음은 해당 날짜 요소를 설정하는 write 메소드입니다.

```

procedure TSampleCalendar.SetDateElement(Index:Integer; Value:Integer);
var
  AYear, AMonth, ADay:Word;
begin
  if Value > 0 then                                { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);          { get current date elements }
    case Index of                                    { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
    else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);        { encode the modified date }
    Refresh;                                         { update the visible calendar }
  end;
end;

```

read 메소드

속성에 대한 read 메소드는 아래에서 언급한 경우를 제외하고 매개변수를 사용하지 않으며 속성과 동일한 타입의 값을 반환하는 함수입니다. 규칙에 따라 이 함수의 이름은 *Get*으로 시작하여 속성의 이름을 뒤에 붙입니다. 예를 들어, *Count*라고 하는 속성에 대한 read 메소드는 *GetCount*입니다. read 메소드는 적절한 타입의 속성 값을 만들기 위해 필요한 경우 내부 저장소 데이터를 조작합니다.

예외적으로 배열 속성과 인덱스 지정자를 사용하는 속성(42-8 페이지의 "배열 속성 생성" 참조)에는 매개변수를 사용하지 않는 규칙이 적용되지 않으며 두 속성 모두 인덱스 값을 매개변수로 전달합니다.(인덱스 지정자를 사용하여 여러 속성이 공유하는 단일 read 메소드를 만들 수 있습니다. 인덱스 지정자에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.)

read 메소드를 선언하지 않으면 속성은 쓰기 전용이 됩니다. 일반적으로 쓰기 전용 속성은 거의 사용되지 않습니다.

write 메소드

속성에 대한 write 메소드는 아래에서 언급한 경우를 제외하고 속성과 동일한 타입의 단일 매개변수를 취하는 프로시저입니다. 매개변수는 참조에 의해서 또는 값에 의해서 전달될 수 있고 사용자가 선택한 어떤 이름도 가질 수 있습니다. 규칙에 따라 write 메소드의 이름은 *Set*으로 시작하여 속성의 이름이 뒤에 붙습니다. 예를 들어, *Count*라는 속성에 대한 write 메소드는 *SetCount*입니다. 매개변수로 전달된 값은 속성의 새 값이 되므로 write 메소드는 속성의 내부 저장소에 적절한 데이터를 두는 데 필요한 모든 조작을 수행해야 합니다.

유일하게 배열 속성 및 인덱스 지정자를 사용하는 속성에는 단일 매개변수 규칙이 적용되지 않으며 두 속성 모두 인덱스 값을 두 번째 매개변수로 전달합니다. (인덱스 지정자를 사용하여 여러 속성이 공유하는 단일 write 메소드를 만듭니다. 인덱스 지정자에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.)

write 메소드를 선언하지 않으면 속성은 읽기 전용이 됩니다.

일반적으로 write 메소드는 속성을 변경하기 전에 새 값이 현재 값과 다른지 여부를 테스트합니다. 예를 들어, 다음은 *FCount*라고 하는 필드에 현재 값을 저장하는 *Count*라는 정수 속성에 대한 간단한 write 메소드입니다.

```
procedure TMyComponent.SetCount(Value:Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

기본 속성 값

속성을 선언할 때 *기본값*을 지정할 수 있습니다. Delphi는 기본값을 사용하여 폼 파일에 속성을 저장할지 여부를 결정합니다. 사용자가 속성에 대한 기본값을 지정하지 않으면 Delphi는 항상 속성을 저장합니다.

속성에 대한 기본값을 지정하려면 속성의 선언 또는 재선언에 **default** 지시어와 기본값을 차례대로 추가합니다. 예를 들면, 다음과 같습니다.

```
property Cool Boolean read GetCool write SetCool default True;
```

참고 기본값을 선언해도 속성이 기본값으로 설정되지 않습니다. 컴포넌트의 생성자 메소드는 적절한 시기에 속성 값을 초기화해야 합니다. 그러나 객체가 항상 자신의 필드를 0으로 초기화하기 때문에 생성자가 반드시 정수 속성을 0으로, 문자열 속성을 Null로, 부울 속성을 *False*로 설정할 필요는 없습니다.

기본값을 갖지 않도록 지정

상속된 속성이 기본값을 지정하더라도 속성을 재선언할 때 속성이 기본값을 갖지 않도록 지정할 수 있습니다.

속성이 기본값을 갖지 않도록 지정하려면 속성의 선언 부분에 **nodefault** 지시어를 추가합니다. 예를 들면, 다음과 같습니다.

```
property FavoriteFlavor string nodefault;
```

속성을 처음 선언할 때 **nodefault**를 포함할 필요는 없습니다. 선언된 기본값이 없다는 것은 기본값이 존재하지 않음을 의미합니다.

다음 컴포넌트 선언에는 기본값이 *True*인 *IsTrue*라고 하는 단일 부울 속성이 포함되어 있습니다. 아래 선언에서 유닛의 **implementation** 섹션에는 속성을 초기화하는 생성자가 있습니다.

```

type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
:
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { call the inherited constructor }
  FIsTrue := True;                    { set the default value }
end;

```

배열 속성 생성

몇몇 속성들은 배열처럼 인덱스를 붙일 수 있습니다. 예를 들어, *TMemo*의 *Lines* 속성은 메모의 텍스트를 구성하는 문자열의 인덱스화된 목록이며, 문자열의 배열처럼 취급할 수 있습니다. *Lines*는 더 큰 데이터 집합(메모 텍스트)의 특정 요소(문자열)에 대한 자연스러운 액세스를 제공합니다.

배열 속성은 다음 사항을 제외하고는 다른 속성과 똑같이 선언됩니다.

- 선언에는 지정된 타입의 인덱스가 하나 이상 포함됩니다. 인덱스는 어떠한 타입도 될 수 있습니다.
- 속성 선언의 **read** 및 **write** 부분은 지정된 경우에 메소드여야 하고 필드가 될 수 없습니다.

배열 속성에 대한 **read** 및 **write** 메소드는 인덱스에 해당하는 추가적인 매개변수를 갖습니다. 이 매개변수는 선언에서 지정한 인덱스와 동일한 순서 및 타입을 가져야 합니다.

배열 속성과 배열 사이에는 몇 가지 중요한 차이점이 있습니다. 배열의 인덱스와는 달리 배열 속성의 인덱스는 정수 타입일 필요가 없습니다. 예를 들어, 문자열로 속성을 인덱스할 수 있습니다. 또한 배열 속성에 대해서는 속성의 전체 범위가 아니라 개별 요소만 참조할 수 있습니다.

다음 예제는 정수 인덱스에 기초하여 문자열을 반환하는 속성 선언을 보여 줍니다.

```

type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
:
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of

```

```

-MaxInt..-1:Result := 'Negative';
0: Result := 'Zero';
1..100: Result := 'Small';
101..MaxInt:Result := 'Large';
end;
end;

```

하위 컴포넌트의 속성 생성

기본적으로 속성 값이 또 다른 컴포넌트인 경우, 다른 컴포넌트의 인스턴스를 폼이나 데이터 모듈에 추가한 다음 해당 컴포넌트를 속성 값으로 할당하여 속성에 값을 할당합니다. 그러나 컴포넌트가 속성 값을 구현하는 객체의 고유한 인스턴스를 만들 수도 있습니다. 이러한 전용 컴포넌트를 하위 컴포넌트라고 합니다.

하위 컴포넌트는 영구적인 객체, 즉 *TPersistent*의 자손이 될 수 있습니다. 속성 값으로 할당될 수 있는 별도의 컴포넌트와 달리 하위 컴포넌트의 published 속성은 하위 컴포넌트를 만드는 컴포넌트와 함께 저장됩니다. 그러나 이러한 작업을 하기 위해서는 다음 조건이 충족되어야 합니다.

- 하위 컴포넌트의 *Owner*는 하위 컴포넌트를 만들어 published 속성 값으로 사용하는 컴포넌트여야 합니다. *TComponent*의 자손인 하위 컴포넌트의 경우, 하위 컴포넌트의 *Owner* 속성을 설정하여 이 작업을 수행할 수 있습니다. 다른 하위 컴포넌트의 경우에는 생성된 컴포넌트를 반환하도록 영구적인 객체의 *GetOwner* 메소드를 오버라이드해야 합니다.
- 하위 컴포넌트가 *TComponent*의 자손인 경우 *SetSubComponent* 메소드를 호출하여 하위 컴포넌트라는 사실을 나타내야 합니다. 일반적으로 이 메소드는 하위 컴포넌트를 만들 때 소유자가 호출하거나 하위 컴포넌트의 생성자가 호출합니다.

일반적으로 값이 하위 컴포넌트인 속성은 대개 읽기 전용입니다. 값이 하위 컴포넌트인 속성을 변경하도록 하는 경우, 속성 설정자는 다른 컴포넌트가 속성 값으로 할당될 때 하위 컴포넌트를 해제해야 합니다. 또한 속성이 nil로 설정되면 컴포넌트는 종종 하위 컴포넌트를 다시 인스턴스화합니다. 그렇지 않으면 속성이 다른 컴포넌트로 바뀔 경우 디자인 타임에 하위 컴포넌트를 복원할 수 없습니다. 다음 예제는 *TTimer* 값을 가지는 속성에 대한 이러한 속성 설정자를 보여 줍니다.

```

procedure TDemoComponent.SetTimerProp(Value:TTimer);
begin
  if Value <> FTimer then
  begin
    if Value <> nil then
    begin
      if (FTimer <> nil and FTimer.Owner = self then
        FTimer.Free;
        FTimer := Value;
        FTimer.FreeNotification(self);
      end
    else { nil value }
    begin
      if FTimer.Owner <> self then

```

```

        {
            FTimer := TTimer.Create(self);
            FTimer.SetSubComponent(True);
            FTimer.FreeNotification(self);
        }
    end;
end;
end;

```

위의 속성 설정자가 속성 값으로 설정된 컴포넌트의 *FreeNotification* 메소드를 호출했다는 점에 유의합니다. 이 호출은 속성 값인 컴포넌트가 소멸되려는 순간에 공지를 보내도록 합니다. *Notification* 메소드를 호출하여 이러한 공지를 보냅니다. 다음과 같이 *Notification* 메소드를 오버라이드하여 이 호출을 처리합니다.

```

procedure TDemoComponent.Notification(AComponent:TComponent; Operation:TOperation);
begin
    inherited Notification(AComponent, Operation);
    if (Operation = opRemove) and (AComponent = FTimer) then
        FTimer := nil;
    end;
end;

```

인터페이스의 속성 생성

published 속성 값으로 객체를 사용하는 것과 마찬가지로 인터페이스를 사용할 수 있습니다. 그러나 컴포넌트가 해당 인터페이스의 구현으로부터 공지를 받는 메커니즘에는 차이가 있습니다. 이전 항목에서 속성 설정자는 속성 값으로 할당된 컴포넌트의 *FreeNotification* 메소드를 호출하였습니다. 이로 인해 속성 값인 컴포넌트가 해제될 때 컴포넌트는 자신을 업데이트할 수 있었습니다. 그러나 속성 값이 인터페이스인 경우에는 해당 인터페이스를 구현하는 컴포넌트에 액세스할 수 없습니다. 결과적으로 컴포넌트의 *FreeNotification* 메소드를 호출할 수 없습니다.

이러한 상황을 처리하려면 다음과 같이 컴포넌트의 *ReferenceInterface* 메소드를 호출합니다.

```

procedure TDemoComponent.SetMyIntfProp(const Value:IMyInterface);
begin
    ReferenceInterface(FIntfField, opRemove);
    FIntfField := Value;
    ReferenceInterface(FIntfField, opInsert);
end;

```

지정된 인터페이스로 *ReferenceInterface*를 호출하면 다른 컴포넌트의 *FreeNotification* 메소드를 호출한 것과 같습니다. 따라서 속성 설정자에서 *ReferenceInterface*를 호출한 후 다음과 같이 *Notification* 메소드를 오버라이드하여 인터페이스 구현에서 공지를 처리할 수 있습니다.

```

procedure TDemoComponent.Notification(AComponent:TComponent; Operation:TOperation);
begin
    inherited Notification(AComponent, Operation);
    if (Assigned(MyIntfProp)) and (AComponent.ImplementorOf(MyIntfProp)) then
        MyIntfProp := nil;
    end;
end;

```


Notification 코드가 private 필드 (*FlntfField*) 가 아닌 *MyIntfProp* 속성에 **nil**을 할당한다는 점에 유의합니다. 이렇게 하면 속성 값이 이미 설정된 경우 만들어지는 공지 요청을 제거하기 위해 *ReferenceInterface*를 호출하는 속성 설정자를 *Notification*에서 호출하게 합니다. 인터페이스 속성에 대한 모든 할당은 속성 설정자를 통해 이루어져야 합니다.

속성 저장 및 로드

Delphi에서는 폼과 컴포넌트가 (VCL의 .dfm 및 CLX의 .xfm) 파일에 저장됩니다. 폼 파일에는 폼과 컴포넌트의 속성이 포함됩니다. Delphi 개발자가 사용자가 작성한 컴포넌트를 폼에 추가할 경우, 사용자가 만든 컴포넌트는 저장될 때 컴포넌트의 속성을 폼 파일에 기록할 수 있는 능력을 가지고 있어야 합니다. 이와 마찬가지로 Delphi로 로드되거나 애플리케이션의 일부로 실행될 때 그 컴포넌트는 폼 파일로부터 자신을 복원할 수 있어야 합니다.

속성을 저장하고 폼 파일에서 로드하는 기능은 컴포넌트의 상속된 동작의 일부이기 때문에 대개의 경우 컴포넌트를 폼 파일에서 작동시키기 위해 추가로 할 일은 없습니다. 하지만 가끔 컴포넌트가 자신을 저장하는 방법이나 로드 시 초기화하는 방법을 변경할 수도 있기 때문에 기본 메커니즘을 이해해야 합니다.

속성 저장에 대해 이해가 필요한 측면은 다음과 같습니다.

- 저장 및 로드 메커니즘 사용
- 기본값 지정
- 저장할 대상 결정
- 로드 후 초기화
- Published가 아닌 속성 저장 및 로드

저장 및 로드 메커니즘 사용

폼에 대한 설명은 폼에 있는 각 컴포넌트에 대한 유사한 설명과 함께 폼 속성의 목록으로 구성됩니다. 폼 자체를 포함하여 각 컴포넌트는 자신의 설명을 저장하고 로드합니다.

기본적으로 컴포넌트는 자신을 저장할 때 기본값과 다른 모든 public 및 published 속성의 값을 선언 순서대로 기록합니다. 로드할 때는 우선 모든 속성을 기본값으로 설정하면서 자기 자신을 생성하고 기본값이 아닌 저장된 속성 값을 읽습니다.

이 기본 메커니즘에서는 거의 모든 컴포넌트의 요구가 충족되고 컴포넌트 작성자가 작업을 수행할 필요가 전혀 없습니다. 그러나 특정 컴포넌트의 요구를 충족시키기 위해서 여러 가지 방법으로 저장 및 로드 프로세스를 사용자 지정할 수 있습니다.

기본값 지정

Delphi 컴포넌트는 자신의 속성 값이 기본값과 다를 경우에만 저장합니다. 기본값이 지정되지 않은 경우, Delphi는 속성에 기본값이 없다고 가정합니다. 따라서 컴포넌트는 값에 상관 없이 항상 속성을 저장합니다.

속성에 기본값을 지정하려면 **default** 지시어와 새 기본값을 속성 선언의 끝 부분에 추가합니다.

또한 속성을 재선언할 때 기본값을 지정할 수도 있습니다. 실제로 속성을 재선언하는 이유 중 하나가 다른 기본값을 지정하기 위한 데 있습니다.

참고 기본값을 지정한다고 해서 기본값이 자동으로 속성에 할당되는 것은 아닙니다. 따라서 컴포넌트의 생성자가 필요한 값을 할당하도록 해야 합니다. 컴포넌트 생성자가 값을 설정하지 않은 속성은 0값, 즉 저장소 메모리가 0으로 설정될 때 가정하는 모든 값을 가정합니다. 따라서 숫자 값에서는 0, 부울 값에서는 *False*, 포인터에서는 **nil**이 기본값이 됩니다. 확실하지 않은 경우에는 생성자 메소드의 값을 할당합니다.

다음 코드는 *Align* 속성의 기본값을 지정하는 컴포넌트 선언과 기본값을 설정하는 컴포넌트 생성자의 구현을 보여 줍니다. 여기서 새 컴포넌트는 창의 상태 표시줄에 사용되는 표준 패널 컴포넌트의 특별한 경우이기 때문에 기본 정렬은 소유자의 맨 아래쪽에 있어야 합니다.

```

type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner:TComponent); override; { override to set new default }
  published
    property Align default alBottom; { redeclare with new default value }
  end;
:
constructor TStatusBar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner); { perform inherited initialization }
  Align := alBottom; { assign new default value for Align }
end;

```

저장할 대상 결정

Delphi에서 사용자 컴포넌트의 각 속성 저장 여부를 제어할 수 있습니다. 기본적으로 클래스 선언의 *published* 부분에 있는 모든 속성이 저장됩니다. 지정된 속성을 저장하지 않도록 선택하거나 속성 저장 여부를 동적으로 결정하는 함수를 지정할 수 있습니다.

Delphi의 속성 저장 여부를 제어하려면 속성 선언에 **stored** 지시어를 추가한 다음 바로 뒤에 *True*, *False* 또는 부울 함수 이름을 추가합니다.

다음 코드는 세 개의 새로운 속성을 선언하는 컴포넌트를 보여 줍니다. 이 중 하나는 항상 저장되고, 다른 하나는 절대 저장되지 않으며, 마지막 하나는 부울 함수 값에 따라 저장 여부가 결정됩니다.

```

type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt:Boolean;
  public
    :
  published
    property Important:Integer stored True; { always stored }

```

```

property Unimportant:Integer stored False; { never stored }
property Sometimes:Integer stored StoreIt; { storage depends on function value }
end;

```

로드 후 초기화

컴포넌트는 저장된 설명에서 모든 속성 값을 읽은 후 필요한 모든 초기화를 수행하는 *Loaded* 라고 하는 가상 메소드를 호출합니다. *Loaded*는 폼과 컨트롤이 나타나기 전에 호출되므로 화면에 깜빡임을 야기하는 초기화에 대해 염려할 필요가 없습니다.

속성 값이 로드된 후 컴포넌트를 초기화하기 위해서는 *Loaded* 메소드를 오버라이드합니다.

참고 *Loaded* 메소드에서 가장 먼저 해야 할 일은 상속된 *Loaded* 메소드를 호출하는 것입니다. 이렇게 하면 자신의 컴포넌트를 초기화하기 전에 상속된 모든 속성을 올바르게 초기화할 수 있습니다.

다음 코드는 *TDatabase* 컴포넌트에서 제공됩니다. 로드 후 데이터베이스는 저장되었을 때 열려 있었던 모든 연결을 다시 시도하며 연결 동안 일어나는 모든 예외에 대한 처리 방법을 지정합니다.

```

procedure TDatabase.Loaded;
begin
  inherited Loaded; { call the inherited method first}
  try
    if FStreamedConnected then Open { reestablish connections }
    else CheckSessionName(False);
  except
    if csDesigning in ComponentState then { at design time... }
      Application.HandleException(Self) { let Delphi handle the exception }
    else raise; { otherwise, reraise }
  end;
end;

```

Published가 아닌 속성 저장 및 로드

기본적으로 *published* 속성들만 컴포넌트와 함께 로드되고 저장됩니다. 하지만 *published* 가 아닌 속성을 로드하고 저장하는 것도 가능합니다. 이는 Object Inspector에 나타나지 않는 영구적인 속성을 가질 수 있도록 합니다. 또한 속성 값이 너무 복잡하기 때문에 컴포넌트가 Delphi에서 읽거나 쓰는 방법을 모르는 속성 값을 저장하고 로드할 수 있게 합니다. 예를 들어, *TStrings* 객체는 Delphi의 자동적 동작에 의존하여 자신이 나타내는 문자열을 저장 및 로드할 수 없고 다음과 같은 메커니즘을 사용해야만 합니다.

사용자 속성 값의 로드 및 저장 방법을 Delphi에게 알려 주는 코드를 추가하여 *published* 가 아닌 속성을 저장할 수 있습니다.

속성을 로드하고 저장하는 코드를 작성하려면 다음 단계를 따릅니다.

- 1 속성 값을 저장하고 로드하는 메소드들을 만듭니다.
- 2 *DefineProperties* 메소드를 파일러 (filer) 객체에 전달하여 오버라이드합니다.

속성 값을 저장 및 로드하는 메소드 생성

published가 아닌 속성을 저장하고 로드하려면 우선 속성 값을 저장하고 로드하기 위한 메소드를 각각 만들어야 합니다. 다음 두 가지 방법 중 하나를 선택할 수 있습니다.

- *TWriterProc* 타입의 메소드를 만들어 속성 값을 저장하고, *TReaderProc* 타입의 메소드를 만들어 속성 값을 로드합니다. 이 방법은 일반 타입을 저장하고 로드할 수 있는 Delphi의 기본 제공 기능을 이용할 수 있습니다. Delphi가 저장 및 로드 방법을 알고 있는 타입으로 속성 값이 작성된 경우 이 방법을 사용합니다.
- *TStreamProc* 타입의 메소드를 두 개 만들어 각각 속성 값을 저장하고 로드하는 데 사용합니다. *TStreamProc*는 스트림을 인수로 가져오며 이 스트림의 메소드를 사용하여 속성 값을 읽고 쓸 수 있습니다.

예를 들어, 런타임 시 생성되는 컴포넌트를 나타내는 속성을 고려해 봅시다. Delphi는 이 값을 기록하는 방법을 알고 있지만 폼 디자이너에서 컴포넌트를 만들었기 때문에 자동으로 기록하지는 않습니다. 스트리밍 시스템은 이미 컴포넌트들을 로드하고 저장할 수 있기 때문에 첫 번째 방법을 사용할 수 있습니다. 다음 메소드는 *MyCompProperty* 라고 하는 속성의 값인 동적으로 생성된 컴포넌트를 로드하고 저장합니다.

```

procedure TSampleComponent.LoadCompProperty(Reader:TReader);
begin
    if Reader.ReadBoolean then
        MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer:TWriter);
begin
    Writer.WriteBoolean(MyCompProperty <> nil);
    if MyCompProperty <> nil then
        Writer.WriteComponent(MyCompProperty);
end;

```

DefineProperties 메소드 오버라이드

속성 값을 저장하고 로드하기 위한 메소드를 만들었다면 컴포넌트의 *DefineProperties* 메소드를 오버라이드할 수 있습니다. Delphi는 컴포넌트를 저장하거나 로드할 때 이 메소드를 호출합니다. *DefineProperties* 메소드에서는 현재 파일러(filer)의 *DefineProperty* 또는 *DefineBinaryProperty* 메소드를 호출하여 속성 값을 저장하거나 로드하는 데 사용할 수 있도록 파일러에 이러한 메소드를 전달해야 합니다. 저장 및 로드 메소드가 각각 *TWriterProc* 및 *TReaderProc*인 경우, 파일러(filer)의 *DefineProperty* 메소드를 호출합니다. *TStreamProc* 타입의 메소드를 만든 경우, *DefineBinaryProperty*를 호출합니다.

속성을 정의하는 데 사용하는 메소드에 상관 없이 속성 값을 저장하고 로드하는 메소드와 속성 값의 기록 여부를 나타내는 부울 값을 파일러에 전달합니다. 상속할 수 있는 값이거나 기본값을 가진 값인 경우에는 그 값을 작성할 필요가 없습니다.

예를 들어, *TReaderProc* 타입의 *LoadCompProperty* 메소드와 *TWriterProc* 타입의 *StoreCompProperty* 메소드가 있는 경우, 다음과 같이 *DefineProperties*를 오버라이드합니다.

```

procedure TSampleComponent.DefineProperties(Filer:TFile);

```

```

function DoWrite:Boolean;
begin
  if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
  begin
    if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
      Result := MyCompProperty <> nil
    else if MyCompProperty = nil or
      TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name
  then
    Result := True
  else Result := False;
  end
  else { no inherited value -- check for default (nil) value }
    Result := MyCompProperty <> nil;
  end;
begin
  inherited; { allow base classes to define properties }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;

```


43

이벤트 생성

이벤트는 사용자의 동작이나 포커스 변경 등과 같은 시스템에서 일어나는 사건과 그러한 사건에 응답하는 코드를 연결합니다. 응답 코드는 *이벤트 핸들러*이고 거의 대부분 애플리케이션 개발자에 의해 작성됩니다. 이벤트를 통해 애플리케이션 개발자는 클래스 자체를 변경하지 않고도 컴포넌트의 행동을 사용자 지정할 수 있습니다. 이것을 *위임 (delegation)*이라고 합니다.

마우스 동작과 같은 가장 일반적인 사용자 동작에 대한 이벤트는 모든 표준 컴포넌트에 내장되어 있지만 사용자가 새로운 이벤트를 정의할 수도 있습니다. 컴포넌트에서 이벤트를 만들려면 다음의 사항들을 이해할 필요가 있습니다.

- 이벤트의 개념
- 표준 이벤트 구현
- 사용자 고유 이벤트 정의

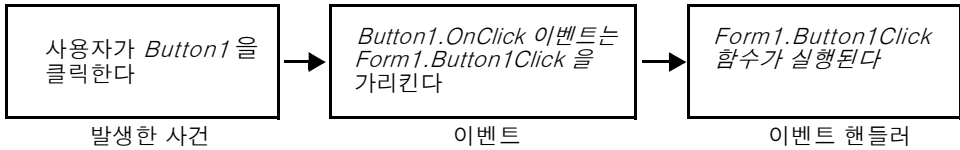
이벤트는 속성으로 구현되므로 컴포넌트의 이벤트를 만들거나 변경하기 전에 42장 "속성 생성"의 내용을 먼저 숙지해야 합니다.

이벤트의 개념

이벤트는 사건을 어떤 코드에 연결하는 메커니즘입니다. 보다 자세히 말하면 이벤트는 특정 클래스 인스턴스의 메소드를 가리키는 메소드 포인터입니다.

애플리케이션 개발자의 관점에서 보면 이벤트는 특정 코드를 덧붙일 수 있는 *OnClick*과 같은 시스템의 사건에 관련된 이름일 뿐입니다. 예를 들어, *Button1*이라는 누름 버튼은 *OnClick* 메소드를 가집니다. 기본적으로 Delphi는 버튼을 포함하는 폼에서 *Button1Click*이라는 이벤트 핸들러를 생성하고 *OnClick*에 할당합니다. 클릭 이벤트가 버튼에서 발생하면 버튼은 *OnClick*에 할당된 메소드를 호출하는데 이 경우에는 *Button1Click*입니다.

이벤트를 작성하려면 다음을 이해해야 합니다.



- 이벤트는 메소드 포인터
- 이벤트는 속성
- 이벤트 타입은 메소드 포인터 타입
- 이벤트 핸들러 타입은 프로시저
- 이벤트 핸들러는 옵션

이벤트는 메소드 포인터

Delphi는 메소드 포인터를 사용하여 이벤트를 구현합니다. 메소드 포인터는 인스턴스 객체에서 특정 메소드를 가리키는 특별한 포인터 타입입니다. 컴포넌트 작성자는 메소드 포인터를 위치 표시자로 취급할 수 있습니다. 코드에서 이벤트가 발생했음을 탐지한 경우 해당 이벤트에 대해 사용자가 지정한 메소드(있는 경우)를 호출합니다.

메소드 포인터는 다른 프로시저 타입처럼 작동하지만 객체에 대한 숨겨진 포인터를 유지합니다. 애플리케이션 개발자가 핸들러를 컴포넌트 이벤트에 할당하면 특정 이름을 갖는 메소드뿐만 아니라 특정 인스턴스 객체의 메소드에 대해서도 할당이 적용됩니다. 그 객체는 일반적으로 컴포넌트를 포함하는 폼이지만 반드시 그럴 필요는 없습니다.

예를 들어, 모든 컨트롤은 클릭 이벤트를 처리하기 위해 *Click*이라는 동적 메소드를 상속합니다.

```
procedure Click; dynamic;
```

*Click*의 구현은 사용자의 클릭 이벤트 핸들러가 있을 경우 그것을 호출합니다. 사용자가 컨트롤의 *OnClick* 이벤트에 핸들러를 할당한 경우, 컨트롤을 클릭하면 해당 메소드가 호출됩니다. 핸들러가 할당되지 않은 경우 아무 일도 일어나지 않습니다.

이벤트는 속성

컴포넌트는 속성을 사용하여 이벤트를 구현합니다. 대부분의 다른 속성들과는 달리 이벤트는 읽기 및 쓰기 부분을 구현하는 데 메소드를 사용하지 않습니다. 그 대신 이벤트 속성은 속성과 동일한 타입의 *private* 클래스 필드를 사용합니다.

규칙에 따라 필드 이름은 *F* 문자가 앞에 오는 속성의 이름입니다. 예를 들면, *OnClick* 메소드의 포인터는 *TNotifyEvent* 타입의 *FOnClick*이라고 하는 필드에 저장되고 *OnClick* 이벤트 속성의 선언은 다음과 같습니다.

```
type
  TControl = class(TComponent)
  private
    FOnClick:TNotifyEvent;           { declare a field to hold the method pointer }
    :
  end;
```



```
protected
property OnClick:TNotifyEvent read FOnClick write FOnClick;
end;
```

TNotifyEvent 및 다른 이벤트 타입에 대한 내용은 다음 단원의 "이벤트 타입은 메소드 포인터 타입"을 참조하십시오.

다른 속성과 마찬가지로 런타임 시 이벤트 값을 설정하거나 변경할 수 있습니다. 그러나 이벤트를 속성으로 만드는 가장 큰 이점은 컴포넌트 사용자가 Object Inspector를 사용하여 디자인 타임에 핸들러를 이벤트에 할당할 수 있다는 것입니다.

이벤트 타입은 메소드 포인터 타입

이벤트는 이벤트 핸들러에 대한 포인터이기 때문에 이벤트 속성의 타입은 메소드 포인터 타입이어야 합니다. 마찬가지로 이벤트 핸들러로 사용된 모든 코드는 적절하게 타입이 지정된 객체 메소드여야 합니다.

모든 이벤트 핸들러 메소드는 프로시저입니다. 주어진 타입의 이벤트와 호환 가능하려면 이벤트 핸들러 메소드는 동일한 순서로 동일한 매개변수의 수와 타입을 가지고 있어서 동일한 방식으로 전달되어야 합니다.

Delphi는 모든 표준 이벤트에 대한 메소드 타입을 정의합니다. 사용자 고유의 이벤트를 만들 경우, 적절한 기존 타입을 사용하거나 사용자 고유 타입을 정의할 수 있습니다.

이벤트 핸들러 타입은 프로시저

컴파일러를 통해 함수인 메소드 포인터 타입을 선언할 수 있지만 이벤트를 처리하기 위해 그렇게 해서는 안 됩니다. 빈 함수는 정의되지 않은 결과를 반환하기 때문에 함수로 작성된 비어 있는 이벤트 핸들러는 항상 유효할 수 없습니다. 이런 이유로 이벤트 및 그와 연결된 이벤트 핸들러는 모두 프로시저여야 합니다.

이벤트 핸들러가 함수일 수는 없지만 **var** 매개변수를 사용하여 애플리케이션 개발자의 코드로부터 여전히 정보를 얻을 수 있습니다. 이 작업 수행 시 값을 변경하는 데 사용자의 코드가 필요하지 않도록 하기 위해 핸들러를 호출하기 전에 매개변수에 유효한 값을 할당하는지 확인해야 합니다.

이벤트 핸들러에 대한 **var** 매개변수 전달의 예는 *TKeyPressEvent* 타입의 *OnKeyPress* 이벤트입니다. *TKeyPressEvent*는 두 개의 매개변수를 정의합니다. 하나의 매개변수는 이벤트를 생성한 객체를 나타내고 다른 매개변수는 눌려진 키를 나타냅니다.

```
type
  TKeyPressEvent = procedure(Sender:TObject; var Key: Char) of object;
```

보통 *Key* 매개변수는 사용자가 누른 문자를 포함합니다. 그러나 어떤 상황에서는 컴포넌트 사용자가 문자를 변경하고자 할 수도 있습니다. 편집기의 모든 문자를 대문자로 바꾸려 할 때가 이러한 경우입니다. 이 경우 사용자는 키 입력에 대한 다음과 같은 핸들러를 정의할 수 있습니다.

```
procedure TForm1.Edit1KeyPressed(Sender:TObject; var Key:Char);
begin
  Key := UpCase(Key);
end;
```

또한 **var** 매개변수를 사용하여 사용자가 기본 처리를 오버라이드하도록 할 수 있습니다.

이벤트 핸들러는 옵션

이벤트를 생성할 때 컴포넌트를 사용하는 개발자가 핸들러를 추가하지 않을 수도 있다는 것을 고려해야 합니다. 이는 특정 이벤트에 연결된 핸들러가 없으므로 컴포넌트가 실패하거나 오류를 발생시키지 않아야 한다는 것을 의미합니다. (핸들러를 호출하고 연결된 핸들러가 없는 이벤트를 처리하는 메커니즘에 대해서는 43-8 페이지의 "이벤트 호출"에서 설명합니다.)

이벤트는 GUI 애플리케이션에서 거의 지속적으로 발생합니다. 비주얼 컴포넌트에 마우스 포인터를 이동하면 컴포넌트가 *OnMouseMove* 이벤트로 번역하는 다수의 마우스 이동 메시지를 발생시킵니다. 대부분의 경우 개발자는 마우스 이동 이벤트를 처리하려고 하지 않는데 이것은 문제가 되지 않습니다. 따라서 사용자가 만든 컴포넌트는 자신의 이벤트에 대한 핸들러를 요구하지 않습니다.

게다가 애플리케이션 개발자는 이벤트 핸들러에 원하는 모든 코드를 작성할 수 있습니다. VCL 및 CLX의 컴포넌트에는 이벤트 핸들러가 오류를 발생시킬 경우를 최소화하기 위해 그러한 방법으로 작성된 이벤트가 있습니다. 애플리케이션 코드에서 로직 오류는 막을 수 없지만 애플리케이션 개발자가 유효하지 않은 데이터를 액세스하지 않도록 이벤트를 호출하기 전에 데이터 구조가 초기화되었는지는 확인할 수 있습니다.

표준 이벤트 구현

Delphi에서 제공하는 컨트롤은 가장 일반적인 발생에 대한 이벤트를 상속합니다. 이러한 이벤트를 *표준 이벤트*라고 합니다. 이러한 이벤트가 모두 컨트롤에 내장되어 있더라도 종종 **protected**이고 이는 개발자가 핸들러를 추가할 수 없다는 것을 의미합니다. 컨트롤을 만들 때 컨트롤 사용자가 이벤트를 볼 수 있도록 선택할 수 있습니다.

표준 이벤트를 컨트롤에 통합시킬 때 고려해야 할 사항은 다음 세 가지입니다.

- 표준 이벤트 식별
- 이벤트 보이기
- 표준 이벤트 처리 변경

표준 이벤트 식별

표준 이벤트에는 모든 컨트롤에 대해 정의된 표준 이벤트 및 표준 Windows 기반 컨트롤에 대해서만 정의된 표준 이벤트의 두 가지 부류가 있습니다

모든 컨트롤에 대한 표준 이벤트

가장 기본적인 이벤트는 *TControl* 클래스에서 정의됩니다. Windows 기반, 그래픽 또는 사용자 지정 여부와 관계 없이 모든 컨트롤은 이러한 이벤트를 상속합니다. 다음 이벤트는 모든 컨트롤에서 사용 가능합니다.

<i>OnClick</i>	<i>OnDragDrop</i>	<i>OnEndDrag</i>	<i>OnMouseMove</i>
<i>OnDbClick</i>	<i>OnDragOver</i>	<i>OnMouseDown</i>	<i>OnMouseUp</i>

표준 이벤트에는 이벤트 이름에 해당하는 이름을 갖는 *TControl*에서 선언된 해당 protected 비주얼 메소드가 있습니다. 예를 들어, *OnClick* 이벤트는 *Click*이라는 메소드를 호출하고, *OnEndDrag* 이벤트는 *DoEndDrag*라는 메소드를 호출합니다.

표준 컨트롤에 대한 표준 이벤트

모든 컨트롤에 공통된 이벤트 외에 표준 Windows 기반 컨트롤(VCL의 *TWinControl* 및 CLX의 *TWidgetControl*의 자손)은 다음 이벤트들을 갖습니다.

<i>OnEnter</i>	<i>OnKeyDown</i>	<i>OnKeyPress</i>
<i>OnKeyUp</i>	<i>OnExit</i>	

*TControl*의 표준 이벤트와 마찬가지로 Windows 기반 이벤트는 해당 메소드를 가집니다. 위에 나열된 표준 핵심 이벤트는 모든 보통의 키스트로크에 응답합니다.

VCL 참고

하지만 Alt 키와 같은 특별한 키스트로크에 응답하기 위해서는 Windows로부터의 WM_GETDLGCODE 또는 CM_WANTSPECIALKEYS 메시지에 응답해야 합니다. 메시지 핸들러 작성에 대한 정보는 46장 "메시지 처리"를 참조하십시오.

이벤트 보이기

TControl 및 *TWinControl*(CLX의 *TWidgetControl*)에서의 표준 이벤트는 protected 로 선언되며 선언은 *TControl* 및 *TWinControl*에 응답하는 메소드입니다. 이러한 추상 클래스 중 하나에서 상속하고 런타임 시 또는 디자인 타임에 이벤트를 액세스할 수 있도록 하려면 이벤트를 public 또는 published로 재선언해야 합니다.

구현을 지정하지 않고 속성을 재선언하면 동일한 구현 메소드가 유지되지만 보호 레벨이 변경됩니다. 그러므로 *TControl*에서 정의된 보이지 않는 이벤트를 가져와서 public 또는 published로 선언하여 이벤트를 표면화할 수 있습니다.

예를 들어, 디자인 타임 시 *OnClick* 이벤트를 표면화하는 컴포넌트를 만들려면 컴포넌트의 클래스 선언에 다음을 추가합니다.

```
type
  TMyControl = class(TCustomControl)
  :
  published
    property OnClick;
  end;
```

표준 이벤트 처리 변경

컴포넌트가 특정 유형의 이벤트에 응답하는 방식을 변경할 경우, 특정 코드를 작성하여 이벤트에 할당하고자 할 수 있습니다. 이것이 바로 애플리케이션 개발자가 하는 일입니다. 그러나 컴포넌트를 만들 때 컴포넌트를 사용하는 다른 개발자들이 이벤트를 사용할 수 있도록 해야 합니다.

이것이 각 표준 이벤트에 연결된 `protected` 구현 메소드에 대한 이유입니다. 구현 메소드를 오버라이드하여 내부 이벤트 처리를 수정할 수 있고 상속된 메소드를 호출하여 애플리케이션 개발자의 코드에 대한 이벤트를 포함하는 표준 처리를 유지 관리할 수 있습니다.

메소드를 호출하는 순서가 중요합니다. 일반적으로 상속된 메소드를 먼저 호출하여 사용자 지정 이전에 애플리케이션 개발자의 이벤트 핸들러가 실행되도록 합니다 (어떤 경우에는 사용자 지정이 실행되지 않도록 합니다.). 그러나 상속된 메소드를 호출하기 전에 자신의 코드를 먼저 실행하고자 할 경우도 종종 있습니다. 예를 들어, 상속된 코드가 컴포넌트의 상태에 다소 중속되어 있고 코드로 그 상태를 변경하는 경우에는 변경하고 나서 사용자 코드가 그 변경 내용에 응답하도록 해야 합니다.

컴포넌트를 작성하고 마우스 클릭에 응답하는 방식을 수정하려 한다고 가정합니다. 애플리케이션 개발자가 하는 것과 같이 `OnClick` 이벤트에 핸들러를 할당하는 대신 `protected` 메소드 `Click`을 오버라이드합니다.

```
procedure click override      { forward declaration }
:
:
procedure TMyControl.Click;
begin
  inherited Click;           { perform standard handling, including calling handler }
  ...                       { your customizations go here }
end;
```

사용자 고유 이벤트 정의

완전히 새로운 이벤트를 정의하는 것은 그다지 일반적이지 않습니다. 그러나 컴포넌트가 다른 컴포넌트의 행동과 완전히 다른 행동을 수행할 때 그에 대한 이벤트를 정의해야 할 필요가 있습니다.

이벤트를 정의할 때 고려해야 할 사항은 다음과 같습니다.

- 이벤트 트리거
- 핸들러 타입 정의
- 이벤트 선언
- 이벤트 호출

이벤트 트리거

이벤트를 트리거하는 것이 무엇인지 알아야 합니다. 일부 이벤트의 경우에는 그 대답이 명백합니다. 예를 들어, 마우스 다운 이벤트는 사용자가 마우스 왼쪽 버튼을 클릭했을

때 발생하며 Windows는 *WMLBUTTONDOWN* 메시지를 애플리케이션에 보냅니다. 그 메시지를 받으면 컴포넌트는 *MouseDown* 메소드를 호출하는데 이 메소드는 사용자가 *OnMouseDown* 이벤트에 연결한 코드를 호출합니다.

하지만 몇몇 이벤트는 특정 외부 사건에 대해 덜 명확하게 연결되어 있습니다. 예를 들어 스크롤 막대는 *OnChange* 이벤트를 가지는데 이 이벤트는 키 입력, 마우스 클릭 및 다른 컨트롤에서의 변경 사항 등을 포함한 여러 가지 종류의 사건에 의해 발생합니다. 이벤트를 정의할 때 모든 적절한 사건들이 적당한 이벤트를 호출하도록 해야 합니다.

두 종류의 이벤트

사용자가 이벤트를 제공할 필요가 있는 두 가지 경우가 있는데 사용자 상호 작용과 상태 변경이 있습니다. 사용자 상호 작용 이벤트는 사용자가 사용자의 컴포넌트가 응답할 필요가 있을 수 있는 무엇인가를 했다는 것을 나타내는 Windows로부터의 메시지에 의해 거의 항상 트리거됩니다. 상태 변경 이벤트도 Windows로부터의 메시지(예를 들어, 포커스 변경 또는 활성화)와 관계가 있을 수 있지만 이 이벤트는 속성 또는 기타 코드의 변경을 통해서도 나타날 수 있습니다.

자신이 정의하는 이벤트의 트리거를 완전히 제어할 수 있습니다. 개발자가 이해하고 사용할 수 있도록 이벤트를 신중하게 정의해야 합니다.

핸들러 타입 정의

이벤트 발생 시기를 결정한 후 이벤트 처리 방법을 정의해야 합니다. 즉, 이벤트 핸들러의 타입을 결정해야 합니다. 대부분의 경우 자신이 직접 정의한 이벤트의 핸들러는 일반 공지 타입 또는 이벤트 특정 타입입니다. 핸들러에서 거꾸로 정보를 얻는 것도 가능합니다.

간단한 공지

공지 이벤트는 이벤트 발생 시기나 위치에 대한 자세한 정보 없이 특정 이벤트가 발생했다는 것만 알려 주는 이벤트입니다. 공지 이벤트는 단 하나의 매개변수, 이벤트를 보내는 사람만 알리는 *TNotifyEvent* 타입을 사용합니다. 공지를 위한 모든 핸들러는 이벤트가 어떤 타입인지, 어떤 컴포넌트에서 이벤트가 발생했는지에 대해 "알고" 있습니다. 예를 들어, 클릭 이벤트는 공지에 해당합니다. 클릭 이벤트의 핸들러를 작성할 때에는 어떤 클릭이 발생했으며 어떤 컴포넌트를 클릭했는지에 대한 것만 알 수 있습니다.

공지는 단방향 프로세스입니다. 피드백을 제공하거나 공지가 더 이상 처리되지 않도록 하는 메커니즘은 없습니다.

이벤트 특정 핸들러

어떤 경우에는 발생한 이벤트와 이벤트가 발생한 컴포넌트를 아는 것만으로 충분하지 않습니다. 예를 들어, 이벤트가 키 입력 이벤트인 경우 핸들러는 사용자가 누른 키를 알고자 합니다. 이러한 경우 추가 정보를 위한 매개변수를 포함하는 핸들러 타입이 있어야 합니다.

메시지에 응답하여 이벤트가 생성된 경우 이벤트 핸들러에 전달하는 매개변수는 메시지 매개변수에서 직접 가져올 수 있습니다.

핸들러의 정보 반환

모든 이벤트 핸들러는 프로시저이기 때문에 핸들러에서 역으로 정보를 전달하는 유일한 방법은 **var** 매개변수를 통하는 것입니다. 컴포넌트는 이 정보를 사용하여 사용자의 핸들러가 실행된 후 이벤트를 처리할지 여부 및 그 방법을 결정합니다.

예를 들어, 모든 키 이벤트 (*OnKeyDown*, *OnKeyUp* 및 *OnKeyPress*)는 *Key*라는 매개변수에서 눌린 키 값을 참조에 의해 전달합니다. 이벤트 핸들러는 애플리케이션에서 이벤트에 관련된 서로 다른 키를 볼 수 있도록 *Key*를 변경할 수 있습니다. 예를 들면, 이렇게 함으로써 입력된 문자를 대문자로 바꿀 수 있습니다.

이벤트 선언

이벤트 핸들러의 타입을 결정했으면 이제 이벤트의 메소드 포인터와 속성을 선언할 수 있습니다. 사용자가 이벤트가 수행하는 작업을 잘 알 수 있도록 이벤트에 의미있고 서술적인 이름을 제공하도록 합니다. 다른 컴포넌트의 비슷한 속성들의 이름과 일관성을 유지하도록 합니다.

"On"으로 시작하는 이벤트 이름

Delphi에 있는 대부분의 이벤트 이름은 "On"으로 시작합니다. 이것은 단지 규칙일 뿐이며 컴파일러에서 이것을 요구하는 것은 아닙니다. Object Inspector는 속성 타입을 보고 이벤트인지 알아냅니다. 모든 메소드 포인터 속성은 이벤트로 간주되어 Events 페이지에 나타납니다.

개발자는 "On"으로 시작하는 이름의 알파벳 순서 목록에서 이벤트를 찾고자 합니다. 다른 종류의 이름을 사용하면 혼동을 일으킬 수 있습니다.

참고 이 규칙에 대한 주요 예외는 "Before" 및 "After"로 시작하는 일부 발생 사건 전후에 발생하는 다양한 이벤트입니다.

이벤트 호출

이벤트에 대한 호출을 중앙 집중화해야 합니다. 즉, 컴포넌트 안에 할당된 것이 있을 경우 애플리케이션의 이벤트 핸들러를 호출하고 기본 처리를 제공하는 가상 메소드를 만듭니다.

모든 이벤트 호출을 한 곳에 두면 사용자의 컴포넌트에서 새 컴포넌트를 파생시키는 누군가가 이벤트가 호출되는 장소를 찾기 위해 사용자의 코드를 검색하지 않고도 하나의 메소드를 오버라이드함으로써 이벤트 처리를 사용자 지정할 수 있습니다.

이벤트를 호출할 때 더 고려해야 할 두 가지 사항은 다음과 같습니다.

- 비어 있는 핸들러도 유효해야 합니다.
- 사용자는 기본 처리를 오버라이드할 수 있습니다.

비어 있는 핸들러도 유효해야 합니다 .

비어 있는 이벤트 핸들러가 오류를 발생하는 상황을 만들거나 컴포넌트의 기능 수행이 애플리케이션의 이벤트 처리 코드의 특정 응답에 종속되어서는 안 됩니다.

비어 있는 핸들러는 핸들러가 전혀 없는 것과 동일한 결과를 만들지 않아야 합니다. 따라서 애플리케이션의 이벤트 핸들러를 호출하기 위한 코드는 다음과 같아야 합니다.

```
if Assigned(OnClick) then OnClick(Self);
... { perform default handling }
```

다음과 같이 코드를 작성해서는 안 됩니다.

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

사용자는 기본 처리를 오버라이드할 수 있습니다 .

어떤 종류의 이벤트에 대해서는 개발자가 기본 처리를 대체하거나 모든 응답을 막습니다. 이렇게 할 수 있으려면 핸들러가 반환할 때 핸들러에 참조에 의한 인수를 전달하고 특정 값을 확인해야 합니다.

이는 비어 있는 핸들러가 핸들러가 없는 경우와 동일한 결과를 내서는 안 된다는 규칙을 지키려는 것입니다. 비어 있는 핸들러는 참조에 의해 전달된 인수 값을 변경하지 않기 때문에 기본 처리는 비어 있는 핸들러를 호출한 후에 항상 발생합니다.

예를 들어, 키 누름 이벤트를 처리할 때 애플리케이션 개발자는 **var** 매개변수 *Key*를 Null 문자(#0)로 설정하여 키스트로크의 컴포넌트 기본 처리를 막을 수 있습니다. 이를 지원하는 로직은 다음과 같습니다.

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then ... { perform default handling }
```

Windows 메시지를 다루기 때문에 실제 코드는 위와 약간 다르지만 로직은 같습니다. 기본적으로 컴포넌트는 사용자 지정 핸들러를 호출한 다음 표준 처리를 수행합니다. 사용자 핸들러가 *Key*를 Null 문자로 설정하면 컴포넌트는 기본 처리를 생략합니다.

44

메소드 생성

컴포넌트 메소드는 클래스 구조에 내장된 프로시저 및 함수입니다. 컴포넌트 메소드를 통해 수행할 수 있는 작업에 특별한 제한은 없지만 Delphi는 준수해야 할 특정 표준을 사용합니다. 다음 내용이 포함되어 있습니다.

- 종속성 피하기
- 메소드 이름 지정
- 메소드 보호
- 가상 메소드 만들기
- 메소드 선언

일반적으로 컴포넌트는 많은 메소드를 포함해서는 안 되며 애플리케이션에서 호출해야 할 메소드 수를 최소화해야 합니다. 메소드로 구현하고 싶은 기능이 속성으로 보다 잘 캡슐화되는 경우가 많습니다. 속성은 Delphi 환경에 맞는 인터페이스를 제공하고 디자인 타임에 액세스할 수 있습니다.

종속성 피하기

컴포넌트를 작성할 때에는 항상 개발자에게 주어지는 전제 조건을 최소화하십시오. 최대한의 확장성을 위해 개발자가 원할 때 언제든지 컴포넌트에서 원하는 모든 작업을 할 수 있어야 합니다. 그럴 수 없는 경우가 있겠지만 가능한 한 조건을 최소화하는 것을 목표로 해야 합니다.

다음 목록은 피해야 할 종속성의 종류입니다.

- 컴포넌트를 사용하기 위해 사용자가 반드시 호출해야 하는 메소드
- 특정 순서로 실행해야 할 메소드
- 컴포넌트를 특정 이벤트나 메소드를 사용할 수 없는 상태나 모드로 가져가는 메소드

이러한 상황을 처리하는 가장 좋은 방법은 그 상황에서 빠져 나올 수 있는 방법을 제공하는 것입니다. 예를 들어 어떤 한 메소드를 호출하는 경우 컴포넌트에서 다른 메소드를

호출하지 못하는 상태에 빠진다면 또 다른 메소드를 작성해서 컴포넌트가 비정상적인 상태에 빠졌을 때 그 메소드를 호출해서 메인 코드 실행 전에 컴포넌트의 비정상적 상태를 바로잡도록 만들 수 있을 것입니다. 사용자가 유효하지 않은 메소드를 호출하는 경우에는 최소한 예외를 발생시켜야 합니다.

즉, 사용자가 코드의 일부가 서로 종속되어 있는 상황을 만든 경우에 코드를 잘못 사용하여 문제가 일어나지 않도록 하는 책임은 *사용자에게* 있습니다. 예를 들어 사용자가 종속성을 적용하지 않는 경우, 경고 메시지를 내보내는 것이 시스템 실패보다는 나은 방법입니다.

메소드 이름 지정

Delphi는 메소드나 그 매개변수의 이름 지정에 대해 제한을 두지 않습니다. 단지 애플리케이션 개발자가 메소드를 더 쉽게 사용할 수 있도록 하는 몇 가지 규칙이 있습니다. 다양한 부류의 사람들이 컴포넌트를 사용할 수 있도록 컴포넌트 아키텍처를 구성해야 합니다.

사용자가 자신이나 소그룹의 프로그래머만 사용하는 코드를 작성하는 데 익숙해져 있다면 이름 지정 방법을 크게 고민하지 않았을지도 모릅니다. 컴포넌트는 사용자의 코드에 익숙하지 않은 사람들과 코드에 익숙하지 않은 사람들이 사용해야 하므로 메소드 이름을 명확하게 지정하는 것이 좋습니다.

다음은 메소드의 이름을 명확하게 지정하기 위한 몇 가지 제안입니다.

- 서술적인 이름을 지정합니다. 의미 있는 동사를 사용합니다.

*PasteFromClipboard*와 같은 이름은 *Paste* 또는 *PFC*보다 더 많은 정보를 알려 줍니다.

- 함수 이름은 반환하는 값의 성격을 반영해야 합니다.

프로그래머들은 *X*라는 함수가 어떤 것의 수평 좌표를 반환한다는 것을 분명히 알 수 있지만 *GetHorizontalPosition*이라는 이름이 보다 보편적으로 이해하기 쉽습니다.

마지막으로 고려할 사항은 메소드가 실제로 메소드여야 하는지 확인합니다. 메소드 이름에 동사가 포함되는 것이 좋습니다. 이름에 동사가 없는 메소드를 많이 만들었다면 그러한 메소드가 속성이어야 할지 고려하십시오.

메소드 보호

필드, 메소드 및 속성 등을 포함한 클래스의 모든 부분은 41-4 페이지의 "액세스 제어"에서 설명한 대로 보호 또는 "가시성" 레벨을 가집니다. 메소드에 대한 적절한 가시성을 선택하는 것은 간단합니다.

컴포넌트에서 작성하는 대부분의 메소드는 **public** 또는 **protected**입니다. 메소드가 실제로 컴포넌트의 그 타입에 특정적이지 않다면 메소드를 **private**으로 만들고 심지어 파생된 컴포넌트에서도 액세스할 수 없게 할 필요가 없습니다.

Public이어야 하는 메소드

애플리케이션 개발자가 호출하고자 하는 모든 메소드는 **public**으로 선언되어야 합니다. 대부분의 메소드 호출은 이벤트 핸들러에서 일어난다는 것을 기억해 두십시오. 따라서 메소드가 시스템 리소스를 잡아 두거나 운영 체제를 사용자에게 응답할 수 없는 상태로 가져가는 것을 피해야 합니다.

참고 생성자와 소멸자는 항상 **public**이어야 합니다.

Protected여야 하는 메소드

애플리케이션이 적절하지 않은 시기에 메소드를 호출하지 않도록 컴포넌트의 구현 메소드는 **protected**여야 합니다. 애플리케이션 코드에서 호출되어서는 안 되지만 파생 클래스에서 호출되는 메소드를 가지고 있는 경우, 이 메소드를 **protected**로 선언합니다.

예를 들어, 이미 설정된 특정 데이터를 가지는 메소드가 있다고 가정합니다. 이 메소드를 **public**으로 만들 경우, 애플리케이션이 데이터를 설정하기 전에 메소드를 호출할 가능성이 있습니다. 그 반면 이 메소드를 **protected**로 만들면 애플리케이션이 직접 메소드를 호출할 수 없습니다. 그럴 경우, **protected** 메소드를 호출하기 전에 데이터가 설정되도록 하는 다른 **public** 메소드를 설정할 수 있습니다.

속성 구현 메소드는 가상 **protected** 메소드로 선언되어야 합니다. 애플리케이션 개발자는 이와 같이 선언된 메소드를 통해 속성 구현을 오버라이드할 수 있으며 그 기능을 확대하거나 완전히 교체할 수 있습니다. 그러한 속성은 완전히 다형적입니다. 메소드 액세스를 **protected**로 둬으로써 개발자가 잘못하여 메소드를 호출하여 부주의하게 속성을 수정하지 않도록 막을 수 있습니다.

추상 메소드

때때로 메소드는 Delphi 컴포넌트에서 **abstract**로 선언됩니다. VCL과 CLX에서 추상 메소드는 *TCustomGrid*와 같이 "custom"으로 시작하는 이름을 가진 클래스에서 주로 나타납니다. 이러한 클래스는 오직 파생된 자손 클래스에 영향을 주기 위해 만들어지므로 그들 자신은 추상적입니다.

추상 멤버를 포함하는 클래스의 인스턴스 객체를 만들 수는 있지만 권장되지 않습니다. 추상 멤버를 호출하면 *EAbstractError* 예외가 발생합니다.

abstract 지시어는 자손 컴포넌트에서 표면화되고 정의되어야 하는 클래스의 부분을 나타내는 데 사용되며 컴포넌트 작성자가 클래스의 실제 인스턴스를 생성하기 전에 자손 클래스에서 추상 멤버를 재선언하도록 합니다.

가상 메소드 만들기

동일한 메소드 호출에 대한 응답으로 서로 다른 타입의 반응을 보이는 코드를 실행할 수 있도록 할 경우 메소드를 **virtual**로 만듭니다.

애플리케이션 개발자가 직접 사용할 수 있는 컴포넌트를 만들 경우, 모든 메소드를 비가상(nonvirtual)으로 만들 수 있습니다. 그 반면 다른 컴포넌트가 파생될 추상 컴포넌트를 만들 경우, 추가된 메소드를 **virtual**로 만들 수 있습니다. 이렇게 하면 파생된 컴포넌트는 상속된 **가상** 메소드를 오버라이드할 수 있습니다.

메소드 선언

컴포넌트에서 메소드를 선언하는 것은 클래스 메소드를 선언하는 것과 동일합니다.

컴포넌트에서 새 메소드를 선언하려면 다음 두 가지 작업을 수행합니다.

- 선언을 컴포넌트의 객체 타입 선언에 추가합니다.
- 컴포넌트 유닛의 **implementation** 부분에 메소드를 구현합니다.

다음 코드는 두 개의 새 메소드, 즉 protected 정적 메소드와 public 가상 메소드를 정의하는 컴포넌트를 보여 줍니다.

```

type
  TSampleComponent = class(TControl)
    protected
      procedure MakeBigger;                { declare protected static method }
    public
      function CalculateArea:Integer; virtual;    { declare public virtual method }
    end;
  :
implementation
  :
  procedure TSampleComponent.MakeBigger;      { implement first method }
  begin
    Height := Height + 5;
    Width := Width + 5;
  end;
  function TSampleComponent.CalculateArea:Integer;    { implement second method }
  begin
    Result := Width * Height;
  end;

```

45

컴포넌트에서 그래픽 사용

Windows는 장치와 무관하게 그래픽을 그릴 수 있는 강력한 그래픽 장치 인터페이스 (GDI)를 제공합니다. 그러나 GDI에는 프로그래머가 그래픽 리소스를 관리해야 하는 것과 같은 별도의 요구 사항이 따릅니다. Delphi는 GDI의 단순한 일들을 처리하므로 사용자가 소실된 핸들이나 배포되지 않은 리소스를 찾는 대신 보다 생산적인 작업에 집중할 수 있게 합니다.

Windows API의 모든 부분에서와 마찬가지로 Delphi 애플리케이션에서 GDI 함수를 직접 호출할 수 있습니다. 그러나 Delphi의 그래픽 기능의 캡슐화가 더 빠르고 쉽다는 것을 알게 될 것입니다.

이 단원에서는 다음과 같은 주제들을 다룹니다.

- 그래픽의 개요
- 캔버스 사용
- 그림 작업
- 오프스크린(off-screen) 비트맵
- 변화에 대한 반응

그래픽의 개요

Delphi는 몇 가지 레벨에서 Windows GDI를 캡슐화합니다. 컴포넌트 작성자에게 가장 중요한 것은 컴포넌트가 이미지를 스크린에 표시하는 방법입니다. GDI 함수를 직접 호출할 때에는 사용자가 펜, 브러시 및 글꼴과 같은 다양한 드로잉 툴을 선택한 장치 컨텍스트에 핸들이 필요합니다. 그래픽 이미지를 렌더링한 다음 그래픽 이미지를 배치하기 전에 반드시 장치 컨텍스트를 원래 상태에 저장해야 합니다.

CLX 참고 GDI 함수는 Windows 특정이므로 CLX 또는 크로스 플랫폼 애플리케이션에 적용되지 않습니다.

Delphi에서는 자세한 수준까지 다루지 않아도 되는 간단하면서도 완벽한 인터페이스, 즉 컴포넌트의 *Canvas* 속성을 제공합니다. 캔버스는 유효한 도구 컨텍스트를 가지고

있으며 사용자가 캔버스를 사용하지 않을 때 컨텍스트를 해제하도록 합니다. 또한 캔버스는 현재 사용하는 펜, 브러시 및 글꼴을 나타내는 속성을 가집니다.

캔버스에서 리소스를 관리하므로 사용자가 펜 핸들과 같은 것을 생성하고, 선택하고, 해제하는 것에 신경을 쓰지 않아도 됩니다. 단지 캔버스에 사용할 펜의 유형을 알려 주면 나머지는 자동으로 처리됩니다.

Delphi에서 그래픽 리소스를 관리할 수 있도록 한 이점 중 하나는 나중에 사용하기 위해 리소스를 캐시로 저장할 수 있어서 반복적인 작업을 신속하게 수행할 수 있다는 것입니다. 예를 들어, 사용자가 특정한 유형의 도구를 반복적으로 생성, 사용 및 처분하는 프로그램을 가지고 있는 경우, 해당 도구를 사용할 때마다 그러한 단계를 반복해야 합니다. Delphi에서는 그래픽 리소스를 캐시하기 때문에 반복적으로 사용하는 도구가 캐시에 아직 남아 있는 경우 Delphi에서 도구를 다시 만들지 않고 기존 도구를 사용하면 됩니다.

이러한 예는 수백 개의 컨트롤을 가지고 수십 개의 폼을 여는 애플리케이션에서 볼 수 있습니다. 이러한 컨트롤 각각에 하나 이상의 *TFont* 속성이 있을 수 있습니다. 결과적으로 *TFont* 객체의 수백, 수천 개의 인스턴스가 생성되지만 대부분의 애플리케이션은 글꼴 캐시 덕분에 두 세 개의 글꼴 핸들만 사용하는 것으로 모든 것이 해결됩니다.

Delphi의 그래픽 코드가 얼마나 쉬운지 보여 주는 두 가지 예가 있습니다. 첫 번째는 사용자가 다른 개발 툴을 사용할 때와 같은 방법으로 표준 GDI 함수를 사용하여 창에서 파란 색 테두리가 있는 노란 생략 버튼을 그리는 것입니다. 두 번째는 캔버스를 사용하여 Delphi로 작성된 애플리케이션에 동일한 생략 버튼을 그리는 것입니다.

```

procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
    PenHandle, OldPenHandle: HPEN;
    BrushHandle, OldBrushHandle: HBRUSH;
begin
    PenHandle := CreatePen(PS_SOLID, 1, RGB(0, 0, 255));           { create blue pen }
    OldPenHandle := SelectObject(PaintDC, PenHandle);           { tell DC to use blue pen }
    BrushHandle := CreateSolidBrush(RGB(255, 255, 0));           { create a yellow brush }
    OldBrushHandle := SelectObject(PaintDC, BrushHandle);       { tell DC to use yellow brush }
    Ellipse(HDC, 10, 10, 50, 50);                               { draw the ellipse }
    SelectObject(OldBrushHandle);                               { restore original brush }
    DeleteObject(BrushHandle);                                  { delete yellow brush }
    SelectObject(OldPenHandle);                                 { restore original pen }
    DeleteObject(PenHandle);                                   { destroy blue pen }
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    with Canvas do
        begin
            Pen.Color := clBlue;                                { make the pen blue }
            Brush.Color := clYellow;                            { make the brush yellow }
            Ellipse(10, 10, 50, 50);                            { draw the ellipse }
        end;
end;

```

캔버스 사용

캔버스 클래스는 개별 선, 모양 및 텍스트를 그리기 위한 수준 높은 함수 및 캔버스의 그리기 기능을 조작하기 위한 중간 속성을 포함하여 몇몇 수준에서 그래픽 컨트롤을 캡슐화하며 VCL에서는 Windows GDI로의 저수준 액세스를 제공합니다.

표 45.1은 캔버스의 기능을 요약합니다.

표 45.1 캔버스 기능 요약

수준	작동	틀
높음	선과 모양 그리기	<i>MoveTo</i> , <i>LineTo</i> , <i>Rectangle</i> 및 <i>Ellipse</i> 와 같은 메소드
	텍스트 표시 및 측정	<i>TextOut</i> , <i>TextHeight</i> , <i>TextWidth</i> 및 <i>TextRect</i> 메소드
	영역 채우기	<i>FillRect</i> 및 <i>FloodFill</i> 메소드
중간	텍스트와 그래픽 사용자 지정	<i>Pen</i> , <i>Brush</i> 및 <i>Font</i> 속성
	픽셀 조작	<i>Pixels</i> 속성
	이미지 복사 및 합치기	<i>Draw</i> , <i>StretchDraw</i> , <i>BrushCopy</i> , 및 <i>CopyRect</i> 메소드, <i>CopyMode</i> 속성
낮음	Windows GDI 함수 호출	<i>Handle</i> 속성

캔버스 클래스 및 그 메소드와 속성에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

그림 작업

Delphi에서 수행하는 대부분의 그래픽 작업은 컴포넌트와 폼의 캔버스에 직접 그리는 것으로 제한됩니다. Delphi에서는 또한 팔레트의 자동 관리를 포함하여 비트맵, 메타파일 및 아이콘과 같은 독립적인 그래픽 이미지를 처리합니다.

Delphi에서 그림 작업을 하는 데에는 다음과 같은 세 가지 중요한 사항이 있습니다.

- 그림, 그래픽 또는 캔버스 사용
- 그래픽 로드 및 저장
- 팔레트 처리

그림, 그래픽 또는 캔버스 사용

Delphi에는 그래픽을 다루는 세 종류의 클래스가 있습니다.

- *canvas*는 폼, 그래픽, 컨트롤, 프린터 또는 비트맵의 비트맵된 드로잉 표면을 나타냅니다. 캔버스는 독립적인 클래스가 아닌 다른 클래스의 속성입니다.
- *graphic*은 비트맵, 아이콘 및 메타파일과 같은 파일이나 리소스에서 일반적으로 발견되는 유형의 그래픽 이미지를 나타냅니다. Delphi에서는 *TBitmap*, *TIcon* 및 *TMetafile* 클래스를 정의하는데 이 모든 클래스는 일반 *TGraphic*의 자손 클래스입니다. 또한 사용자는 자신의 그래픽 클래스를 정의할 수도 있습니다. 모든 그래픽의 최소 표준 인터페이스를 정의하여 *TGraphic*은 애플리케이션의 단순한 메커니즘을 제공하여 다양한 유형의 그래픽을 쉽게 이용할 수 있습니다.
- *picture*는 그래픽을 담아 두는 컨테이너로서 임의의 그래픽 클래스를 포함할 수 있습니다. 즉, *TPicture* 타입의 항목은 비트맵, 아이콘, 메타파일 또는 사용자 정의 그래픽 타입을 포함할 수 있고, 애플리케이션은 그림 클래스를 통해 동일한 방법으로 모든 그래픽에 액세스할 수 있습니다. 예를 들어, 이미지 컨트롤은 *TPicture* 타입의 *Picture*라는 속성을 가지며 이를 통해 컨트롤은 다양한 종류의 그래픽 이미지를 표시할 수 있습니다.

picture 클래스는 항상 *graphic*을 가지고 있고 *graphic*은 *canvas*를 가질 수 있다는 것을 명심하십시오. (*canvas*를 가진 유일한 표준 그래픽은 *TBitmap*입니다.) 일반적으로 그림을 처리할 때 *TPicture*에 의해 드러나는 그래픽 클래스의 부분만 사용합니다. 그래픽 클래스 자체의 특정 부분에 액세스해야 하는 경우, *picture*의 *Graphic* 속성을 참조할 수 있습니다.

그래픽 로드 및 저장

Delphi의 모든 그림과 그래픽은 파일에서 이미지를 로드하고 다시 그 파일(또는 다른 파일)에 저장할 수 있습니다. 언제든지 그림의 이미지를 로드하거나 저장할 수 있습니다.

CLX 참고 CLX 컴포넌트를 만들면 이미지를 로드하여 Qt MIME 소스나 스트림 객체로 저장할 수도 있습니다.

이미지를 파일의 그림으로 로드하려면 *LoadFromFile* 메소드를 호출합니다. 그림의 이미지를 파일에 저장하려면 *SaveToFile* 메소드를 호출합니다.

*LoadFromFile*과 *SaveToFile*은 각각 유일한 매개변수로서 파일의 이름을 취합니다. *LoadFromFile*은 생성하고 로드할 그래픽 객체의 유형을 결정하기 위해 파일 이름의 확장자를 사용합니다. *SaveToFile*은 저장되는 그래픽 객체 타입에 적절한 파일 타입으로 저장합니다.

예를 들어, 비트맵을 이미지 컨트롤의 그림으로 로드하려면 비트맵 파일의 이름을 그림의 *LoadFromFile* 메소드에 전달합니다.

```
procedure TForm1.LoadBitmapClick(Sender:TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.BMP');
end;
```


그림은 bmp를 비트맵 파일의 표준 확장자로 인식하므로 *TBitmap*으로 그래픽을 생성한 다음 그래픽의 *LoadFromFile* 메소드를 호출합니다. 그래픽이 비트맵이기 때문에 파일의 이미지를 비트맵으로 로드합니다.

팔레트 처리

VCL 컴포넌트가 팔레트 기반 장치(일반적으로 256색 비디오 모드)에서 작동하면 Delphi의 컨트롤은 자동으로 팔레트를 실현합니다. 즉, 팔레트가 들어 있는 컨트롤이 있다면 사용자는 *TControl*에서 상속된 두 가지 메소드를 사용하여 Windows에서 팔레트를 사용하도록 조정할 수 있습니다.

컨트롤에 대한 팔레트 지원에는 다음 두 가지 용도가 있습니다.

- 컨트롤에 팔레트 지정
- 팔레트 변경에 대한 응답

대부분의 컨트롤에는 팔레트가 필요하지 않지만 이미지 컨트롤과 같은 "rich color" 그래픽 이미지를 포함한 컨트롤은 Windows 및 스크린 장치 드라이버와 상호 작용하여 컨트롤의 적절한 모습을 확인해야 합니다. Windows는 이러한 과정을 참조하여 팔레트를 실현합니다.

팔레트를 실현한다는 것은 가장 중요한 창이 자신의 팔레트를 충분히 사용하고 배경이 되는 창이 팔레트를 최대한 많이 사용하고 나서 색상들을 "real" 팔레트의 사용 가능한 가장 가까운 색상으로 매핑하는 과정입니다. 창들이 다른 창 앞에서 움직이므로 Windows는 계속적으로 팔레트를 실현합니다.

참고 Delphi는 비트맵 외에서는 팔레트를 생성하거나 유지하는 특정한 지원을 제공하지 않습니다. 그러나 팔레트 핸들이 있으면 Delphi 컨트롤이 사용자 대신 팔레트를 관리해 줍니다.

컨트롤에 팔레트 지정

팔레트의 핸들을 반환하도록 컨트롤의 *GetPalette* 메소드를 오버라이드하여 VCL 컨트롤에 팔레트를 지정합니다.

컨트롤에 팔레트를 지정하면 다음과 같은 작업을 수행합니다.

- 사용자 컨트롤의 팔레트가 실현될 필요가 있음을 애플리케이션에 알립니다.
- 실현하는 데 사용할 팔레트를 지정합니다.

팔레트 변경에 대한 응답

VCL 컨트롤이 *GetPalette*를 오버라이드하여 팔레트를 지정하면 Delphi는 Windows의 팔레트 메시지에 자동적으로 응답합니다. 팔레트 메시지를 처리하는 메소드는 *PaletteChanged*입니다.

*PaletteChanged*의 가장 중요한 역할은 컨트롤의 팔레트를 전경에서 실현할지 배경에서 실현할지 결정하는 것입니다. Windows에서는 맨 위의 창이 전경의 팔레트를 갖게 하고 다른 창들은 배경 팔레트가 되게 하여 이러한 팔레트의 실현을 처리합니다. Delphi는 한 단계 더 나아가 창 안에서 탭 순서에 따라 팔레트를 실현하기도 합니다. 이렇게 기

본 동작을 오버라이드하는 것은 사용자가 탭 순서에서 첫 번째가 아닌 컨트롤이 전경 팔레트를 갖는 것을 원할 경우에만 필요합니다.

오프스크린(off-screen) 비트맵

복잡한 그래픽 이미지를 그릴 때 그래픽 프로그래밍의 일반적인 방법은 오프스크린 비트맵을 생성하여 비트맵에 이미지를 그리고 비트맵으로부터 완성된 이미지를 최종 대상 온스크린 (onscreen) 에 복사하는 것입니다. 오프스크린 이미지를 사용하면 스크린에 직접 그리기를 반복함으로써 야기되는 화면 떨림을 줄일 수 있습니다.

리소스와 파일의 비트맵된 이미지를 나타내는 Delphi의 비트맵 클래스는 오프스크린 이미지로 사용될 수도 있습니다.

오프스크린 비트맵을 사용하는 데에는 두 가지 용도가 있습니다.

- 오프스크린 비트맵 생성 및 관리.
- 비트맵 이미지 복사.

오프스크린 비트맵 생성 및 관리

복잡한 그래픽 이미지를 생성할 때 온스크린 (onscreen)에 나타나는 캔버스에 직접 그리는 것을 피하는 것이 좋습니다. 폼이나 컨트롤을 캔버스에 그리는 대신 비트맵 객체를 구성하고 캔버스에 그린 다음 완성된 이미지를 온스크린 캔버스에 복사하면 됩니다.

오프스크린 비트맵의 가장 일반적인 사용은 그래픽 컨트롤의 *Paint* 메소드에서 이루어 집니다. 임시 객체로의 비트맵 객체는 **try..finally** 블록 안에서 보호되어야 합니다.

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;           { override the Paint method }
  end;

procedure TFancyControl.Paint;
var
  Bitmap:TBitmap;                       { temporary variable for the off-screen bitmap }
begin
  Bitmap := TBitmap.Create;             { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                         { destroy the bitmap object }
  end;
end;
```

비트맵 이미지 복사

Delphi는 하나의 캔버스에서 다른 곳으로 이미지를 복사하는 방법 네 가지를 제공합니다. 만들어 내고자 하는 효과에 따라서 다른 메소드를 호출합니다.

표 45.2는 캔버스 객체의 이미지 복사 메소드를 요약한 것입니다.

표 45.2 이미지 복사 메소드

효과	호출하는 메소드
전체 그래픽 복사	Draw
그래픽 복사과 크기 조정	StretchDraw
캔버스 일부분 복사	CopyRect
래스터 연산으로 비트맵 복사	BrushCopy (VCL)
그래픽을 반복적으로 복사해서 영역에 타일 모양으로 나타내기	TiledDraw (CLX)

변화에 대한 반응

캔버스와 그에 속한 객체(펜, 브러시 및 글꼴)들을 포함해서 모든 그래픽 객체에는 객체의 변화에 반응하기 위한 이벤트가 내장되어 있습니다. 이러한 이벤트를 사용하여 이미지를 다시 그려서 컴포넌트나 컴포넌트를 사용하는 애플리케이션이 변화에 반응하도록 할 수 있습니다.

그래픽 객체의 변화에 반응하는 것은 컴포넌트의 디자인 타임 인터페이스의 일부로 게시하는 경우 매우 중요합니다. 컴포넌트의 디자인 타임 외관이 Object Inspector에 설정된 속성과 일치하는지 확인하는 유일한 방법은 객체의 변화에 반응하는 것입니다.

그래픽 객체의 변화에 반응하려면 클래스의 *OnChange* 이벤트에 메소드를 할당합니다.

Tshape 컴포넌트는 형태를 그릴 때 사용하는 펜과 브러시를 나타내는 속성을 게시합니다. 컴포넌트의 생성자는 *OnChange* 이벤트에 펜이나 브러시가 변경되는 경우 컴포넌트가 자신의 이미지를 새로 고치도록 하는 메소드를 할당합니다.

```

type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender:TObject);
  end;
  ..
implementation
  ..
constructor TShape.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                 { construct the pen }
  FPen.OnChange := StyleChanged;      { assign method to OnChange event }
  FBrush := TBrush.Create;            { construct the brush }

```

변화에 대한 반응

```
    FBrush.OnChange := StyleChanged;           { assign method to On Change }  
end;  
procedure TShape.StyleChanged(Sender:TObject);  
begin  
    Invalidate();                             { erase and repaint the component }  
end;
```

46

메시지 처리

이전 Windows 프로그래밍의 핵심 중 하나는 Windows에서 애플리케이션으로 보낸 메시지를 처리하는 것입니다. Delphi에서는 대부분의 일반적인 메시지를 처리합니다. 그러나 Delphi가 처리하지 않는 메시지나 사용자 고유의 메시지는 사용자가 처리해야 할 수도 있습니다. CLX 컴포넌트는 Windows 메시지를 처리하지 않지만 사용자 고유의 메시지에 대한 메시지 핸들러를 만들 수 있습니다.

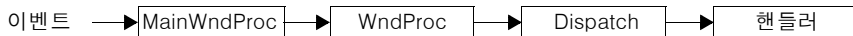
메시지 작업에는 다음과 같은 세 가지 측면이 있습니다.

- 메시지 처리 시스템에 대한 이해
- 메시지 처리 변경
- 새 메시지 핸들러 생성

메시지 처리 시스템에 대한 이해

모든 Delphi 클래스에는 *메시지 처리 메소드* 또는 *메시지 핸들러*라는 기본 제공되는 메시지 처리 메커니즘이 있습니다. 메시지 핸들러의 기본 개념은 클래스가 특정 종류의 메시지를 받고 디스패치하며 받은 메시지에 따라 특정 메시지 집합 중 하나를 호출한다는 것입니다. 특정 메시지에 대한 관련 메소드가 없으면 기본 핸들러가 있습니다.

다음 다이어그램은 메시지 디스패치 시스템을 보여 줍니다.



비주얼 컴포넌트 라이브러리 (VCL)는 특정 클래스로 전달된 사용자 정의 메시지를 포함하여 모든 Windows 메시지를 메소드 호출로 변환하는 메시지 디스패칭 시스템을 정의합니다. CLX의 경우 디스패치 시스템에는 MainWndProc 및 WndProc가 포함되지 않습니다. 이 메시지 디스패치 메커니즘을 변경해서는 안 됩니다. 사용자는 메시지 처리 메소드를 만들지만 하면 됩니다. 자세한 내용은 46-7 페이지의 "새 메시지 처리 메소드 선언"을 참조하십시오.

Windows 메시지 종류

참고 이 정보는 VCL 컴포넌트 작성 시에만 적용됩니다.

Windows 메시지는 여러 필드가 포함된 데이터 레코드입니다. 여기서 가장 중요한 부분은 메시지를 식별하는 integer 크기 값입니다. Windows는 많은 메시지를 정의하고 *Messages* 유닛은 모든 메시지에 대한 식별자를 선언합니다. 그 밖의 메시지에 유용한 정보는 두 가지 매개변수 필드와 하나의 결과 필드에 나타납니다.

한 매개변수에는 16비트가 포함되고 다른 매개변수에는 32비트가 포함됩니다. 종종 "word 매개변수"와 "long 매개변수"에 대해 *wParam*과 *lParam*으로 해당 값을 참조하는 Windows 코드를 봅니다. 종종 각 매개변수는 둘 이상의 정보를 포함하고 있으므로 사용자는 매개변수의 상위 word를 참조하는 *lParamHi*와 같은 이름에 대한 참조를 보게 됩니다.

예전에는 Windows 프로그래머들이 각 매개변수가 포함된 Windows API를 기억하거나 조회해야 했습니다. 그러나 최근에 Microsoft는 매개변수에 이름을 지정했습니다. 이 "메시지 크래킹"이라는 매개변수로 각 메시지에 포함된 정보를 훨씬 더 쉽게 이해할 수 있게 되었습니다. 예를 들어, *WM_KEYDOWN* 메시지에 대한 매개변수는 이제 *nVirtKey* 및 *lKeyData*라고 하며 *wParam* 및 *lParam*보다 더 구체적인 정보를 제공합니다.

각 메시지 유형에 대해 Delphi는 각 매개 변수에 기억하기 쉬운 이름을 제공하도록 레코드 타입을 정의합니다. 예를 들어, 마우스 메시지는 long 매개변수, 상위 word 및 상위 word의 다른 매개변수로 마우스 이벤트의 x 및 y 좌표를 전달합니다. 마우스 메시지 구조를 사용하면 *lParamLo* 및 *lParamHi* 대신 *XPos* 및 *YPos*를 사용하기 때문에 어떤 word가 어디에 해당되는지 걱정하지 않아도 됩니다.

메시지 디스패칭

참고 이 정보는 VCL 컴포넌트 작성 시에만 적용됩니다.

애플리케이션에서 윈도우를 만들 때는 Windows 커널과 함께 *윈도우 프로시저*를 등록합니다. 윈도우 프로시저는 윈도우에 대한 메시지를 처리하는 루틴입니다. 전통적으로 윈도우 프로시저에는 윈도우가 처리해야 하는 각 메시지에 대한 항목이 있는 거대한 **case** 문이 포함됩니다. 이런 의미의 "윈도우"는 각 윈도우, 각 컨트롤 등과 같이 화면에 나타나는 것들을 의미합니다. 새로운 종류의 윈도우를 만들 때마다 전체 윈도우 프로시저를 만들어야 합니다.

Delphi는 다음과 같은 여러 가지 방법으로 메시지 디스패칭을 간소화합니다.

- 각 컴포넌트는 전체 메시지 디스패칭 시스템을 상속합니다.
- 디스패치 시스템에는 기본 처리가 있습니다. 특수하게 응답해야 하는 메시지에 대해서만 핸들러를 정의합니다.
- 메시지 처리에서 작은 부분을 수정하고 대부분의 처리를 상속된 메소드에 의존할 수 있습니다.

이 메시지 디스패치 시스템의 가장 큰 이점은 언제든지 안전하게 컴포넌트로 메시지를 보낼 수 있다는 점입니다. 컴포넌트에 메시지에 대해 정의된 핸들러가 없으면 일반적으로 모든 메시지를 무시하여 기본적으로 처리합니다.

메시지 흐름 추적

Delphi는 애플리케이션에 있는 각 컴포넌트 타입에 대한 윈도우 프로시저로 *MainWndProc* 라는 메소드를 등록합니다. *MainWndProc*에는 메시지 구조를 Windows에서 *WndProc* 라는 가상 메소드로 전달하고 애플리케이션 클래스의 *HandleException* 메소드를 호출하여 모든 예외를 처리하는 예외 처리 블록이 들어 있습니다.

*MainWndProc*는 특정 메시지에 대한 특수 처리가 포함되지 않은 가상이 아닌 메소드입니다. 각 컴포넌트 타입이 특정 요구에 맞게 메소드를 오버라이드할 수 있으므로 *WndProc*를 사용자 지정할 수 있습니다.

WndProc 메소드는 처리에 영향을 주는 특수한 조건을 검사하므로 원하지 않는 메시지를 "트래핑(trapping)"할 수 있습니다. 예를 들어, 끌어오는 동안 컴포넌트는 키보드 이벤트를 무시하므로 *TWinControl*의 *WndProc* 메소드는 컴포넌트를 끌어오지 않는 경우에만 키보드 이벤트를 전달합니다. 결국 *WndProc*는 메시지를 처리하기 위해 호출할 메소드를 결정하는 *TObject*에서 상속된 가상이 아닌 메소드, *Dispatch*를 호출합니다.

*Dispatch*는 메시지 구조의 *Msg* 필드를 사용하여 특정 메시지를 디스패치하는 방법을 결정합니다. 컴포넌트가 특정 메시지에 대한 핸들러를 정의하면 *Dispatch*는 메소드를 호출합니다. 컴포넌트가 해당 메시지에 대한 핸들러를 정의하지 않으면 *Dispatch*는 *DefaultHandler*를 호출합니다.

메시지 처리 변경

참고 이 정보는 VCL 컴포넌트 작성 시에만 적용됩니다.

컴포넌트의 메시지 처리를 변경하려면 실제로 원하는 것인지 확인해야 합니다. Delphi는 대부분의 Windows 메시지를 컴포넌트 작성자와 컴포넌트 사용자 모두가 처리할 수 있는 이벤트로 변환합니다. 메시지 처리 동작을 변경하는 대신 이벤트 처리 동작을 변경해야 합니다.

VCL 컴포넌트에서 메시지 처리를 변경하려면 메시지 처리 메소드를 오버라이드합니다. 메시지를 트래핑하면 특정 환경에서 컴포넌트가 메시지를 처리할 수 없도록 할 수 있습니다.

핸들러 메소드 오버라이드

컴포넌트가 특정 메시지를 처리하는 방법을 변경하려면 해당 메시지에 대한 메시지 처리 메소드를 오버라이드합니다. 컴포넌트가 특정 메시지를 처리하지 않으면 새 메시지 처리 메소드를 선언해야 합니다.

메시지 처리 메소드를 오버라이드하려면 오버라이드하는 메소드와 같은 메시지 인덱스로 컴포넌트에서 새 메소드를 선언합니다. **override** 지시어는 사용하지 *마십시오*. **message** 지시어와 일치하는 메시지 인덱스를 사용해야 합니다.

단일 **var** 매개변수의 타입과 메소드 이름은 오버라이드된 메소드와 일치해서는 안 됩니다. 메시지 인덱스만 의미가 있습니다. 그러나 명확성을 위해서는 처리한 메시지 이름에 따라 메시지 처리 메소드 이름을 지정하는 규칙을 따르는 것이 가장 좋습니다.

예를 들어, *WM_PAINT* 메시지에 대한 컴포넌트의 처리를 오버라이드하려면 *WMPaint* 메소드를 재선언해야 합니다.

```
type
  TMyComponent = class(...)
  :
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

메시지 매개변수 사용

컴포넌트가 메시지 처리 메소드 내에 있으면 메시지 구조의 모든 매개변수에 액세스할 수 있습니다. 메시지 핸들러로 전달된 매개변수는 **var** 매개변수이기 때문에 핸들러는 필요한 경우에 매개변수의 값을 변경할 수 있습니다. 자주 변경되는 매개변수만 메시지의 *결과 필드*, 즉 메시지를 보내는 *SendMessage* 호출로 반환된 값입니다.

참고 이 정보는 VCL 컴포넌트 작성 시에만 적용됩니다.

메시지 처리 메소드의 *Message* 매개변수 종류는 처리되는 메시지에 따라 다양하므로 개별 매개변수의 이름과 의미에 대해서는 Windows 메시지에 대한 설명서를 참조해야 합니다. 어떤 이유에서 이전 스타일 이름 (*WParam*, *LParam* 등)으로 메시지 매개변수를 참조해야 하는 경우에는 해당 매개변수 이름을 사용하는 일반 타입 *TMessage*로 *Message*를 타입 변환할 수 있습니다.

메시지 트래핑(Trapping)

특정 환경에서는 컴포넌트가 메시지를 무시하게 할 수 있습니다. 즉, 컴포넌트가 핸들러로 메시지를 디스패치할 수 없게 합니다. 메시지를 트래핑하려면 가상 메소드 *WndProc*를 오버라이드합니다.

VCL 컴포넌트의 경우 *WndProc* 메소드는 *Dispatch* 메소드로 메시지를 전달하기 전에 메시지를 검사하여 메시지를 처리할 메소드를 차례로 결정합니다. *WndProc*를 오버라이드하면 컴포넌트는 메시지를 디스패치하기 전에 필터링할 수 있습니다. *TWinControl*에서 파생된 컨트롤에 대한 *WndProc*의 오버라이드는 다음과 같이 나타납니다.

```
procedure TMyControl.WndProc(var Message:TMessage);
begin
  { tests to determine whether to continue processing }
  inherited WndProc(Message);
end;
```


TControl 컴포넌트는 사용자가 컨트롤을 끌어서 놓을 때 필터링하는 마우스 메시지의 전체 범위를 정의합니다. *WndProc*를 오버라이드하면 다음 두 가지 방법으로 이 작업에 도움을 줍니다.

- 각 메시지에 대한 핸들러를 지정하지 않고도 메시지의 범위를 필터링할 수 있습니다.
- 메시지 디스패칭을 방해하여 핸들러가 호출되지 않도록 할 수 있습니다.

CLX의 경우 컨트롤은 *TWidgetControl*의 자손일 수 있으며 *WndProc* 대신 *EventFilter*를 오버라이드합니다.

다음은 *TControl*에 대한 *WndProc* 메소드의 일부입니다. 예를 들면, 다음과 같습니다.

```

procedure TControl.WndProc(var Message:TMessage);
begin
  :
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
      : { handle others normally }
    end;
  : { otherwise process normally }
end;

```

새 메시지 핸들러 생성

Delphi는 대부분의 일반적인 메시지에 핸들러를 제공하기 때문에 사용자가 새 메시지 핸들러를 만들고자 하는 시기는 사용자 고유의 메시지를 정의하는 경우입니다. 사용자 정의 메시지 작업에는 두 가지 측면이 있습니다.

- 사용자 고유의 메시지 정의
- 새 메시지 처리 메소드 선언

CLX 컴포넌트는 Windows 메시지를 처리하지 않지만 사용자 고유의 메시지에 대한 메시지 핸들러를 만들 수 있습니다. Qt 이벤트는 메시지 ID가 아니라 객체이기 때문에 이 이벤트에 대해서는 메시지 핸들러를 생성할 수 없습니다.

사용자 고유의 메시지 정의

많은 표준 컴포넌트는 내부적으로 사용하기 위해 메시지를 정의합니다. 메시지를 정의하는 가장 일반적인 이유는 표준 메시지에서 처리하지 못하는 정보와 상태 변경에 대한 알림을 브로드캐스트하기 위해서입니다. VCL과 CLX 모두에서 사용자 고유의 메시지를 정의할 수 있습니다.

메시지 정의 과정은 두 단계입니다. 그 단계는 다음과 같습니다.

- 1 메시지 식별자 선언
- 2 메시지 레코드 타입 선언

메시지 식별자 선언

메시지 식별자는 integer 크기의 상수입니다. Windows는 자체적인 사용을 위해 메시지를 1,024개 이하로 유지하므로 사용자 고유의 메시지를 정의할 때는 이 수준 위에서 시작해야 합니다.

상수 *WM_APP*는 사용자 정의 메시지의 시작 번호를 나타냅니다. 메시지 식별자를 정의할 때는 *WM_APP*의 숫자를 기반으로 해야 합니다.

일부 표준 Windows 컨트롤은 사용자 정의 범위 내의 메시지를 사용해야 합니다. 이 컨트롤에는 리스트 박스, 콤보 박스, 편집 상자 및 명령 버튼이 포함됩니다. 이 컨트롤 중 하나에서 컴포넌트를 파생시키고 그 컴포넌트에 대한 새 메시지를 정의하려면 Messages 유닛을 검사하여 Windows가 해당 컨트롤에 이미 정의한 메시지를 확인해야 합니다.

다음 코드는 두 개의 사용자 정의 메시지를 보여 줍니다.

```
const
    WM_MYFIRSTMESSAGE = WM_APP + 400;
    WM_MYSECONDMESSAGE = WM_APP + 401;
```

메시지 레코드 타입 선언

메시지의 매개변수에 유용한 이름을 제공하려면 해당 메시지에 대해 메시지 레코드 타입을 선언해야 합니다. 메시지 레코드는 메시지 처리 메소드로 전달된 매개변수 타입입니다. 메시지의 매개변수를 사용하지 않거나 이전 스타일 매개변수 표기법 (*wParam*, *lParam* 등)을 사용하려는 경우에는 기본 메시지 레코드, *TMessage*를 사용할 수 있습니다.

다음 규칙에 따라 메시지 레코드 타입을 선언합니다.

- 1 메시지 이름에 따라 *T*로 시작하는 레코드 타입 이름을 지정합니다.
- 2 타입 *TMsgParam*의 레코드 *Msg*에 있는 첫 번째 필드를 호출합니다.
- 3 *Word* 매개변수에 해당하는 다음 2바이트를 정의하고 그 다음 2바이트는 사용하지 않은 것으로 정의합니다.

또는

Longint 매개변수에 해당하도록 다음 4바이트를 정의합니다.

- 4 *Longint* 타입의 *결과*라는 최종 필드를 추가합니다.

예를 들어, 가변 레코드를 사용하여 같은 매개변수에 대한 이름 집합을 정의하는 모든 마우스 메시지인 *TWMMouse*에 대한 메시지 레코드는 다음과 같습니다.

```
type
    TWMMouse = record
        Msg: TMsgParam;           ( first is the message ID )
        Keys: Word;              ( this is the wParam )
        case Integer of         ( two ways to look at the lParam )
            0: {
                XPos: Integer;   ( either as x- and y-coordinates... )
                YPos: Integer;
            }
            1: {
                Pos: TPoint;     ( ... or as a single point )
```

```

    Result := ''; Longint); ( and finally, the result field )
end;

```

새 메시지 처리 메소드 선언

새 메시지 처리 메소드를 선언해야 하는 환경은 두 가지입니다.

- 컴포넌트는 표준 컴포넌트에서 처리되지 않은 Windows 메시지를 처리해야 합니다.
- 컴포넌트에서 사용하도록 사용자 고유의 메시지를 정의합니다.

다음과 같이 메시지 처리 메소드를 선언합니다.

- 1 컴포넌트 클래스 선언의 **protected** 부분에서 메소드를 선언합니다.
- 2 메소드를 프로시저로 만듭니다.
- 3 처리한 메시지 이름에 따라 메소드 이름을 지정합니다. 밑줄 문자는 사용하지 않습니다.
- 4 메시지 레코드 타입 중에서 *Message*라는 단일 **var** 매개변수를 전달합니다.
- 5 메시지 메소드 구현 내에서 컴포넌트와 관련된 모든 처리에 대해 코드를 작성합니다.
- 6 상속된 메시지 핸들러를 호출합니다.

예를 들어, *CM_CHANGECOLOR*라는 사용자 정의 메시지에 대한 메시지 핸들러의 선언이 있습니다.

```

const
    CM_CHANGECOLOR = WM_APP + 400;

type
    TMyComponent = class(TControl)
    :
    protected
        procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
    end;

procedure TMyComponent.CMChangeColor(var Message:TMessage);
begin
    Color := Message.lParam;
    inherited;
end;

```


디자인 타임 시 컴포넌트 사용

이 장에서는 작성한 컴포넌트를 IDE에서 사용 가능하도록 만들기 위한 단계를 설명합니다. 디자인 타임에 컴포넌트를 사용 가능하게 하려면 여러 단계가 필요합니다.

- 컴포넌트 등록
- 팔레트 비트맵 추가
- 사용자의 컴포넌트에 도움말 제공
- 속성 편집기 추가
- 컴포넌트 에디터 추가
- 컴포넌트를 패키지로 컴파일

모든 컴포넌트에 이러한 모든 단계가 적용되는 것은 아닙니다. 예를 들어, 새로운 속성이나 이벤트를 정의하지 않는 경우 해당 속성이나 이벤트에 대한 도움말을 제공할 필요가 없습니다. 반드시 필요한 단계는 등록과 컴파일입니다.

일단 컴포넌트가 패키지에 등록되고 컴파일되면 다른 개발자에게 배포될 수 있고 IDE에 설치될 수 있습니다. IDE에서의 패키지 설치에 대한 내용은 11-5 페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

컴포넌트 등록

등록 작업은 유닛 기반 컴파일을 통해 이루어지므로 하나의 컴파일 유닛에 여러 컴포넌트를 만들면 한꺼번에 모두 등록할 수 있습니다.

컴포넌트를 등록하려면 *Register* 프로시저를 유닛에 추가합니다. *Register* 프로시저 내에서 컴포넌트를 등록하고 컴포넌트 팔레트에 설치할 위치를 결정합니다.

참고 IDE에서 Component|New Component를 선택하여 컴포넌트를 생성하는 경우, 컴포넌트를 등록하는 데 필요한 코드가 자동으로 추가됩니다.

컴포넌트를 수동으로 등록하는 단계는 다음과 같습니다.

- Register 프로시저 선언
- Register 프로시저 작성

Register 프로시저 선언

등록에는 유닛에서 *Register*라는 이름의 단일 프로시저 작성 단계가 포함됩니다. *Register* 프로시저는 유닛의 인터페이스 부분에 나타나야 하며 오브젝트 파스칼의 나머지 부분과 달리 이름의 대소문자를 구별합니다.

다음 코드는 새로운 컴포넌트를 생성하고 등록하는 단순한 유닛에 대한 개요를 보여 줍니다.

```
unit MyBtns;
interface
type
    ...                               { declare your component types here }
procedure Register;                   { this must appear in the interface section }
implementation
    ...                               { component implementation goes here }

procedure Register;
begin
    ...                               { register the components }
end;
end.
```

Register 프로시저 내에서 컴포넌트 팔레트에 추가하고자 하는 각각의 컴포넌트에 대해 *RegisterComponents*를 호출합니다. 유닛이 여러 컴포넌트를 포함하고 있는 경우 한꺼번에 모두 등록할 수 있습니다.

Register 프로시저 작성

컴포넌트를 포함하는 유닛의 *Register* 프로시저 내에서 컴포넌트 팔레트에 추가하고자 하는 각각의 컴포넌트를 등록해야 합니다. 유닛이 여러 컴포넌트를 포함하고 있는 경우, 동시에 모두 등록할 수 있습니다.

컴포넌트를 등록하려면 컴포넌트를 추가하고자 하는 컴포넌트 팔레트의 각 페이지에 대해 한 번씩 *RegisterComponents* 프로시저를 호출합니다. *RegisterComponents*에 관련된 세 가지 중요한 내용은 다음과 같습니다.

- 1 컴포넌트 지정
- 2 팔레트 페이지 지정
- 3 RegisterComponents 함수 사용

컴포넌트 지정

Register 프로시저 내에서 RegisterComponents에 대한 호출에서 구성할 수 있는 개방형 배열로 컴포넌트 이름을 전달합니다.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

다음 코드에서 보듯이 여러 컴포넌트를 같은 페이지에 한꺼번에 등록하거나 다른 페이지에 등록할 수 있습니다.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);      { two on this page... }
  RegisterComponents('Assorted', [TThird]);                    { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]); { ...and one on the Standard page }
end;
```

팔레트 페이지 지정

팔레트 페이지 이름은 문자열입니다. 팔레트 페이지에 지정한 이름이 이미 존재하는 이름이 아닌 경우, Delphi는 그 이름을 갖는 새 페이지를 만듭니다. Delphi는 제품의 국제 버전이 해당 모국어로 페이지를 이름 지정할 수 있도록 문자열 목록 리소스에 표준 페이지의 이름을 저장합니다. 표준 페이지 중 하나에 컴포넌트를 설치할 경우, *LoadStr* 함수를 호출하여 페이지 이름의 문자열을 얻어야 하는데 이 함수는 System 페이지의 *srSystem*과 같은 해당 페이지의 문자열 리소스를 나타내는 상수를 전달합니다.

RegisterComponents 함수 사용

Register 프로시저 내에서 *RegisterComponents*를 호출하여 클래스 배열에 컴포넌트를 등록합니다. *RegisterComponents*는 두 매개변수, 즉 컴포넌트 팔레트 페이지의 이름과 컴포넌트 클래스의 배열을 취하는 함수입니다.

Page 매개변수를 컴포넌트가 나타나야 할 컴포넌트 팔레트의 페이지 이름으로 설정합니다. 명명된 페이지가 이미 있는 경우 컴포넌트는 그 페이지에 추가됩니다. 명명된 페이지가 없는 경우 Delphi는 그 이름을 갖는 새 팔레트 페이지를 만듭니다.

사용자 지정 컴포넌트를 정의하는 유닛 중 하나의 Register 프로시저 구현에서 RegisterComponents를 호출합니다. 그런 다음 컴포넌트를 정의하는 유닛이 패키지로 컴파일되어 사용자 지정 컴포넌트가 컴포넌트 팔레트에 추가되기 전에 패키지를 설치해야 합니다.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);          {add to system page}
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);    { new page}
end;
```

팔레트 비트맵 추가

모든 컴포넌트에는 컴포넌트 팔레트에 컴포넌트를 나타내는 비트맵이 필요합니다. 비트맵을 따로 지정하지 않으면 Delphi는 기본 비트맵을 사용합니다.

팔레트 비트맵은 디자인 타임에만 필요하므로 컴포넌트의 컴파일 유닛 안으로 비트맵을 컴파일하지 않습니다. 그 대신 유닛과 동일한 이름을 갖지만 확장자가 .DCR(dynamic component resource)인 Windows 리소스 파일에 팔레트 비트맵을 제공합니다. Delphi

에서 이미지 편집기를 사용하여 이 리소스 파일을 만들 수 있습니다. 각 비트맵은 가로 세로 24픽셀이어야 합니다.

설치할 컴포넌트마다 팔레트 비트맵 파일을 제공하고 각 팔레트 비트맵 파일 내에 사용자가 등록하는 각 컴포넌트를 위한 비트맵을 제공합니다. 비트맵 이미지는 컴포넌트와 동일한 이름을 가집니다. 컴파일된 파일이 있는 동일한 디렉토리에 팔레트 비트맵 파일을 보관하므로 Delphi는 컴포넌트 팔레트에 컴포넌트를 설치할 때 비트맵을 찾을 수 있습니다.

예를 들어, ToolBox 라는 이름의 유닛에서 *TMyControl*이라는 이름의 컴포넌트를 생성할 경우 비트맵 TMYCONTROL을 포함하는 리소스 파일 TOOLBOX.DCR을 만들어야 합니다. 리소스 이름은 대소문자를 구별하지 않지만 일반적으로 규칙에 따라 대문자로 되어 있습니다.

사용자의 컴포넌트에 도움말 제공

폼에서 표준 컴포넌트를 선택하거나 Object Inspector에서 속성 또는 이벤트를 선택하면 *F1*을 눌러 그 항목에 관한 도움말을 얻을 수 있습니다. 해당 도움말 파일을 작성할 경우 개발자에게 같은 종류의 컴포넌트 설명서를 제공할 수 있습니다.

컴포넌트를 설명하는 작은 도움말 파일을 제공할 수 있고 도움말 파일은 사용자의 전체적인 Delphi 도움말 시스템의 일부가 됩니다.

컴포넌트와 함께 사용하기 위해 도움말 파일을 구성하는 방법에 대한 정보는 47-4 페이지의 "도움말 파일 생성" 단원을 참조하십시오.

도움말 파일 생성

원하는 툴을 사용하여 (.rtf 형식의) Windows 도움말 파일의 소스 파일을 만들 수 있습니다. Delphi는 사용자의 도움말 파일을 컴파일하고 온라인 도움말 저작 안내서를 제공하는 Microsoft Help Workshop을 포함하고 있습니다. Help Workshop의 온라인 설명서에서는 도움말 파일 작성에 관해 자세히 설명하고 있습니다.

컴포넌트의 도움말 파일 작성은 다음의 단계들로 구성됩니다.

- 항목 생성
- 문맥에 따른 컴포넌트 도움말 만들기 컴포넌트 도움말 파일 추가

항목 생성

컴포넌트의 도움말이 라이브러리의 나머지 컴포넌트들의 도움말과 매끄럽게 통합되기 위해서는 다음과 같은 규칙을 준수해야 합니다.

1 각 컴포넌트는 도움말 항목을 가지고 있어야 합니다.

컴포넌트 항목은 컴포넌트가 선언된 유닛을 보여 주고 컴포넌트를 간략하게 설명해야 합니다. 컴포넌트 항목은 객체 계층 구조에서 컴포넌트의 위치를 설명하고 모든 속성, 이벤트, 메소드를 나열하는 보조 창에 연결되어야 합니다. 애플리케이션 개발

자는 폼에서 컴포넌트를 선택하고 *F1*을 눌러 이 항목에 액세스합니다. 컴포넌트 항목의 예를 보려면 폼에 임의의 컴포넌트를 놓고 *F1*을 눌러 보십시오.

컴포넌트 항목은 항목에 고유한 값을 갖는 # 각주를 가져야 합니다. # 각주는 도움말 시스템에 의해 각 항목을 고유하게 식별합니다.

컴포넌트 항목은 컴포넌트 클래스의 이름을 포함하는 도움말 시스템 색인에서의 키워드 검색을 위해 K 각주를 가져야 합니다. 예를 들어, *TMemo* 컴포넌트에 대한 키워드 각주는 "TMemo"입니다.

컴포넌트 항목은 항목의 제목을 제공하는 \$ 각주도 가져야 합니다. 제목은 Topics Found 대화 상자, Bookmark 대화 상자, History 창에 나타납니다.

2 각 컴포넌트는 다음과 같은 보조 탐색 항목을 가져야 합니다.

- 컴포넌트 계층 구조에서 컴포넌트의 모든 조상과 링크되어 있는 계층 항목.
- 해당 속성을 설명하는 항목에 대한 링크를 가지고 있으며 컴포넌트에서 사용할 수 있는 모든 속성의 목록.
- 해당 이벤트를 설명하는 항목에 대한 링크를 가지고 있으며 컴포넌트에서 사용할 수 있는 모든 이벤트의 목록.
- 해당 메소드를 설명하는 항목에 대한 링크를 가지고 있으며 컴포넌트에서 사용할 수 있는 메소드의 목록.

Delphi 도움말 시스템에서 객체 클래스, 속성, 메소드, 이벤트에 대한 링크는 Alinks 를 사용하여 만들 수 있습니다. 객체 클래스에 링크할 경우, Alink는 밑줄(_)과 문자열 "object"가 뒤에 붙는 객체의 클래스 이름을 사용합니다. 예를 들어, *TCustomPanel* 객체에 링크하려면 다음을 사용합니다.

```
!AL(TCustomPanel_object,1)
```

속성이나 메소드, 이벤트에 링크할 때는 속성이나 메소드, 이벤트의 이름 앞에 그것을 구현하는 객체의 이름과 밑줄을 붙입니다. 예를 들어, *TControl*이 구현하는 *Text* 속성에 연결하려면 다음을 사용합니다.

```
!AL(TControl_Text,1)
```

보조 탐색 항목의 예를 보려면 컴포넌트의 도움말을 표시하고 계층이나 속성, 메소드, 이벤트 등의 레이블이 붙은 링크를 클릭합니다.

3 컴포넌트에 선언되는 각 속성, 메소드, 이벤트는 항목을 가져야 합니다.

속성이나 이벤트, 메소드 항목은 항목의 선언을 보여 주고 그 용도를 설명해야 합니다. 애플리케이션 개발자는 이 항목들을 보기 위해 Object Inspector에서 항목을 선택하고 *F1*을 누르거나 코드 에디터에서 항목의 이름에 커서를 놓고 *F1*을 누릅니다. 속성 항목의 예를 보려면 Object Inspector에서 임의의 항목을 선택하고 *F1*을 누릅니다.

속성, 이벤트, 메소드 항목은 속성이나 메소드, 이벤트의 이름과 컴포넌트의 이름을 조합한 이름을 나열하는 K 각주를 포함해야 합니다. 따라서 *TControl*의 *Text* 속성은 다음과 같은 K 각주를 갖습니다.

```
Text,TControl;TControl,Text;Text,
```

속성, 메소드, 이벤트 항목은 TControl.Text와 같이 항목의 제목을 나타내는 \$ 각주도 포함해야 합니다.

이 모든 항목은 # 각주로 입력된 항목에 고유한 항목 ID를 가져야 합니다.

문맥에 따른 컴포넌트 도움말 만들기

각 컴포넌트, 속성, 메소드, 이벤트 항목은 A 각주를 가져야 합니다. A 각주는 사용자가 컴포넌트를 선택하고 *F1*을 누를 때 또는 Object Inspector에서 속성이나 이벤트가 선택되어 있고 사용자가 *F1*을 누를 때 항목을 표시하기 위해 사용됩니다. A 각주는 다음과 같은 이름 지정 규칙을 따라야 합니다.

컴포넌트에 대한 도움말 항목인 경우 A 각주는 다음 구문을 사용하여 세미콜론으로 구분된 두 개의 항목으로 구성됩니다.

```
ComponentClass_Object;ComponentClass
```

여기서 *ComponentClass*는 컴포넌트 클래스의 이름입니다.

속성 또는 이벤트에 대한 도움말 항목인 경우 A 각주는 다음 구문을 사용하여 세미콜론으로 구분된 세 개의 항목으로 구성됩니다.

```
ComponentClass_Element;Element_Type;Element
```

여기서 *ComponentClass*는 컴포넌트 클래스의 이름이고 *Element*는 속성이나 메소드, 이벤트의 이름이고 *Type*은 Property, Method, Event 중 하나입니다.

예를 들어, *TMyGrid*라는 이름을 갖는 컴포넌트의 *BackgroundColor* 속성의 경우 A 각주는 다음과 같습니다.

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

컴포넌트 도움말 파일 추가

Delphi에 사용자의 도움말 파일을 추가하려면 bin 디렉토리에 있거나 IDE에서 Help! Customize를 사용하여 액세스되는 OpenHelp 유틸리티(oh.exe)를 사용합니다.

도움말 시스템에 사용자의 도움말 파일을 추가하는 것을 포함하여 OpenHelp.hlp 파일에서 OpenHelp 사용에 관한 정보를 찾을 수 있습니다.

속성 편집기 추가

Object Inspector는 모든 타입의 속성에 기본 편집 기능을 제공합니다. 그러나 속성 편집기를 작성하고 등록하여 특정 속성의 대체 편집기를 제공할 수 있습니다. 작성하는 컴포넌트의 속성에만 적용되는 속성 편집기를 등록할 수 있고, 특정 타입의 모든 속성에 적용되는 편집기를 만들 수도 있습니다.

가장 간단한 수준에서는 속성 편집기가 다음 두 가지 방법 중 하나 또는 모두에 의해 작동됩니다. 사용자가 현재 값을 텍스트 문자열로 편집하게 하여 표시하거나 다른 종류의 편집을 할 수 있는 대화 상자를 표시합니다. 편집할 속성에 따라 둘 중 한 가지를 사용하든지 두 가지 모두 사용하는 것이 좋습니다.

속성 편집기를 작성하려면 다음의 다섯 단계를 거쳐야 합니다.

- 1 속성 편집기 클래스 파생
- 2 텍스트로 속성 편집
- 3 전체 속성 편집
- 4 편집기 속성 지정
- 5 속성 편집기 등록

속성 편집기 클래스 파생

CLX와 VCL 모두 여러 유형의 속성 편집기를 정의하는데 모두 *TPropertyEditor*에서 파생됩니다. 속성 편집기를 만들 때 속성 편집기 클래스는 *TPropertyEditor*의 직계 자손이 되거나 표 47.1에 설명되어 있는 속성 편집기 클래스 중 하나의 간접 자손이 될 수 있습니다. DesignEditors 유닛의 클래스는 VCL과 CLX 애플리케이션에서 모두 사용될 수 있습니다. 하지만 일부 속성 편집기 클래스는 특화된 대화 상자를 제공하고 VCL 또는 CLX로 특화됩니다. 이는 각각 WinEditors와 CLXEditors 유닛에서 찾을 수 있습니다.

참고 속성 편집기는 반드시 *TBasePropertyEditor*의 자손이어야 하고 *IProperty* 인터페이스를 지원해야 합니다. 그러나 *TPropertyEditor*는 *IProperty* 인터페이스의 기본 구현을 제공합니다.

표 47.1의 목록은 완전하지는 않습니다. WinEditors와 CLXEditors 유닛은 컴포넌트 이름과 같은 고유 속성에 사용되는 매우 특화된 일부 속성 편집기도 정의합니다. 아래 목록에 나열된 속성 편집기는 사용자 정의 속성에 가장 유용합니다.

표 47.1 이미 정의된 속성 편집기 타입

타입	편집될 속성
TOrdinalProperty	정수, 문자 및 열거 타입 속성을 위한 모든 순서 타입 속성 편집기는 <i>TOrdinalProperty</i> 의 자손입니다.
TIntegerProperty	이미 정의되고 사용자 지정된 부분 범위를 포함한 모든 정수 타입.
TCharProperty	<i>Char</i> 타입 및 'A'...'Z'와 같은 <i>Char</i> 의 부분 범위.
TEnumProperty	열거 타입.
TFloatProperty	모든 부동 소수점 숫자.
TStringProperty	문자열.
TSetElementProperty	부울 값으로 표시되는 개별 요소 집합.
TSetProperty	모든 집합. 집합은 직접 편집할 수는 없지만 집합 요소 속성의 목록으로 확장할 수 있습니다.
TClassProperty	클래스. 클래스 이름을 표시하고 클래스 속성을 확장할 수 있습니다.
TMethodProperty	메소드 포인터들로 가장 확실한 이벤트입니다.
TComponentProperty	동일한 품의 컴포넌트. 사용자는 컴포넌트 속성을 편집할 수 없지만 호환 가능한 타입의 특정 컴포넌트를 가리킬 수 있습니다.
TColorProperty	컴포넌트 색상. 적용되는 경우 색상 상수를 표시하고, 그렇지 않은 경우 16진수 값을 표시합니다. 드롭다운 목록에 색상 상수가 포함되어 있습니다. 더블 클릭하면 색상 선택 대화 상자가 나타납니다.

표 47.1 이미 정의된 속성 편집기 타입 (계속)

타입	편집될 속성
TFontNameProperty	글꼴 이름. 드롭다운 목록에 현재 설치되어 있는 글꼴이 모두 표시됩니다.
TFontProperty	글꼴. 글꼴 대화 상자에 액세스할 수 있을 뿐만 아니라 개별 글꼴 속성을 확장할 수 있습니다.

다음 예제는 *TMyPropertyEditor*라는 간단한 속성 편집기의 선언을 보여 줍니다.

```

type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual:Boolean; override;
    function GetValue:string; override;
    procedure SetValue(const Value:string); override;
  end;

```

텍스트로 속성 편집

모든 속성은 Object Inspector에 표시할 수 있도록 값에 대한 문자열 표현을 제공해야 합니다. 대부분의 속성은 사용자가 새로운 속성 값을 입력할 수 있도록 합니다. 속성 편집기 클래스는 텍스트 표현과 실제 값 사이에서 변환하기 위해 오버라이드할 수 있는 가상 메소드를 제공합니다.

오버라이드해야 할 메소드는 *GetValue* 및 *SetValue*입니다. 또한 사용자의 속성 편집기는 표 47.2에 표시된 대로 다른 종류의 값을 할당하고 읽을 때 사용되는 일련의 메소드를 상속합니다.

표 47.2 속성 값을 읽고 쓰기 위한 메소드

속성 타입	Get 메소드	Set 메소드
부동 소수점	GetFloatValue	SetFloatValue
메소드 포인터(이벤트)	GetMethodValue	SetMethodValue
순서 타입	GetOrdValue	SetOrdValue
문자열	GetStrValue	SetStrValue

GetValue 메소드를 오버라이드하는 경우 Get 메소드 중 하나를 호출하고 *SetValue*를 오버라이드하는 경우 Set 메소드 중 하나를 호출합니다.

속성 값 표시

속성 편집기의 *GetValue* 메소드는 속성의 현재 값을 나타내는 문자열을 반환합니다. Object Inspector는 속성의 값 열에 이 문자열을 사용합니다. 기본적으로 *GetValue*는 "unknown"을 반환합니다.

속성에 대한 문자열 표현을 제공하려면 속성 편집기의 *GetValue* 메소드를 오버라이드합니다.

속성이 문자열 값이 아닌 경우, *GetValue*는 그 값을 문자열 표현으로 변환해야 합니다.

속성 값 설정

속성 편집기의 *SetValue* 메소드는 Object Inspector에서 사용자가 입력한 문자열을 취하여 적절한 타입으로 변환한 다음 속성 값을 설정합니다. 문자열이 속성에 적합한 값을 나타내지 않는 경우, *SetValue*는 예외를 버리고 부적절한 값을 사용하지 않아야 합니다.

문자열 값을 속성으로 읽어 들이려면 속성 편집기의 *SetValue* 메소드를 오버라이드합니다.

*SetValue*는 Set 메소드 중 하나를 호출하기 전에 값을 변환하고 검사해야 합니다.

다음은 *TIntegerProperty*의 *GetValue* 및 *SetValue* 메소드입니다. 정수는 순서 타입이므로 *GetValue*는 *GetOrdValue*를 호출하고 그 결과를 문자열로 변환합니다. *SetValue*는 문자열을 정수로 변환하고 일부 범위 점검을 수행한 다음 *SetOrdValue*를 호출합니다.

```
function TIntegerProperty.GetValue:string;
begin
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then // unsigned
      Result := IntToStr(Cardinal(GetOrdValue))
    else
      Result := IntToStr(GetOrdValue);
  end;
end;

procedure TIntegerProperty.SetValue(const Value:string);
  procedure Error(const Args:array of const);
  begin
    raise EPropertyError.CreateResFmt(@SOutOfRange, Args);
  end;
var
  L:Int64;
begin
  L := StrToInt64(Value);
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then
      begin // unsigned compare and reporting needed
        if (L < Cardinal(MinValue)) or (L > Cardinal(MaxValue)) then
          // bump up to Int64 to get past the %d in the format string
          Error([Int64(Cardinal(MinValue)), Int64(Cardinal(MaxValue))]);
        end
      else if (L < MinValue) or (L > MaxValue) then
        Error([MinValue, MaxValue]);
      SetOrdValue(L);
    end;
end;
```

여기서 설명한 예제보다는 다음 원칙이 더 중요합니다. *GetValue*는 값을 문자열로 변환하고, *SetValue*는 문자열을 변환하고 값을 검사한 다음 "Set" 메소드 중 하나를 호출합니다.

전체 속성 편집

사용자가 비주얼하게 속성을 편집할 수 있는 대화 상자를 옵션으로 제공할 수 있습니다. 속성 편집기는 자신이 클래스인 속성에 가장 많이 사용됩니다. 그 예로 *Font* 속성에서 사용자는 글꼴 대화 상자를 열어 글꼴의 속성을 한꺼번에 모두 선택할 수 있습니다.

전체 속성 편집기 대화 상자를 제공하려면 속성 편집기 클래스의 *Edit* 메소드를 오버라이드합니다.

Edit 메소드는 *GetValue*와 *SetValue* 메소드 작성 시 동일한 *Get* 메소드와 *Set* 메소드를 사용합니다. 실제로 *Edit* 메소드는 *Get* 메소드와 *Set* 메소드를 모두 호출합니다. 편집기는 타입별로 정의되므로 속성 값을 문자열로 변환할 필요가 없습니다. 편집기는 일반적으로 값을 "가져온 상태"대로 처리합니다.

사용자가 속성 옆에 있는 '...' 버튼을 클릭하거나 값 옆을 더블 클릭하면 Object Inspector 는 속성 편집기의 *Edit* 메소드를 호출합니다.

Edit 메소드의 구현 내에서 다음 단계를 따릅니다.

- 1 속성에 사용할 편집기를 생성합니다.
- 2 현재 값을 읽고 *Get* 메소드를 사용하여 그 값을 속성에 지정합니다.
- 3 사용자가 새로운 값을 선택하는 경우, *Set* 메소드를 사용하여 그 값을 속성에 지정합니다.
- 4 편집기를 소멸시킵니다.

대부분의 컴포넌트에 있는 *Color* 속성은 표준 Windows 색상 대화 상자를 속성 편집기로 사용합니다. 다음은 대화 상자를 불러내서 결과를 사용하는 *TColorProperty*의 *Edit* 메소드입니다.

```

procedure TColorProperty.Edit;
var
    ColorDialog:TColorDialog;
begin
    ColorDialog := TColorDialog.Create(Application);           { construct the editor }
    try
        ColorDialog.Color := GetOrdValue;                     { use the existing value }
        if ColorDialog.Execute then                          { if the user OKs the dialog... }
            SetOrdValue(ColorDialog.Color);                    { ...use the result to set value }
        finally
            ColorDialog.Free;                                  { destroy the editor }
    end;
end;

```

편집기 속성 지정

속성 편집기는 Object Inspector가 표시할 도구를 결정하는 데 사용할 수 있는 정보를 제공해야 합니다. 예를 들어, Object Inspector는 속성이 하위 속성을 가지고 있는지 또는 가능한 값의 목록을 표시할 수 있는지 여부를 알아야 합니다.

편집기 속성을 지정하려면 속성 편집기의 *GetAttributes* 메소드를 오버라이드합니다.

*GetAttributes*는 다음 값 중 일부 또는 전부를 포함할 수 있는 *TPropertyAttributes* 타입의 값 집합을 반환하는 메소드입니다.

표 47.3 속성 편집기 특성 플래그

플래그	관련 메소드	포함된 경우의 의미
paValueList	GetValues	편집기는 열거 타입 값의 목록을 제공할 수 있습니다.
paSubProperties	GetProperties	속성은 표시할 수 있는 하위 속성을 가집니다.
paDialog	Edit	편집기는 전체 속성을 편집하기 위한 대화 상자를 표시할 수 있습니다.
paMultiSelect	해당 없음	속성은 사용자가 하나 이상의 컴포넌트를 선택할 때 표시되어야 합니다.
paAutoUpdate	SetValue	값이 승인되기를 기다리는 대신 변경이 일어날 때마다 컴포넌트를 업데이트합니다.
paSortList	해당 없음	Object Inspector는 값 목록을 정렬해야 합니다.
paReadOnly	해당 없음	사용자는 속성 값을 수정할 수 없습니다.
paRevertable	해당 없음	Object Inspector의 컨텍스트 메뉴에 있는 Inherited 메뉴 항목에 Revert를 사용 가능하게 합니다. 메뉴 항목은 현재 속성 값을 버리고 이전에 설정된 기본값이나 표준 값으로 반환하도록 속성 편집기에 알립니다.
paFullWidthName	해당 없음	값을 표시할 필요가 없습니다. 그 대신 Object Inspector는 속성 이름에 전체 폭을 사용합니다.
paVolatileSubProperties	GetProperties	Object Inspector는 속성 값이 변경될 때 모든 하위 속성의 값을 다시 페치합니다.
paReference	GetComponentValue	값은 기타 다른 것에 대한 참조입니다. paSubProperties와 연결되어 사용되는 경우, 참조된 객체는 이 속성에 대한 하위 속성으로 표시되어야 합니다.

Color 속성은 사용자가 Object Inspector에서 몇 가지 방법, 즉 속성 값을 입력하고 목록과 사용자 지정된 편집기로부터 선택하는 것을 허용한다는 의미에서 대부분의 다른 속성들보다 더 많은 유연성이 있습니다. 따라서 *TColorProperty*의 *GetAttributes* 메소드는 그 반환 값에 다음과 같은 몇 가지 속성을 포함합니다.

```
function TColorProperty.GetAttributes:TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList, paRevertable];
end;
```

속성 편집기 등록

일단 속성 편집기를 생성하면 Delphi에 등록해야 합니다. 속성 편집기를 등록하려면 속성 타입을 특정 속성 편집기에 연결해야 합니다. 지정된 타입의 모든 속성을 갖는 편집기를 등록하거나 특정 컴포넌트 타입의 특정 속성만 갖는 편집기를 등록할 수 있습니다.

속성 편집기를 등록하려면 *RegisterPropertyEditor* 프로시저를 호출합니다.

*RegisterPropertyEditor*는 다음의 네 가지 매개변수를 취합니다.

- 편집할 속성의 타입에 대한 타입 정보 포인터.
기본 제공 함수인 *TypeInfo*는 항상 *TypeInfo(TMyComponent)*의 형태로 호출합니다.
- 이 편집기가 적용되는 컴포넌트 타입. 이 매개변수가 **nil**인 경우, 편집기는 지정된 타입의 모든 속성에 적용됩니다.
- 속성의 이름. 이 매개변수는 이전의 매개변수가 특정 컴포넌트 타입을 지정하는 경우에만 의미를 가집니다. 이 경우 이 편집기가 적용되는 컴포넌트 타입의 특정 속성의 이름을 지정할 수 있습니다.
- 지정된 속성을 편집하는 데 사용할 속성 편집기의 타입.

다음은 컴포넌트 팔레트에 표준 컴포넌트의 편집기를 등록하는 프로시저에서 발췌한 것입니다.

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

이 프로시저의 세 문장은 *RegisterPropertyEditor*가 다르게 사용된 내용을 다룹니다.

- 첫 번째 문장은 가장 일반적입니다. 이 문장은 *TComponent*(또는 고유 편집기가 등록되어 있지 않은 *TComponent*의 자손) 타입의 모든 속성에 대해 *TComponentProperty* 속성 편집기를 등록합니다. 일반적으로 속성 편집기를 등록할 때 특정 타입의 편집기를 만든 다음 그 타입의 모든 속성에 대해 그 편집기를 사용하고자 하므로 두 번째와 세 번째 매개변수는 각각 **nil**과 비어 있는 문자열입니다.
- 두 번째 문장은 가장 특정한 등록 유형입니다. 이 문장은 특정 컴포넌트 타입에서 특정 속성에 대한 편집기를 등록합니다. 이런 경우 편집기는 모든 컴포넌트의 *TComponentName* 타입의 *Name* 속성에 대한 것입니다.
- 세 번째 문장은 첫 번째 문장보다 특정적이면서 두 번째 문장만큼 제한되어 있지 않습니다. 이 문장은 *TMenu* 타입의 컴포넌트에서 *TMenuItem* 타입의 모든 속성에 대한 편집기를 등록합니다.

속성 범주

IDE에서 Object Inspector를 통해 속성 범주별로 속성을 선택적으로 표시하고 숨길 수 있습니다. 새로운 사용자 지정 컴포넌트의 속성은 범주에 속성을 등록하여 이 스키마에 맞출 수 있습니다. `RegisterPropertyInCategory` 또는 `RegisterPropertiesInCategory`를 호출하여 컴포넌트를 등록하는 동시에 이 작업을 수행합니다. `RegisterPropertyInCategory`를 사용하여 단일 속성을 등록합니다. `RegisterPropertiesInCategory`를 사용하여 단일 함수 호출에서 여러 속성을 등록합니다. 이러한 함수는 DesignIntf 유닛에 정의되어 있습니다.

속성을 등록하거나 또는 일부가 등록될 때 사용자 지정 컴포넌트의 모든 속성을 등록하는 것은 필수 사항이 아닙니다. 범주에 명시적으로 연결되어 있지 않은 속성은 `TMiscellaneousCategory` 범주에 포함됩니다. 이러한 속성은 기본 범주에 따라 Object Inspector에서 표시되거나 숨겨집니다.

속성을 등록하기 위한 이 두 가지 함수 외에도 `IsPropertyInCategory` 함수가 있습니다. 이 함수는 지역화 유틸리티를 생성하는 데 유용합니다. 이 유틸리티에서 지정된 속성 범주에 속성이 등록되어 있는지 결정해야 합니다.

한 번에 하나의 속성 등록

한 번에 하나의 속성을 등록하고 `RegisterPropertyInCategory` 함수를 사용하여 그 속성을 속성 범주에 연결합니다. `RegisterPropertyInCategory`는 네 가지 오버로드된 변형으로 나타나는데 각 변형은 속성 범주에 연결할 사용자 지정 컴포넌트의 속성을 식별하기 위해 다른 기준 집합을 제공합니다.

첫 번째 변형을 사용하면 속성 이름에 의해서 속성을 식별할 수 있습니다. 아래 행은 컴포넌트의 시각적 표시에 관련된 속성을 등록하여 "AutoSize"라는 이름으로 속성을 식별합니다.

```
RegisterPropertyInCategory('Visual', 'AutoSize');
```

두 번째 변형은 지정된 타입의 컴포넌트에 나타나는 지정된 이름의 속성에만 범주를 제한한다는 점을 제외하면 첫 번째 변형과 매우 유사합니다. 아래 예제에서는 `TMyButton` 사용자 지정 클래스 컴포넌트의 "HelpContext"라는 속성을 "AutoSize" 범주로 등록합니다.

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

세 번째 변형은 이름이 아닌 타입을 사용하여 속성을 식별합니다. 아래 예제에서는 정수 타입에 기반하여 속성을 등록합니다.

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

마지막 변형은 속성 타입과 이름 모두를 사용하여 속성을 식별합니다. 아래 예제에서는 `TBitmap` 타입과 "Pattern" 이름 모두에 기반하여 속성을 등록합니다.

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

유용한 속성 범주와 사용 설명은 속성 범주 지정 단원을 참조하십시오.

한 번에 여러 속성 등록

한 번에 여러 속성을 등록하고 `RegisterPropertiesInCategory` 함수를 사용하여 그 속성들을 속성 범주에 연결합니다. `RegisterPropertiesInCategory`는 세 가지 오버로드된 변형으로 나타나는데 각 변형은 속성 범주에 연결할 사용자 지정 컴포넌트의 속성을 식별하기 위해 다른 기준 집합을 제공합니다.

첫 번째 변형을 사용하면 속성 이름이나 타입에 기반하여 속성을 식별할 수 있습니다. 목록은 상수의 배열로 전달됩니다. 아래 예제에서 "Text" 이름을 갖거나 `TEdit` 타입의 클래스에 속하는 속성은 "Pattern" 범주에 등록됩니다.

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

두 번째 변형을 사용하면 특정 컴포넌트에 속하는 속성 범주에 대해 등록된 속성을 제한할 수 있습니다. 등록할 속성의 목록에 타입은 포함되지 않고 이름만 포함됩니다. 예를 들어, 다음 코드는 모든 컴포넌트에 대해 다수의 속성을 'Help and Hints' 범주에 등록합니다.

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint', 'ParentShowHint', 'ShowHint']);
```

세 번째 변형을 사용하면 사용자가 특정 타입을 갖는 속성 범주에 대해 등록된 속성을 제한할 수 있습니다. 두 번째 변형에서 등록할 속성의 목록에는 이름만 포함될 수 있습니다.

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

유용한 속성 범주와 사용 설명은 속성 범주 지정 단원을 참조하십시오.

속성 범주 지정

범주에서 속성을 등록할 때 원하는 모든 문자열을 범주 이름으로 사용할 수 있습니다. 이전에 사용되지 않았던 문자열을 사용하는 경우, Object Inspector는 그 이름을 갖는 새로운 속성 범주 클래스를 생성합니다. 하지만 기본 제공된 범주 중 하나에 속성을 등록할 수도 있습니다. 기본 제공 속성 범주는 표 47.4에 설명되어 있습니다.

표 47.4 속성 범주

범주	용도
<i>Action</i>	<i>TEdit</i> 의 <i>Enabled</i> 및 <i>Hint</i> 속성은 런타임 액션에 관련된 속성입니다.
<i>Database</i>	<i>TQuery</i> 의 <i>DatabaseName</i> 및 <i>SQL</i> 속성은 데이터베이스 연산에 관련된 속성입니다.
<i>Drag, Drop, and Docking</i>	<i>TImage</i> 의 <i>DragCursor</i> 및 <i>DragKind</i> 속성은 끌어서 가져다 놓기 및 도킹 연산에 관련된 속성입니다.
<i>Help and Hints</i>	<i>TMemo</i> 의 <i>HelpContext</i> 및 <i>Hint</i> 속성은 온라인 도움말이나 힌트 사용에 관련된 속성입니다.
<i>Layout</i>	<i>TDBEdit</i> 의 <i>Top</i> 및 <i>Left</i> 속성은 디자인 타입에 컨트롤의 시각적 표시와 관련된 속성입니다.

표 47.4 속성 범주 (계속)

범주	용도
<i>Legacy</i>	<i>TComboBox</i> 의 <i>Ctl3D</i> 및 <i>ParentCtl3D</i> 속성은 폐기된 연산에 관련된 속성입니다.
<i>Linkage</i>	<i>TDataSource</i> 의 <i>DataSet</i> 속성은 한 컴포넌트를 다른 컴포넌트에 연결하는 것과 관련된 속성입니다.
<i>Locale</i>	<i>TMainMenu</i> 의 <i>BiDiMode</i> 및 <i>ParentBiDiMode</i> 속성은 국제 로케일에 관련된 속성입니다.
<i>Localizable</i>	애플리케이션의 지역화 버전에서 수정이 필요한 속성. <i>Caption</i> 과 같은 문자열 속성, 즉 컨트롤의 크기 및 위치를 결정하는 속성들이 이 범주에 많이 포함됩니다.
<i>Visual</i>	<i>TScrollBar</i> 의 <i>Align</i> 및 <i>Visible</i> 속성은 런타임 시 컨트롤의 가시적인 표시와 관련된 속성입니다.
<i>Input</i>	<i>TEdit</i> 의 <i>Enabled</i> 및 <i>ReadOnly</i> 속성은 데이터베이스 연산에 관련될 필요는 없고 데이터 입력과 관련이 있는 속성입니다.
<i>Miscellaneous</i>	<i>TSpeedButton</i> 의 <i>AllowAllUp</i> 및 <i>Name</i> 속성은 범주에 맞지 않거나 범주화될 필요가 없는 속성 및 특정 범주에 명시적으로 등록되지 않은 속성입니다.

IsPropertyInCategory 함수 사용

애플리케이션에서 기존 등록된 속성을 쿼리하여 지정된 속성이 이미 특정 범주에 등록되어 있는지 여부를 결정합니다. 이런 작업은 지역화 작업을 수행하기 위해 속성의 범주화를 확인하는 지역화 유틸리티와 같은 상황에서 특히 유용합니다. *IsPropertyInCategory* 함수의 두 가지 오버로드된 변형은 속성이 범주에 있는지 여부를 결정할 때 다른 기준을 고려합니다.

첫 번째 변형을 사용하여 소유하고 있는 컴포넌트의 클래스 타입과 속성 이름의 결합에 대한 비교 기준을 만들 수 있습니다. 아래 명령줄에서 *True*를 반환하는 *IsPropertyInCategory*에 대한 속성은 *TCustomEdit* 자손에 속해야 하고 "Text"라는 이름을 가지며 'Localizable' 속성 범주에 포함되어야 합니다.

```
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
```

두 번째 변형을 사용하여 소유하고 있는 컴포넌트의 클래스 이름과 속성 이름의 결합에 대한 비교 기준을 만들 수 있습니다. 아래 명령줄에서 *True*를 반환하는 *IsPropertyInCategory*에 대한 속성은 *TCustomEdit*의 자손이어야 하고 "Text"라는 이름을 가지며 'Localizable' 속성 범주에 포함되어야 합니다.

```
IsItThere := IsPropertyInCategory('Localizable', 'CustomEdit', 'Text');
```

컴포넌트 에디터 추가

컴포넌트 에디터는 디자이너에서 컴포넌트를 더블 클릭했을 때 일어나는 작업을 결정하고 마우스 오른쪽 버튼으로 컴포넌트를 클릭하면 나타나는 컨텍스트 메뉴에 명령을 추가합니다. 또한 컴포넌트 에디터는 컴포넌트를 사용자 지정 형식으로 Windows 클립 보드에 복사할 수 있습니다.

컴포넌트 에디터에 컴포넌트를 제공하지 않은 경우 Delphi는 기본 컴포넌트 에디터를 사용합니다. 기본 컴포넌트 에디터는 *TDefaultEditor* 클래스에 의해 구현됩니다. *TDefaultEditor*는 컴포넌트의 컨텍스트 메뉴에 새 항목을 추가하지 않습니다. 컴포넌트를 더블 클릭하면 *TDefaultEditor*는 해당 컴포넌트의 속성을 검색하고 첫 번째로 찾은 이벤트 핸들러를 생성하거나 탐색합니다.

컨텍스트 메뉴에 항목을 추가하려면 컴포넌트를 더블 클릭할 때 행동을 변경하거나 새 클립보드 형식을 추가하고 *TComponentEditor*에서 새 클래스를 파생한 후 사용자의 컴포넌트에 그 용도를 등록합니다. 오버라이드된 메소드에서 *TComponentEditor*의 *Component* 속성을 사용하여 편집 중인 컴포넌트에 액세스할 수 있습니다.

사용자 지정 컴포넌트 에디터 추가 작업은 다음 단계로 이루어집니다.

- 컨텍스트 메뉴에 항목 추가
- 더블 클릭 동작 변경
- 클립보드 형식 추가
- 컴포넌트 에디터 등록

컨텍스트 메뉴에 항목 추가

마우스 오른쪽 버튼으로 컴포넌트를 클릭하면 컴포넌트 에디터의 *GetVerbCount* 및 *GetVerb* 메소드가 호출되어 컨텍스트 메뉴를 만듭니다. 이러한 메소드를 오버라이드하여 컨텍스트 메뉴에 명령(동사)을 추가할 수 있습니다.

컨텍스트 메뉴에 항목을 추가하려면 다음 단계가 필요합니다.

- 메뉴 항목 지정
- 명령 구현

메뉴 항목 지정

GetVerbCount 메소드를 오버라이드하여 컨텍스트 메뉴에 추가할 명령의 수를 반환합니다. *GetVerb* 메소드를 오버라이드하여 이러한 명령 각각에 대해 추가할 문자열을 반환합니다. *GetVerb*를 오버라이드할 때 문자열에 앰퍼샌드(&)를 추가하면 다음 문자가 컨텍스트 메뉴에 밑줄이 나타나며 메뉴 항목을 선택하기 위한 바로가기 키로 사용됩니다. 대화 상자가 나타날 경우에는 명령 끝에 생략 기호(...)를 추가해야 합니다. *GetVerb*는 명령 인덱스를 나타내는 하나의 매개변수를 가집니다.

다음 코드는 *GetVerbCount* 및 *GetVerb* 메소드를 오버라이드하여 컨텍스트 메뉴에 두 개의 명령을 추가합니다.

```
function TMyEditor.GetVerbCount:Integer;
begin
    Result := 2;
end;

function TMyEditor.GetVerb(Index:Integer):String;
begin
    case Index of
```

```

0: Result := "&DoThis ...";
1: Result := "Do&That";
end;
end;

```

참고 *GetVerb* 메소드는 *GetVerbCount*에 의해 표시된 모든 가능한 인덱스의 값을 반환해야 합니다.

명령 구현

*GetVerb*가 제공하는 명령을 디자이너에서 선택하면 *ExecuteVerb* 메소드가 호출됩니다. *GetVerb* 메소드에서 제공하는 모든 명령에 대해 *ExecuteVerb* 메소드에서 동작을 구현합니다. 에디터의 *Component* 속성을 사용하여 편집 중인 컴포넌트에 액세스할 수 있습니다.

예를 들어, 다음 *ExecuteVerb* 메소드는 이전 예제에서 *GetVerb* 메소드의 명령을 구현합니다.

```

procedure TMyEditor.ExecuteVerb(Index: Integer);
var
  MySpecialDialog: TMyDialog;
begin
  case Index of
    0: begin
      MyDialog := TMySpecialDialog.Create(Application); { instantiate the editor }
      if MySpecialDialog.Execute then; { if the user OKs the dialog... }
        MyComponent.FThisProperty := MySpecialDialog.ReturnValue; { ...use the value }
        MySpecialDialog.Free; { destroy the editor }
      end;
    1: That; { call the That method }
  end;
end;

```

더블 클릭 동작 변경

컴포넌트를 더블 클릭하면 컴포넌트 에디터의 *Edit* 메소드가 호출됩니다. 기본적으로 *Edit* 메소드는 컨텍스트 메뉴에 추가된 첫 번째 명령을 실행합니다. 따라서 이전 예제에서 컴포넌트를 더블 클릭하면 *DoThis* 명령이 실행됩니다.

일반적으로 첫 번째 명령을 실행하는 것이 바람직하지만 이 기본 동작을 변경할 수도 있습니다. 예를 들어, 다음과 같은 경우에 대체 동작을 제공할 수 있습니다.

- 컨텍스트 메뉴에 명령을 추가하지 않을 경우
- 컴포넌트를 더블 클릭했을 때 여러 개의 명령이 결합된 대화 상자를 표시하기 원할 경우

컴포넌트를 더블 클릭한 경우, *Edit* 메소드를 오버라이드하여 새로운 동작을 지정합니다. 예를 들어, 다음 *Edit* 메소드는 컴포넌트를 더블 클릭할 때 글꼴 대화 상자를 나타냅니다.

```

procedure TMyEditor.Edit;
var
  FontDlg:TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
    finally
      FontDlg.Free;
    end;
end;

```

참고 컴포넌트를 더블 클릭해서 이벤트 핸들러의 코드 에디터를 표시하고자 하는 경우, *TComponentEditor* 대신 *TDefaultEditor*를 컴포넌트 에디터의 기본 클래스로 사용합니다. 그런 다음 *Edit* 메소드를 오버라이드하지 않고 보호되는 *TDefaultEditor.EditProperty* 메소드를 사용합니다. *EditProperty*는 컴포넌트의 이벤트 핸들러를 통해 검색하고 첫 번째로 찾은 이벤트를 나타냅니다. 특정 이벤트를 보기 위해서 이 작업을 변경할 수도 있습니다. 예를 들면, 다음과 같습니다.

```

procedure TMyEditor.EditProperty(PropertyEditor:TPropertyEditor;
  Continue, FreeEditor:Boolean)
begin
  if (PropertyEditor.ClassName = 'MethodProperty' and
    (PropertyEditor.GetName = 'nSpecialEvent' then
      // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;

```

클립보드 형식 추가

기본적으로 컴포넌트가 IDE에 선택되어 있는 동안 사용자가 Copy를 선택한 경우 해당 컴포넌트는 Delphi의 내부 형식으로 복사됩니다. 그런 다음 다른 폼이나 데이터 모듈로 붙여넣을 수 있습니다. 컴포넌트는 *Copy* 메소드를 오버라이드하여 추가 형식을 클립보드에 복사할 수 있습니다.

예를 들어, 다음 *Copy* 메소드를 통해 *TImage* 컴포넌트에서 해당 그림을 클립보드에 복사할 수 있습니다. 이 그림은 Delphi IDE에서 무시되지만 다른 애플리케이션에 붙여넣을 수 있습니다.

```

procedure TMyComponent.Copy;
var
  MyFormat : Word;
  AData,APalette : THandle;
begin
  TImage(Component).Picture.Bitmap.SaveToClipboardFormat(MyFormat, AData, APalette);
  Clipboard.SetAsHandle(MyFormat, AData);
end;

```

컴포넌트 에디터 등록

일단 컴포넌트 에디터가 정의되면 특정 컴포넌트 클래스를 사용하기 위해 등록될 수 있습니다. 등록된 컴포넌트 에디터가 폼 디자이너에서 선택되어 있는 경우, 해당 클래스의 각 컴포넌트에 대해 등록된 컴포넌트 에디터를 생성합니다.

컴포넌트 에디터와 컴포넌트 클래스를 연결하려면 *RegisterComponentEditor*를 호출합니다. *RegisterComponentEditor*는 편집기를 사용하는 컴포넌트 클래스의 이름과 정의된 컴포넌트 에디터 클래스의 이름을 취합니다. 예를 들어, 다음 문장은 *TMyComponent* 타입의 모든 컴포넌트를 사용하는 *TMyEditor*라는 컴포넌트 에디터 클래스를 등록합니다.

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

컴포넌트를 등록하는 *Register* 프로시저에서 *RegisterComponentEditor*를 호출합니다. 예를 들어, *TMyComponent*라는 새로운 컴포넌트 및 해당 컴포넌트 에디터인 *TMyEditor*가 모두 동일한 유닛에서 구현되는 경우, 다음 코드는 컴포넌트 및 컴포넌트 에디터에 연결된 것을 등록합니다.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent]);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
```

컴포넌트를 패키지로 컴파일

일단 컴포넌트가 등록되면 IDE에 설치하기 전에 패키지로 컴파일해야 합니다. 패키지에 는 사용자 지정 속성 편집기뿐만 아니라 하나 이상의 컴포넌트를 포함할 수 있습니다. 패키지에 대한 자세한 내용은 11장 "패키지와 컴포넌트 사용"을 참조하십시오.

패키지를 생성하고 컴파일하려면 11-6 페이지의 "패키지 생성 및 편집"을 참조하십시오. 패키지의 포함 목록에 사용자 지정 컴포넌트의 소스 코드 유닛을 포함시킵니다. 컴포넌트가 다른 패키지에 종속되어 있는 경우 해당 패키지를 요청 목록에 포함시킵니다.

IDE에 컴포넌트를 설치하려면 11-5 페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

48

기존 컴포넌트 수정

컴포넌트를 만드는 가장 쉬운 방법은 사용자가 원하는 대부분의 기능을 가진 컴포넌트를 가져와서 필요에 맞게 변경하는 것입니다. 다음 예는 표준 메모 컴포넌트를 수정하여 기본값으로 줄 바꿈하지 않는 메모를 만드는 것을 보여 줍니다.

메모 컴포넌트의 *WordWrap* 속성 값은 *True*로 초기화됩니다. 줄 바꿈하지 않는 메모를 자주 사용하는 경우 기본값으로 줄 바꿈 하지 않는 새 메모 컴포넌트를 만들 수 있습니다.

참고 *published* 속성을 수정하거나 기존 컴포넌트의 특정 이벤트 핸들러를 저장하려면 새로운 클래스를 만드는 것보다 *컴포넌트 템플릿*을 사용하는 것이 더 쉬운 경우가 있습니다.

기존 컴포넌트 수정은 다음 두 단계만 거치면 됩니다.

- 컴포넌트 생성 및 등록
- 컴포넌트 클래스 수정

컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 다음과 동일한 방법으로 시작됩니다. 유닛을 만들고 컴포넌트 클래스를 파생시켜 등록한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 40-8 페이지의 "새 컴포넌트 생성"에 개괄적으로 설명되어 있습니다.

이 예제에서는 다음과 같은 특성을 사용하여 컴포넌트를 만드는 일반적인 절차를 따릅니다.

- 컴포넌트의 유닛 *Memos*를 호출합니다.
- *TMemo*의 자손인 *TWrapMemo*라는 새 컴포넌트 타입을 파생시킵니다.
- 컴포넌트 팔레트의 *Samples* 페이지에 *TWrapMemo*를 등록합니다.
- 결과 유닛은 다음과 같습니다.

```

unit Memos;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, StdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.

```

새 컴포넌트를 지금 컴파일하고 설치하면 그 조상인 *TMemo*와 똑같은 역할을 수행합니다. 다음 단원에서 이 컴포넌트를 간단히 변경하게 됩니다.

컴포넌트 클래스 수정

일단 새 컴포넌트 클래스를 생성하면 어떤 방식으로든 수정할 수 있습니다. 여기서는 메모 컴포넌트의 한 속성의 초기 값만 변경합니다. 이를 위해서는 컴포넌트 클래스에 두 가지 작은 변경을 수행해야 합니다.

- 생성자 오버라이드
- 새 기본 속성 값 지정

생성자는 실제로 속성의 값을 설정합니다. 기본값은 Delphi가 폼(VCL에서는 .dfm 및 CLX에서는 .xfm) 파일에 저장할 값을 알려 줍니다. Delphi는 기본값과 다른 값만 저장하기 때문에 두 단계를 모두 수행하는 것이 중요합니다.

생성자 오버라이드

디자인 타임에 컴포넌트가 폼에 위치해 있거나 런타임에 애플리케이션이 컴포넌트를 생성할 때 컴포넌트의 생성자는 속성 값을 설정합니다. 컴포넌트가 폼 파일에서 로드될 때 애플리케이션은 디자인 타임에 변경된 모든 속성을 설정합니다.

참고 생성자를 오버라이드할 때 새 생성자는 상속된 생성자를 호출한 후 다른 작업을 수행해야 합니다. 자세한 내용은 41-8 페이지의 "메소드 오버라이드"를 참조하십시오.

이 예제에서 새 컴포넌트가 *WordWrap* 속성을 *False*로 설정하기 위해서 *TMemo*에서 상속된 생성자를 오버라이드해야 합니다. 이렇게 하려면 생성자 오버라이드를 forward 선언에 첨가한 다음 **implementation** 부분에서 새 생성자를 작성합니다.

```

type
  TWrapMemo = class(TMemo)
  public
    constructor Create(AOwner:TComponent); override; { constructors are always public }
  end; { this syntax is always the same }

```

```

:
constructor TWrapMemo.Create(AOwner:TComponent); { this goes after implementation }
begin
  inherited Create(AOwner); { ALWAYS do this first! }
  WordWrap := False; { set the new desired value }
end;

```

이제 컴포넌트 팔레트에 새 컴포넌트를 설치한 다음 폼에 추가할 수 있습니다. *WordWrap* 속성이 이제는 *False*로 초기화되어 있음에 유의하십시오.

초기 속성 값을 변경하려면 그 값을 기본값으로 지정해야 합니다. 생성자에 의해 설정된 값이 지정된 기본값과 일치하지 않는 경우, Delphi는 적절한 값을 저장하거나 재저장할 수 없습니다.

새 기본 속성 값 지정

Delphi가 폼 파일에서 폼에 대한 설명 저장 시 기본값과 다른 속성 값만 저장합니다. 기본값과 다른 값만 저장하면 폼 파일 크기를 작게 유지할 수 있고 폼을 더 빨리 로드할 수 있습니다. 속성을 만들거나 기본값을 변경할 때 속성 선언이 새 기본값을 포함하도록 업데이트하는 것이 좋습니다. 폼 파일, 로드 및 기본값은 47장 "디자인 타임 시 컴포넌트 사용"에 자세하게 설명되어 있습니다.

속성의 기본값을 변경하려면 **default** 지시어가 뒤에 오는 속성 이름과 새 기본값을 재선언합니다. 전체 속성을 재선언할 필요 없이 이름과 기본값만 재선언합니다.

줄 바꿈 메모 컴포넌트에서 객체 선언의 **published** 부분에 있는 *WordWrap* 속성을 *False* 기본값으로 재선언합니다.

```

type
  TWrapMemo = class(TMemo)
  :
  published
    property WordWrap default False;
  end;

```

기본 속성 값 지정이 컴포넌트의 작업에 영향을 주지는 않습니다. 따라서 컴포넌트의 생성자에서 값을 초기화해야 합니다. 기본값 재선언은 Delphi에게 폼 파일에 *WordWrap* 을 언제 기록할지 확실히 알려 주는 것입니다.

49

그래픽 컴포넌트 생성

그래픽 컨트롤은 간단한 유형의 컴포넌트입니다. 순수한 그래픽 컨트롤은 포커스를 받지 않으므로 창 핸들을 갖거나 필요로 하지 않습니다. 사용자는 마우스로 컨트롤을 조작할 수 있지만 키보드 인터페이스는 없습니다.

이 장에 설명되어 있는 그래픽 컴포넌트는 컴포넌트 팔레트의 추가 페이지에 있는 도형 컴포넌트인 *TShape*입니다. 생성된 컴포넌트가 표준 도형 컴포넌트와 동일하더라도 중복 식별자를 피하기 위해서는 생성된 컴포넌트를 다른 이름으로 불러야 합니다. 이 장에서는 도형 컴포넌트를 *TSampleShape*라고 하며 도형 컴포넌트의 생성에 관련된 모든 단계를 설명합니다.

- 컴포넌트 생성 및 등록
- 상속된 속성 게시
- 그래픽 기능 추가

컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 다음과 동일한 방법으로 시작됩니다. 유닛을 만들고 컴포넌트 클래스를 파생시켜 파생된 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 40-8 페이지의 "새 컴포넌트 생성"에 개괄적으로 설명되어 있습니다.

이 예제에서는 다음과 같은 특성을 사용하여 컴포넌트를 만드는 일반적인 절차를 따릅니다.

- 컴포넌트의 유닛의 이름을 *Shapes*라고 합니다.
- *TGraphicControl*의 자손인 *TSampleShape*라는 새 컴포넌트 타입을 파생시킵니다.
- 컴포넌트 팔레트의 Sample 페이지에 *TSampleShape*를 등록합니다.

결과 유닛은 다음과 같습니다.

```

unit Shapes;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TSampleShape = class(TGraphicControl)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.

```

상속된 속성 게시

일단 컴포넌트 타입을 파생시키면 새 컴포넌트에서 표면에 나타내고자 하는 조상 클래스의 protected 부분에서 선언된 속성과 이벤트를 결정할 수 있습니다. *TGraphicControl* 은 컴포넌트가 컨트롤로 작용하도록 하는 모든 속성을 이미 게시하고 있으므로 사용자는 마우스 이벤트에 응답하고 드래그 앤 드롭 작업을 처리하는 기능을 게시해야 합니다.

상속된 속성 및 이벤트 게시에 대해서는 42-3 페이지의 "상속된 속성 게시"와 43-5 페이지의 "이벤트 보이기"에서 설명합니다. 두 가지 프로세스에서는 클래스 선언의 published 부분에 있는 속성 이름만 재정의하면 됩니다.

도형 컨트롤에서 세 개의 마우스 이벤트, 세 개의 드래그 앤 드롭 이벤트, 두 개의 드래그 앤 드롭 속성을 게시할 수 있습니다.

```

type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;           { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;         { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;       { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;

```

이제 사용자는 예제 도형 컨트롤을 통해 마우스 및 드래그 앤 드롭 상호 작용을 사용할 수 있습니다.

그래픽 기능 추가

일단 그래픽 컴포넌트를 선언하고 사용하고자 하는 상속된 속성을 게시하면 컴포넌트를 구별하는 그래픽 기능을 추가할 수 있습니다. 그래픽 컨트롤 생성 시 수행할 두 가지 작업은 다음과 같습니다.

- 1 그릴 대상 결정
- 2 컴포넌트 이미지 그리기

또한 도형 컨트롤 예제에서는 애플리케이션 개발자가 디자인 타임에 도형 모양을 사용자 지정할 수 있는 속성을 추가합니다.

그릴 대상 결정

그래픽 컨트롤을 통해 사용자 입력을 포함한 동적 조건을 반영하도록 모양을 변경할 수 있습니다. 항상 똑같이 보이는 그래픽 컨트롤은 컴포넌트가 아닐 수 있습니다. 정적 이미지를 원할 경우, 컨트롤을 사용하는 대신 이미지를 가져올 수 있습니다.

일반적으로 그래픽 컨트롤의 모양은 그 속성의 조합에 따라 달라집니다. 예를 들어, 계측기(gauge) 컨트롤에는 모양과 방향, 진행 상황을 그래픽뿐만 아니라 수치적으로 표시할지 여부를 결정하는 속성이 있습니다. 이와 마찬가지로 도형 컨트롤에는 그려야 할 도형의 유형을 결정하는 속성이 있습니다.

그릴 도형을 결정하는 속성을 컨트롤에 제공하려면 *Shape*라는 속성을 추가합니다. 이 작업을 수행하려면 다음 단계가 필요합니다.

- 1 속성 타입 선언
- 2 속성 선언
- 3 구현 메소드 작성

속성 생성에 대해서는 42장 "속성 생성"에 자세히 설명되어 있습니다.

속성 타입 선언

사용자 정의 타입의 속성을 선언할 경우, 속성을 포함하는 클래스를 선언하기 전에 먼저 그 타입을 선언해야 합니다. 속성에 대한 사용자 정의 타입의 가장 일반적인 유형은 열거(enumerated) 타입입니다.

도형 컨트롤에서 컨트롤이 그릴 수 있는 각 도형의 종류에 대한 요소를 갖는 열거 타입이 필요합니다.

도형 컨트롤 클래스의 선언 위에 다음과 같은 타입 정의를 추가합니다.

```
type
    TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
        sstEllipse, sstCircle);
    TSampleShape = class(TGraphicControl) { this is already there }
```

이제 이 타입을 사용하여 클래스에서 새 속성을 선언할 수 있습니다.

속성 선언

속성을 선언할 때 속성의 데이터를 저장하려면 private 필드를 선언한 다음 속성 값을 읽고 쓰기 위한 메소드를 지정해야 합니다. 값을 읽기 위해 메소드를 사용할 필요가 없는 경우가 종종 있으며 대신 저장된 데이터를 가리킬 수 있습니다.

도형 컨트롤에서는 현재 도형을 유지하는 필드를 선언한 다음 그 필드를 읽는 속성을 선언하고 메소드 호출을 통해 그 필드에 씁니다.

다음 선언을 *TSampleShape*에 추가합니다.

```

type
  TSampleShape = class(TGraphicControl)
  private
    FShape:TSampleShapeType; { field to hold property value }
    procedure SetShape(Value:TSampleShapeType);
  published
    property Shape:TSampleShapeType read FShape write SetShape;
  end;

```

이제 남은 작업은 *SetShape*의 구현을 추가하는 것입니다.

구현 메소드 작성

속성 정의의 **read** 또는 **write** 부분이 저장된 속성 데이터에 직접 액세스하는 대신 메소드를 사용하는 경우에는 그 메소드를 구현해야 합니다.

SetShape 메소드의 구현을 유닛의 **implementation** 부분에 추가합니다.

```

procedure TSampleShape.SetShape(Value:TSampleShapeType);
begin
  if FShape <> Value then { ignore if this isn't a change }
  begin
    FShape := Value; { store the new value }
    Invalidate; { force a repaint with the new shape }
  end;
end;

```

생성자 및 소멸자 오버라이드

기본 속성 값을 변경하고 컴포넌트의 소유된 클래스를 초기화하려면 상속된 생성자와 소멸자를 오버라이드합니다. 두 가지 경우 모두 새로운 생성자나 소멸자에서 상속된 메소드를 항상 호출해야 합니다.

기본 속성 값 변경

그래픽 컨트롤의 기본 크기가 다소 작기 때문에 생성자에서 너비와 높이를 변경할 수 있습니다. 기본 속성 값 변경에 대해서는 48장 "기존 컴포넌트 수정"에 자세히 설명되어 있습니다.

이 예에서 도형 컨트롤의 크기는 각 변이 65픽셀인 사각형으로 설정됩니다.

컴포넌트 클래스의 선언에 오버라이드된 생성자를 추가합니다.

```

type
  TSampleShape = class(TGraphicControl)
  public { constructors are always public }
    constructor Create(AOwner:TComponent); override { remember override directive }
  end;

```


- 1 새 기본값을 갖는 *높이*와 *너비* 속성을 재선언합니다.

```
type
  TSampleShape = class(TGraphicControl)
  :
  published
    property Height default 65;
    property Width default 65;
  end;
```

- 2 유닛의 **implementation** 부분에 새로운 생성자를 작성합니다.

```
constructor TSampleShape.Create(AOwner:TComponent);
begin
  inherited Create(AOwner); { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;
```

펜과 브러시 게시

기본적으로 캔버스에는 가는 검정색 펜과 굵은 흰색 브러시가 있습니다. 개발자가 펜과 브러시를 변경하도록 하려면 디자인 타임에 조작할 수 있는 클래스를 제공한 다음 색칠하는 도중에 캔버스에 클래스를 복사해야 합니다. 컴포넌트는 보조 펜이나 브러시를 소유하고 그 생성과 소멸을 책임지므로 보조 펜이나 브러시와 같은 클래스를 *소유된 클래스*라고 합니다.

소유된 클래스 관리에는 다음과 같은 단계가 필요합니다.

- 1 클래스 필드 선언
- 2 액세스 속성 선언
- 3 소유된 클래스 초기화
- 4 소유된 클래스의 속성 설정

클래스 필드 선언

컴포넌트가 소유하는 각 클래스에는 컴포넌트에서 클래스에 대해 선언된 클래스 필드가 있어야 합니다. 클래스 필드는 컴포넌트가 자신을 소멸하기 전에 클래스를 소멸할 수 있도록 소유된 객체에 대한 포인터를 항상 가지도록 합니다. 일반적으로 컴포넌트는 자신의 생성자에서 소유된 객체를 초기화하고 자신의 소멸자에서 소유된 객체를 소멸합니다.

소유된 객체의 필드는 거의 항상 `private`으로 선언됩니다. 애플리케이션이나 다른 컴포넌트가 소유된 객체에 액세스해야 하는 경우에는 이러한 목적을 위해서 **published** 또는 **public** 속성을 선언할 수 있습니다.

도형 컨트롤에 펜과 브러시의 필드를 추가합니다.

```

type
  TSampleShape = class(TGraphicControl)
  private
    { fields are nearly always private }
    FPen:TPen;      { a field for the pen object }
    FBrush:TBrush; { a field for the brush object }
    :
  end;

```

액세스 속성 선언

객체 타입의 속성을 선언하여 컴포넌트의 소유된 객체에 대한 액세스를 제공할 수 있습니다. 이렇게 하면 개발자가 디자인 타임이나 런타임 시 객체에 액세스할 수 있습니다. 일반적으로 속성의 읽기 부분은 클래스 필드를 참조하지만 쓰기 부분은 컴포넌트가 소유된 객체의 변경 내용에 반응할 수 있도록 하는 메소드를 호출합니다.

펜과 브러시 필드에 대한 액세스를 제공하는 속성을 도형 컨트롤에 추가합니다. 또한 펜이나 브러시의 변경 사항에 반응하기 위한 메소드를 선언합니다.

```

type
  TSampleShape = class(TGraphicControl)
  :
  private
    { these methods should be private }
    procedure SetBrush(Value:TBrush);
    procedure SetPen(Value:TPen);
  published
    { make these available at design time }
    property Brush:TBrush read FBrush write SetBrush;
    property Pen:TPen read FPen write SetPen;
  end;

```

그런 다음 유닛의 구현 부분에 *SetBrush* 및 *SetPen* 메소드를 작성합니다.

```

procedure TSampleShape.SetBrush(Value:TBrush);
begin
  FBrush.Assign(Value);           { replace existing brush with parameter }
end;

procedure TSampleShape.SetPen(Value:TPen);
begin
  FPen.Assign(Value);            { replace existing pen with parameter }
end;

```

다음과 같은 방법으로 *Value*의 값을 *FBrush*에 직접 할당하면

```
FBrush := Value;
```

*FBrush*의 내부 포인터를 덮어쓰고, 메모리가 손실되며, 소유권 문제가 많이 일어납니다.

소유된 클래스 초기화

클래스를 컴포넌트에 추가할 때 컴포넌트의 생성자는 사용자가 런타임에 객체와 상호 작용할 수 있도록 객체를 초기화해야 합니다. 이와 마찬가지로 컴포넌트의 소멸자는 소유된 객체를 소멸하고 나서 컴포넌트 자체를 소멸해야 합니다.

펜과 브러시를 도형 컨트롤에 추가했기 때문에 해당 펜과 브러시를 도형 컨트롤 생성자에서 초기화하고 컨트롤 소멸자에서 소멸해야 합니다.

- 1 도형 컨트롤 생성자에서 펜과 브러시를 생성합니다.

```

constructor TSampleShape.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                { construct the pen }
  FBrush := TBrush.Create;           { construct the brush }
end;

```

- 2 컴포넌트 클래스의 선언에 오버라이드된 소멸자를 추가합니다.

```

type
  TSampleShape = class(TGraphicControl)
  public
    constructor Create(AOwner:TComponent); override;   { destructors are always public }
    destructor Destroy; override;                   { remember override directive }
  end;

```

- 3 유닛의 **implementation** 부분에 새 생성자를 작성합니다.

```

destructor TSampleShape.Destroy;
begin
  FPen.Free;                            { destroy the pen object }
  FBrush.Free;                          { destroy the brush object }
  inherited Destroy;                    { always call the inherited destructor, too }
end;

```

소유된 클래스의 속성 설정

펜과 브러시 클래스를 처리하는 최종 단계에서는 펜과 브러시의 변경으로 도형 컨트롤이 그 자체를 다시 그리는지 확인할 필요가 있습니다. 펜과 브러시 클래스는 모두 *OnChange* 이벤트를 가지므로 도형 컨트롤에서 메소드를 만들어서 두 *OnChange* 이벤트가 모두 메소드를 가리키도록 할 수 있습니다.

다음 메소드를 도형 컨트롤에 추가하고 컴포넌트 생성자를 업데이트하여 펜과 브러시 이벤트를 새로운 메소드로 설정합니다.

```

type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender:TObject);
  end;
  ..
implementation
  ..

```

```

constructor TSampleShape.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                 { construct the pen }
  FPen.OnChange := StyleChanged;      { assign method to OnChange event }
  FBrush := TBrush.Create;            { construct the brush }
  FBrush.OnChange := StyleChanged;    { assign method to OnChange event }
end;

procedure TSampleShape.StyleChanged(Sender:TObject);
begin
  Invalidate;                          { erase and repaint the component }
end;

```

이렇게 변경하면 컴포넌트는 펜이나 브러시에 대한 변경 사항을 반영하기 위해 다시 그립니다.

컴포넌트 이미지 그리기

그래픽 컨트롤의 필수 요소는 그래픽 컨트롤이 화면에서 자신의 이미지를 그리는 방식입니다. *TGraphicControl* 추상 타입은 컨트롤에 원하는 이미지를 그리기 위해 오버라이드하는 *Paint*라는 메소드를 정의합니다.

도형 컨트롤의 *Paint* 메소드는 몇 가지 해야 할 일들이 있습니다.

- 사용자가 선택한 펜과 브러시를 사용합니다.
- 선택한 도형을 사용합니다.
- 정사각형과 원이 동일한 너비와 높이를 사용하도록 조정합니다.

Paint 메소드 오버라이드에는 다음 두 단계의 작업이 필요합니다.

- 1 *Paint*를 컴포넌트 선언에 추가합니다.
- 2 유닛의 **implementation** 부분에 *Paint* 메소드를 작성합니다.

도형 컨트롤의 경우, 클래스 선언에 다음 선언을 추가합니다.

```

type
  TSampleShape = class(TGraphicControl)
  :
  protected
    procedure Paint; override;
  :
  end;

```

그런 다음 유닛의 **implementation** 부분에 메소드를 작성합니다.

```

procedure TSampleShape.Paint;
begin
  with Canvas do
    begin
      Pen := FPen;           { copy the component's pen }
      Brush := FBrush;      { copy the component's brush }
      case FShape of

```

```

    sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height);           { draw rectangles and squares }
    sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4);{ draw rounded shapes }
    sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height);             { draw round shapes }
    end;
end;
end;

```

*Paint*는 컨트롤이 자신의 이미지를 업데이트해야 할 때마다 호출됩니다. 컨트롤이 처음 나타나거나 컨트롤 앞의 창이 없어질 때 컨트롤이 그려집니다. 또한 *StyleChanged* 메소드가 하는 것과 같이 *Invalidate* 메소드를 호출하여 다시 그리도록 할 수 있습니다.

정교한 도형 그리기

표준 도형 컨트롤은 예제 도형 컨트롤이 아직 수행하지 않은 작업을 한 가지 더 수행합니다. 표준 도형 컨트롤은 직사각형과 타원뿐만 아니라 정사각형과 원을 처리합니다. 그렇게 하려면 가장 짧은 면을 찾아 이미지를 가운데에 배치하는 코드를 작성해야 합니다.

다음은 정사각형과 타원에 맞게 조정하는 정교한 *Paint* 메소드입니다.

```

procedure TSampleShape.Paint;
var
    X, Y, W, H, S:Integer;
begin
    with Canvas do
        begin
            Pen := FPen;           { copy the component's pen }
            Brush := FBrush;       { copy the component's brush }
            W := Width;            { use the component width }
            H := Height;           { use the component height }
            if W < H then S := W else S := H;   { save smallest for circles/squares }

            case FShape of         { adjust height, width and position }
                sstRectangle, sstRoundRect, sstEllipse:
                    begin
                        X := 0;     { origin is top-left for these shapes }
                        Y := 0;
                    end;
                sstSquare, sstRoundSquare, sstCircle:
                    begin
                        X := (W - S) div 2;   { center these horizontally... }
                        Y := (H - S) div 2;   { ...and vertically }
                        W := S;               { use shortest dimension for width... }
                        H := S;               { ...and for height }
                    end;
            end;

            case FShape of
                sstRectangle, sstSquare:
                    Rectangle(X, Y, X + W, Y + H);           { draw rectangles and squares }
                sstRoundRect, sstRoundSquare:
                    RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);   { draw rounded shapes }
            end;

```

그래픽 기능 추가

```
        sstCircle, sstEllipse:
            Ellipse(X, Y, X + W, Y + H);           { draw round shapes }
    end;
end;
end;
```

50

그리드 사용자 지정

Delphi는 사용자 지정 컴포넌트의 기반으로 사용할 수 있는 추상 컴포넌트를 제공합니다. 추상 컴포넌트에서 가장 중요한 것은 그리드와 리스트 박스입니다. 이 장에서는 기본 그리드 컴포넌트인 *TCustomGrid*로부터 한 달을 출력하는 작은 달력을 만드는 방법을 보여 줍니다.

달력을 만드는 데에는 다음 작업들이 포함됩니다.

- 컴포넌트 생성 및 등록
- 상속된 속성 게시
- 초기 값 변경
- 셀 크기 조정
- 셀 채우기
- 월과 연도 탐색
- 일(day) 탐색

결과 컴포넌트는 컴포넌트 팔레트의 Samples 페이지에 있는 *TCalendar* 컴포넌트와 유사합니다.

컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 다음과 동일한 방법으로 시작됩니다. 유닛을 만들고 컴포넌트 클래스를 파생시켜 파생된 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 40-8 페이지의 "새 컴포넌트 생성"에 개괄적으로 설명되어 있습니다.

이 예제에서는 다음 특성을 사용하여 컴포넌트를 생성하는 일반적인 절차를 따릅니다.

- 컴포넌트의 *CalSamp* 유닛을 호출합니다.
- *TCustomGrid*의 자손인 *TSampleCalendar*라는 새 컴포넌트 타입을 파생시킵니다.
- 컴포넌트 팔레트의 Samples 페이지에 *TSampleCalendar*를 등록합니다.

VCL의 *TCustomGrid*로부터 상속 받은 결과 유닛은 다음과 같습니다.

```

unit CalSamp;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;

type
  TSampleCalendar = class(TCustomGrid)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;

end.

```

CLX 버전의 *TCustomGrid*로부터 상속 받는 경우, **uses** 절만 CLX 유닛을 대신 보여주는 것이 다릅니다.

달력 컴포넌트를 지금 설치한다면 Sample 페이지에 나타날 것입니다. 가장 기본적인 컨트롤 속성만 사용할 수 있습니다. 다음 단계는 보다 특화된 일부 속성을 달력 사용자가 사용할 수 있게 합니다.

참고 방금 컴파일한 샘플 달력 컴포넌트를 바로 설치할 수는 있지만 아직은 그 컴포넌트를 폼에 두지 마십시오. *TCustomGrid* 컴포넌트는 인스턴스 객체를 만들기 전에 재선언해야 하는 추상 *DrawCell* 메소드를 가집니다. *DrawCell* 메소드 오버라이드에 대해서는 이후에 나오는 "셀 채우기"에서 설명합니다.

상속된 속성 게시

추상 그리드 컴포넌트인 *TCustomGrid*는 다수의 **protected** 속성을 제공합니다. 이러한 속성들 중에서 달력 컨트롤의 사용자가 사용할 수 있는 속성을 선택할 수 있습니다.

컴포넌트의 사용자가 상속된 **protected** 속성을 사용할 수 있도록 하려면 컴포넌트 선언의 **published** 부분의 속성을 재선언합니다.

여기에 표시된 대로 달력 컨트롤에서 다음의 속성과 이벤트를 게시합니다.

```

type
  TSampleCalendar = class(TCustomGrid)
  published
    property Align; { publish properties }
    property BorderStyle;
    property Color;
    property Font;
    property GridLineWidth;
    property ParentColor;
    property ParentFont;

```



```

property OnClick; { publish events }
property OnDbClick;
property OnDragDrop;
property OnDragOver;
property OnEndDrag;
property OnKeyDown;
property OnKeyPress;
property OnKeyUp;
end;

```

사용자가 어떤 그리드 선을 그릴지 선택할 수 있게 해주는 *Options* 속성과 같이 게시할 수 있는 다른 많은 속성들도 있지만 달력에 적용하지는 못합니다.

컴포넌트 팔레트에 수정된 달력 컴포넌트를 설치하고 그 컴포넌트를 애플리케이션에서 사용하는 경우, 완전한 기능을 갖는 달력에서 다양한 속성과 이벤트를 사용할 수 있습니다. 이제 사용자는 사용자가 만든 디자인에 새 기능을 추가할 수 있습니다.

초기 값 변경

달력은 고정된 수의 행과 열을 갖는 그리드가 필수적이지만 모든 행이 날짜를 가지는 것은 아닙니다. 이런 이유에서 *ColCount* 및 *RowCount* 그리드 속성을 게시하지 않습니다. 달력의 사용자가 주 7일 이외에 다른 것을 표시하려고 하지는 않을 것이기 때문입니다. 그러나 일 주일이 항상 7일이 되도록 이러한 속성의 초기 값을 설정해야 합니다.

컴포넌트 속성의 초기 값을 변경하려면 생성자를 오버라이드하여 원하는 값을 설정합니다. 생성자는 가상이어야 합니다.

생성자를 컴포넌트 객체 선언의 **public** 부분에 추가한 다음 컴포넌트 유닛의 **implementation** 부분에 새 생성자를 작성해야 합니다. 새 생성자의 첫 번째 문장은 항상 상속된 생성자에 대한 호출이어야 합니다.

```

type
  TSampleCalendar = class(TCustomGrid
  public
    constructor Create(AOwner:TComponent); override;
    :
    end;
  :
  constructor TSampleCalendar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner); { call inherited constructor }
  ColCount := 7; { always seven days/week }
  RowCount := 7; { always six weeks plus the headings }
  FixedCols := 0; { no row labels }
  FixedRows := 1; { one row for day names }
  ScrollBars := ssNone; { no need to scroll }
  Options := Options - [goRangeSelect] + [goDrawFocusSelected]; {disable range selection}
end;

```

이제 달력에는 열과 행이 각각 일곱 개씩 있으며 맨 위의 행이 고정되어 있거나 스크롤되지 않습니다.

셀 크기 조정

VCL 사용자나 애플리케이션이 창이나 컨트롤의 크기를 변경할 때 Windows는 *WM_SIZE*라는 메시지를 해당 창이나 컨트롤에 보내 나중에 새 창에서의 그리기에 필요한 설정을 조정할 수 있게 합니다. 사용자의 VCL 컴포넌트는 셀의 크기를 변경하는 것으로 해당 메시지에 대해 응답할 수 있으므로 컨트롤 경계 내에 모두 맞춰집니다. *WM_SIZE* 메시지에 응답하려면 컴포넌트에 메시지 처리 메소드를 추가합니다.

메시지 처리 메소드 생성에 관한 자세한 내용은 46-5 페이지의 "새 메시지 핸들러 생성"을 참조하십시오.

이 경우 달력 컨트롤은 *WM_SIZE*에 응답하여 *WMSize*라는 protected 메소드를 *WM_SIZE* 메시지에 인덱스된 컨트롤에 추가한 다음 모든 셀들이 새로운 크기 내에서 보일 수 있도록 적당한 셀 크기를 계산하는 메소드를 작성합니다.

```

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
    :
  end;
  :
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines:Integer;           { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;      { calculate combined size of all lines }
  DefaultColWidth := (Message.Width - GridLines) div 7; { set new default cell width }
  DefaultRowHeight := (Message.Height - GridLines) div 7; { and cell height }
end;

```

이제 크기가 조정된 달력은 컨트롤에 맞는 최대 크기로 모든 셀을 표시합니다.

CLX CLX에서는 창 또는 컨트롤의 크기에 대한 변경이 protected *BoundsChanged* 메소드의 호출에 의해 자동적으로 통지됩니다. CLX 컴포넌트는 셀의 크기를 변경하여 이 공지에 대해 응답할 수 있으므로 컨트롤 경계 내에 모두 맞춰집니다.

이 경우 달력 컨트롤은 모든 셀이 새로운 크기로 보여질 수 있는 적합한 셀 크기를 계산하기 위해서 *BoundsChanged*를 오버라이드해야 합니다.

```

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
    :
  end;
  :
procedure TSampleCalendar.BoundsChanged;
var
  GridLines:Integer;           { temporary local variable }

```

```

begin
  GridLines := 6 * GridLineWidth;           { calculate combined size of all lines }
  DefaultColWidth := (Width - GridLines) div 7; { set new default cell width }
  DefaultRowHeight := (Height - GridLines) div 7; { and cell height }
  inherited; {now call the inherited method }
end;

```

셀 채우기

그리드 컨트롤은 셀 단위로 내용을 입력합니다. 달력의 경우는 각 셀에 속하는 날짜를 계산하는 것을 의미합니다. 그리드 셀의 기본 그리기는 *DrawCell*이라는 가상 메소드에서 일어납니다.

그리드 셀의 내용을 채우려면 *DrawCell* 메소드를 오버라이드합니다.

채우기가 가장 쉬운 부분은 고정 행의 헤더 셀입니다. 런타임 라이브러리에는 짧은 요일 이름의 배열이 들어 있으므로 달력의 경우 각 열에 적절한 요일 이름을 사용합니다.

```

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow:Longint; ARect:TRect; AState:TGridDrawState);
    override;
  end;
:
procedure TSampleCalendar.DrawCell(ACol, ARow:Longint; ARect:TRect;
  AState:TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);{ use RTL strings }
  end;
end;

```

날짜 추적

달력 컨트롤이 유용하게 쓰이도록 하려면 사용자와 애플리케이션은 일, 월, 연도를 설정하기 위한 메커니즘이 있어야 합니다. Delphi는 *TDateTime* 타입의 변수로 날짜와 시간을 저장합니다. *TDateTime*은 날짜와 시간의 인코딩된 숫자 표시로 프로그래밍 조작에는 유용하나 사람이 사용하기에는 불편합니다.

따라서 인코딩된 폼으로 날짜를 저장하고 그 값에 런타임 액세스를 제공할 수 있으며 달력 컴포넌트의 사용자가 디자인 타임에 설정할 수 있는 *Day*, *Month* 및 *Year* 속성을 제공할 수 있습니다.

달력에서의 날짜 추적은 다음과 같은 단계로 수행됩니다.

- 내부 날짜 저장
- 일, 월, 연도 액세스
- 일(day) 수 생성
- 오늘 날짜 선택

내부 날짜 저장

달력의 날짜를 저장하려면 날짜를 담은 private 필드와 그 날짜에 액세스를 제공하는 런타임 전용 속성이 필요합니다.

내부 날짜를 달력에 추가하려면 다음과 같은 세 단계를 거쳐야 합니다.

- 1 날짜를 유지할 private 필드를 선언합니다.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate:TDateTime;
    ;
```

- 2 생성자에서 날짜 필드를 초기화합니다.

```
constructor TSampleCalendar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);           { this is already here }
  ;                                     { many statements setting element values }
  FDate := Date;                       { get current date from RTL }
end;
```

- 3 인코드된 날짜에 액세스하도록 하는 런타임 속성을 선언합니다.

날짜를 설정하려면 컨트롤의 온스크린 이미지를 업데이트해야 하기 때문에 날짜를 설정하기 위한 메소드가 필요합니다.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value:TDateTime);
  public
    property CalendarDate:TDateTime read FDate write SetCalendarDate;
    ;
  procedure TSampleCalendar.SetCalendarDate(Value:TDateTime);
  begin
    FDate := Value;                    { set new date value }
    Refresh;                           { update the onscreen image }
  end;
```

일, 월, 연도 액세스

인코드된 숫자로 표현된 날짜는 애플리케이션에는 적합하지만 사람들은 일, 월, 연도 표시를 사용하는 것을 더 선호합니다. 속성을 생성함으로써 인코드로 저장된 요소들에 대하여 대체 액세스를 제공할 수 있습니다.

날짜의 각 요소(일, 월, 연도)가 정수이고 각 요소를 설정하려면 설정 시 날짜를 인코딩해야 하기 때문에 세 개의 속성 모두에서 구현 메소드를 공유하면 번번이 코드를 중복시키는 것을 피할 수 있습니다. 즉, 두 개의 메소드를 작성할 수 있는데 하나는 요소를 읽는 메소드이고 다른 하나는 요소를 쓰는 메소드입니다. 이러한 메소드를 사용하여 세 개의 속성을 모두 만들고 설정할 수 있습니다.

일, 월, 연도에 디자인 타임 액세스를 제공하려면 다음을 수행하십시오.

- 1 세 개의 속성을 선언하고 각 속성에 고유한 인덱스 번호를 할당합니다.

```

type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day:Integer index 3 read GetDateElement write SetDateElement;
    property Month:Integer index 2 read GetDateElement write SetDateElement;
    property Year:Integer index 1 read GetDateElement write SetDateElement;
    :
  end;

```

- 2 구현 메소드를 선언하고 작성하여 인덱스 값마다 다른 요소를 설정합니다.

```

type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index:Integer):Integer;           { note the Index parameter }
    procedure SetDateElement(Index:Integer; Value:Integer);
    :
  public
    function TSampleCalendar.GetDateElement(Index:Integer):Integer;
  var
    AYear, AMonth, ADay:Word;
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                    { break encoded date into elements }
    case Index of
      1: Result := AYear;
      2: Result := AMonth;
      3: Result := ADay;
      else Result := -1;
    end;
  end;

  procedure TSampleCalendar.SetDateElement(Index:Integer; Value:Integer);
  var
    AYear, AMonth, ADay:Word;
  begin
    if Value > 0 then                                         { all elements must be positive }
    begin
      DecodeDate(FDate, AYear, AMonth, ADay);                 { get current date elements }
      case Index of                                          { set new element depending on Index }
        1: AYear := Value;
        2: AMonth := Value;
        3: ADay := Value;
        else Exit;
      end;
      FDate := EncodeDate(AYear, AMonth, ADay);               { encode the modified date }
      Refresh;                                               { update the visible calendar }
    end;
  end;
end;

```

이제 디자인 타임에 Object Inspector를 사용하거나 런타임에 코드를 사용하여 달력의 일, 월, 연도를 설정할 수 있습니다. 물론 셀에 날짜를 쓰기 위한 코드는 아직 추가하지 않았지만 이제 필요한 데이터는 가지고 있습니다.

일(day) 수 생성

달력에 숫자를 넣기 위해서는 여러 가지 고려해야 할 것이 있습니다. 월의 일(day) 수는 어떤 달인지와 해당 연도가 윤년인지에 따라 달라집니다. 또한 월은 월과 연도에 따라 다른 요일에서 시작됩니다. *IsLeapYear* 함수를 사용하여 해당 연도가 윤년인지 여부를 확인합니다. SysUtils unit의 *MonthDays* 배열을 사용하여 월의 일 수를 알아냅니다.

윤년과 월별 일 수에 대한 정보가 있다면 각 날짜가 입력되는 그리드 내에서의 위치를 계산할 수 있습니다. 첫째 주, 첫째 날에 근거하여 계산합니다.

채우려는 각각의 셀에 대해 달 오프셋(month-offset) 숫자가 필요하기 때문에 월이나 연도를 변경할 때마다 계산하고 참조하는 것이 좋습니다. 클래스 필드에 그 값을 저장하여 날짜가 변경될 때마다 그 필드를 업데이트할 수 있습니다.

다음과 같은 방법으로 적절한 셀에 날(day)들을 채웁니다.

- 1 필드 값을 업데이트하는 메소드와 객체에 달 오프셋 필드를 추가합니다.

```

type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;           { storage for the offset }
    :
  protected
    procedure UpdateCalendar; virtual; { property for offset access }
  end;
  :
  procedure TSampleCalendar.UpdateCalendar;
  var
    AYear, AMonth, ADay: Word;
    FirstDate: TDateTime;           { date of the first day of the month }
  begin
    if FDate <> 0 then               { only calculate offset if date is valid }
    begin
      DecodeDate(FDate, AYear, AMonth, ADay); { get elements of date }
      FirstDate := EncodeDate(AYear, AMonth, 1); { date of the first }
      FMonthOffset := 2 - DayOfWeek(FirstDate); { generate the offset into the grid }
    end;
    Refresh;                         { always repaint the control }
  end;

```

- 2 생성자와 날짜가 변경될 때마다 새로운 업데이트 메소드를 호출하는 *SetCalendarDate* 및 *SetDateElement* 메소드와 생성자에 다음 문장을 추가합니다.

```

constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);          { this is already here }
  :                                  { many statements setting element values }
  UpdateCalendar;                   { set proper offset }
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                   { this was already here }
  UpdateCalendar;                   { this previously called Refresh }
end;

```

```

procedure TSampleCalendar.SetDateElement(Index:Integer; Value:Integer);
begin
  :
  FDate := EncodeDate(AYear, AMonth, ADay);           { encode the modified date }
  UpdateCalendar;                                   { this previously called Refresh }
end;
end;

```

- 3 셀의 행과 열 조합을 전달할 때 일 수를 반환하는 메소드를 달력에 추가합니다.

```

function TSampleCalendar.DayNum(ACol, ARow:Integer):Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;     { calculate day for this cell }
  if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1;                                     { return -1 if invalid }
end;

```

*DayNum*의 선언을 컴포넌트의 타입 선언에 추가해야 합니다.

- 4 이제 날짜가 입력되는 위치를 계산할 수 있으므로 *DrawCell*을 업데이트하여 날짜를 입력할 수 있습니다.

```

procedure TCalendar.DrawCell(ACol, ARow:Longint; ARect:TRect; AState:TGridDrawState);
var
  TheText:string;
  TempDay:Integer;
begin
  if ARow = 0 then                                   { if this is the header row ... }
    TheText := ShortDayNames[ACol + 1]                { just use the day name }
  else begin
    TheText := '';                                   { blank cell is the default }
    TempDay := DayNum(ACol, ARow);                    { get number for this cell }
    if TempDay <> -1 then TheText := IntToStr(TempDay); { use the number if valid }
  end;
  with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;

```

달력 컴포넌트를 다시 설치하여 폼에 두면 현재 월에 대한 적절한 정보가 나타납니다.

오늘 날짜 선택

이제 달력 셀에 숫자를 가지게 되었기 때문에 오늘 날짜가 있는 셀로 선택 반전(selection highlighting)을 이동하는 것이 바람직합니다. 기본적으로 선택은 맨 위의 왼쪽 셀에서 시작하므로 달력을 처음 만들 때와 날짜를 변경할 때 *Row* 및 *Column* 속성을 설정해야 합니다.

오늘 날짜에 대한 선택을 설정하려면 *Refresh*를 호출하기 전에 *Row* 및 *Column*을 설정하는 *UpdateCalendar* 메소드를 변경합니다.

```

procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
    begin
      : { existing statements to set FMonthOffset }
    end

```

```

    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
end;
Refresh; { this is already here }
end;

```

날짜를 디코딩하여 이전에 설정된 *ADay* 변수를 재사용하고 있다는 것에 유의하십시오.

월과 연도 탐색

속성들은 특히 디자인 타임에 컴포넌트를 조작할 때 유용합니다. 그러나 때때로 하나 이상의 속성을 포함하는 일반적이거나 자연스러운 조작 타임이 있는 경우 이 조작들을 처리하기 위한 메소드를 제공하는 것이 바람직합니다. 이러한 자연스러운 조작의 한 예로 달력의 "다음 달(next month)" 기능이 있습니다. 월의 랩어라운드 및 연도의 증분 처리는 간단하면서도 컴포넌트를 사용하는 개발자에게는 매우 편리합니다.

일반적인 조작들을 메소드로 캡슐화하는 유일한 단점은 메소드가 런타임에만 사용될 수 있다는 것입니다. 하지만 그러한 조작들은 보통 반복적으로 수행될 때만 귀찮은 작업이 되고 디자인 타임에는 아주 드문 일입니다.

달력에 대해, 다음 월 및 연도, 이전 월 및 연도에 대한 다음의 네 가지 메소드를 추가합니다. 이러한 메소드마다 약간씩 다른 방법으로 *IncMonth* 함수를 사용하여 월 또는 연도의 증분에 의해 *CalendarDate*를 증가시키거나 감소시킵니다. *CalendarDate*를 증가시키거나 감소시킨 후 날짜 값을 디코딩하여 Year, Month 및 Day 속성에 해당하는 새 값을 입력합니다.

```

procedure TCalendar.NextMonth;
begin
    DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;

procedure TCalendar.PrevMonth;
begin
    DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;

procedure TCalendar.NextYear;
begin
    DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;

procedure TCalendar.PrevYear;
begin
    DecodeDate(IncMonth(CalendarDate, -12), Year, Month, Day);
end;

```

클래스 선언에 새 메소드의 선언을 추가해야 합니다.

이제 달력 컴포넌트를 사용하는 애플리케이션을 생성할 때 월 또는 연도를 검색하는 것을 쉽게 구현할 수 있습니다.

일(day) 탐색

주어진 월에서 일(day)을 탐색하는 두 가지 확실한 방법이 있습니다. 한 가지 방법은 화살표 키를 사용하는 것이고 또 다른 방법은 마우스 클릭에 반응하는 것입니다. 표준 그리드 컴포넌트는 두 가지 모두 클릭처럼 처리합니다. 즉, 화살표 이동은 인접한 셀을 클릭하는 것처럼 처리됩니다.

일(day) 탐색 과정은 다음과 같습니다.

- 선택 이동
- OnChange 이벤트 제공
- 빈 셀 배제

선택 이동

그리드의 상속된 동작은 화살표 키나 클릭에 대한 반응으로 선택 이동을 처리하지만 선택된 일(day)을 변경할 경우 그 기본 동작을 수정해야 합니다.

달력에서의 이동을 처리하려면 그리드의 *Click* 메소드를 오버라이드합니다.

사용자 상호 작용과 연결된 *Click*과 같은 메소드를 오버라이드할 때 표준 동작을 잃지 않도록 상속된 메소드에 대한 호출을 거의 항상 포함시킵니다.

다음은 달력 그리드의 오버라이드된 *Click* 메소드입니다. 나중에 **override** 지시어를 포함하여 *TSampleCalendar*에 *Click*의 선언을 추가해야 합니다.

```

procedure TSampleCalendar.Click;
var
    TempDay: Integer;
begin
    inherited Click;           { remember to call the inherited method! }
    TempDay := DayNum(Col, Row); { get the day number for the clicked cell }
    if TempDay <> -1 then Day := TempDay; { change day if valid }
end;

```

OnChange 이벤트 제공

달력 사용자가 달력의 날짜를 변경할 수 있기 때문에 애플리케이션이 이러한 변경 사항에 반응할 수 있도록 하는 것이 바람직합니다.

*TSampleCalendar*에 *OnChange* 이벤트를 추가합니다.

1 이벤트, 이벤트를 저장할 필드 및 이벤트를 호출할 동적 메소드를 선언합니다.

```

type
    TSampleCalendar = class(TCustomGrid)
    private
        FOnChange: TNotifyEvent;
    protected
        procedure Change; dynamic;
    :

```

```

published
  property OnChange:TNotifyEvent read FOnChange write FOnChange;
  :

```

2 *Change* 메소드를 작성합니다.

```

procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;

```

3 *SetCalendarDate* 및 *SetDateElement* 메소드의 끝에 *Change*를 호출하는 문장을 추가합니다.

```

procedure TSampleCalendar.SetCalendarDate(Value:TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change; { this is the only new statement }
end;

procedure TSampleCalendar.SetDateElement(Index:Integer; Value:Integer);
begin
  : { many statements setting element values }
  FDate := EncodeDate(AYear, AMonth, ADay);
  UpdateCalendar;
  Change; { this is new }
end;

```

이제 달력 컴포넌트를 사용하는 애플리케이션은 *OnChange* 이벤트에 핸들러를 첨부함으로써 컴포넌트 날짜의 변경 사항에 반응할 수 있습니다.

빈 셀 배제

달력을 작성할 때 사용자가 빈 셀을 선택할 수 있지만 날짜는 변경되지 않습니다. 그러므로 빈 셀의 선택을 하지 않도록 하는 것이 바람직합니다.

지정한 셀의 선택 가능 여부를 조정하려면 그리드의 *SelectCell* 메소드를 오버라이드합니다.

*SelectCell*은 열과 행을 매개변수로 취하고 지정한 셀이 선택 가능한지 여부를 나타내는 부울 값을 반환합니다.

셀에 유효한 날짜가 없으면 *SelectCell*을 오버라이드하여 *False*를 반환할 수 있습니다.

```

function TSampleCalendar.SelectCell(ACol, ARow:Longint):Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False { -1 indicates invalid date }
  else Result := inherited SelectCell(ACol, ARow); { otherwise, use inherited value }
end;

```

이제 사용자가 빈 셀을 클릭하거나 화살표 키에 의해 빈 셀로 이동하려고 하면 달력에 현재 셀이 선택된 상태로 남겨집니다.

Data-aware 컨트롤 만들기

데이터베이스 연결을 사용할 때 *data-aware* 컨트롤이 있으면 편리합니다. 즉, 애플리케이션에서 컨트롤과 데이터베이스의 일부를 연결할 수 있습니다. Delphi에는 *data-aware* 레이블, 편집 상자, 리스트 박스, 콤보 박스, 조회 컨트롤 및 그리드가 있습니다. 또한 사용자 고유의 *data-aware* 컨트롤을 만들 수도 있습니다. *data-aware* 컨트롤 사용에 대한 자세한 내용은 15장 "데이터 컨트롤 사용"을 참조하십시오.

Data awareness에는 여러 단계가 있습니다. 가장 간단한 단계는 읽기 전용 *data awareness* 또는 *데이터 찾아보기*인데 현재 데이터베이스의 상태를 반영하는 기능입니다. 보다 복잡한 단계는 편집 가능 *data awareness* 또는 *데이터 편집*인데 사용자가 컨트롤을 조작하여 데이터베이스의 값을 편집할 수 있습니다. 데이터베이스 관련 단계는 가장 간단한 경우인 단일 필드와의 연결부터 가장 복잡한 경우인 다중 레코드 컨트롤에 이르기까지 다양하다는 점에 유의합니다.

이 장에서는 먼저 데이터셋의 단일 필드와 연결하는 읽기 전용 컨트롤을 만드는 가장 간단한 방법을 설명합니다. 사용된 특정 컨트롤은 50장 "그리드 사용자 지정"에서 만들어진 *TSampleCalendar* 달력입니다. Component 팔레트의 Samples 페이지에 있는 표준 달력 컨트롤인 *TCalendar*를 사용할 수도 있습니다.

그런 다음 계속해서 이 장에서는 새로운 데이터 찾아보기 컨트롤을 데이터 편집 컨트롤로 만드는 방법을 설명합니다.

데이터 찾아보기 컨트롤 생성

Data-aware 달력 컨트롤 생성은 읽기 전용 컨트롤인지 또는 데이터셋의 원본으로 사용할 데이터를 변경할 수 있는 컨트롤인지 여부에 관계 없이 다음과 같은 단계로 이루어집니다.

- 컴포넌트 생성 및 등록
- 데이터 연결 추가
- 데이터 변경 내용에 응답

컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 다음과 동일한 방법으로 시작됩니다. 유닛을 만들고 컴포넌트 클래스를 파생시켜 파생된 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 40-8 페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음 특성을 사용하여 컴포넌트를 생성하는 일반적인 절차를 따릅니다.

- 컴포넌트 유닛 *DBCAl*을 호출합니다.
- VCL 컴포넌트 *TSampleCalendar*에서 파생된 *TDBCcalendar*라는 새로운 컴포넌트를 파생시킵니다. 50장 "그리드 사용자 지정"에서 *TSampleCalendar* 컴포넌트를 생성하는 방법을 보여 줍니다.
- 컴포넌트 팔레트의 Samples 페이지에 *TDBCcalendar*를 등록합니다.

결과 유닛은 다음과 같습니다.

```
unit DBCal;

interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
     Forms, Grids, Calendar;

type
  TDBCcalendar = class(TSampleCalendar)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TDBCcalendar]);
end;

end.
```

이제 계속해서 새 달력을 데이터 브라우저로 만들 수 있습니다.

읽기 전용 컨트롤 만들기

이 데이터 달력은 데이터에 대해 읽기 전용이므로 읽기 전용 컨트롤이 만들어지며 사용자가 컨트롤 내에 변경을 만들어서 데이터베이스에 그 변경 내용을 반영하지 않게 합니다.

다음과 같은 방법으로 읽기 전용 달력을 만듭니다.

- *ReadOnly* 속성 추가
- 필요한 업데이트 허용

TSampleCalendar 대신 Delphi의 Samples 페이지에 있는 *TCalendar*로 시작했으면 이미 *ReadOnly* 속성을 가지므로 이러한 단계를 생략할 수 있습니다.

ReadOnly 속성 추가

ReadOnly 속성을 추가하면 디자인 타임에 읽기 전용 컨트롤을 만드는 방법을 제공합니다. 읽기 전용 속성이 *True*로 설정되어 있으면 선택 불가능한 컨트롤에 모든 셀을 만들 수 있습니다.

- 1 속성 선언과 값을 유지할 **private** 필드를 추가합니다.

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FReadOnly:Boolean;           { field for internal storage }
  public
    constructor Create(AOwner:TComponent); override; { must override to set default }
  published
    property ReadOnly:Boolean read FReadOnly write FReadOnly default True;
  end;
:
constructor TDBCalendar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor! }
  FReadOnly := True;                 { set the default value }
end;
```

- 2 컨트롤이 읽기 전용인 경우, 선택을 허용하지 않도록 *SelectCell* 메소드를 오버라이드합니다. *SelectCell* 사용에 대해서는 50-12 페이지의 "빈 셀 배제"에서 설명합니다.

```
function TDBCalendar.SelectCell(ACol, ARow:Longint):Boolean;
begin
  if FReadOnly then Result := False           { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow); { otherwise, use inherited method }
end;
```

*SelectCell*의 선언을 *TDBCalendar*의 타입 선언에 추가하고 **override** 지시어를 첨부해야 합니다.

이제 달력을 폼에 추가하는 경우 컴포넌트는 클릭과 키 입력을 무시합니다. 또한 날짜 변경 시 선택 위치를 업데이트하지 않습니다.

필요한 업데이트 허용

읽기 전용 달력은 *Row* 및 *Col* 속성을 포함하여 모든 종류의 변경에 대해 *SelectCell* 메소드를 사용합니다. *UpdateCalendar* 메소드는 날짜가 변경될 때마다 *Row* 및 *Col*을 설정하지만 *SelectCell*이 변경을 허용하지 않기 때문에 날짜가 변경되더라도 선택은 그대로 유지됩니다.

변경에 대한 절대적 금지를 피하려면 내부 부울 플래그를 달력에 추가하고 그 플래그가 *True*로 설정되었을 때 변경을 허용하도록 합니다.

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FUpdating:Boolean;           { private flag for internal use }
  protected
```

```

    function SelectCell(ACol, ARow :Longint) :Boolean; override;
    public
    procedure UpdateCalendar; override;           { remember the override directive }
    end;
    :
    function TDBCcalendar.SelectCell(ACol, ARow:Longint):Boolean;
    begin
    if (not FUpdating) and FReadOnly then Result := False { allow select if updating }
    else Result := inherited SelectCell(ACol, ARow); { otherwise, use inherited method }
    end;

    procedure TDBCcalendar.UpdateCalendar;
    begin
    FUpdating := True;                               { set flag to allow updates }
    try
    inherited UpdateCalendar;                       { update as usual }
    finally
    FUpdating := False;                             { always clear the flag }
    end;
    end;

```

달력은 여전히 사용자 변경을 허용하지 않지만 이제 날짜 속성을 변경하여 해당 날짜에서 만들어진 변경을 제대로 반영합니다. 읽기 전용 달력 컨트롤이 있으므로 데이터 찾아보기 기능을 추가할 수 있습니다.

데이터 연결 추가

컨트롤과 데이터베이스 사이의 연결은 *data link*라는 클래스에 의해 처리됩니다. 데이터베이스에서 컨트롤을 단일 필드에 연결하는 데이터 연결 클래스는 *TFieldDataLink* (VCL 또는 CLX)입니다. 전체 테이블에 대한 데이터 연결도 있습니다.

data-aware 컨트롤은 데이터 연결 클래스를 소유(*own*)합니다. 즉, 컨트롤이 데이터 연결을 생성하고 소멸하는 것을 담당합니다. 소유된 클래스의 관리에 대한 자세한 내용은 49장 "그래픽 컴포넌트 생성"을 참조하십시오.

데이터 연결을 소유된 클래스로 만드는 데에는 다음 세 단계가 필요합니다.

- 1 클래스 필드 선언
- 2 액세스 속성 선언
- 3 데이터 연결 초기화

클래스 필드선언

49-5 페이지의 "클래스 필드 선언"에서 설명된 대로 컴포넌트는 자신의 소유된 클래스 각각에 대해 필드를 필요로 합니다. 이 경우 달력은 그 데이터 연결에 대해 *TFieldDataLink* 타입의 필드를 필요로 합니다.

달력의 데이터 연결에 대한 필드를 선언합니다.

```

type
    TDBCcalendar = class(TSampleCalendar)
    private

```

```

    FDataLink:TFieldDataLink;
  :
end;

```

애플리케이션을 컴파일하려면 DB 및 DBCtrls를 유닛의 **uses** 절에 추가해야 합니다.

액세스 속성 선언

모든 data-aware 컨트롤에는 데이터를 컨트롤에 제공하는 애플리케이션의 데이터 소스 클래스를 지정하는 *DataSource* 속성이 있습니다. 또한 단일 필드를 액세스하는 컨트롤은 데이터 소스에서 그 필드를 지정하기 위해서 *DataField* 속성이 필요합니다.

49장 "그래픽 컴포넌트 생성"의 예에 있는 소유된 클래스에 대한 액세스 속성과 달리 이러한 액세스 속성들은 소유된 클래스 자체에 액세스를 제공하지는 않지만 소유된 클래스의 해당 속성에 액세스를 제공합니다. 즉, 컨트롤과 그 데이터 연결이 동일한 데이터 소스와 필드를 공유할 수 있도록 하는 속성을 만듭니다.

DataSource 및 *DataField* 속성과 그 구현 메소드를 선언한 다음 그 메소드를 데이터 연결 클래스의 해당 속성에 대한 "전달(pass-through)" 메소드로 작성합니다.

액세스 속성 선언의 예제

```

type
  TDBCcalendar = class(TSampleCalendar)
  private
    ...
    { implementation methods are private }
  ...
  function GetDataField: string;           { returns the name of the data field }
  function GetDataSource: TDataSource;     { returns reference to the data source }
  procedure SetDataField(const Value: string); { assigns name of data field }
  procedure SetDataSource(Value:TDataSource); { assigns new data source }
  published
    { make properties available at design time }
  property DataField: string read GetDataField write SetDataField;
  property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
:
function TDBCcalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCcalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCcalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCcalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;

```

달력과 데이터 연결 사이에 연결이 이루어졌으므로 한 가지 중요한 단계가 더 있습니다. 달력 컨트롤이 생성되었을 때 데이터 연결 클래스를 생성하고 달력을 소멸하기 전에 데이터 연결을 소멸해야 합니다.

데이터 연결 초기화

Data-aware 컨트롤은 그 데이터 연결에 대한 액세스가 필요하므로 데이터 연결 객체를 그 자신의 생성자 일부로 생성하고 자신이 소멸되기 전에 데이터 연결 객체를 소멸해야 합니다.

달력의 *Create* 및 *Destroy* 메소드를 오버라이드하여 데이터 연결 객체를 각각 생성하고 소멸합니다.

```

type
  TDBCcalendar = class(TSampleCalendar)
  public                                { constructors and destructors are always public }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    :
  end;
:
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                { always call the inherited constructor first }
  FDataLink := TFieldDataLink.Create;      { construct the datalink object }
  FDataLink.Control := self;               {let the datalink know about the calendar }
  FReadOnly := True;                       { this is already here }
end;

destructor TDBCcalendar.Destroy;
begin
  FDataLink.Free;                          { always destroy owned objects first... }
  inherited Destroy;                       { ...then call inherited destructor }
end;

```

이제 완벽한 데이터 연결이 갖추어졌지만 연결된 필드에서 읽어야 하는 데이터를 컨트롤에 아직 알리지 않았습니다. 다음 단원에서 이 작업을 수행하는 방법을 설명합니다.

데이터 변경 내용에 응답

컨트롤에 데이터 소스와 데이터 필드를 지정할 데이터 연결 및 속성을 갖추면 다른 레코드로의 이동이나 그 필드의 변경에 기인하여 그 필드에서 데이터 변경에 응답해야 합니다.

데이터 연결에는 모두 *OnChange*라는 이벤트가 있습니다. 데이터 소스가 데이터의 변경을 표시하면 데이터 연결 객체는 *OnChange* 이벤트에 첨부된 이벤트 핸들러를 호출합니다.

데이터 변경에 대한 응답으로 컨트롤을 업데이트하기 위해 데이터 연결의 *OnChange* 이벤트에 핸들러를 첨부합니다.

이 경우 메소드를 달력에 추가한 다음 그 메소드를 데이터 연결의 *OnDataChange*에 대한 핸들러로 지정합니다.

DataChange 메소드를 선언하고 구현한 다음 그 메소드를 생성자에서 데이터 연결의 *OnDataChange* 이벤트에 할당합니다. 소멸자에서 객체를 소멸하기 전에 *OnDataChange* 핸들러를 분리합니다.

```

type
  TDBCalendar = class(TSampleCalendar)
  private { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject); { must have proper parameters for event }
  end;
:
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor first }
  FReadOnly := True;                  { this is already here }
  FDataLink := TFieldDataLink.Create; { construct the datalink object }
  FDataLink.OnDataChange := DataChange; { attach handler to event }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.OnDataChange := nil;      { detach handler before destroying object }
  FDataLink.Free;                    { always destroy owned objects first... }
  inherited Destroy;                 { ...then call inherited destructor }
end;

procedure TDBCalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then      { if there is no field assigned... }
    CalendarDate := 0                { ...set to invalid date }
  else CalendarDate := FDataLink.Field.AsDateTime; { otherwise, set calendar to the date }
end;

```

이제 데이터 찾아보기 컨트롤이 만들어졌습니다.

데이터 편집 컨트롤 생성

데이터 편집 컨트롤을 만들 때 컴포넌트를 생성하고 등록한 다음 데이터 찾아보기 컨트롤에서 행한 대로 데이터 연결을 추가합니다. 또한 비슷한 방법으로 원본으로 사용한 필드의 데이터 변경 내용에 응답하며 몇 가지 문제를 더 처리해야 합니다.

예를 들어, 컨트롤이 키와 마우스 이벤트에 모두 응답하도록 하고자 합니다. 컨트롤은 사용자가 컨트롤의 내용을 변경할 때 응답해야 합니다. 사용자가 컨트롤을 종료할 때 컨트롤의 변경 내용이 데이터셋에 반영되어야 합니다.

여기에서 설명되는 데이터 편집 컨트롤은 이 장의 첫 번째 부분에 설명된 컨트롤과 동일한 달력 컨트롤입니다. 컨트롤은 연결 필드에서 데이터를 보고 편집할 수 있도록 수정됩니다.

기존 컨트롤을 수정하여 데이터 편집 컨트롤로 만들려면 다음 단계가 필요합니다.

- FReadOnly의 기본값 변경
- 마우스 다운 및 키 다운 메시지 처리
- 필드 데이터 연결 클래스 업데이트
- Change 메소드 수정
- 데이터셋 업데이트

FReadOnly의 기본값 변경

이것은 데이터 편집 컨트롤이므로 *ReadOnly* 속성은 *False*로 기본 설정되어야 합니다. *ReadOnly* 속성을 *False*로 만들려면 생성자에서 *FReadOnly*의 값을 변경합니다.

```

constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  :
  FReadOnly := False; { set the default value }
  :
end;
    
```

마우스 다운 및 키 다운 메시지 처리

컨트롤의 사용자가 컨트롤과 상호 작용을 하기 시작하면 컨트롤은 Windows로부터 마우스 다운 메시지 (*WM_LBUTTONDOWN*, *WM_MBUTTONDOWN* 또는 *WM_RBUTTONDOWN*) 또는 키 다운 메시지 (*WM_KEYDOWN*)를 받습니다. CLX를 사용하는 경우, 공지는 시스템 이벤트의 형식으로 운영 체제로부터 보내집니다. 컨트롤이 이러한 메시지에 응답하도록 하려면 이러한 메시지에 응답하는 핸들러를 작성해야 합니다.

- 마우스 다운 메시지에 응답
- 키 다운 메시지에 응답

마우스 다운 메시지에 응답

MouseDown 메소드는 컨트롤의 *OnMouseDown* 이벤트에 대한 protected 메소드입니다. 컨트롤 자체에서 *MouseDown*을 호출하여 Windows 마우스 다운 메시지에 응답합니다. 상속된 *MouseDown* 메소드를 오버라이드할 때 *OnMouseDown* 이벤트를 호출하는 것은 물론 기타 응답을 제공하는 코드를 포함시킬 수 있습니다.

*MouseDown*을 오버라이드하려면 *MouseDown* 메소드를 *TDBCcalendar* 클래스에 추가합니다.

```

type
  TDBCcalendar = class(TSampleCalendar);
  :
  protected
  procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
    override;
  :
  end;

procedure TDBCcalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
    
```

```

var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
    begin
      MyMouseDown := OnMouseDown;
      if Assigned(MyMouseDown) then MyMouseDown(Self, Button, Shift, X, Y);
    end;
  end;
end;

```

*MouseDown*이 마우스 다운 메시지에 응답할 때 상속된 *MouseDown* 메소드는 컨트롤의 *ReadOnly* 속성이 *False*이고 데이터 연결 객체가 필드를 편집할 수 있는 편집 모드에 있는 경우에만 호출됩니다. 필드를 편집할 수 없는 경우, 프로그래머가 *OnMouseDown* 이벤트 핸들러에 입력한 코드가 실행됩니다.

키 다운 메시지에 응답

KeyDown 메소드는 컨트롤의 *OnKeyDown* 이벤트에 대한 protected 메소드입니다. 컨트롤 자체에서 *KeyDown*을 호출하여 Windows 키 다운 메시지에 응답합니다. 상속된 *KeyDown* 메소드를 오버라이드할 때 *OnKeyDown* 이벤트를 호출하는 것은 물론 기타 응답을 제공하는 코드를 포함시킬 수 있습니다.

*KeyDown*을 오버라이드하려면 다음 단계를 따릅니다.

- 1 *KeyDown* 메소드를 *TDBCcalendar* 클래스에 추가합니다.

```

type
  TDBCcalendar = class(TSampleCalendar);
  ..
protected
  procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
    override;
  ..
end;

```

- 2 *KeyDown* 메소드를 구현합니다.

```

procedure KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, VK_END,
    VK_HOME, VK_PRIOR, VK_NEXT]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
    begin
      MyKeyDown := OnKeyDown;
      if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
    end;
  end;
end;

```

*KeyDown*이 마우스 다운 메시지에 응답할 때 상속된 *KeyDown* 메소드는 컨트롤의 *ReadOnly* 속성이 *False*이고 선택된 키가 커서 컨트롤 키 중 하나이며 데이터 연결 객체가 필드를 편집할 수 있는 편집 모드에 있는 경우에만 호출됩니다. 필드를 편집할 수 없거나 다른 키가 선택된 경우, 프로그래머가 *OnKeyDown* 이벤트 핸들러에 입력한 코드가 실행됩니다.

필드 데이터 연결 클래스 업데이트

데이터 변경에는 두 가지 유형이 있습니다.

- Data-aware 컨트롤에 반영되어야 하는 필드 값 변경
- 필드 값에 반영되어야 하는 data-aware 컨트롤 변경

TDBCcalendar 컴포넌트에는 *CalendarDate* 속성에 값을 할당하여 데이터셋의 필드 값 변경을 처리하는 *DataChange* 메소드가 이미 있습니다. *DataChange* 메소드는 *OnDataChange* 이벤트의 핸들러입니다. 그러므로 달력 컴포넌트는 데이터 변경의 첫 번째 유형을 처리할 수 있습니다.

이와 유사하게 필드 데이터 연결 클래스에는 또한 컨트롤 사용자가 data-aware 컨트롤의 내용을 수정할 때 발생하는 *OnUpdateData* 이벤트가 있습니다. 달력 컨트롤에는 *OnUpdateData* 이벤트에 대한 이벤트 핸들러가 되는 *UpdateData* 메소드가 있습니다. *UpdateData*는 data-aware 컨트롤의 변경된 값을 필드 데이터 연결에 할당합니다.

- 1 필드 값에 달력의 값 변경을 반영하려면 *UpdateData* 메소드를 달력 컴포넌트의 private 섹션에 추가합니다.

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
    ;
  end;
```

- 2 *UpdateData* 메소드를 구현합니다.

```
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;    { set field link to calendar date }
end;
```

- 3 *TDBCcalendar*의 생성자에서 *UpdateData* 메소드를 *OnUpdateData* 이벤트에 할당합니다.

```
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

Change 메소드 수정

*TDBCcalendar*의 *Change* 메소드는 새로운 날짜 값이 설정될 때마다 호출됩니다. *Change*는 *OnChange* 이벤트 핸들러를 호출합니다. 컴포넌트 사용자는 *OnChange* 이벤트 핸들러에 코드를 작성하여 날짜 변경에 응답할 수 있습니다.

달력의 날짜가 변경되면 원본으로 사용한 데이터셋에 변경이 발생했음을 공지해야 합니다. *Change* 메소드를 오버라이드하고 코드를 한 줄 더 추가하여 이 작업을 수행할 수 있습니다. 다음과 같은 단계를 따릅니다.

- 1 새로운 *Change* 메소드를 *TDBCcalendar* 컴포넌트에 추가합니다.

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure Change; override;
    :
  end;
```

- 2 데이터가 변경된 데이터셋을 공지하는 *Modified* 메소드를 호출하여 *Change* 메소드를 작성한 다음 상속된 *Change* 메소드를 호출합니다.

```
procedure TDBCcalendar.Change;
begin
  FDataLink.Modified;           { call the Modified method }
  inherited Change;             { call the inherited Change method }
end;
```

데이터셋 업데이트

지금까지 data-aware 컨트롤 내의 변경은 필드 데이터 연결 클래스의 값을 변경시켰습니다. 데이터 편집 컨트롤 생성의 최종 단계는 새 값을 갖는 데이터셋을 업데이트하는 것입니다. 이것은 data-aware 컨트롤의 값을 변경한 사람이 컨트롤 외부로 클릭하거나 *Tab* 키를 눌러 컨트롤을 종료한 후 발생합니다. 이러한 프로세스는 VCL과 CLX에서 다르게 동작합니다.

VCL VCL은 컨트롤의 연산에 대해 정의된 메시지 컨트롤 ID를 갖습니다. 예를 들어, 사용자가 컨트롤에서 빠져 나갈 때 *CM_EXIT* 메시지가 컨트롤에 보내집니다. 사용자는 메시지 핸들러를 작성하여 이 메시지에 응답할 수 있습니다. 이 경우 사용자가 컨트롤에서 빠져 나가면 *CM_EXIT*의 메시지 핸들러인 *CMExit* 메소드는 필드 데이터 연결 클래스의 변경된 값으로 데이터셋의 레코드를 업데이트하여 응답합니다. 메시지 핸들러에 대한 더 자세한 내용은 46장 "메시지 처리"를 참조하십시오.

메시지 핸들러의 데이터셋을 업데이트하려면 다음 단계를 따릅니다.

- 1 *TDBCcalendar* 컴포넌트에 메시지 핸들러를 추가합니다.

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
    :
  end;
```

2 다음과 같이 *CMExit* 메소드를 구현합니다.

```
procedure TDBCcalendar.CMExit(var Message:TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;           { tell data link to update database }
  except
    on Exception do SetFocus;       { if it failed, don't let focus leave }
  end;
  inherited;
end;
```

CLX CLX 에서 *TWidgetControl*에는 컨트롤에서 입력 포커스가 벗어났을 때 호출되는 protected *DoExit* 메소드가 있습니다. 이 메소드는 *OnExit* 이벤트에 대한 이벤트 핸들러를 호출합니다. 이 메소드를 오버라이드하여 *OnExit* 이벤트 핸들러를 생성하기 전에 데이터셋의 레코드를 업데이트할 수 있습니다.

사용자가 컨트롤을 종료할 때 데이터셋을 업데이트하려면 다음 단계를 따릅니다.

1 *DoExit* 메소드에 대한 오버라이드를 *TDBCcalendar* 컴포넌트에 추가합니다.

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
    :
  end;
```

2 다음과 같이 *DoExit* 메소드를 구현합니다.

```
procedure TDBCcalendar.CMExit(var Message:TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;           { tell data link to update database }
  except
    on Exception do SetFocus;       { if it failed, don't let focus leave }
  end;
  inherited;                         { let the inherited method generate an OnExit event }
end;
```

52

대화 상자를 컴포넌트로 만들기

자주 사용되는 대화 상자를 컴포넌트로 만들어 컴포넌트 팔레트에 추가하여 사용하면 편리합니다. 사용자의 대화 상자 컴포넌트는 표준 공통 대화 상자를 나타내는 컴포넌트 처럼 동작합니다. 이 장의 목표는 사용자가 프로젝트에 추가하고 디자인 타임 시 속성을 설정할 수 있는 간단한 컴포넌트를 만들어 보는 것입니다.

다음 단계에 따라 대화 상자를 컴포넌트로 만듭니다.

- 1 컴포넌트 인터페이스 정의
- 2 컴포넌트 생성 및 등록
- 3 컴포넌트 인터페이스 생성
- 4 컴포넌트 테스트

대화 상자에 연결된 Delphi의 "랩퍼(wrapper)" 컴포넌트는 사용자가 지정한 데이터를 전달하는 대화 상자를 런타임 시 생성하고 실행합니다. 그러므로 대화 상자 컴포넌트는 재사용할 수 있고 사용자 지정할 수 있습니다.

이 장에서는 Delphi Object Repository에 제공된 일반 About Box 폼을 둘러싸는 래퍼 컴포넌트를 만드는 방법을 설명합니다.

참고 ABOUT.PAS와 ABOUT.DFM 파일을 사용자가 작업하는 디렉토리에 복사합니다.

컴포넌트에 둘러싸이는 대화 상자를 디자인할 때 특별히 고려해야 할 사항은 많지 않습니다. 거의 모든 폼이 대화 상자로 작동할 수 있습니다.

컴포넌트 인터페이스 정의

대화 상자의 컴포넌트를 만들기 전에 개발자가 어떻게 사용하도록 할지 결정해야 합니다. 대화 상자과 그 대화 상자를 사용하는 애플리케이션 사이에 인터페이스를 만듭니다.

예를 들어, 공통 대화 상자 컴포넌트의 속성을 봅니다. 개발자는 이 속성으로 캡션이나 초기 컨트롤 설정과 같은 대화 상자의 초기 상태를 설정한 다음 대화 상자가 닫힌 후에 필요

한 정보를 다시 읽을 수 있습니다. 래퍼 컴포넌트의 속성과 마찬가지로 이 속성도 대화 상자의 각 컨트롤과 직접 상호 작용하지 않습니다.

따라서 개발자가 애플리케이션에서 요구하는 정보를 지정하고 반환하는 방식으로 대화 상자 폼이 나타낼 수 있도록 인터페이스에는 충분한 정보가 들어 있어야 합니다. 래퍼 컴포넌트의 속성을 일시적인 대화 상자의 영구적인 데이터로 생각할 수 있습니다.

About 상자의 경우에는 정보를 반환할 필요가 없으므로 래퍼의 속성에 About 상자를 적절히 표시하는 데 필요한 정보만 있으면 됩니다. About 상자에는 애플리케이션이 영향을 미치는 개별 필드가 네 개 있기 때문에 해당하는 네 개의 문자열 타입 속성을 제공해야 합니다.

컴포넌트 생성 및 등록

컴포넌트의 생성은 모두 동일한 방법으로 시작됩니다. 즉, 유닛을 만들고 컴포넌트 클래스를 파생시켜 파생된 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 40-8 페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음 특성을 사용하여 컴포넌트를 생성하는 일반적인 절차를 따릅니다.

- 컴포넌트 유닛의 이름을 *AboutDlg*라고 합니다.
- *TComponent*의 자손인 새 컴포넌트 타입 *TAboutBoxDlg*를 파생시킵니다.
- 컴포넌트 팔레트의 Samples 페이지에 *TAboutBoxDlg*를 등록합니다.

결과 유닛은 다음과 같습니다.

```
unit AboutDlg;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TAboutBoxDlg = class(TComponent)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TAboutBoxDlg]);
end;
end.
```

이제 새 컴포넌트는 *TComponent*에 기본 제공된 기능만 가지게 되었습니다. 이것은 가장 간단한 논비주얼(nonvisual) 컴포넌트입니다. 다음 단원에서는 컴포넌트와 대화 상자 사이에 인터페이스를 만듭니다.

컴포넌트 인터페이스 생성

컴포넌트 인터페이스를 만드는 단계는 다음과 같습니다.

- 1 폼 유닛 포함
- 2 인터페이스 속성 추가
- 3 Execute 메소드 추가

폼 유닛 포함

래퍼 컴포넌트가 둘러싸인 대화 상자를 초기화하고 표시하는 경우 래퍼 컴포넌트 유닛의 **uses** 절에 폼의 유닛을 추가해야 합니다.

AboutDlg 유닛의 **uses** 절에 *About*을 추가합니다.

그러면 **uses** 절은 다음과 같이 나타납니다.

```
uses
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,
    About;
```

폼 유닛은 항상 폼 클래스의 인스턴스를 선언합니다. *About* 상자의 경우에 폼 클래스는 *TAboutBox*이며 *About* 유닛에는 다음과 같은 선언이 포함됩니다.

```
var
    AboutBox: TAboutBox;
```

따라서 **uses** 절에 *About*을 추가하면 *AboutBox*를 래퍼 컴포넌트에 사용할 수 있습니다.

인터페이스 속성 추가

계속 진행하기 전에 개발자가 애플리케이션에서 대화 상자를 컴포넌트로 사용할 수 있도록 하기 위해 래퍼가 필요로 하는 속성을 결정합니다. 그런 다음 해당 속성 선언을 컴포넌트의 클래스 선언에 추가할 수 있습니다.

래퍼 컴포넌트의 속성은 보통의 컴포넌트를 작성할 때 만드는 속성보다 더 간단한 편입니다. 이 경우 기억해야 할 것은 래퍼가 대화 상자에 자유롭게 전달할 수 있는 영구적인 일부 데이터를 생성한다는 것입니다. 그 데이터를 속성 폼에 넣으면 디자인 타임 시 개발자가 데이터를 설정할 수 있고 런타임 시 래퍼가 대화 상자에 전달할 수 있습니다.

인터페이스 속성 선언은 컴포넌트의 클래스 선언에 다음 두 가지를 추가해야 합니다.

- 속성 값을 저장하기 위해 래퍼가 사용하는 변수인 `private` 클래스 필드
- 속성의 이름을 지정하고 저장을 위해 사용할 필드를 나타내는 `published` 속성 선언 자체

이러한 종류의 인터페이스 속성은 액세스 메소드를 필요로 하지 않고 저장된 데이터를 직접 액세스합니다. 규칙에 따라 속성의 값을 저장하는 클래스 필드는 속성과 동일한 이름을 갖지만 앞에 *F*가 붙습니다. 필드와 속성이 동일한 타입이어야 합니다.

예를 들어, *Year*라는 이름의 정수 타입 인터페이스 속성을 선언하려면 다음과 같이 선언합니다.

```
type
  TMyWrapper = class(TComponent)
  private
    FYear: Integer;           { field to hold the Year-property data }
  published
    property Year: Integer read FYear write FYear; { property matched with storage }
  end;
```

이 About 상자의 경우 제품 이름, 버전 정보, 저작권 정보, 주석에 대해 각각 하나씩 네 개의 문자열 타입 속성이 필요합니다.

```
type
  TAboutBoxDlg = class(TComponent)
  private
    FProductName, FVersion, FCopyright, FComments: string; { declare fields }
  published
    property ProductName: string read FProductName write FProductName;
    property Version: string read FVersion write FVersion;
    property Copyright: string read FCopyright write FCopyright;
    property Comments: string read FComments write FComments;
  end;
```

컴포넌트 팔레트에 컴포넌트를 설치하고 폼에 컴포넌트를 놓을 때 속성을 설정할 수 있고 해당 값은 폼과 함께 자동으로 유지될 것입니다. 그런 다음 랩퍼는 둘러싸인 대화 상자를 실행할 때 해당 값을 사용할 수 있습니다.

Execute 메소드 추가

컴포넌트 인터페이스의 마지막 부분은 대화 상자를 열고 그 대화 상자가 닫힐 때 결과를 반환하는 방식입니다. 공통 대화 상자(common-dialog-box) 컴포넌트를 사용할 때처럼 사용자가 OK를 클릭하면 *True*를 반환하고 사용자가 대화 상자를 취소하면 *False*를 반환하는 부울 함수 *Execute*를 사용합니다.

Execute 메소드의 선언은 항상 다음과 같습니다.

```
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
```

*Execute*의 최소한의 구현에서는 대화 상자 폼을 생성하고, 모달(modal) 대화 상자로 표시하고, *ShowModal*의 반환 값에 따라 *True* 또는 *False*를 반환해야 합니다.

다음은 *TMyDialogBox* 타입의 대화 상자 폼에 대한 최소한의 *Execute* 메소드입니다.

```
function TMyWrapper.Execute: Boolean;
begin
  DialogBox := TMyDialogBox.Create(Application);           { construct the form }
  try
    Result := (DialogBox.ShowModal = IDOK); { execute; set result based on how closed }
```

```

finally
    DialogBox.Free;                                { dispose of the form }
end;
end;

```

예외가 발생할 경우에도 애플리케이션이 대화 상자 객체를 처리하도록 **try..finally** 블록을 사용합니다. 일반적으로 이와 같이 객체를 생성할 때마다 **try..finally** 블록을 사용하여 코드 블록을 보호하고 애플리케이션이 할당된 리소스를 해제하도록 해야 합니다.

실제로는 **try..finally** 블록 내에 코드가 더 많이 있게 됩니다. 특히 *ShowModal*을 호출하기 전에 랩퍼는 랩퍼 컴포넌트의 인터페이스 속성에 기반하여 대화 상자의 속성 일부를 설정하고, *ShowModal*이 반환하면 대화 상자 실행 결과에 기반하여 인터페이스 속성 일부를 설정할 것입니다.

About 상자의 경우 랩퍼 컴포넌트의 네 가지 인터페이스 속성을 사용하여 About 상자 폼에 있는 레이블의 내용을 설정해야 합니다. About 상자는 애플리케이션에 정보를 반환하지 않기 때문에 *ShowModal*을 호출한 후 아무 것도 할 필요가 없습니다. About 상자 랩퍼의 *Execute* 메소드를 다음과 같이 작성합니다.

TAboutDlg 클래스의 public 부분에 *Execute* 메소드의 선언을 추가합니다.

```

type
    TAboutDlg = class(TComponent)
public
    function Execute: Boolean;
end;

function TAboutBoxDlg.Execute: Boolean;
begin
    AboutBox := TAboutBox.Create(Application);           { construct About box }
    try
        if ProductName = '' then                       { if product name's left blank... }
            ProductName := Application.Title;           { ...use application title instead }
        AboutBox.ProductName.Caption := ProductName;   { copy product name }
        AboutBox.Version.Caption := Version;           { copy version info }
        AboutBox.Copyright.Caption := Copyright;       { copy copyright info }
        AboutBox.Comments.Caption := Comments;         { copy comments }
        AboutBox.Caption := 'About ' + ProductName;    { set About-box caption }
        with AboutBox do begin
            ProgramIcon.Picture.Graphic := Application.Icon;           { copy icon }
            Result := (ShowModal = IDOK);                               { execute and set result }
        end;
    finally
        AboutBox.Free;                                               { dispose of About box }
    end;
end;

```

컴포넌트 테스트

대화 상자 컴포넌트를 설치했으면 폼에 놓고 실행하여 공통 대화 상자의 컴포넌트처럼 사용할 수 있습니다. About 상자를 빠르게 테스트하는 방법은 폼에 명령 버튼을 추가하고 사용자가 버튼을 클릭하면 대화 상자를 실행하게 하는 것입니다.

예를 들어, About 대화 상자를 만들고 그 대화 상자를 컴포넌트로 만들어서 컴포넌트 팔레트에 추가한 후 다음과 같이 테스트할 수 있습니다.

- 1 새 프로젝트를 만듭니다.
- 2 About 상자(About-box) 컴포넌트를 메인 폼에 놓습니다.
- 3 폼에 명령 버튼을 놓습니다.
- 4 명령 버튼을 더블 클릭하여 비어 있는 클릭 이벤트 핸들러를 생성합니다.
- 5 클릭 이벤트 핸들러에 다음 코드를 입력합니다.

```
AboutBoxDlg1.Execute;
```

- 6 애플리케이션을 실행합니다.

메인 폼이 나타나면 명령 버튼을 클릭합니다. 기본 프로젝트 아이콘과 Project1 이름을 갖는 About 상자가 나타납니다. OK를 선택하여 대화 상자를 닫습니다.

About 상자 컴포넌트의 다양한 속성을 설정하고 애플리케이션을 다시 실행하여 컴포넌트를 더 자세히 테스트할 수 있습니다.

색인

기호

& (앰퍼샌드) 문자 3-20, 6-34
...(생략) 버튼 15-21

숫자

2 계층 애플리케이션 14-3, 14-9, 14-12
2 단계 커밋 25-19
3 계층 애플리케이션 다계층 애플리케이션

가

가변

사용자 지정 4-27 ~ 4-39

가상

메소드 41-8, 44-3

속성 42-2

속성 편집기 47-8 ~ 47-9

메소드 테이블 41-8

지시어 41-8

가속키 3-20, 6-34

가시성(visibility) 3-9

값 42-2

기본 데이터 15-10

기본 속성 42-7, 42-11 ~ 42-12

재정의 48-2, 48-3

부울 42-2, 42-12, 51-3

테스트 42-7

개발자 지원 1-3

객체 3-1, 3-5 ~ 3-12, 4-1

년비주얼 3-11

드래그 앤 드롭 7-1

사용자 지정 3-8

상속 3-8 ~ 3-12

생성 3-11

소멸 3-11

소유된 49-5 ~ 49-8

초기화 49-7

속성 3-5

스크립트 29-11

액세스 3-8 ~ 3-9

여러 인스턴스 3-6

이벤트 3-7

인스턴스화 3-6, 43-2

입시 45-6

정의 3-5

타입 선언 3-10

COM 객체 참조

helper 3-48

TObject 3-14

객체 변수 3-10

객체 생성자 10-13

객체 잠금

스레드 9-7

호출 중첩 9-7

객체 지향 프로그래밍 3-4 ~

3-12, 41-1 ~ 41-9

상속 3-8

선언 41-3, 41-9

메소드 41-7, 41-8,

41-9

클래스 41-5, 41-6

정의 3-4

객체 지향 프로그래밍

(OOP) 3-1

객체 컨텍스트 39-4

ASP 37-3

트랜잭션 39-9

객체 포인터 참조 41-9

객체 풀링 39-8

사용 불가능 39-8

원격 데이터 모듈 25-8

객체 필드 19-22 ~ 19-28

타입 19-22

검색 경로 10-16

검색 경로 구분자 10-16

검색 목록(도움말

시스템) 47-5

결과 매개변수 18-51

경계를 이루는 사각형 8-11

경로(URL) 27-3

경로명 10-16

계산된 필드 18-22 ~ 18-23,

19-6

값 할당 19-8

정의 19-7 ~ 19-8

조회 필드 및 19-9

클라이언트 데이터셋

23-11

계층 25-1

계층 구조(클래스) 41-3

공급 24-1, 25-4

공유 객체 10-22

공유 객체 파일 10-9, 10-15

공유 속성 그룹 39-6

공지 이벤트 43-7

공통 대화 상자 3-47, 52-1

생성 52-2

실행 52-4

관계형 데이터베이스 14-1

구분 표시 12-10

구분자 표시줄(메뉴) 6-34

구성 파일 10-15

국제적인 애플리케이션

약자 12-9

지역화 12-13

키보드 입력 변환 12-9

그래픽 45-1 ~ 45-8

객체 형식 8-3

교체 8-20

국제화 12-10

그리기와 색칠 비교 8-4

독립적인 45-3

드로잉 툴 45-1, 45-7,

45-8, 49-5

변경 49-7

로드 8-19, 45-4

메소드 45-3, 45-4, 45-6

이미지 복사 45-7

팔레트 45-5

문자열에 연결 3-54

복사 8-22

복잡 45-6

붙여넣기 8-23

삭제 8-22

선 그리기 8-5, 8-10 ~

8-11, 8-27 ~ 8-29

이벤트 핸들러 8-26

펜 너비 변경 8-6

양단 묶음 예제 8-23 ~

8-29

이미지 다시 그리기 45-7

이미지 변경 8-20

저장 8-20, 45-4

컨테이너 45-4

컨트롤 추가 8-17

크기 조정 8-20, 15-10,

45-7

파일 8-19 ~ 8-21

파일 형식 8-3

표시 3-46

프로그래밍 개요 8-1 ~

8-3

프레임 6-15

함수, 호출 45-1

HTML에 추가 28-14

owner-draw 컨트롤 7-12

그래픽 객체

스프레드 9-5
 그래픽 상자 15-2
 그래픽 컨트롤 40-4, 45-4,
 49-1 ~ 49-10
 그리기 49-2 ~ 49-10
 비트맵과 비교 49-3
 생성 40-4, 49-2
 시스템 리소스 절감 40-4
 이벤트 45-7
 그룹 상자 3-41
 그룹화 수준 23-10
 유지 관리되는 집계 23-13
 그리기 그리드 3-44
 그리기 모드 8-29
 그리기 상자 3-47
 그리드 3-44 ~ 3-45, 15-2,
 50-1, 50-2, 50-5, 50-11
 값 얻기 15-17
 그리기 15-26
 기본 상태 15-16
 복원 15-21
 데이터 편집 15-6, 15-25
 데이터 표시 15-15,
 15-17, 15-27
 런타임 옵션 15-24 ~
 15-25
 사용자 지정 15-17 ~
 15-22
 색상 8-6
 열 삽입 15-18
 열 재정렬 15-19
 열 제거 15-16, 15-19
 행 추가 18-18
 data-aware 15-14,
 15-27
 그림 8-17, 45-3 ~ 45-6
 교체 8-20
 로드 8-19
 변경 8-20
 저장 8-20
 그림 상자 3-24
 글꼴 13-14
 높이 8-5
 금칙 처리 7-8
 기능
 포팅 불가능 Windows
 10-9
 기본
 값 15-10
 속성 값 42-7
 변경 48-2, 48-3
 지정 42-11 ~ 42-12
 예약어 42-7
 조상 클래스 41-4
 지시어 42-12, 48-3

프로젝트 옵션 5-3
 핸들러
 메시지 46-3
 오버라이드 43-9
 이벤트 43-9
 기본 단위 4-60, 4-62
 기본 인덱스
 배치 이동 20-51
 기본 클라이언트 39-2
 기술 지원 1-3
 기존 컨트롤 40-4
 기하학적 도형
 그리기 49-9
 긴 문자열 4-41
 끝점
 소켓 연결 32-5
나
 날짜
 국제화 12-10
 입력 3-40
 calendar 컴포넌트 3-40
 날짜 필드
 서식 19-15
 내부 객체 33-9
 내장 프로시저 14-5, 18-23,
 18-50 ~ 18-55
 데이터베이스 지정 18-50
 매개변수 18-51 ~ 18-53
 디자인 타임 18-52 ~
 18-53
 런타임 18-53
 속성 18-52 ~ 18-53
 클라이언트 데이터
 셋 23-29
 생성 22-12
 실행 18-54
 열거 17-14
 오버로드 20-12
 준비 18-54
 BDE 기반 20-2, 20-12 ~
 20-13
 매개변수 바인딩 20-12
 dbExpress 22-8
 년비주얼 객체 3-11
 년비주얼 컴포넌트 40-5,
 40-11, 52-2
 네임스페이스
 호출 가능한
 인터페이스 31-4
 네트워크
 데이터베이스에 연결
 20-16
 네트워크 제어 파일 20-24
 노트북 분할기(divider) 3-42

논리값 15-2, 15-13
 달력 50-1 ~ 50-12
 날짜 추가 50-5 ~ 50-10
 속성과 이벤트 정의 50-2,
 50-6, 50-11
 오늘 날짜 선택 50-9
 이동 50-10 ~ 50-12
 읽기 전용으로 만들기 51-2
 ~ 51-4
 크기 조정 50-4
 뉴스그룹 1-3
 느린 프로세스
 스프레드 사용 9-1

다

다각선 8-10
 그리기 8-10
 다각형 8-12
 그리기 8-12
 다계층 아키텍처 25-4, 25-5
 웹 기반 25-33
 다계층 애플리케이션 14-3,
 14-13, 25-1 ~ 25-44
 개요 25-3 ~ 25-4
 마스터/디테일 관계 25-19
 만들기 25-11
 매개변수 23-28
 맵 13-10
 빌드 25-32
 서버 라이센스 25-3
 아키텍처 25-4, 25-5
 웹 애플리케이션 25-33 ~
 25-44
 빌드 25-34 ~ 25-35,
 25-44
 이점 25-2
 컴포넌트 25-2 ~ 25-3
 콜백 25-18
 다중 문서 인터페이스 5-1 ~
 5-2
 다중 스프레드 애플리케이션 9-1
 세션 20-13, 20-29 ~
 20-30
 다중 읽기 배타적 쓰기 동기화
 장치 9-8
 사용 경고 9-8
 다차원 크로스탭 16-3
 다형성 3-2, 3-5
 단방향 데이터셋 22-1 ~
 22-19
 데이터 페치 22-8
 데이터 편집 22-11
 메타데이터 페치 22-13 ~
 22-18
 명령 실행 22-10 ~ 22-11

- 바인딩 22-6 ~ 22-8
- 서버에 연결 22-2
- 제한 사항 22-1
- 준비 22-9
- 타입 22-2
- 단방향 커서 18-49
- 단일 계층 애플리케이션 14-3, 14-9, 14-12
- 파일 기반 14-10
- 단일 문서 인터페이스 5-1 ~ 5-2
- 단위, 변환 4-60
- 단축키
 - 메뉴에 추가 6-34
- 대상 데이터셋, 정의 20-49
- 대상, 작업 목록 6-18
- 대소문자 구별 10-14
- 인덱스 23-9
- 대화 상자 52-1 ~ 52-6
- 공통 3-47
- 국제화 12-9, 12-10
- 멀티페이지 3-42
- 생성 52-1
- 속성 편집기 47-10
- 초기 상태, 설정 52-1
- Windows 공통 52-1
- 생성 52-2
- 실행 52-4
- 더블 클릭
 - 응답 47-17 ~ 47-18
 - 컴포넌트 47-15
- 데이터
 - 기본값 15-10, 19-20
 - 변경 18-17 ~ 18-22
 - 보고 14-16
 - 분석 14-15, 16-2
 - 액세스 51-1
 - 인쇄 14-16
 - 입력 18-18
 - 파악 14-15
 - 폼 동기화 15-3
 - 표시 19-17, 19-17
 - 그리드에서 15-15, 15-27
 - 다시 그리기 사용 불가능 15-6
 - 현재값 15-8
 - 표시 전용 15-8
 - 형식, 국제화 12-10
- 데이터 압축
 - TSocketConnection 25-26
- 데이터 액세스
 - 메커니즘 5-10 ~ 5-11, 14-1 ~ 14-2, 18-2
 - 컴포넌트 5-10, 14-1
 - 스레드 9-4
 - 크로스 플랫폼 13-7, 14-2
 - 데이터 그리드 15-2, 15-14, 15-15 ~ 15-27
 - 값 얻기 15-17
 - 그리기 15-26
 - 기본 상태 15-16
 - 복원 15-21
 - 데이터 편집 15-6, 15-25
 - 데이터 표시 15-15, 15-17, 15-27
 - 배열 필드 15-22
 - ADT 필드 15-22
 - 런타임 옵션 15-24 ~ 15-25
 - 사용자 지정 15-17 ~ 15-22
 - 속성 15-28
 - 열 삽입 15-18
 - 열 재정렬 15-19
 - 열 제거 15-16, 15-19
 - 이벤트 15-26 ~ 15-27
 - 데이터 동기화
 - 여러 폼 15-3
 - 데이터 멤버 3-2
 - 데이터 모듈 14-6, 29-6
 - 데이터베이스 컴포넌트 20-16
 - 세션 20-18
 - 원격과 표준 비교 5-15
 - 웹 29-3, 29-5
 - 웹 애플리케이션 28-2
 - 작성 5-16
 - 편집 5-16
 - 폼에서 액세스 5-18
 - Web Broker
 - 애플리케이션 28-4
 - 데이터 무결성 14-5, 24-13
 - 데이터 바인딩 38-10
 - 데이터 브로커 23-26, 25-1
 - 데이터 사전 19-13 ~ 19-14, 20-53 ~ 20-54, 25-3
 - 계약 조건 24-13
 - 데이터 소스 14-7, 15-3 ~ 15-4
 - 사용 가능 15-4
 - 사용 불가능 15-4
 - 이벤트 15-4
 - 데이터 연결 51-4 ~ 51-6
 - 초기화 51-6
 - 데이터 입력 검증 19-16
 - 데이터 저장소 21-2
 - 데이터 정의 랭귀지 17-10, 18-43, 20-9, 22-11
 - 데이터 제약 조건 제약 조건 참조
 - 데이터 처리 랭귀지 17-10, 18-43, 20-9
 - 데이터 타입
 - 영구적 필드 19-6
 - 데이터 패킷 26-4
 - 고유한 레코드 보장 24-5
 - 복사 23-14 ~ 23-15
 - 애플리케이션 정의 정보 23-16, 24-7
 - 업데이트된 레코드 새로 고침 24-6
 - 읽기 전용 24-5
 - 클라이언트 편집 제한 24-5
 - 페치 23-26 ~ 23-28, 24-7 ~ 24-8
 - 편집 24-7
 - 필드 속성 포함 24-6
 - 필드 제어 24-5
 - XML 25-35, 25-38
 - 페치 25-38 ~ 25-39
 - 편집 25-39
 - XML 문서로 매핑 26-2
 - XML 문서로 변환 26-1 ~ 26-8
 - 데이터 필드 19-6
 - 정의 19-6 ~ 19-7
 - 데이터 필터 18-12 ~ 18-16
 - 런타임 시 설정 18-15
 - 복마크 사용 21-11
 - 빈 필드 18-14
 - 사용 가능/사용 불가능 18-13
 - 연산자 18-14
 - 정의 18-13 ~ 18-15
 - 쿼리와 비교 18-13
 - 클라이언트 데이터셋 23-3 ~ 23-5
 - 매개변수 사용 23-29 ~ 23-30
 - 데이터 형식
 - 기본 19-15
 - 데이터 형식화
 - 국제적인 애플리케이션 12-10
 - 데이터베이스 5-10, 14-1 ~ 14-5, 51-1
 - 관계형 14-1
 - 데이터 추가 18-21
 - 로그인 14-4, 17-4 ~ 17-5
 - 별칭 20-14
 - 보안 14-4
 - 선택 14-3

승인되지 않은 액세스 17-4
 식별 20-14 ~ 20-15
 암시적인 연결 17-2
 액세스 18-1
 액세스 속성 51-5 ~ 51-6
 연결 17-1 ~ 17-14
 웹 애플리케이션 28-17
 이름 지정 20-14
 타입 14-2
 트랜잭션 14-4 ~ 14-5
 파일 기반 14-3
 HTML 응답 생성 28-17 ~ 28-21
 데이터베이스 애플리케이션
 5-10, 14-1
 다계층 25-3 ~ 25-4
 배포 13-6
 분산 5-11
 아키텍처 14-6 ~ 14-14, 25-33
 파일 기반 14-9 ~ 14-10, 21-15 ~ 21-16, 23-33 ~ 23-36
 포팅 10-26
 확장 14-11
 XML 26-1 ~ 26-11
 데이터베이스 관리 시스템 25-1
 데이터베이스 데스크탑 20-55
 데이터베이스 드라이버
 BDE 20-1, 20-3, 20-14
 dbExpress 22-3 ~ 22-4
 데이터베이스 서버 5-10, 17-3, 20-15
 설명 17-2
 연결 14-8 ~ 14-9
 제약 조건 19-21, 19-21 ~ 19-22, 24-13
 타입 14-2
 데이터베이스 엔진
 협력업체 13-7
 데이터베이스 연결 17-2 ~ 17-5
 영구적 20-19
 유지 17-3
 제한 25-8
 풀링 25-7, 39-5 ~ 39-6
 해제 17-3, 17-3 ~ 17-4
 데이터베이스 컴포넌트 20-3, 20-13 ~ 20-16
 공유 20-16
 데이터베이스 식별 20-14 ~ 20-15
 세션 20-13 ~ 20-14, 20-21 ~ 20-22
 임시 20-20
 끊기 20-21
 캐시된 업데이트 적용 20-36
 데이터베이스 탐색기 15-2, 15-28 ~ 15-31, 18-5, 18-6
 데이터 삭제 18-20
 도움말 힌트 15-30
 버튼 15-29
 버튼 사용 가능/사용 불가능 15-29, 15-30
 편집 18-18
 데이터셋 14-7, 18-1 ~ 18-55
 검색 18-10 ~ 18-12
 검색 확장 18-30
 부분 키 18-30
 여러 열 18-11, 18-12
 인덱스 사용 18-11, 18-12, 18-28 ~ 18-30
 내장 프로시저 18-23, 18-50 ~ 18-55
 단방향 22-1 ~ 22-19
 단기 18-4 ~ 18-5
 레코드 포스트 18-20
 데이터 변경 18-17 ~ 18-22
 레코드 삭제 18-19 ~ 18-20
 레코드 추가 18-18 ~ 18-19, 18-21
 레코드 포스트 18-20
 레코드 표시 18-9 ~ 18-10
 레코드 필터링 18-12 ~ 18-16
 만들기 18-38 ~ 18-41
 모드 18-3 ~ 18-4
 반복 17-12
 범주 18-23 ~ 18-24
 변경 취소 18-21
 사용자 지정 18-2
 상태 18-3 ~ 18-4
 열기 18-4
 의사 결정 컴포넌트 16-4 ~ 16-6
 인덱싱되지 않은
 데이터셋 18-21
 읽기 전용
 업데이트 20-11
 커서 18-5
 쿼리 18-23, 18-42 ~ 18-50
 탐색 15-28, 18-5 ~ 18-9, 18-16
 테이블 18-23, 18-25 ~ 18-42
 편집 18-17 ~ 18-18
 프로바이더 24-2
 필드 18-1
 현재 행 18-5
 ADO 기반 21-9 ~ 21-17
 BDE 기반 20-2 ~ 20-13
 HTML 문서 28-20
 데이터셋 페이지 프로듀서 28-18
 필드 값 변환 28-19
 데이터셋 프로바이더 14-12
 데이터셋 필드 19-22, 19-26 ~ 19-27
 영구적 18-37
 표시 15-24
 델타 패킷 24-8, 24-9
 업데이트 스크린 24-11
 편집 24-8, 24-9 ~ 24-10
 XML 25-38, 25-39 ~ 25-40
 도움말 47-4
 문맥에 따른 도움말 3-44
 타입 정보 34-8
 툴팁 3-44
 힌트 3-44
 도움말 객체 등록 5-28
 도움말 뷰어 5-22
 도움말 선택기 5-28, 5-31
 도움말 시스템 5-22, 47-4
 객체 등록 5-28
 인터페이스 5-23
 키워드 47-5
 툴 버튼 6-49
 파일 47-4
 도킹 7-4
 도킹 공간 7-5
 도형 3-46, 8-11 ~ 8-12, 8-14
 그리기 8-11, 8-14
 비트맵 속성으로 채우기 8-9
 윤곽 8-5
 채우기 8-8
 동작 속성 3-18
 동적 메모리 3-58
 동적 메소드 41-9
 동적 바인딩 25-31
 동적 열 15-16
 속성 15-16
 동적 필드 19-2 ~ 19-3

드라이버 이름 20-14
 드라이브 문자 10-15
 드로잉 툴 45-1, 45-7,
 45-8, 49-5
 기본값으로 할당 6-45
 변경 8-13, 49-7
 애플리케이션에서 여러 개
 처리 8-12
 테스트 8-12, 8-13
 드래그 앤 도킹 3-20, 3-22,
 7-4 ~ 7-7
 드래그 앤 드롭 3-20, 7-1 ~
 7-4
 마우스 포인터 7-4
 사용자 지정 7-3
 이벤트 49-2
 DLL 7-4
 드래그 객체 7-3
 드롭다운 메뉴 6-34 ~ 6-35
 드롭다운 목록
 데이터 그리드 15-21
 드릴다운 폼 15-15
 등록
 변환 패밀리 4-60
 속성 편집기 47-12
 컴포넌트 40-12
 컴포넌트 에디터 47-19
 Active Server Object
 37-7 ~ 37-8
 ActiveX 컨트롤 38-14
 COM 객체 36-17
 디렉토리, Linux 10-16
 디버깅
 웹 서버 애플리케이션 27-7
 ~ 27-9, 29-2
 트랜잭션 객체 39-21 ~
 39-22
 Active Server Object
 37-8
 ActiveX 컨트롤 38-15
 COM 객체 36-18
 dbExpress
 애플리케이션 22-18 ~
 22-19
 디스패치
 액션 항목 29-18
 디스패치 컴포넌트 29-4,
 29-13
 어댑터 29-13
 디스패치, 웹 28-2, 28-4 ~
 28-5
 디스패치 액션 29-4
 디스패치 인터페이스 36-12,
 36-14

메소드 호출 35-13 ~
 35-14
 식별자 36-14
 타입 라이브러리 34-9
 타입 호환성 36-15
 Type Library 에디터
 34-15
 디자인
 애플리케이션 2-2
 디자인 타임 인터페이스 41-6
 디자인 타임 패키지 11-1,
 11-5 ~ 11-6
 디자인 툴 2-2
 디지털 오디오 테이프 8-32
라
 라디오 그룹 3-41
 라디오 버튼 3-37, 15-2
 그룹화 3-41
 선택 15-14
 data-aware 15-14
 라이브러리
 사용자 지정 컨트롤 40-4
 라이선스
 ActiveX 컨트롤 38-5,
 38-6 ~ 38-7
 Internet Explorer
 38-7
 라이선스 키 38-6
 라이선스 패키지 파일 38-7
 라이선싱
 ActiveX 컨트롤 38-6 ~
 38-7
 래스터 연산 45-7
 랩퍼 40-4, 52-2
 초기화 52-3
 컴포넌트 랩퍼 참조
 런타임 인터페이스 41-6
 런타임 타입 정보 41-6
 런타임 패키지 11-1,
 11-2 ~ 11-5
 레이블 3-43, 12-10, 15-2,
 40-4
 열 15-17
 레지스트리 10-15
 레코드
 객체와 비교 3-5
 검색 반복 18-30
 검색 조건 18-11
 반복 18-8
 복사
 배치 작업 20-8, 20-51
 삭제 18-19 ~ 18-20,
 18-41
 배치 작업 20-8, 20-51

새로 고침 15-7, 23-31
 업데이트 18-21 ~ 18-22,
 24-8 ~ 24-11, 25-39
 ~ 25-40
 업데이트 스크린 24-11
 델타 패킷 24-8, 24-9
 배치 작업 20-8, 20-51
 여러 24-6
 쿼리 20-11
 클라이언트
 데이터셋 23-20 ~
 23-25
 테이블 식별 24-12
 XML 문서 26-10 ~
 26-11
 업데이트 해결 23-23
 웹 어댑터 29-8
 이동 15-28, 18-5 ~
 18-9, 18-16
 정렬 18-26 ~ 18-27
 찾기 18-10 ~ 18-12,
 18-28 ~ 18-30
 추가 18-18, 18-19,
 18-21
 배치 작업 20-8,
 20-50, 20-51
 페지 22-8, 23-26 ~
 23-28
 비동기화 21-12
 포스트 15-6, 18-20
 데이터 그리드 15-25
 데이터셋을 닫을 때
 포스트 18-20
 표시 15-27, 18-9 ~
 18-10
 필터링 18-12 ~ 18-16
 현재 동기화 18-42
 Type Library 에디터
 34-10, 34-17, 34-23 ~
 34-24
 레코드 전송 52-2
 레포지토리 5-20 ~ 5-22,
 6-12
 항목 사용 5-21
 항목 추가 5-20
 로그인
 웹 연결 25-27
 SOAP 연결 25-27
 로그인 스크립트 17-4 ~
 17-5
 로그인 이벤트 17-5
 로그인 정보
 지정 17-4
 로컬 SQL 20-9, 20-10
 이중 쿼리 20-9

- 로컬 데이터베이스 14-3
 - 별칭 20-26
 - 액세스 20-5
 - 테이블 이름 재지정 20-8
 - BDE 지원 20-5 ~ 20-8
- 로컬 트랜잭션 20-32 ~ 20-33
- 로케일 12-1
 - 데이터 형식 12-10
 - 리소스 모듈 12-11
- 로케일 설정 4-44
- 루트 디렉토리 10-17
- 리소스 40-7, 45-1
 - 문자열 12-10
 - 분리 12-10
 - 시스템, 최적화 40-4
 - 지역화 12-11, 12-12, 12-14
 - 캐시로 저장 45-2
 - 해제 52-5
- 리소스 DLL
 - 동적 전환 12-13
 - 마법사 12-11
- 리소스 디스펜서 39-5
 - ADO 39-6
 - BDE 39-6
- 리소스 모듈 12-10, 12-11, 12-12
- 리소스 파일 6-42 ~ 6-43
 - 로드 6-43
- 리소스 풀링 39-5 ~ 39-7
- 리소스 해제 52-5
- 리소스를 캐시로 저장 45-2
- 리스닝 연결 32-2, 32-3, 32-7, 32-9
 - 닫기 32-7
 - 포트 번호 32-5
- 리스트 박스 3-38, 15-2, 15-12, 50-1
 - 속성 저장
 - 예제 6-9
 - 채우기 15-11
 - 항목 드래그 7-2, 7-3
 - 항목 그룹 7-3
 - data-aware 15-10 ~ 15-13
 - owner-draw 7-12
 - draw-item 이벤트 7-16
 - measure-item 이벤트 7-15
- 리스트 뷰
 - owner draw 7-12
- 리스트 컨트롤 3-38 ~ 3-4
- 리포트 14-16

마

- 메뉴 6-17, 6-29 ~ 6-41
 - 국제화 12-9, 12-10
 - 명령 액세스 6-34
 - 사이 이동 6-38
 - 이름 지정 6-32
 - 이미지 추가 6-36
 - 이벤트 처리 3-28, 6-41
 - 재사용 6-38
 - 추가 6-32 ~ 6-37
 - 다른 애플리케이션에서 가져옴 6-42
 - 그룹다운 6-34 ~ 6-35
 - 템플릿 6-32, 6-38, 6-39 ~ 6-41
 - 로드 6-39
 - 삭제 6-40
 - 템플릿으로 저장 6-39, 6-40 ~ 6-41
 - 팝업 7-11, 7-12
 - 표시 6-37, 6-38
 - 항목 비활성화 7-10
 - 항목 이동 6-35
 - owner-draw 7-12
- 메뉴 디자이너 3-28, 6-30 ~ 6-31
 - 컨텍스트 메뉴 6-37
- 메뉴 컴포넌트 6-30
- 메뉴 항목 6-33 ~ 6-34
 - 구분자 표시줄 6-34
 - 그룹화 6-34
 - 문자에 밑줄 긋기 6-34
 - 삭제 6-33, 6-38
 - 속성 설정 6-37
 - 위치 표시자 6-38
 - 이동 6-35
 - 이름 지정 6-32, 6-41
 - 중첩 6-34
 - 정의 6-29
 - 추가 6-33, 6-41
 - 편집 6-37
- 메뉴, 작업 목록 6-18
- 메모 컨트롤 7-7, 42-8
 - 속성 3-32
- 메모 필드 15-2, 15-9
 - rich edit 15-9 ~ 15-10
- 메모리 관리
 - 동적 메모드와 가상 메모드 비교 41-9
 - 인터페이스 4-25
 - 의사 결정 컴포넌트 16-8, 16-19
 - 컴포넌트 3-12
 - 폼 6-6
- COM 객체 4-20
- 메소드 3-3, 8-15, 40-6, 44-1, 50-10
 - 가상 41-8, 44-3
 - 객체 3-5, 3-7
 - 그래픽 45-3, 45-4, 45-6, 45-7
 - 팔레트 45-5
 - 그리기 49-8, 49-9
 - 디스패칭 41-7
 - 메시지 처리 46-1, 46-3, 46-4
 - 삭제 3-29
 - 상속 43-6
 - 선언 8-15, 44-4
 - 가상 41-8
 - 동적 41-9
 - 정적 41-7
 - public 44-3
 - 속성 42-5 ~ 42-7, 44-1, 44-2, 49-3
 - 오버라이드 41-8, 46-3, 46-4, 50-11
 - 이름 지정 44-2
 - 이벤트 핸들러 43-3, 43-5
 - 오버라이드 43-6
 - 인터페이스에 추가 36-10
 - 재정의 41-8, 46-7
 - 초기화 42-13
 - 호출 43-6, 44-3, 49-4
 - ActiveX 컨트롤 추가 38-8 ~ 38-9
 - protected 44-3
 - public 44-3
- 메소드 재정의 41-8
- 메소드 포인터 43-2, 43-3, 43-8
- 메시지 46-1 ~ 46-7, 50-4
 - 레코드 46-2, 46-4
 - 타입, 선언 46-6
 - 마우스 51-8
 - 마우스 다운 및 키 다운 51-8
 - 사용자 정의 46-5, 46-7
 - 식별자 46-6
 - 정의 46-2
 - 처리 46-3 ~ 46-5
 - 크래킹 46-2
 - 키 51-8
 - 트래핑 46-4
 - 핸들러 46-1, 46-2, 50-4
 - 기본 46-3
 - 메소드, 재정의 46-7
 - 생성 46-5 ~ 46-7

- 선언 46-4, 46-5, 46-7
- 오버라이드 46-3
- Windows 6-5
- 메시지 기반 서버
- 웹 서버 애플리케이션 참조
- 메시지 루프
- 스레드 9-4
- 메시지 헤더(HTTP) 27-3, 27-4
- 메시징 10-21
- 메인 VCL 스레드 9-4
- OnTerminate 이벤트 9-6
- 메인 폼 6-1
- 메타데이터 17-13 ~ 17-14
- 수정 22-11 ~ 22-12
- 프로바이더에서 얻기 23-27
- dbExpress 22-13 ~ 22-18
- 메타파일 3-46, 8-1, 8-19, 45-4
- 사용 시기 8-3
- 릴리스 노트 13-15
- 마법사 5-20
- 리소스 DLL 12-11
- 속성 페이지 33-21, 38-12
- 원격 데이터 모듈 25-13 ~ 25-14
- 타입 라이브러리 34-19
- 트랜잭션 객체 33-21, 39-15 ~ 39-18
- Active Server Object 33-20, 37-2 ~ 37-3
- ActiveForm 33-21, 38-5 ~ 38-6
- ActiveX 라이브러리 33-21
- ActiveX 컨트롤 33-21, 38-4 ~ 38-5
- Automation 객체 33-20, 36-4 ~ 36-9
- COM 33-18 ~ 33-23, 36-1
- COM 객체 33-20, 34-19, 36-2 ~ 36-4, 36-5 ~ 36-9
- COM+ 이벤트 객체 33-21, 39-19
- Component 40-9
- Console Wizard 5-3
- CORBA Data Module 25-16 ~ 25-17
- SOAP Data Module 25-16
- Transactional Data Module 25-15
- WebSnap
- 애플리케이션 29-1
- XML 데이터 바인딩 30-5 ~ 30-8
- 마살링 33-8
- 사용자 지정 36-16
- 웹 서비스 31-5
- 트랜잭션 객체 39-3
- COM 인터페이스 33-8 ~ 33-9, 36-4, 36-15 ~ 36-16
- IDispatch 인터페이스 33-13, 36-15
- 마스킹 편집 컨트롤 3-31
- 마스터/디테일 관계 15-15, 18-35 ~ 18-37, 18-47 ~ 18-48
- 다계층 애플리케이션 25-19
- 단방향 데이터셋 22-12 ~ 22-13
- 연쇄 업데이트 24-6
- 연쇄 삭제 24-6
- 인덱스 18-35
- 중첩 테이블 18-37, 25-20
- 참조 무결성 14-5
- 클라이언트 데이터셋 23-19
- 마스터/디테일 폼 15-15
- 예제 18-36 ~ 18-37
- 마우스 다운 메시지 51-8
- 마우스 메시지 46-2, 51-8
- 마우스 버튼 8-24
- 마우스 이동 이벤트 8-26
- 클릭 8-25
- 마우스 버튼 놓기 8-25
- 마우스 이벤트 8-24 ~ 8-26, 49-2
- 드래그 앤 드롭 7-1 ~ 7-4
- 매개변수 8-24
- 상태 정보 8-24
- 정의 8-24
- 테스트 8-26
- 마우스 포인터
- 드래그 앤 드롭 7-4
- 매개변수
- 결과 18-51
- 마우스 이벤트 8-24, 8-25
- 메시지 46-2, 46-3, 46-4, 46-6
- 모드 바인딩 20-12
- 속성 설정 42-6
- 배열 속성 42-8
- 이벤트 핸들러 43-3, 43-7, 43-8, 43-9
- 이중 인터페이스 36-16
- 입/출력 18-51
- 입력 18-51
- 출력 18-51, 23-28
- 클라이언트 데이터셋 23-28 ~ 23-30
- 레코드 필터링 23-29 ~ 23-30
- 클래스 41-9
- HTML 태그 28-14
- TXMLTransformClient 26-9
- XML 브로커 25-39
- 매개변수 컬렉션 에디터 18-45
- 매개변수화된 쿼리 18-43, 18-45 ~ 18-47
- 생성
- 디자인 타임 시 18-45
- 런타임 시 18-46
- 패킹
- XML 26-2 ~ 26-3
- 정의 26-4
- 멀티미디어 8-33
- 멀티바이트 문자 집합 12-2
- 멀티바이트 문자 코드 12-2
- 멀티바이트 문자(MBCS) 10-18, 10-23
- 멀티태스킹 10-16
- 멀티페이지 대화 상자 3-42
- 멀티프로세싱
- 스레드 9-1
- 멤버 함수 3-3
- 명령 객체 21-17 ~ 21-20
- 반복 17-13
- 명령 스위치 10-15
- 명령, 작업 목록 6-18
- 명명된 연결 22-4 ~ 22-5
- 런타임 시 로드 22-5
- 삭제 22-5
- 이름 재지정 22-5
- 추가 22-5
- 모달 폼 6-6
- 모달리스 6-6, 6-8
- 모듈 40-11
- Type Library 에디터 34-10, 34-18, 34-24
- 모바일 컴퓨팅 14-14
- 모서리가 둥근 사각형 8-11
- 목록
- 문자열 3-49 ~ 3-54
- 스레드에서 사용 9-5

- 무결성 위반 20-53
- 문자 42-2
- 문자 집합 4-43, 12-2, 12-2 ~ 12-4
 - 국제적인 정렬 순서 12-10
 - 기본 12-2
 - 멀티바이트 12-2
 - 멀티바이트 변환 12-2
 - ANSI 12-2
 - OEM 12-2
- 문자 타입 4-39, 12-2
- 문자열 4-39, 42-2, 42-8
 - 2바이트 변환 12-3
 - 그래픽 연결 7-14
 - 긴 4-41
 - 루틴
 - 대소문자 구별 4-44
 - 런타임 라이브러리 4-42
 - 멀티바이트 문자 지원 4-44
 - 메모리 손상 4-50
 - 반환 42-8
 - 번역 12-2, 12-9, 12-10
 - 변수 매개변수 4-49
 - 선언 및 초기화 4-46
 - 시작 위치 7-9
 - 잘라내기 12-3
 - 정렬 12-10
 - 지역 변수 4-48
 - 참조 카운팅 문제 4-41, 4-47
 - 컴파일러 지시어 4-49
 - 크기 7-9
 - 타입 개요 4-40
 - 타입의 혼합 및 변환 4-47
 - 파일 4-57
 - 확장 문자 집합 4-50
 - PChar 변환 4-47
- 문자열 그리드 3-45
- 문자열 목록 3-49 ~ 3-54
 - 객체 추가 7-14
 - 기간이 긴 문자열 3-51
 - 기간이 짧은 문자열 3-50
 - 문자열 삭제 3-53
 - 문자열 이동 3-53
 - 문자열 찾기 3-52
 - 반복 3-53
 - 복사 3-54
 - 부분 문자열 3-53
 - 생성 3-50 ~ 3-52
 - 연결 객체 3-54
 - 위치 3-52, 3-53
 - 정렬 3-53
 - 추가 3-53
 - 파일에 저장 3-50

- 파일에서 로드 3-50
- owner-draw 컨트롤 7-13 ~ 7-14
- 문자열 번역 12-2, 12-9, 12-10
 - 2바이트 변환 12-3
- 문자열 연산자 4-50
- 문자열 예약어 4-41
 - 기본 타입 4-40
- 문자열 필드
 - 크기 19-6
- 문체 테이블 20-53
- 미디어 장치 8-31
- 미디어 플레이어 3-24, 8-31 ~ 8-33
 - 예제 8-33

바

- 배경 12-10
- 배열 42-2, 42-8
 - 안전 34-13
- 배열 필드 19-22, 19-25 ~ 19-26
 - 영구적 필드 19-25
 - 평평하게 하기 (flattening) 15-22
 - 표시 15-22, 19-23
- 배치 업데이트 21-12 ~ 21-15
 - 적용 21-14
 - 취소 21-14 ~ 21-15
- 배치 작업 20-8 ~ 20-9, 20-49 ~ 20-53
 - 다른 데이터베이스 20-51
 - 데이터 업데이트 20-51
 - 데이터 추가 20-50, 20-51
 - 데이터 타입 매핑 20-51 ~ 20-52
 - 데이터셋 복사 20-51
 - 레코드 삭제 20-51
 - 모드 20-8, 20-50
 - 설정 20-49 ~ 20-50
 - 실행 20-52
 - 오류 처리 20-53
- 배치 파일 10-15
- 배타적 잠금
- 테이블 20-6
- 배포
 - 글꼴 13-14
 - 데이터베이스 애플리케이션 13-6
 - 애플리케이션 13-1
 - 웹 애플리케이션 13-10
 - 일반 애플리케이션 13-1

- 패키지 파일 13-3
- ActiveX 컨트롤 13-5
- Borland Database Engine 13-8
- CLX 애플리케이션 13-6
- dbExpress 22-1
- DLL 파일 13-5
- MIDAS 애플리케이션 13-10
- 백그라운드 6-20
- 버전 정보
 - 타입 정보 34-8
 - ActiveX 컨트롤 38-5
- 버전 조정 2-5
- 버튼 3-35 ~ 3-37
 - 탐색기 15-28
 - 툴바 6-43
 - 툴바에 추가 6-44 ~ 6-46, 6-47
 - 툴바에서 사용 불가능 6-47
 - glyph 할당 6-45
- 번역 12-9
- 범위 18-30 ~ 18-34
 - 경계 18-33
 - 변경 18-33 ~ 18-34
 - 인덱스 18-31
 - 적용 18-34
 - 지정 18-31 ~ 18-33
 - 취소 18-34
 - 필터와 비교 18-30
 - Null 값 18-31, 18-32, 18-33
- 북마크 18-9 ~ 18-10
 - 데이터셋 타입에 의한 지원 18-9
 - 레코드 필터링 21-11
- 부모 속성 3-19
- 부모 컨트롤 3-19
- 부분 키
 - 검색 18-30
 - 범위 설정 18-33
- 부울 값 42-2, 42-12, 51-3
- 부울 필드 15-2, 15-13
- 변경 로그 23-5, 23-20, 23-35
 - 변경 내용 저장 23-6
 - 변경 취소 23-6
- 변수
 - 객체 3-10, 3-10 ~ 3-11
 - 선언
 - 예제 3-10
- 변환
 - 문자열 4-47
 - 복잡 4-61
 - 통화 4-63

- 필드 값 19-16, 19-18 ~ 19-19
- PChar 4-47
- 변환 계수 4-63
- 변환 유틸리티 4-59
- 변환 클래스 4-63 ~ 4-65
- 변환 패밀리 4-59, 4-60
 - 생성 예제 4-60
- 변환 파일 26-1 ~ 26-6
 - 사용자 정의 노트 26-5, 26-7 ~ 26-8
 - TXMLTransform 26-7
 - TXMLTransformClient 26-9
 - TXMLTransformProvider 26-8
- 별칭
 - BDE 20-3, 20-14, 20-25 ~ 20-27
 - 로컬 20-26
 - 삭제 20-26
 - 생성 20-25 ~ 20-26
 - 지정 20-14, 20-15
 - Type Library 에디터 34-10, 34-17, 34-23
- 병렬 프로세스
 - 스레드 9-1
- 보안
 - 다계층 애플리케이션 25-2
 - 데이터베이스 14-4, 17-4 ~ 17-5
 - 로컬 테이블 20-22 ~ 20-24
 - 소켓 연결 등록 25-9
 - 웹 연결 25-10, 25-27
 - 트랜잭션 객체 39-14 ~ 39-15
 - 트랜잭션 데이터 모듈 25-7, 25-9
 - DCOM 25-38
 - SOAP 연결 25-27
- 복사(Object Repository) 5-21
- 분리
 - 트랜잭션 14-5, 39-9
- 분산 데이터 처리 25-2
- 분산 애플리케이션
 - 데이터베이스 5-11
 - MTS 및 COM+ 5-15
- 분산 COM 33-7, 33-8
- 브러시 8-8 ~ 8-9, 49-5
 - 변경 49-7
 - 비트맵 속성 8-9
 - 색상 8-8
 - 스타일 8-8

- 브리프케이스 모델 14-14
- 비디오 카세트 8-33
- 비디오 클립 8-29, 8-31
- 비생성 인텍스 파일 20-6
- 비즈니스 롤 25-2, 25-13
 - 트랜잭션 객체 39-2
 - ASP 37-1
- 비차단 연결 32-10
 - 차단 연결과 비교 32-10
- 비트맵 3-46, 8-18 ~ 8-19, 45-4
 - 교체 8-20
 - 국제화 12-10
 - 그래픽 컨트롤과 비교 49-3
 - 그리기 8-18
 - 드로잉 표면 45-4
 - 로드 45-4
 - 문자열에 연결 3-54, 7-13
 - 브러시 8-9
 - 브러시 속성 8-8, 8-9
 - 빈 8-17
 - 소멸 8-21
 - 스크롤 8-17
 - 스크롤 기능 추가 8-17
 - 애플리케이션에
 - 나타날 때 8-2
 - 오프스크린 45-6 ~ 45-7
 - 임시 8-18
 - 초기 크기 설정 8-17
 - 컴포넌트에 추가 47-3
 - 틀바 6-47
 - 프레임 6-15
 - draw-item 이벤트 7-16
 - ScanLine 속성 8-9
- 비트맵 객체 8-3
- 비트맵 버튼 3-36
- 빗면 3-46

사

- 사각형
 - 그리기 8-11, 49-9
- 사용권 계약서 13-15
- 사용자 목록 서비스 29-4
- 사용자 인터페이스 3-23, 14-15 ~ 14-16
 - 단일 레코드 15-7
 - 데이터 구성 15-7 ~ 15-8, 15-14 ~ 15-15
 - 레이아웃 6-4 ~ 6-5
- 분리 14-6
- 여러 레코드 15-14
- 폼 6-1 ~ 6-2
- 사용자 정의 메시지 46-5, 46-7
- 사용자 정의 타입 49-3

- 사용자 지정 가변 4-27 ~ 4-39
 - 값 로드 및 저장 4-35 ~ 4-36
 - 단항 연산자 4-33 ~ 4-34
 - 데이터 저장 4-27 ~ 4-28, 4-31, 4-34
 - 메모리 4-34
 - 메소드 4-37 ~ 4-39
 - 복사 4-34
 - 비교 연산자 4-32 ~ 4-33
 - 사용 가능 4-36
 - 삭제 4-34 ~ 4-35
 - 생성 4-27, 4-28 ~ 4-36
 - 속성 4-37 ~ 4-39
 - 유틸리티 작성 4-36 ~ 4-37
 - 이항 연산 4-30 ~ 4-32
 - 타입 변환 4-28 ~ 4-30, 4-31, 4-36
- 사용자 지정 컨트롤 40-4
 - 라이브러리 40-4
- 사용자 지정 컴포넌트 3-31
- 사용자별 예약 35-16
- 삼각형 8-12
- 상속 3-5, 3-8
 - 메소드 43-6
 - 속성 49-2, 50-2
 - 게시 42-3
 - 이벤트 43-5
- 상속(Object Repository) 5-21
- 상태 없음(stateless) 객체 39-11
- 상태 정보 3-43, 3-44
 - 공유 속성 39-6
 - 관리 39-5
 - 마우스 이벤트 8-24
 - 통신 24-8, 25-20 ~ 25-22
 - 트랜잭션 39-11
 - 트랜잭션 객체 39-11
- 상태 표시줄 3-43, 3-44
 - 국제화 12-9
 - owner-draw 7-12
- 상호 배타적 옵션 6-46
- 새로운 WebSnap
 - 애플리케이션 29-2
- 색상
 - 국제화 12-10
 - 펜 8-6
 - 색상 그리드 8-6
 - 색상 수 13-12
 - 프로그래밍 13-14
- 생략(...)

- 그리드의 버튼 15-21
- 생성자 3-11, 40-13, 42-12, 44-3, 50-3, 50-4, 51-6
- 소유된 객체 49-5, 49-7
- 여러 6-9
- 오버라이드 48-2
- 크로스 플랫폼 10-22
- 생성자 클래스
 - CoClass 35-5
 - CoClasses 35-13
- 셀 스크립트 10-15
- 서버
 - 웹 애플리케이션
 - 디버거 29-2
 - 인터넷 27-1 ~ 27-9
 - 서버 애플리케이션
 - 다계층 25-5 ~ 25-11
 - 등록 25-11, 25-23
 - 서비스 32-1
 - 아키텍처 25-5
 - 웹 서비스 31-2 ~ 31-8
 - 인터페이스 32-2
 - COM 33-5 ~ 33-9, 36-1 ~ 36-18
 - 서버 소켓 32-7
 - 오류 메시지 32-8
 - 이벤트 처리 32-9
 - 지정 32-6
 - 클라이언트 승인 32-9
 - 클라이언트 요청 승인 32-7
 - Windows 소켓 객체 32-7
 - 서버 연결 32-2, 32-3
 - 포트 번호 32-5
 - 서버 타입 29-2
 - 서버측 29-9
 - 서버측 스크립팅 29-9
 - 서비스 5-4 ~ 5-8
 - 구현 32-1 ~ 32-2, 32-7
 - 네트워크 서버 32-1
 - 설치 5-4
 - 설치 해제 5-4
 - 예제 5-6
 - 예제 코드 5-4, 5-6
 - 요청 32-6
 - 이름 속성 5-8
 - 포트 32-2
 - 서비스 애플리케이션 5-4 ~ 5-8
 - 예제 5-6
 - 예제 코드 5-4, 5-6
 - 서비스 스레드 5-6
 - 서비스 제어 관리자 5-4
 - 서식있는 텍스트 컨트롤 3-33, 15-9 ~ 15-10
 - 속성 3-32

- 서식있는 텍스트 편집 컨트롤
 - 속성 3-32
- 서식있는 텍스트(rich text) 컨트롤 7-7
- 선
 - 그리기 8-5, 8-10, 8-10 ~ 8-11, 8-27 ~ 8-29
 - 이벤트 핸들러 8-26
 - 팬 너비 변경 8-6
 - 지우기 8-28
- 선언
 - 메소드 8-15, 44-4
 - 가상 41-8
 - 동적 41-9
 - 정적 41-7
 - public 44-3
 - 메시지 핸들러 46-4, 46-5, 46-7
 - 변수
 - 예제 3-10
 - 새 컴포넌트 타입 41-3
 - 속성 42-3, 42-3 ~ 42-7, 42-12, 43-8, 49-3
 - 사용자 정의 타입 49-3
 - stored 42-12
 - 이벤트 핸들러 43-5, 43-8, 50-11
 - 클래스 41-9, 49-5
 - protected 41-5
 - public 41-6
 - published 41-6
- 선택기
 - 도움말 5-28
- 설명서
 - 정렬 1-3
- 설치 프로그램 13-2
- 세션 20-17 ~ 20-30
 - 기본 20-3, 20-13, 20-17 ~ 20-18
 - 기본 연결 속성 20-19
 - 다중 20-13, 20-29 ~ 20-30
 - 다중 스레드
 - 애플리케이션 20-13, 20-29 ~ 20-30
 - 단기 20-18
 - 데이터베이스 20-13 ~ 20-14
 - 데이터셋 20-3 ~ 20-4
 - 메소드 20-14
 - 빌칭 관리 20-25
 - 생성 20-28, 20-29
 - 암시적 데이터베이스
 - 연결 20-13
 - 암호 20-22

- 여러 20-28 ~ 20-30
- 연결 관리 20-19 ~ 20-22
- 연결 단기 20-20
- 연결 열기 20-20
- 연결된 데이터베이스
 - 20-21 ~ 20-22
- 웹 애플리케이션 28-18
- 이름 지정 20-29, 28-18
- 재시작 20-19
- 정보 얻기 20-27 ~ 20-28
- 현재 상태 20-18
- 활성화 20-18 ~ 20-19
- 세션 서비스 29-4
- 셀(그리드) 3-44
- 소멸자 3-11, 44-3, 51-6
 - 소유된 객체 49-5, 49-7
- 소스 데이터셋, 정의 20-49
- 소스 코드
 - 보기
 - 특정 이벤트 핸들러 3-27
 - 재사용 6-12
 - 최적화 8-15
 - 편집 2-3
- 소스 파일
 - 변경 2-3
 - 패키지 11-2, 11-6, 11-8, 11-12
 - 소스 파일, 공유 10-14
 - 소유된 객체 49-5 ~ 49-8
 - 초기화 49-7
- 소켓 32-1 ~ 32-10
 - 네트워크 주소 32-3, 32-4
 - 서비스 구현 32-1 ~ 32-2, 32-7
 - 설명 32-3
 - 쓰기 32-10
 - 오류 처리 32-8
 - 이벤트 처리 32-8 ~ 32-9, 32-10
 - 읽기 32-10
 - 읽기/쓰기 32-9 ~ 32-10
 - 정보 제공 32-4
 - 클라이언트 요청 승인 32-3
 - 호스트 지정 32-4
- 소켓 객체
 - 클라이언트 32-6
- 소켓 디스패처
 - 애플리케이션 25-9, 25-13, 25-26
- 소켓 연결 25-9 ~ 25-10, 25-25, 32-2 ~ 32-3
 - 끝점 32-3, 32-5
 - 다중 32-5
 - 단기 32-7

- 일기 32-6, 32-7
- 정보 보내기/받기 32-9
- 타입 32-2
- 소켓 컴포넌트 32-5 ~ 32-7
- 소프트웨어 사용권 요구 사항 13-15
- 속성 3-2, 42-1 ~ 42-13
 - 값 쓰기 42-6, 47-8
 - 값 읽기 47-8
 - 값 지정 42-11, 47-9
 - 개요 40-6
 - 객체 3-5
 - 공통 대화 상자 52-1
 - 기본값 42-7, 42-11 ~ 42-12
 - 재정의 48-2, 48-3
 - 내부 데이터 저장소 42-4, 42-6
 - 도움말 제공 47-4
 - 로드 42-13
 - 랩퍼 컴포넌트 52-3
 - 배열 42-2, 42-8
 - 변경 47-6 ~ 47-12, 48-2, 48-3
 - 보기 47-8
 - 상속 42-3, 49-2, 50-2
 - 서식있는 텍스트 컨트롤 3-32
 - 선언 42-3, 42-3 ~ 42-7, 42-12, 43-8, 49-3
 - stored 42-12
 - 사용자 정의 타입 49-3
 - 설정 3-24 ~ 3-25
 - 속성 편집기 47-10
 - 쓰기 전용 42-6
 - 액세스 42-5 ~ 42-7
 - 업데이트 40-7
 - 이벤트 43-1, 43-2
 - 인터페이스 42-10
 - 인터페이스에 추가 36-9 ~ 36-10
 - 읽기 전용 41-6, 41-7, 42-7, 51-2
 - 재선언 42-12, 43-5
 - 저장 42-12
 - 클래스 42-2
 - 타입 42-2, 42-8, 47-8, 49-3
 - 편집
 - 텍스트로 47-8
 - 하위 컴포넌트 42-9
 - ActiveX 컨트롤 추가 38-8 ~ 38-9
 - COM 33-3, 34-8
 - 참조별 작성 34-8
 - 참조별 전용 34-8
 - HTML 테이블 28-19
 - no default 42-7
 - published 50-2
 - published가 아닌 저장 및 로드 42-13 ~ 42-15
 - read 및 write 42-5
 - 속성 사양
 - 타입 라이브러리 34-13
 - 속성 설정
 - 쓰기 42-8
 - 읽기 42-8
 - 속성 설정 읽기 42-6
 - 속성 저장 명령 19-13
 - 속성 페이지 38-12 ~ 38-14
 - 생성 38-12 ~ 38-14
 - 업데이트 38-13
 - 컨트롤 추가 38-12 ~ 38-14
 - ActiveX 컨트롤 35-7, 38-3, 38-14
 - ActiveX 컨트롤 속성에 연결 38-13
 - ActiveX 컨트롤 업데이트 38-13
 - import된 컨트롤 35-4
 - 속성 페이지 마법사 38-12
 - 속성 편집기 3-25, 42-2, 47-6 ~ 47-12
 - 대화 상자 47-10
 - 등록 47-12
 - 속성 47-10
 - 파생된 클래스 47-7
 - 수직 트랙 표시줄 3-33
 - 수평 트랙 표시줄 3-33
 - 순환 참조 6-2
 - 숨겨진 필드 24-5
 - 숫자 42-2
 - 국제화 12-10
 - 속성 값 42-12
 - 숫자 필드
 - 서식 19-15
 - 스레드 9-1 ~ 9-12
 - 객체 잠금 9-7
 - 그래픽 객체 9-5
 - 대기 9-9
 - 여러 9-10
 - 데이터 액세스 컴포넌트 9-4
 - 동시 액세스 피하기 9-7
 - 메시지 루프 9-4
 - 목록 사용 9-5
 - 반환 값 9-9
 - 생성 9-11
 - 서비스 5-6
 - 수 제한 9-11
 - 실행 9-11
 - 실행 막기 9-7
 - 예외 9-6
 - 우선 순위 9-1, 9-2
 - 오버라이드 9-11
 - 이벤트 대기 9-9
 - 임계 구역 9-7
 - 조정 9-4, 9-7 ~ 9-10
 - 종료 9-6
 - 증지 9-11
 - 초기화 9-2
 - 프로세스 공간 9-4
 - 해제 9-2, 9-3
 - 활동 39-18
 - BDE 20-13
 - ids 9-12
 - ISAPI/NSAPI
 - 프로그램 28-3, 28-18
 - VCL 스레드 9-4
 - 스레드 객체 9-1
 - 정의 9-2
 - 제한 사항 9-2
 - 초기화 9-2
 - 스레드 로컬 변수 9-5
 - OnTerminate 이벤트 9-6
 - 스레드 모델 36-6 ~ 36-9
 - 시스템 레지스트리 36-7
 - 원격 데이터 모듈 25-14
 - 트랜잭션 객체 39-17
 - 트랜잭션 데이터 모듈 25-15
 - ActiveX 컨트롤 38-5
 - Automation 객체 36-5
 - COM 객체 36-3
 - CORBA 데이터 모듈 25-17
 - 스레드 변수 9-5
 - 스레드 함수 9-4
 - 스레드에 대해 안전한 객체 9-4
 - 스크롤 가능한 비트맵 8-17
 - 스크롤 막대 3-33
 - 텍스트 창 7-8
 - 스크립트 29-9
 - 편집과 보기 29-11
 - 활성 29-9
 - WebSnap에서 생성 29-10
 - 스크립트 객체 29-11
 - 스크립트 보기 29-11
 - 스크립트 편집 29-11
 - 스크립트(URL) 27-3
 - 스키마 정보 22-13 ~ 22-18
 - 내장 프로시저 22-15, 22-17 ~ 22-18

- 인덱스 22-17
- 테이블 22-15
- 필드 22-16
- 스타일 10-7
- 스타일 시트 25-42
- 스텝 (stub)
 - 트랜잭션 객체 39-2
- COM 33-8
- 스트림 3-58
- 스플리터 3-34
- 스피드 버튼 3-36
 - 가운데 맞춤 6-45
 - 그룹화 6-46
 - 드로잉 툴 8-13
 - 이벤트 핸들러 8-13
 - 작동 모드 6-44
 - 초기 상태, 설정 6-45
 - 토글로 사용 6-46
 - 툴바에 추가 6-44 ~ 6-46
 - glyph 할당 6-45
- 스핀 편집 컨트롤 3-34
- 쓰기 전용 속성 42-6
- 시간
 - 국제화 12-10
 - 입력 3-40
- 시간 변환 4-60
- 시간 초과 이벤트 9-10
- 시간 필드
 - 서식 19-15
- 시그널 10-15
- 시스템 공지 10-21
- 시스템 리소스 최적화 40-4
- 시스템 리소스, 절감 40-4
- 시스템 이벤트 10-21
- 식별자
 - 메소드 44-2
 - 메시지 레코드 타입 46-6
 - 속성 설정 42-6
 - 이벤트 43-8
 - 잘못된 6-32
 - 클래스 필드 43-2
- 썬 클라이언트
 - 애플리케이션 25-2, 25-33
- 실행 파일 10-15
 - 국제화 12-12, 12-14
 - 스레드 모델 36-7
 - COM 서버 33-7
- 심볼릭 링크 10-16

아

- 암호
 - 암시적 연결 20-13
 - dBASE 테이블 20-22 ~ 20-24

- Paradox 테이블 20-22 ~ 20-24
- 암호화
 - TSocketConnection 25-26
- 아날로그 비디오 8-32
- 아이콘 3-46, 6-21, 45-4
- 그래픽 객체 8-3
- 툴바 6-47
- 트리 뷰 3-39
- 아키텍처
 - 다계층 25-4, 25-5
 - 웹 기반 25-33
- 데이터베이스
 - 애플리케이션 14-6 ~ 14-14, 20-1 ~ 20-2
 - 서버 25-5
 - 클라이언트 25-4
 - BDE 기반 애플리케이션 20-1 ~ 20-2
 - Web Broker 서버 애플리케이션 28-3
- 안전 배열 34-13
- 알기 쉬운 툴바 6-47, 6-49
- 애니메이션 컨트롤 3-47, 8-29 ~ 8-31
- 예제 8-30
- 애플리케이션
 - 국제화 12-1
 - 그래픽 40-7, 45-1
 - 다계층 25-1 ~ 25-44
 - 개요 25-3 ~ 25-4
 - 다중 스레드 9-1
 - 데이터베이스 14-1
 - 만들기 3-23
 - 배포 13-1
 - 상태 정보 3-43, 3-44
 - 서비스 5-4
 - 웹 기반 클라이언트 애플리케이션 25-33 ~ 25-44
 - 웹 서버 5-11, 29-2
 - 크로스 플랫폼 10-1 ~ 10-30
 - 생성 10-1
 - 클라이언트/서버 25-1
 - 네트워크 프로토콜 20-16
 - 파일 13-2
 - 팔레트 실현 45-5
 - 포팅 10-17
 - 포팅 가능 10-1 ~ 10-30
 - Apache 27-7, 28-2, 29-2
 - CGI 독립형 29-2

- COM 5-14, 33-3 ~ 33-10, 33-18, 36-1 ~ 36-18
- ISAPI 27-6, 27-7, 28-1, 29-2
- MDI 5-2
- MTS 5-15
- NSAPI 27-6, 28-1, 28-2, 29-2
- SDI 5-2
- Web Broker 28-1 ~ 28-21
- WebSnap 29-1 ~ 29-4
- WebSnap 자습서 29-19
- Win-CGI 독립형 29-2
- 애플리케이션 국제화 12-1
- 애플리케이션 모듈 페이지 옵션 29-3
- 애플리케이션 변수 6-3, 28-3
- 애플리케이션 서버 14-13, 25-1, 25-12 ~ 25-23
- 다중 데이터 모듈 25-22
- 등록 25-11, 25-23
- 식별 25-24
- 연결 끊기 25-30
- 연결 열기 25-30
- 원격 데이터 모듈 5-19
- 인터페이스 25-17 ~ 25-18, 25-30
- 작성 25-13
- 애플리케이션 어댑터 29-4
- 콜백 25-18
- 애플리케이션 컴포넌트, 웹 애플리케이션 29-4
- 애플리케이션 포팅 10-1 ~ 10-30
- 액세스 위반
 - 문자열 4-46
- 액션
 - 웹 어댑터 29-8
- 액션 에디터
 - 액션 변경 28-6
 - 액션 추가 28-5
- 액션 요청
 - HTML 29-15
- 액션 응답 29-16
- 액션 항목 28-3, 28-4, 28-6 ~ 28-8
- 기본 28-5, 28-7
- 변경시 주의 28-3
- 선택 28-6, 28-7
- 연결 28-8
- 요청에 응답 28-8
- 웹 디스패처 29-18
- 이벤트 핸들러 28-4

- 추가 28-5
- 페이지 프로듀서 28-15
- 활성화 및 비활성화 28-7
- 액자 모양의 패널 3-46
- 앰퍼샌드(&) 문자 3-20, 6-34
- 양단 묶음 예제 8-23 ~ 8-29
- 양방향 애플리케이션
 - 메소드 12-8
 - 속성 12-6
- 양방향 속성 10-9
- 양방향 커서 18-49
- 어댑터
 - 레코드 29-8
 - 액션 29-8
 - 오류 29-8
 - 웹 애플리케이션 29-8
 - 필드 29-8
- 어댑터 디스패처 29-4
- 어댑터 디스패처 요청 29-15
- 어댑터 디스패처 컴포넌트 29-4
- 어댑터 컴포넌트
 - 사용 29-13
- 어셈블러 코드 10-21
- 업다운 컨트롤 3-34
- 업데이트
 - 작업 6-26
- 업데이트 객체 20-40 ~ 20-49, 23-19
 - 다중 사용 20-45 ~ 20-48
 - 매개변수 20-43, 20-47, 20-48 ~ 20-49
 - 실행 20-46 ~ 20-48
 - 쿼리 20-48 ~ 20-49
 - 프로바이더 20-11
 - SQL 문 20-41 ~ 20-45
- 업데이트 오류
 - 응답 메시지 25-40
 - 해결 23-21, 23-23 ~ 23-25, 24-8, 24-11 ~ 24-12
- 여러 줄의 텍스트 컨트롤 15-9
- 역할 기반 보안 39-14
- 연결
 - 끊기 25-30
 - 데이터베이스 17-2 ~ 17-5
 - 관리 20-19 ~ 20-22
 - 네트워크 프로토콜 20-16
 - 닫기 20-20
 - 비동기화 21-5
 - 열기 20-18, 20-20
 - 영구적 20-19
 - 이름 지정 22-4 ~ 22-5
 - 임시 20-21
 - 제한 25-8
 - 폴링 25-7, 39-5 ~ 39-6
 - 데이터베이스 서버 17-3, 20-15
 - 열기 25-30, 32-6
 - 종료 32-7
 - 클라이언트 32-3
 - 프로토콜 25-9 ~ 25-11, 25-24
 - CORBA 25-11, 25-28
 - DCOM 25-9, 25-25
 - HTTP 25-10, 25-27
 - SOAP 25-10, 25-27
 - TCP/IP 25-9 ~ 25-10, 25-25, 32-2 ~ 32-3
- 연결 매개변수 20-15
- 로그인 정보 17-4, 21-4
- ADO 21-3 ~ 21-4
- dbExpress 22-4, 22-5
- 연결 브로커링 (brokering) 25-28
- 연결 이름 22-4 ~ 22-5
 - 변경 22-5
 - 삭제 22-5
 - 정의 22-5
- 연결 컴포넌트
 - 데이터베이스 14-8 ~ 14-9, 17-1 ~ 17-14, 20-3
 - 각 연결에 대한 문장 22-3
 - 메타데이터 액세스 17-13 ~ 17-14
 - 바인딩 20-14 ~ 20-15, 21-2 ~ 21-4, 22-3 ~ 22-5
 - 암시적 17-2, 20-3, 20-13, 20-20, 21-3
 - 원격 데이터 모듈 25-6
 - ADO 21-2 ~ 21-9
 - BDE 20-13 ~ 20-16
 - dbExpress 22-2 ~ 22-5
 - SQL 명령 실행 17-10 ~ 17-12, 21-6
 - DataSnap 14-14, 25-3, 25-5, 25-23, 25-24 ~ 25-32
 - 연결 관리 25-29
 - 연결 끊기 25-30
 - 연결 열기 25-30
- 이름 지정 25-9 ~ 25-11, 25-24
- 연결 해제된 모델 14-14
- 연결된 선분 8-10
- 연쇄 삭제 24-6
- 연쇄 업데이트 24-6
- 열 3-44
 - 기본 상태 15-16, 15-21
 - 삭제 15-16
 - 속성 15-17, 15-20
 - 재설정 15-22
 - 영구적 15-15, 15-17 ~ 15-18
 - 삭제 15-19
 - 삽입 15-19
 - 생성 15-18 ~ 15-22
 - 재정렬 15-19
 - 의사 결정 그리드 16-11
 - HTML 테이블에 포함 28-20
- 열 헤더 3-42, 15-17, 15-20
- 열거 타입 42-2, 49-3
 - 상수 8-13
 - 선언 8-12
 - Type Library 에디터 34-9 ~ 34-10, 34-16, 34-23
- 영구적 열 15-15, 15-17 ~ 15-18
 - 삭제 15-16, 15-19
 - 삽입 15-19
 - 생성 15-18 ~ 15-22
 - 재정렬 15-19
- 영구적 필드 15-16, 19-3 ~ 19-16
 - 데이터 타입 19-6
 - 데이터 패킷 24-5
 - 데이터셋 필드 18-37
 - 동적으로 변환 19-3
 - 배열 필드 19-25
 - 삭제 19-10
 - 생성 19-4 ~ 19-5, 19-5 ~ 19-10
 - 속성 19-11 ~ 19-15
 - 열기 19-4, 19-5
 - 이름 지정 19-5
 - 정렬 19-5
 - 정의 19-5 ~ 19-10
 - 테이블 만들기 18-39
 - 특별 타입 19-5, 19-6
 - ADT 필드 19-24
- 예약어
 - 우선 바인딩 25-31
- 예약어 쓰기 49-4
- 예약어 발생 4-16

- 예약어 읽기 49-4
- 예약어 입력 8-12
- 예약어입니다. 3-10
- 예외 4-4 ~ 4-17, 10-15, 44-2, 46-3, 52-5
 - 발생 4-16
 - 사용자 정의 4-16
 - 스레드 9-6
 - 응답 4-5
 - 인스턴스 4-11
 - 재발생 4-13
 - 조용한 4-15
 - 중첩 4-6
 - 처리 4-5
 - 컴포넌트 4-14
 - 클래스 4-13
 - COM 인터페이스 34-9
 - RTL 4-9
- 예외 처리 4-4 ~ 4-17
 - 개요 4-4 ~ 4-17
 - 객체 선언 4-16
 - 기본 핸들러 4-12
 - 리소스 보호 블록 4-8
 - 리소스 할당 보호 4-7
 - 문장 4-10
 - 범위 4-12
 - 제어 흐름 4-6
 - 코드 블록 보호 4-4
 - 클린업 코드 실행 4-5
 - 핸들러 작성 4-10
 - TApplication 4-14
- 오디오 클립 8-31
- 오류
 - 소켓 32-8
 - 웹 어댑터 29-8
- 오류 메시지
 - 국제화 12-10
- 오버라이드
 - 메소드 41-8, 46-3, 46-4, 50-11
- 오버로드 내장 프로시저 20-12
- 오브젝트 파스칼 10-22
 - 개요 3-4
- 오프스크린 비트맵 45-6 ~ 45-7
- 온도 단위 4-62
- 온라인 도움말 47-4
- 읍선
 - 웹 애플리케이션 모듈 29-3
- 읍선 매개변수 23-16, 24-7
- 와이드 문자 12-3
 - 런타임 라이브러리 루틴 4-43
- 외국어 번역 12-1
- 외부 객체 33-9
- 요구불 폐치 23-27
- 요약 값
 - 유지 관리되는 집계 23-14
 - 의사 결정 그래프 16-15
 - 의사 결정 큐브 16-19
 - 크로스탭 16-3
- 요청
 - 디스패치 29-13
 - 액션과 HTML 29-15
 - 어댑터 29-15
 - 이미지
 - HTTP 요청 이미지 29-17
- 요청 객체
 - 헤더 정보 28-4
- 요청 디스패치
 - WebSnap 29-13
- 요청 메시지 28-3, 37-4
 - 디스패칭 28-5
 - 액션 항목 28-6
 - 응답 28-8, 28-12
 - 처리 28-5
 - 컨텐츠 28-11
 - 타입 28-10
 - 헤더 정보 28-9 ~ 28-11
 - HTTP 개요 27-5 ~ 27-6
 - XML 브로커 25-39
- 요청 목록(패키지) 11-6, 11-7, 11-9, 47-19
- 요청 헤더 28-9
- 우선 바인딩 25-31
 - Automation 33-18, 36-13
 - COM 33-16
- 우선 순위
 - 스레드 사용 9-1, 9-2
- 원, 그리기 49-9
- 원격 가능 클래스
 - 등록 31-5
 - 예외 31-7
- 원격 가능 클래스
 - 레지스트리 31-5
- 원격 데이터 모듈 5-19, 25-3, 25-5, 25-12, 25-13 ~ 25-17
 - 다중 25-22, 25-32 ~ 25-33
 - 부모 25-22
 - 상태 없는 25-20 ~ 25-22
 - 상태 없음 25-7, 25-20 ~ 25-22
 - 스레드 모델 25-14, 25-15
 - 인스턴스 25-14
- 자식 25-22
- 폴링 25-8
- 원격 데이터 모듈 마법사 25-13 ~ 25-14
- 원격 서버 20-9, 33-7
 - 승인되지 않은 액세스 17-4
- 원격 애플리케이션
 - TCP/IP 32-1
 - 연결 유지 20-19
- 원격 연결 32-2 ~ 32-3
 - 다중 32-5
 - 열기 32-6, 32-7
 - 정보 보내기/받기 32-9
 - 종료 32-7
- 원자성
 - 트랜잭션 14-4, 39-9
- 웹 데이터 모듈 29-3, 29-5
 - 구조 29-5
 - 인터페이스 29-6
- 웹 디스패처 28-2, 28-4 ~ 28-5
 - 액션 항목 29-18
 - 액션 항목 선택 28-6, 28-7
 - 요청 처리 28-3, 28-8
 - 자동 디스패칭 객체 25-39, 28-5
 - DLL 기반
 - 애플리케이션 28-3
- 웹 모듈 28-2 ~ 28-3, 28-4, 29-5 ~ 29-8
 - 데이터베이스 세션 추가 28-18
 - 타입 29-5
 - DLL 및 주의 28-3
- 웹 배포 38-15 ~ 38-17
 - 다계층 애플리케이션 25-35
- 웹 브라우저 27-4
 - URL 27-5
- 웹 브로커 5-11
- 웹 사이트 (Delphi 지원) 1-3
- 웹 서버 25-34, 27-1 ~ 27-9, 37-6
 - 디버깅 28-2
 - 클라이언트 요청 27-5
 - 타입
 - 웹 서버 타입 29-2
- 웹 서버 애플리케이션 5-11, 27-1 ~ 27-9
 - 개요 27-6 ~ 27-9
 - 다계층 25-32 ~ 25-44
 - 디버깅 27-7 ~ 27-9
 - 리소스 위치 27-3
 - 생성 29-2

- 타입 27-6
- 표준 27-2
- ASP 37-1
- 웹 서버 애플리케이션
 - 디버깅 28-2
- 웹 서비스 31-1 ~ 31-10
 - 구현 클래스 31-6 ~ 31-7
 - 등록 31-6
 - 네임스페이스 31-4
 - 복잡한 타입 31-5 ~ 31-6
 - 서버 31-2 ~ 31-8
 - 작성 31-2 ~ 31-3
 - 예외 31-7
 - 클라이언트 31-8 ~ 31-10
- 웹 서비스 importer 31-9
- 웹 스크립팅 29-9
- 웹 애플리케이션
 - 객체 28-3
 - 데이터베이스 25-33 ~ 25-44
 - 배포 13-10
 - 어댑터 29-8
 - ActiveX 33-13, 38-1, 38-15 ~ 38-17
 - 다계층 클라이언트 25-34
 - ASP 33-13, 37-1
- 웹 애플리케이션 디버거 27-7, 28-2, 29-2
- 웹 애플리케이션 모듈
 - 인터페이스
 - 인터페이스
 - 웹 모듈 29-7
- 웹 어댑터
 - 레코드 29-8
 - 액션 29-8
 - 오류 29-8
 - 필드 29-8
- 웹 연결 25-10, 25-27
- 웹 페이지 27-4
 - InternetExpress 페이지
 - 프로듀서 25-40 ~ 25-44
- 웹 페이지 모듈 29-3, 29-6
 - 인터페이스 29-7
- 웹 페이지 모듈 작성 29-24
- 웹 페이지 에디터 25-41 ~ 25-42
- 웹 항목 25-41
 - 속성 25-42 ~ 25-43
- 웹에 배포 38-15 ~ 38-17
- 위임(delegation) 43-1
- 위치 독립 코드(PIC) 10-9, 10-21
- 윈도우
 - 프로시저 46-2, 46-3
 - 유니코드 문자 4-39, 12-3
 - 문자열 4-41, 4-43
 - 유닛
 - 기존
 - 컴포넌트 추가 40-11
 - 다른 유닛에서 액세스 3-9
 - 컴포넌트 추가 40-11
 - 패키지 포함 11-3
 - 유로화 변환 4-63
 - 유럽 통화 변환 4-65
 - 유지 관리되는 집계 14-15, 23-12 ~ 23-14
 - 값 23-14
 - 요약 연산자 23-12
 - 지정 23-12 ~ 23-13
 - 집계(aggregate) 필드 19-10
 - 합계 23-13
 - 유효 범위(객체) 3-8 ~ 3-9
 - 윤곽, 그리기 8-5
 - 윤년 50-8
 - 응답
 - 액션 29-16
 - 어댑터 29-15
 - 이미지
 - HTTP 응답
 - 이미지 29-17
 - 응답 메시지 28-3, 37-5
 - 데이터베이스 정보 28-17 ~ 28-21
 - 보내기 28-8, 28-13
 - 상태 정보 28-11
 - 생성 28-11 ~ 28-13, 28-13 ~ 28-21
 - 컨텐츠 28-12, 28-13 ~ 28-21
 - 헤더 정보 28-11 ~ 28-12
 - 응답 템플릿 28-14
 - 응답 헤더 28-12
 - 의사 결정 그래프 16-13 ~ 16-18
 - 그래프 타입 16-16
 - 데이터 시리즈 16-17 ~ 16-18
 - 런타임 동작 16-19
 - 사용자 지정 16-15 ~ 16-18
 - 차원 16-14
 - 템플릿 16-16
 - 표시 옵션 16-15
 - 피벗 상태 16-8, 16-9
 - 의사 결정 그리드 16-10 ~ 16-13
 - 런타임 동작 16-19
- 속성 16-12
- 이벤트 16-12
- 차원
 - 드릴 다운 16-11
 - 선택 16-12
 - 열기/닫기 16-11
 - 재정렬 16-11
 - 피벗 상태 16-8, 16-9, 16-11
- 의사 결정 데이터셋 16-4 ~ 16-6
- 의사 결정 소스 16-8 ~ 16-9
 - 속성 16-9
 - 이벤트 16-9
- 의사 결정 지원 컴포넌트
 - 14-15, 16-1 ~ 16-20
 - 데이터 할당 16-4 ~ 16-6
 - 디자인 옵션 16-8
 - 런타임 16-18 ~ 16-19
 - 메모리 관리 16-19
 - 추가 16-3 ~ 16-4
- 의사 결정 쿼리
 - 정의 16-6
- 의사 결정 큐브 16-7 ~ 16-8
 - 데이터 가져오기 16-4
 - 데이터 표시 16-9, 16-11
 - 드릴 다운 16-5, 16-9, 16-11, 16-20
 - 디자인 옵션 16-8
 - 메모리 관리 16-8
 - 부분 합계 16-5
 - 새로 고침 16-7
 - 속성 16-7
- 차원
 - 열기/닫기 16-9
 - 페이지화된 16-20
 - 차원 맵 16-5, 16-7, 16-19, 16-20
 - 피벗 16-5, 16-9
- 의사 결정 피벗 16-9 ~ 16-10
 - 런타임 동작 16-18
 - 방향 16-10
 - 속성 16-10
 - 차원 버튼 16-10
- 유니온
 - Type Library 에디터
 - 34-10, 34-18, 34-23 ~ 34-24
- 이름 지정 규칙
 - 메소드 44-2
 - 메시지 레코드 타입 46-6
 - 속성 42-6
 - 이벤트 43-8
 - 필드 43-2

- 이미지 3-46, 15-2, 45-3
 - 국제화 12-10
 - 그리기 49-8
 - 다시 그리기 45-7
 - 다시 생성 8-2
 - 메뉴에 추가 6-36
 - 변경 8-20
 - 브러시 8-9
 - 스크롤 8-17
 - 저장 8-20
 - 지우기 8-22
 - 추가 8-17
 - 컨트롤 8-2, 8-17
 - 컨트롤 추가 7-14
 - 툴 버튼 6-47
 - 표시 3-46
 - 프레임 6-15
 - 화면 펼침 감소 45-6
- 이미지 다시 그리기 45-7
- 이미지 요청 29-17
- 이벤트 3-26 ~ 3-29, 10-21, 40-6, 43-1 ~ 43-9
 - 객체 3-10
 - 검색 43-3
 - 공유 3-28
 - 구현 43-2, 43-4
 - 그래픽 컨트롤 45-7
 - 기본 3-26
 - 대기 9-9
 - 데이터 그리드 15-26 ~ 15-27
 - 데이터 소스 15-4
 - 도움말 제공 47-4
 - 로그인 17-5
 - 마우스 8-24 ~ 8-26
 - 테스트 8-26
 - 메시지 처리 46-3, 46-6
 - 사용자 3-3
 - 상속 43-5
 - 새로운 정의 43-6 ~ 43-9
 - 시간 초과 9-10
 - 시스템 3-4
 - 신호 지정 9-10
 - 애플리케이션 레벨 6-3
 - 액세스 43-5
 - 응답 43-6, 43-7, 43-9, 51-6
 - 이름 지정 43-8
 - 인터페이스 36-11
 - 크로스 플랫폼 10-23
 - 타입 3-3
 - 표준 43-4, 43-4 ~ 43-6
 - 필드 객체 19-15 ~ 19-16
 - 핸들러에 연결 3-27
- ActiveX 컨트롤 38-10
- ADO 연결 21-8 ~ 21-9
- Automation 객체 36-5
- Automation 컨트롤러 35-10, 35-14 ~ 35-16
- COM 36-10, 36-12
- COM 객체 36-10 ~ 36-12
 - 컴포넌트 랩퍼 35-2
- COM+ 35-15 ~ 35-16, 39-18 ~ 39-20
 - 발생 39-20
 - data-aware 컨트롤 사용 가능 15-7
- XML 브로커 25-40
- 이벤트 객체 9-9
 - COM+ 39-19
- 이벤트 공지 6-5
- 이벤트 신호 지정 9-10
- 이벤트 싱크 36-12
 - 정의 35-14 ~ 35-15
- 이벤트 핸들러 3-7, 3-26 ~ 3-29, 10-22, 40-6, 43-2, 51-6
 - 공유 3-27 ~ 3-28, 8-15
 - 기본, 오버라이드 43-9
 - 매개변수 43-3, 43-7, 43-8, 43-9
 - 공지 이벤트 43-7
- 메뉴 3-28, 7-12
 - 템플릿 6-41
- 메소드 43-3, 43-5
 - 오버라이드 43-6
- 버튼 클릭에 응답 8-13
- 빈 43-9
 - 삭제 3-29
- 선 그리기 8-26
- 선언 43-5, 43-8, 50-11
- 쓰기 3-26
- 위치 3-27
- 이벤트에 연결 3-27
- 작성 3-9, 3-26
- 정의 3-26
 - 참조에 의한 매개변수 전달 43-9
- 코드 에디터 표시 47-18
- 타입 43-3 ~ 43-4, 43-7 ~ 43-8
 - 포인터 43-2, 43-3, 43-8
- Sender 매개변수 3-27
- 이중 쿼리 20-9 ~ 20-10
 - 로컬 SQL 20-9
- 이중 인터페이스 36-13 ~ 36-14
 - 매개변수 36-16
- 메소드 호출 35-13
- 타입 호환성 36-15
- 트랜잭션 객체 39-3, 39-16
- Active Server Object 37-3
- 인덱스 18-26 ~ 18-37, 42-8
 - 데이터 그룹화 23-10
 - 레코드 정렬 18-26 ~ 18-27, 23-8
 - 마스터/디테일 관계 18-35
 - 배치 이동 20-51
 - 범위 18-31
 - 부분 키 검색 18-30
 - 삭제 23-9
 - 열거 17-14, 18-26
 - 지정 18-27
 - 클라이언트 데이터셋 23-8 ~ 23-10
 - dBASE 테이블 20-6 ~ 20-7
- 인덱스 기반 검색 18-11, 18-12, 18-28 ~ 18-30
- 인덱스 정의 18-39
 - 복사 18-40
- 인덱스 파일 20-6
- 인덱싱되지 않은 데이터셋 18-19, 18-21
- 인쇄 3-57
- 인스턴스 43-2
 - 원격 데이터 모듈 25-14
 - COM 객체 36-5 ~ 36-6
 - CORBA 데이터 모듈 25-16
- 인터넷 서버 27-1 ~ 27-9
- 인터넷 표준 및 프로토콜 27-2
- 인터페이스 4-17 ~ 4-26, 41-4, 41-5, 52-1, 52-3
 - 개요 4-17 ~ 4-26
 - 객체 소멸 4-24
 - 구현 33-6, 36-3
 - 국제화 12-9, 12-10, 12-14
 - 내부 객체 4-23
 - 년비주얼 프로그램 요소 40-5
- 다형성 4-18
- 단일 상속 확장 4-17, 4-18
- 도움말 시스템 5-10, 5-23
- 동적 바인딩 4-21, 34-9, 36-12
- 동적 쿼리 4-20
- 동적 호출 인터페이스 4-26

디스패치 36-14
 디자인 타임 41-6
 랭귀지 기능 4-17
 런타임 41-6
 메모리 관리 4-21, 4-24
 메소드 추가 36-10
 마샬링 4-26
 분산 애플리케이션 4-26
 사용 4-17 ~ 4-26
 사용자 지정 36-15
 속성 42-10
 속성 추가 36-9 ~ 36-10
 속성, 선언 52-3
 수명 관리 4-20, 4-24
 애플리케이션 서버 25-17
 ~ 25-18, 25-30
 예제 코드 4-18, 4-22,
 4-24
 우선 바인딩 25-31
 웹 데이터 모듈 29-6
 웹 서비스 31-1
 웹 페이지 모듈 29-7
 인터페이스, 구현 4-20
 지연 바인딩 25-31
 참조 카운팅 4-20, 4-21,
 4-23 ~ 4-25
 추상화(aggregation)
 4-22, 4-23
 컴포넌트 4-25
 코드 재사용 4-22
 코드 최적화 4-25
 클래스 간 공유 4-18
 타입 라이브러리 33-12,
 33-17, 35-5, 36-9
 파생 4-20
 프로시저 4-19
 호출 가능 4-26, 31-2,
 31-3 ~ 31-4
 ActiveX 33-19
 사용자 지정 38-7 ~
 38-11
 as 연산자 4-21
 Automation 36-12 ~
 36-15
 CLSID 4-26
 COM 4-26, 5-14, 33-1,
 33-3 ~ 33-5, 34-8,
 35-1, 36-3, 36-9 ~
 36-15
 선언 35-5
 이벤트 36-11
 COM+ 이벤트 객체 39-19
 CORBA 4-26
 Ctrl+Shift+G 4-22
 DOM 30-2

IID 4-22, 4-25
 outgoing 36-11, 36-12
 SOAP 4-26
 TComponent 4-25
 Type Library 에디터
 34-15, 34-20, 36-9
 Unknown 제어 4-23,
 4-25
 XML 노트 30-4
 인터페이스 및 WebSnap
 29-6, 29-7
 인터페이스 키워드 4-17
 인터페이스 포인터 33-5
 인트라넷
 호스트 이름 32-4
 일관성
 트랜잭션 14-4, 39-9
 일대다 관계 18-35, 22-12
 일반(simple) 타입 42-2
 일시적 예약 35-15
 읽기 전용 데이터셋
 업데이트 14-10
 읽기 전용 속성 41-6, 41-7,
 42-7, 51-2
 읽기 전용 테이블 18-38
 읽기 전용 필드 15-5
 임계 구역 9-7
 사용 경고 9-8
 임시 객체 45-6
 입/출력 매개변수 18-51
 입력 매개변수 18-51
 입력 컨트롤 3-33
 입력 포커스 40-4
 필드 19-16

자

자바스크립트 라이브러리
 25-37
 위치 25-36, 25-37
 자손 클래스 3-8, 41-3 ~
 41-4
 메소드 재정의 41-8
 자습서
 WebSnap 29-19
 자식 컨트롤 3-19
 작업 6-23 ~ 6-50
 등록 6-28
 실행 6-24
 업데이트 6-26
 이미 정의된 6-28
 작업 클래스 6-27
 작업 목록 3-27, 6-17,
 6-19, 6-23 ~ 6-50
 작업 밴드 6-18
 작업 클라이언트 6-17

장치 독립형 그래픽 45-1
 장치 컨텍스트 8-1, 40-7,
 45-1
 재배치 가능 코드 10-21
 저장 매체 3-58
 전역 오프셋 테이블
 (GOT) 10-21
 절단 주소 10-9
 중단 유지 21-6
 중첩 디테일 18-37, 19-26 ~
 19-27, 25-20
 요구 즉시 페치 24-6
 중첩 테이블 18-37, 19-26 ~
 19-27, 25-20
 점진적 페치 23-27, 25-20
 정렬 순서 12-10
 내림차순 23-9
 설정 18-27
 클라이언트 데이터셋 23-8
 TSQLTable 22-7
 정사각형, 그리기 49-9
 정적 메소드 41-7
 정적 바인딩 25-31
 COM 33-16
 정적 텍스트 3-43, 3-44
 정적 텍스트 컴포넌트 3-43
 제약 조건
 데이터 19-21 ~ 19-22
 사용 불가능 23-30
 생성 19-21
 클라이언트 데이터
 셋 23-7 ~ 23-8,
 23-30 ~ 23-31
 import하기 19-21 ~
 19-22, 23-30,
 24-13
 컨트롤 6-4 ~ 6-5
 조건부 컴파일 10-18, 10-19
 조상 클래스 3-5, 3-8, 41-3
 ~ 41-4
 기본 41-4
 조회 값 15-18
 조회 리스트 박스 15-2,
 15-12 ~ 15-13
 보조 데이터 소스 15-12
 조회 필드 15-12
 조회 콤보 박스 15-2, 15-12
 ~ 15-13
 데이터 그리드 15-21
 보조 데이터 소스 15-12
 조회 필드 15-12
 채우기 15-21
 조회 필드 15-12, 19-6
 값을 캐시로 저장 19-9
 데이터 그리드 15-21

- 성능 19-9
- 정의 19-8 ~ 19-10
- 지정 15-21
- 프로그램으로 값 제공 19-9
- 좌표
 - 현재 그리기 위치 8-25
- 주소
 - 소켓 연결 32-3, 32-4
- 줄 끝 문자 10-14
- 증가 검색 15-11
- 지속성
 - 리소스 디스펜서 39-5
 - 트랜잭션 14-5, 39-9
- 지속적 예약 35-15
- 지시어 10-19
 - \$ELSEIF 10-19
 - \$ENDIF 10-19
 - \$H 컴파일러 4-41, 4-49
 - \$IF 10-19
 - \$IFDEF 10-18
 - \$IFEND 10-19
 - \$IFNDEF 10-19
 - \$LIBPREFIX 컴파일러 5-9
 - \$LIBSUFFIX 컴파일러 5-9
 - \$LIBVERSION 컴파일러 5-9
 - \$MESSAGE 컴파일러 10-20
 - \$P 컴파일러 4-49
 - \$V 컴파일러 지시어 4-50
 - \$X 컴파일러 4-50
 - 가상 41-8
 - 기본 42-12, 48-3
 - 동적 41-9
 - 문자열 관련 4-49
 - 조건부 컴파일 10-18
 - override 41-8, 46-4
 - protected 43-5
 - public 43-5
 - published 42-3, 43-5, 52-3
 - stored 42-12
- 지역화 12-14
 - 리소스 12-11, 12-12, 12-14
 - 애플리케이션 지역화 12-1
- 자연 바인딩 25-31
 - Automation 36-12, 36-14
- 지원 서비스 1-3
- 진행 표시줄 3-44
- 집계 필드
 - 정의 19-10

- 집계(aggregate) 필드 19-6, 23-14
 - 표시 19-10
- 집합 42-2
- 집합 타입 42-2
- 집합체
 - COM 33-9
- 짧은 문자열 4-40
- as-soon-as-possible
- 비활성화 25-7

차

- 차단 연결 32-10
 - 비차단 연결과 비교 32-9
 - 이벤트 처리 32-9
- 참조
 - 패키지 11-3
 - 폼 6-2
 - 참조 무결성 14-5
 - 참조 카운팅
 - 인터페이스 4-23 ~ 4-25
 - COM 객체 4-20, 33-4
 - 참조 필드 19-22, 19-27 ~ 19-28
 - 표시 15-24
 - 참조별 작성
 - COM 인터페이스 속성 34-8
 - 참조별 전용
 - COM 인터페이스 속성 34-8
- 창 3-34
 - 메시지 처리 50-4
 - 컨트롤 40-3
 - 크기 조정 3-34
 - 핸들 40-3, 40-5
- 찾기
 - 파일 4-57
- 채우기 패턴 8-8
- 체크 리스트 박스 3-38
- 체크 박스 3-36, 15-2
 - data-aware 15-13 ~ 15-14
- 최종 사용자 어댑터 29-4
- 추상 클래스 40-3
- 추상화(aggregation)
 - 인터페이스 4-23
 - 클라이언트 데이터셋 23-12 ~ 23-14
- 출력 매개변수 18-51, 23-28
- 측정
 - 변환 4-59

카

- 컴보 박스 3-39, 10-8, 15-2, 15-12
 - 조회 15-21
 - data-aware 15-10 ~ 15-13
 - owner-draw 7-12
 - measure-item
 - 이벤트 7-15
- 캐비닛 38-16
- 캐시된 업데이트 23-16 ~ 23-25
 - 개요 23-17 ~ 23-18
 - 마스터/디테일 관계 23-19
 - 업데이트 객체 23-19
 - 클라이언트 데이터셋 14-10, 23-16, 23-20 ~ 23-25
 - 업데이트 오류 23-23 ~ 23-25, 24-11 ~ 24-12
 - 읽기 전용 데이터셋
 - 업데이트 20-11
 - 적용 20-11, 23-21 ~ 23-22
 - 다중 테이블 20-41, 20-45
 - 트랜잭션 17-6
 - 프로마이더 24-8 ~ 24-9
 - ADO 21-12 ~ 21-15
 - 적용 21-14
 - 취소 21-14 ~ 21-15
 - BDE 20-33 ~ 20-49
 - 오류 처리 20-38 ~ 20-40
 - 읽기 전용 데이터셋
 - 업데이트 20-11
 - 적용 20-11, 20-35 ~ 20-38
 - 다중 테이블 20-41, 20-45
- 캔버스 40-7, 45-1
 - 개요 8-1 ~ 8-3
 - 그리기와 색칠 비교 8-4
 - 기본 드로잉 툴 49-5
 - 도형 추가 8-11 ~ 8-12, 8-14
 - 선 그리기 8-5, 8-10 ~ 8-11, 8-27 ~ 8-29
 - 이벤트 핸들러 8-26
 - 펜 너비 변경 8-6
 - 일반적인 속성, 메소드 8-4
 - 팔레트 45-5 ~ 45-6
 - 화면 새로 고침 8-2

캡슐화 3-5
 커밋 유지 21-6
 커서 18-5
 양방향 18-49
 단방향 18-49
 동기화 18-42
 복제 23-15
 연결 18-35, 22-12 ~ 22-13
 이동 18-6, 18-7, 18-28, 18-29
 마지막 행으로 이동 18-6, 18-7
 조건 18-11
 첫 행으로 이동 18-6, 18-8
 커서 드래그 7-2
 컨택스트 ID 5-26
 컨택스트 메뉴
 메뉴 디자이너 6-37
 툴바 6-49
 항목 추가 47-16 ~ 47-17
 컨택스트 번호(도움말) 3-44
 콘텐츠 속성
 웹 응답 객체 28-12
 콘텐츠 프로듀서 28-4, 28-13
 이벤트 처리 28-15, 28-16, 28-17
 컨트롤 3-2, 3-12 ~ 3-21
 그래픽 45-4, 49-1 ~ 49-10
 그리기 49-2 ~ 49-10
 생성 40-4, 49-2
 이벤트 45-7
 그룹화 3-40 ~ 3-42
 다시 그리기 49-7, 49-9, 50-4
 데이터 찾아보기 51-1 ~ 51-7
 데이터 편집 51-7 ~ 51-11
 도형 49-7
 변경 40-3
 사용자 지정 40-4
 위치 3-19
 이동 3-19, 3-22
 창 40-3
 크기 3-19
 크기 조정 45-7, 50-4
 팔레트 45-5 ~ 45-6
 포커스 받기 40-4
 표시 옵션 3-19
 표준
 데이터 표시 15-4, 19-17, 19-17
 ActiveX 컨트롤 구현 38-3
 ActiveX 컨트롤 생성 38-2, 38-4 ~ 38-6
 data-aware 15-1 ~ 15-31
 owner-draw 7-12, 7-15
 선언 7-13
 컨트롤 다시 그리기 49-7, 49-9, 50-4
 컨트롤 도킹 해제 7-6
 컨트롤 크기 조정 3-34, 13-13, 50-4
 그래픽 45-7
 컴파일 타임 오류
 override 지시어 41-8
 컴파일러 옵션 5-3
 컴파일러 지시어 10-19
 라이브러리 5-9
 문자열 4-49
 패키지 특정 11-10
 컴파일러 지시어, 문자열 및 문자 타입 4-49
 컴포넌트 3-1, 3-12 ~ 3-21, 40-1, 41-1, 42-3
 공통 속성 3-18 ~ 3-20, 3-21 ~ 3-22
 그룹화 3-40 ~ 3-42
 기존 유닛에 추가 40-11
 년비주얼 3-48, 40-5, 40-11, 52-2
 더블 클릭 47-15, 47-17 ~ 47-18
 데이터 찾아보기 51-1 ~ 51-7
 데이터 편집 51-7 ~ 51-11
 등록 40-12, 47-2
 디스패처 29-13
 리소스, 해제 52-5
 메모리 관리 3-12
 변경 48-1 ~ 48-3
 사용자 지정 3-31, 6-12, 40-3, 42-1, 43-1
 생성 40-2, 40-8
 설치 3-31, 11-5 ~ 11-6, 47-19
 소유권 3-12
 온라인 도움말 47-4
 웹 애플리케이션 29-4
 유닛에 추가 40-11
 이름 재지정 3-7 ~ 3-8
 이벤트에 응답 43-6, 43-7, 43-9, 51-6
 인터페이스 41-4, 41-5, 52-1
 디자인 타임 41-6
 런타임 41-6
 종속성 40-5
 초기화 42-13, 49-7, 51-6
 추상 40-3
 컨택스트 메뉴 47-15, 47-16 ~ 47-17
 컴포넌트 팔레트에 추가 47-1
 크기 조정 3-34
 크로스 플랫폼 3-29
 테스트 40-12, 40-14, 52-6
 파생된 클래스 40-3, 40-11, 49-2
 팔레트 비트맵 47-3
 패키지 47-19
 페이지 프로듀서 29-6
 표준 3-29 ~ 3-31
 data-aware 51-1
 컴포넌트 그룹화 3-40 ~ 3-42
 컴포넌트 랩퍼 40-4, 52-2
 초기화 52-3
 ActiveX 컨트롤 35-6 ~ 35-7, 35-8 ~ 35-9
 Automation 객체 35-7 ~ 35-8
 예제 35-10 ~ 35-12
 COM 객체 35-1, 35-2, 35-3, 35-6 ~ 35-12
 컴포넌트 사용자 지정 42-1
 컴포넌트 인터페이스
 생성 52-3
 속성, 선언 52-3
 컴포넌트 템플릿 6-12, 6-13, 41-2
 프레임 6-14, 6-15
 컴포넌트 팔레트 3-29
 컴포넌트 추가 11-6, 47-1, 47-3
 페이지 목록 3-29
 프레임 6-14
 ActiveX 페이지 3-31, 35-4
 Additional 페이지 3-29
 ADO 페이지 3-30, 14-1, 21-1
 BDE 페이지 3-30, 14-1
 Data Access 페이지 3-29, 14-2, 25-2

Data Controls 페이지 14-15, 15-1, 15-2
 DataSnap 페이지 3-30, 25-2, 25-6
 dbDirect 페이지 14-2
 dbExpress 페이지 3-30, 22-2
 Decision Cube 페이지 14-15, 16-1
 Dialogs 페이지 3-30
 FastNet 페이지 3-30
 Indy Clients 페이지 3-30
 Indy Misc 페이지 3-31
 Indy Servers 페이지 3-31
 InterBase 페이지 3-30, 14-2
 Internet 페이지 3-30
 InternetExpress 페이지 3-30
 QReport 페이지 3-30
 Samples 페이지 3-30, 3-31
 Servers 페이지 3-30, 3-31
 Standard 페이지 3-29
 System 페이지 3-29
 Win 3.1 페이지 3-30
 Win32 페이지 3-29
 컴포넌트 에디터 47-15 ~ 47-19
 기본 47-16
 등록 47-19
 코드 44-3
 포팅 가능 10-16, 10-17
 코드 디버깅 2-5
 코드 에디터 2-4
 개요 2-4
 이벤트 핸들러 3-27
 패키지 열기 11-8
 표시 47-18
 코드 최적화
 인터페이스 4-25
 코드 컴파일 2-4
 코드 템플릿 5-3
 코드 페이지 12-2
 코드 편집 2-3, 2-4
 콘솔 애플리케이션 5-3
 CGI 27-7
 콜백
 다계층 애플리케이션 25-18
 제한 25-10, 25-11
 트랜잭션 객체 39-21
 쿨마 3-37, 6-43
 구성 6-49
 디자인 6-43 ~ 6-50
 숨기기 6-50
 추가 6-48 ~ 6-49
 쿼리 18-23, 18-42 ~ 18-50
 결과 집합 18-49
 단방향 커서 18-49
 데이터베이스 지정 18-42
 마스터/디테일 관계 18-47 ~ 18-48
 매개 변수화된 18-43
 매개변수 18-45 ~ 18-47
 디자인 타임 시 설정 18-45
 런타임 시 설정 18-46
 마스터/디테일 관계 18-47 ~ 18-48
 명명 18-45
 명명되지 않은 18-45
 바인딩 18-45
 속성 18-46
 클라이언트 데이터 셋 23-29
 실행 18-49
 업데이트 객체 20-47
 양방향 커서 18-49
 업데이트 객체 20-48 ~ 20-49
 웹 애플리케이션 28-21
 이중 20-9 ~ 20-10
 준비 18-48
 지정 18-43 ~ 18-44, 22-6
 최적화 18-48, 18-49
 필터와 비교 18-13
 BDE 기반 20-2, 20-9 ~ 20-11
 동시 20-18
 라이브 결과 집합 20-10 ~ 20-11
 HTML 테이블 28-21
 쿼리 부분(URL) 27-3
 크로스 플랫폼
 애플리케이션 3-29, 10-1 ~ 10-30
 생성 10-1
 크로스 플랫폼 개발 6-17, 6-19
 크로스탭 16-2 ~ 16-3, 16-10
 1차원 16-3
 다차원 16-3
 요약 값 16-3
 정의 16-2
 클라이언트 애플리케이션
 네트워크 프로토콜 20-16
 다계층 25-2, 25-4
 사용자 인터페이스 25-1
 생성 25-23 ~ 25-32, 35-1 ~ 35-16
 소켓 32-1
 윈 25-2, 25-33
 아키텍처 25-4
 웹 서버 애플리케이션 25-33
 웹 서비스 31-8 ~ 31-10
 인터페이스 32-2
 쿼리 제공 24-6
 타입 라이브러리 34-20, 35-2 ~ 35-4
 트랜잭션 객체 39-2
 COM 33-3, 33-9, 35-1 ~ 35-16
 클라이언트 데이터셋 23-1 ~ 23-36, 25-3
 계산된 필드 23-11
 다른 데이터셋에 연결 14-10 ~ 14-12, 23-25 ~ 23-32
 단방향 데이터셋 22-11
 데이터 공유 23-15
 데이터 그룹화 23-10
 데이터 복사 23-14 ~ 23-15
 데이터 집계 23-12 ~ 23-14
 데이터 합병 23-15
 레코드 업데이트 23-20 ~ 23-25
 레코드 새로 고침 23-31
 레코드 제한 23-29 ~ 23-30
 레코드 필터링 23-3 ~ 23-5
 매개변수 23-28 ~ 23-30
 배포 13-6
 변경 내용 저장 23-6
 변경 취소 23-6
 변경 합병 23-35
 업데이트 오류 해결 23-21, 23-23 ~ 23-25
 업데이트 적용 23-21 ~ 23-22
 인덱스 23-8 ~ 23-10
 추가 23-8
 인덱스 기반 검색 18-28
 인덱스 삭제 23-9
 인덱스 전환 23-9
 제약 조건 23-7 ~ 23-8, 23-30 ~ 23-31

- 사용 불가능 23-30
- 쿼리 제공 23-32 ~ 23-33
- 타입 23-18 ~ 23-19
- 탐색 23-2
- 테이블 생성 23-34
- 파일 기반
 - 애플리케이션 23-33 ~ 23-36
- 파일 로드 23-34
- 파일 저장 23-35 ~ 23-36
- 편집 23-5
- 프로바이더 23-25 ~ 23-32
- 프로바이더 지정 23-25 ~ 23-26
- 클라이언트 소켓 32-3, 32-6 ~ 32-7
 - 서버 식별 32-6
 - 서버에 연결 32-8
 - 서비스 요청 32-6
 - 속성 32-6
 - 오류 메시지 32-8
 - 이벤트 처리 32-8
 - 호스트 지정 32-4
 - Windows 소켓 객체 32-6
- 클라이언트 연결 32-2, 32-3
- 열기 32-6
 - 요청 승인 32-7
 - 포트 번호 32-5
- 클라이언트 요청 27-5 ~ 27-6, 28-9
- 클라이언트
 - 애플리케이션 참조
- 클라이언트, 작업 목록 6-18
- 클라이언트/서버
 - 애플리케이션 5-10
- 클래스 4-1, 40-2, 40-3, 41-1, 42-2
 - 계층 구조 41-3
 - 기본 41-4
 - 매개변수로 전달 41-9
 - 상속 41-7
 - 새로 파생 41-2, 41-3, 41-8
 - 생성 41-1
 - 속성 42-2
 - 속성 편집기 47-7
 - 액세스 41-4 ~ 41-7, 49-6
 - 인스턴스화 41-2
 - 자손 41-3 ~ 41-4, 41-8
 - 정의 40-11, 41-2
 - 가상 메소드 41-8
 - 정적 메소드 41-7
 - 조상 41-3 ~ 41-4

- 추상 40-3
- 파생 41-8
- protected 부분 41-5
- public 부분 41-6
- published 부분 41-6
- 클래스 파생 41-8
- 클래스 팩토리 33-6
- 마법사에 의해 추가 36-2
- 클래스 포인터 41-9
- 클래스 필드 49-3
 - 선언 49-5
 - 이름 지정 43-2
- 클래스에서 상속 3-8 ~ 3-12, 41-7
- 클릭 이벤트 8-25, 43-1, 43-2, 43-7
- 클립보드 7-8, 7-10, 15-9
 - 그래픽 8-21 ~ 8-23
 - 그래픽 객체 8-3, 15-10
 - 내용 확인 7-11
 - 선택 부분 지우기 7-10
 - 이미지 테스트 8-23
- 형식
 - 추가 47-15, 47-18
- 키 다운 메시지 43-5, 51-8
- 키 위반 20-53
- 키 필드 18-33
 - 여러 18-32, 18-33
- 키보드 단축키 3-34
- 키보드 매핑 12-9, 12-10
- 키보드 이벤트 43-3, 43-9
 - 국제화 12-9
- 키워드 47-5
 - protected 43-5
- 키워드 방식 도움말 5-26

타

- 타원
 - 그리기 8-11, 49-9
- 타이머 3-24
- 타입
 - 메시지 레코드 46-6
 - 사용자 정의 49-3
 - 속성 42-2, 42-8, 47-8
 - 웹 모듈 29-5
 - 웹 서비스 31-5 ~ 31-6
 - 타입 라이브러리 34-11 ~ 34-13
 - Automation 36-15 ~ 36-16
 - Char 12-2
 - 타입 라이브러리 33-11, 33-12, 33-15 ~ 33-18, 34-1 ~ 34-27

- _TLB 유닛 33-23, 34-2, 34-20, 35-2, 35-5, 36-15
- 객체 등록 33-18
- 내용 35-5
- 도구 33-18
- 등록 33-18, 34-26
- 등록 해제 33-18
- 리소스로 포함 34-26 ~ 34-27, 38-3
- 마법사에서 생성 34-1
- 배포 34-26 ~ 34-27
- 브라우저 33-18
- 사용 시기 33-16 ~ 33-17
- 생성 33-16, 34-19
- 설치 해제 33-18
- 성능 최적화 34-8
- 액세스 33-17, 34-19 ~ 34-20, 35-2
- 열기 34-19 ~ 34-20
- 유효한 타입 34-11 ~ 34-13
- 이점 33-17 ~ 33-18
- 인터페이스 33-17
- 인터페이스 수정 34-20 ~ 34-22
- 저장 34-25
- 찾아보기 33-18
- 추가
 - 메소드 34-21 ~ 34-22
 - 속성 34-21 ~ 34-22
 - 인터페이스 34-20
- 컨텐츠 33-16, 34-1
- 타입 확인 33-17
- 트랜잭션 객체 39-3
- Active Server Object 37-3
- ActiveX 컨트롤 38-3
- IDL 및 ODL 33-16
- import하기 35-2 ~ 35-3
- 타입 선언
 - 객체 3-10
 - 속성 49-3
 - 열거 타입 8-12
- 타입 정보 33-16, 34-1
 - dispinterfaces 36-13
 - IDispatch 인터페이스 36-14
 - import하기 35-2 ~ 35-3
 - 도움말 34-8
- 타입 정의
 - Type Library 에디터 34-9 ~ 34-10
- 탐색기 15-2, 15-28 ~ 15-31, 18-5, 18-6

- 데이터 삭제 18-20
- 데이터셋끼리 공유 15-31
- 도움말 힌트 15-30
- 버튼 15-29
- 버튼 사용 가능/사용 불가능 15-29, 15-30
- 편집 18-18
- 탭
 - draw-item 이벤트 7-16
- 탭 순서 3-22
- 탭 집합 3-42
- 탭 컨트롤 3-42
 - owner-draw 7-12
- 터널 유형 10-15
- 테두리
 - 패널 3-41
- 테스트
 - 값 42-7
 - 컴포넌트 40-12, 40-14, 52-6
- 테스트 서버, 웹 애플리케이션 디버거 29-2
- 테이블 18-23, 18-25 ~ 18-42
 - 검색 18-28 ~ 18-30
 - 그리드에 표시 15-16
 - 데이터베이스 이외의
 - 그리드 3-44
 - 데이터베이스 지정 18-25
 - 동기화 18-42
 - 레코드 삽입 18-18 ~ 18-19, 18-21
 - 마스터/디테일 관계 18-35 ~ 18-37
 - 만들기 18-38 ~ 18-41
 - 영구적 필드 18-39
 - 인덱스 18-39
 - 범위 18-30 ~ 18-34
 - 비우기 18-41
 - 삭제 18-41
 - 열거 17-13
 - 인덱스 18-26 ~ 18-37
 - 읽기 전용 18-38
 - 정렬 18-26, 22-7
 - 정의 18-38 ~ 18-39
 - 중첩 18-37
 - 필드 및 인덱스 정의 18-39
 - 미리 로드 18-40
- 테이블 액세스
 - 로컬 트랜잭션 20-32
- 테이블 프로듀서 28-19 ~ 28-21
 - 속성 설정 28-19
- 텍스트
 - 검색 3-33

- 국제화 12-9
 - 복사, 잘라내기, 붙여넣기 7-10
 - 사용 7-7 ~ 7-12
 - 삭제 7-10
 - 선택 7-9, 7-9
 - 오른쪽에서 왼쪽으로 읽기 12-6
 - 인쇄 3-33
 - 컨트롤에서 텍스트 사용 7-7
- BDE 기반 20-2, 20-4 ~ 20-9
 - 단기 20-5
 - 레코드 복사 20-8
 - 레코드 삭제 20-8
 - 레코드 업데이트 20-8
 - 레코드 추가 20-8
 - 바인딩 20-5
 - 배치 작업 20-8 ~ 20-9
 - 배타적 잠금 20-6
 - 액세스 권한 20-6
 - 인덱스 기반 검색 18-28
- dbExpress 22-7
 - owner-draw 컨트롤 7-12
- 텍스트 컨트롤 3-31 ~ 3-33
- 템플릿 5-20, 5-21
 - 메뉴 6-32, 6-38, 6-39 ~ 6-41
 - 로드 6-39
 - 의사 결정 그래프 16-16
 - 컴포넌트 6-12, 6-13
 - 페이지 프로듀서 29-6
 - 프로그래밍 5-3
 - 프로듀서 29-6
- 토글 6-46, 6-48
- 통신 32-1
 - 표준 27-2
 - 프로토콜 20-16, 27-2, 32-2
- 통합 디버거 2-5
- 통화
 - 국제화 12-10
 - 형식 12-10
- 통화 변환
 - 예제 4-63
 - HTML 28-14 ~ 28-17, 29-9
- Web Broker
 - 애플리케이션 28-3
- 투명한 배경 12-10
- 툴 버튼 6-47
 - 그룹화/그룹 해제 6-48
 - 도움말 얻기 6-49
 - 사용 불가능 6-47

- 여러 행 6-47
- 이미지 추가 6-47
- 줄 바꿈(wrapping) 6-47
- 초기 상태, 설정 6-47
- 토글로 사용 6-48
- 툴바 3-37, 6-17, 6-18, 6-43
 - 기본 드로잉 툴 6-45
 - 디자인 6-43 ~ 6-50
 - 버튼 사용 불가능 6-47
 - 버튼 삽입 6-44 ~ 6-46, 6-47
 - 숨기기 6-50
 - 스피드 버튼 3-36
 - 알기 쉬운 6-47, 6-49
 - 여백 설정 6-45
 - 추가 6-46 ~ 6-48
 - 컨텍스트 메뉴 6-49
 - 패널 추가 6-44 ~ 6-46
 - owner-draw 7-12
- 툴바, 작업 목록 6-18
- 툴팁 도움말 3-44
- 트랙 표시줄 3-33
- 트랜잭션 14-4 ~ 14-5, 17-6 ~ 17-10
 - 객체 컨텍스트 39-9
 - 끝 39-11 ~ 39-12
 - 다계층 애플리케이션 25-19
 - 롤 백 17-8 ~ 17-9
 - 로컬 20-32 ~ 20-33
 - 로컬 테이블 17-6
 - 분리 14-5, 39-9
 - 레벨 17-9 ~ 17-10
 - 서버 제어 39-12, 39-13
 - 시간 초과 39-14, 39-21
 - 시작 17-6 ~ 17-7
 - 업데이트 적용 17-6, 25-19
 - 여러 객체로 구성 39-9
 - 여러 데이터베이스 확장 39-9
 - 오버랩 17-7
 - 원자성 14-4, 39-9
 - 일관성 14-4, 39-9
 - 자동 39-12
 - 종료 17-8 ~ 17-9
 - 중첩 17-7
 - 커밋 17-8
 - 지속성 14-5, 39-9
 - 캐시된 업데이트 20-35
 - 커밋 17-8
 - 클라이언트 제어 39-12, 39-12 ~ 39-13
 - 트랜잭션 객체 39-5

- 트랜잭션 데이터 모듈
 - 25-7, 25-15, 25-19
- 트랜잭션 컴포넌트 17-7
- ADO 21-6 ~ 21-7, 21-8
 - 중단 유지 21-6
 - 커밋 유지 21-6
- BDE 20-31 ~ 20-33
 - 암시적 20-31
 - 제어 20-31 ~ 20-32
- IAppServer 25-19
- MTS 및 COM+ 39-8 ~ 39-14
- SQL 명령 사용 17-6, 20-31
- 트랜잭션 객체 33-11, 33-14 ~ 33-15, 39-1 ~ 39-24
 - 객체 컨텍스트 39-4
 - 관리 33-15, 39-23
 - 데이터베이스 연결
 - 풀링 39-5 ~ 39-6
 - 디버깅 39-21 ~ 39-22
 - 리소스 관리 39-3 ~ 39-8
 - 리소스 해제 39-8
 - 마샬링 39-3
 - 보안 39-14 ~ 39-15
 - 상태 없음(stateless) 39-11
 - 생성 39-15 ~ 39-18
 - 설치 39-22 ~ 39-23
 - 속성 공유 39-6 ~ 39-7
 - 요구 사항 39-3
 - 이중 인터페이스 39-3
 - 롤백 39-21
 - 타입 라이브러리 39-3
 - 트랜잭션 39-5, 39-8 ~ 39-14
 - 서버 제어 39-12, 39-13
 - 속성 39-9 ~ 39-11
 - 시간 초과 39-14, 39-21
 - 자동 39-12
 - 클라이언트 제어 39-12, 39-12 ~ 39-13
 - 특징 39-2 ~ 39-3
 - 활동 39-17 ~ 39-18
- 트랜잭션 데이터 모듈 25-5, 25-6 ~ 25-8
 - 데이터베이스 연결 25-6, 25-8
 - 풀링 25-7
 - 보안 25-9
 - 스레드 모델 25-15
 - 인터페이스 25-18
 - 트랜잭션 속성 25-15

- 트랜잭션 매개변수
 - 분리 레벨 17-10
- 트랜잭션 분리 레벨 17-9 ~ 17-10
 - 로컬 트랜잭션 20-32
 - 지정 17-10
- 트랜잭션 속성 39-9 ~ 39-11
 - 설정 39-10
 - 트랜잭션 데이터 모듈 25-15
- 트리 뷰 3-39
 - owner-draw 7-12
- 트리거 14-5

파

- 파일 4-50 ~ 4-58
 - 그래픽 8-19 ~ 8-21, 45-4
 - 루틴
 - 런타임 라이브러리 4-51
 - date-time 루틴 4-53
 - Windows API 4-54
 - 모드 4-56
 - 문자열 4-57
 - 바이트 복사 4-58
 - 복사 4-53
 - 비호환 타입 4-54
 - 삭제 4-51
 - 쓰기 4-56
 - 위치 4-58
 - 웹으로 보내기 28-13
 - 이름 재지정 4-53
 - 읽기 4-56
 - 작업 4-50 ~ 4-58
 - 찾기 4-51, 4-57
 - 처리 4-51 ~ 4-54
 - 크기 4-58
 - 폼 10-2
 - 핸들 4-53, 4-54, 4-56
 - 형식
 - 타입이 지정되지 않은 4-54
 - 타입이 지정된 4-54
 - 텍스트 4-54
 - I/O 4-54
 - date-time 루틴 4-53
 - resource 6-42 ~ 6-43
- 파일 권한 10-16
- 파일 기반 애플리케이션 14-9 ~ 14-10
 - 클라이언트 데이터셋 23-33 ~ 23-36
- 파일 끝 문자 10-14
- 파일 목록
 - 항목 드래그 7-2, 7-3
- 항목 드롭 7-3
- 파일 스트림 4-54 ~ 4-58
 - 생성 4-55
 - 열기 4-55
 - 예외 4-57
 - 파일 I/O 4-54 ~ 4-58
 - 포팅 가능 4-54
 - 핸들 얻기 4-53
 - TMemoryStream 4-55
- 파일 I/O
 - 형식 4-54
- 팔레트 45-5 ~ 45-6
 - 기본 동작 45-5
 - 지정 45-5
- 팔레트 비트맵 파일 47-3
- 팔레트 실현 45-5
- 팝업 메뉴 7-11 ~ 7-12
 - 드롭다운 메뉴 6-35
- 패널
 - 스피드 버튼 3-36
 - 스피드 버튼 추가 6-45
 - 액자 모양 3-46
 - 폼 위에 추가 6-44
- 패턴 8-9
- 패키지 11-1, 47-19
 - 국제화 12-12, 12-14
 - 기본 설정 11-7
 - 디자인 전용 옵션 11-7
 - 디자인 타임 11-1, 11-5 ~ 11-6
 - 런타임 11-1, 11-2 ~ 11-5, 11-7
 - 모음 11-13
 - 사용 5-9
 - 사용자 지정 11-4
 - 생성 5-9, 11-6 ~ 11-11
 - 설치 11-5 ~ 11-6
 - 소스 파일 11-2, 11-12
 - 애플리케이션 배포 11-2, 11-13
 - 애플리케이션에서 사용 11-3 ~ 11-5
 - 옵션 11-7
 - 요청 목록 11-6, 11-7, 11-9, 47-19
 - 중복 참조 11-9
 - 참조 11-3
 - 컴파일 11-10 ~ 11-12
 - 옵션 11-10
 - 컴파일러 지시어 11-10
 - 컴포넌트 47-19
 - 파일명 확장자 11-1
 - 편집 11-7 ~ 11-8
 - 포함 목록 11-6, 11-7, 11-9, 47-19

- DLL 11-1, 11-2
- 패키지 모음 파일 11-13
- 패키지 파일 13-3
- 팩토리 29-5
- 페이지 디스패처 29-4
 - 디스패처
 - 페이지 29-18
- 페이지 모듈 29-3, 29-7
 - 웹 29-6
- 페이지 이름 29-6
- 페이지 컨트롤 3-42
 - 페이지 추가 3-42
- 페이지 프로듀서 28-14 ~ 28-17, 29-6, 29-9
 - 연결 28-16
 - 이벤트 처리 28-15, 28-16, 28-17
 - 컴포넌트 29-6
 - 타입 29-3
 - 템플릿 29-6, 29-9
 - 템플릿 변환 28-15
 - Content 메소드 28-15
 - ContentFromStream 28-15
 - ContentFromString 28-15
 - data-aware 25-40 ~ 25-44, 28-18
- 웹 8-5, 49-5
 - 그리기 모드 8-29
 - 기본 설정 8-6
 - 너비 8-6
 - 변경 49-7
 - 브러시 8-5
 - 색상 8-6
 - 스타일 8-6
 - 위치 파악 8-7
 - 위치, 설정 8-7, 8-25
- 편집 모드 18-17
 - 취소 18-18
- 편집 컨트롤 3-31 ~ 3-33, 7-7, 15-2, 15-8
 - 여러 줄 15-9
 - 텍스트 선택 7-9
 - rich edit 형식 15-9
- 평균
 - 의사 결정 큐브 16-5
- 포인터
 - 메소드 43-2, 43-3, 43-8
 - 클래스 41-9
- 포커스 3-18, 40-4
 - 이동 3-34
 - 필드 19-16
- 포트 32-5
 - 다중 연결 32-5
- 서버 소켓 32-7
 - 서비스 32-2
 - 클라이언트 소켓 32-6
- 포팅 가능 코드 10-17
- 포함 목록(패키지) 11-6, 11-7, 11-9, 47-19
- 포함된 HTML 태그 (<EMBED>) 28-14
- 포함된 객체 33-9
- 폼 2-2, 3-23
 - 다른 폼에서 액세스 3-9
 - 데이터 검색 6-9 ~ 6-12
 - 데이터 동기화 15-3
 - 드릴 다운 15-15
 - 로컬 변수로 생성 6-8
 - 런타임에 생성 6-7
 - 마스터/디테일 테이블 15-15
 - 메모리 관리 6-6
 - 메인 6-1
 - 모달 6-6
 - 모달리스 6-6, 6-8
 - 새 객체 타입 3-5 ~ 3-7
 - 속성 쿼리
 - 예제 6-9
 - 연결 6-2
 - 유닛 참조 추가 6-2
 - 이벤트 핸들러 공유 8-15
 - 인수 전달 6-8 ~ 6-9
 - 인스턴스화 3-6
 - 전역 변수 6-6
 - 참조 6-2
 - 컨트롤 간 탐색 3-19, 3-22
 - 컴포넌트로 52-1
 - 표시 6-6
 - 프로젝트에 추가 3-8, 6-1 ~ 6-2
 - 필드 추가 8-26 ~ 8-27
- 폼 연결 6-2
- 폼 파일 3-16, 10-2, 12-14
- 폼과 대화 상자 공유 5-20 ~ 5-22
- 표 모양의 그리드 15-27
- 표 형식으로 표시(그리드) 3-44
- 표의 문자 12-2, 12-3
- 약자 12-9
- 표준 이벤트 43-4, 43-4 ~ 43-6
 - 사용자 지정 43-6
- 표준 컴포넌트 3-29 ~ 3-31
- 풍선 도움말 15-30
- 프로그래밍 템플릿 5-3
- 프로바이더 24-1 ~ 24-14
- 내부 23-19, 23-25, 24-1
- 데이터 제약 조건 24-13
- 데이터셋 관련 24-2
- 업데이트 객체 사용 20-11
- 업데이트 적용 24-4, 24-8 ~ 24-11
 - 업데이트 스크린 24-11
- 오류 처리 24-11
- 외부 14-11
- 원격 23-26, 24-3, 25-6
- 지역 23-26, 24-3
- 클라이언트 데이터셋 23-25 ~ 23-32
- 클라이언트 생성
 - 이벤트 24-12
- external 23-19, 23-25, 24-1
- XML 26-8 ~ 26-9
- XML 문서에 데이터 제
 - 공 26-9 ~ 26-11
- XML 문서와 연결 24-2, 26-8
- 프로세스 10-16
- 프로시저 40-6, 43-3
- 속성 설정 47-12
 - 이름 지정 44-2
- 프로젝트
 - 폼 추가 6-1 ~ 6-2
- 프로젝트 옵션 5-3
 - 기본 5-3
- 프로젝트 템플릿 5-21
- 프로젝트 파일
 - 변경 2-3
 - 분산 2-5
- 프로토크
 - 네트워크 연결 20-16
 - 선택 25-9 ~ 25-11
 - 연결 컴포넌트 25-9 ~ 25-11, 25-24
 - 인터넷 27-2, 32-1
- 프록시 33-7, 33-8
 - 트랜잭션 객체 39-2
- 프레임 6-12, 6-14 ~ 6-16
 - 공유 및 배포 6-16
 - 그래픽 6-15
 - 리소스 6-15
 - 컴포넌트 템플릿 6-14, 6-15
- 플라이바이 도움말 3-44
- 플래그 51-3
- 픽셀
 - 읽기 및 설정 8-9
- 필드 19-1 ~ 19-28
 - 값 변경 15-6

- 값 업데이트 15-5
- 값 표시 15-11, 19-17
- 값 할당 18-21
- 기본 형식 19-15
- 기본값 19-20
- 데이터 검색 19-17
- 데이터 입력 18-18, 19-14
- 데이터베이스 51-5, 51-6
- 메시지 레코드 46-2, 46-4, 46-6
- 상호 배타적인 옵션 15-2
- 속성 19-1
- 숨겨진 24-5
- 열거 17-14
- 영구적 열 15-17
- 웹 어댑터 29-8
- 유효한 데이터 제한 19-21 ~ 19-22
- 입기 전용 15-5
- 추상 데이터 타입 19-22 ~ 19-28
- 폼에 추가 8-26 ~ 8-27
- 활성화 19-16
- Null 값 18-21
- 필드 객체 19-1 ~ 19-28
- 동적 19-2 ~ 19-3
- 영구적과 비교 19-2
- 삭제 19-10
- 속성 19-1, 19-11 ~ 19-15
- 공유 19-13
- 런타임 19-12
- 액세스 값 19-19 ~ 19-20
- 영구적 19-3 ~ 19-16
- 동적과 비교 19-2
- 이벤트 19-15 ~ 19-16
- 정의 19-5 ~ 19-10
- 표시 및 편집 속성 19-11
- 필드 속성 19-13 ~ 19-14
- 데이터 패킷 24-6
- 제거 19-14
- 지정 19-13
- 필드 정의 18-39
- 복사 18-40
- 필드 타입
- 변환 19-16, 19-18 ~ 19-19
- 필터 18-12 ~ 18-16
- 대소문자 구별 18-15
- 런타임 시 설정 18-15
- 문자열 비교 18-15
- 범위와 비교 18-30
- 북마크 사용 21-11
- 빈 필드 18-14

- 사용 가능/사용 불가능 18-13
- 연산자 18-14
- 정의 18-13 ~ 18-15
- 쿼리와 비교 18-13
- 클라이언트 데이터셋 23-3 ~ 23-5
- 매개변수 사용 23-29 ~ 23-30
- 텍스트 필드의 옵션 18-15

하

- 하위 메뉴 6-34
- 하위 컴포넌트
- 속성 42-9
- 하이퍼텍스트 링크
- HTML에 추가 28-14
- 한정자 3-8 ~ 3-9
- 할당문 42-2
- 할당문
- 객체 변수 3-10
- 함수 40-6
- 그래픽 45-1
- 속성 읽기 42-6, 47-8, 47-10
- 이름 지정 44-2
- 이벤트 43-3
- Windows API 45-1
- 핫 키 3-34
- 해결 24-1, 25-4
- 핸들
- 리소스 모듈 12-13
- 소켓 연결 32-6
- 행 3-44
- 웹 어댑터 29-8
- 의사 결정 그리드 16-11
- 헤더
- HTTP 요청 27-4
- owner-draw 7-12
- 헤더 컨트롤 3-42
- 현재 윌 반환 50-8
- 확장성 14-11
- 호스트 25-25, 32-4
- URL 27-3
- 주소 32-4
- 호스트 이름 32-4
- IP 주소와 비교 32-4
- 호출 가능 인터페이스
- 구현 31-6 ~ 31-7
- 호출 31-9 ~ 31-10
- 호출 가능한 인터페이스 4-26, 31-2, 31-3 ~ 31-4
- 네임스페이스 31-4
- 등록 31-4
- 정의 31-2

- URI 31-9
- 호출 동기화 39-18
- 호출 레지스트리 31-4, 31-6
- 홈 디렉토리 10-15
- 화면
- 새로 고침 8-2
- 해상도 13-12
- 프로그래밍 13-12, 13-13
- 활동
- 트랜잭션 객체 39-17 ~ 39-18
- 활성 문서 33-14
- 활성 스크립팅 29-9
- 활성화 속성
- 공유 속성 39-6
- 힌트 3-44

A

- Abort 프로시저
- 편집 방지 18-20
- AbortOnKeyViol 속성 20-53
- AbortOnProblem 속성 20-53
- About 상자 52-2, 52-3
- 속성 추가 52-4
- 실행 52-5
- ActiveX 컨트롤 추가 38-5
- About 유닛 52-3
- AboutDlg 유닛 52-2
- Acquire 메소드 9-7
- Action 6-17
- Action List 에디터 6-18
- Action Manager 6-16, 6-17, 6-18, 6-19, 6-21
- Action Manager 에디터 6-19, 6-22
- ActionBand 6-18
- ActionLink 속성 3-18
- Actions 속성 28-5
- Active Document 33-10
- IOleDocumentSite
- 인터페이스 참조
- Active Server Object 37-1 ~ 37-8
- 등록 37-7 ~ 37-8
- 디버깅 37-8
- Active Server Object
- 마법사 37-2 ~ 37-3
- Active Server Objects
- 등록 37-7 ~ 37-8
- 생성 37-2 ~ 37-7
- in-process 서버 37-7
- out-of-process 서버 37-7

Active Server Page 33-10, 33-13, 37-1 ~ 37-8
 생성 37-3
 HTML 문서 37-1
 UI 디자인 37-1
 Active 속성
 데이터셋 18-4
 서버 소켓 32-7
 세션 20-18
 클라이언트 소켓 32-6
 ActiveAggs 속성 23-14
 ActiveFlag 속성 16-19, 16-20
 ActiveForms 38-5 ~ 38-6
 다계층 애플리케이션 25-34
 데이터베이스
 웹 애플리케이션 25-34
 마법사 38-5 ~ 38-6
 생성 38-2
 InternetExpress와 비교 25-33
 ActiveX 33-13 ~ 33-14, 38-1
 인터페이스 33-19
 웹 애플리케이션 33-13, 38-1, 38-15 ~ 38-17
 ASP와 비교 37-7
 InternetExpress와 비교 25-33 ~ 25-34
 ActiveX 컨트롤 13-5, 33-10, 33-13, 33-22, 38-1 ~ 38-17
 등록 38-14
 디버깅 38-15
 디자인 38-4
 라이선스 38-5, 38-7
 라이선싱 38-6
 마법사 38-4 ~ 38-5
 메소드 추가 38-8 ~ 38-9
 생성 38-2, 38-4 ~ 38-6
 속성 추가 38-8 ~ 38-9
 속성 페이지 35-7, 38-3, 38-12 ~ 38-14
 스래드 모델 38-5
 영구적인 속성 38-12
 요소 38-2 ~ 38-3
 웹 애플리케이션 33-13, 38-1, 38-15 ~ 38-17
 웹 배포 38-17
 웹에 배포 38-15
 이벤트 처리 38-10
 인터페이스 38-7 ~ 38-11
 컴포넌트 랩퍼 35-5, 35-6 ~ 35-7, 35-8 ~ 35-9
 타입 라이브러리 33-16, 38-3
 Automation 호환 타입 사용 38-4, 38-8
 data-aware 35-8 ~ 35-9, 38-8, 38-10 ~ 38-11
 HTML에 포함 28-14
 import하기 35-4
 VCL 컨트롤 38-4 ~ 38-6
 ActiveX 페이지
 (컴포넌트 팔레트) 3-30
 ActiveX 페이지
 (컴포넌트 팔레트) 3-31, 35-4
 ActiveX Data Objects
 ADO 참조
 ActnList 유닛 6-28
 AdapterPageProducer 29-3
 Add 메소드
 메뉴 6-41
 문자열 3-53
 영구적 열 15-19
 Add Fields 대화 상자 19-4
 Add ~ Interface 명령 25-17
 Add To Repository 명령 5-20
 AddAlias 메소드 20-25
 AddFieldDef 메소드 18-39
 AddFontResource 함수 13-14
 AddIndexDef 메소드 18-39
 Additional 페이지
 (컴포넌트 팔레트) 3-29
 AddObject 메소드 3-54
 AddParam 메소드 18-53
 AddPassword 메소드 20-22
 AddRef 메소드 4-20, 4-23, 4-25, 33-4
 Address 속성
 TSocketConnection 25-26
 AddStandardAlias 메소드 20-26
 AddStrings 메소드 3-53, 3-54
 ADO 14-1, 18-2, 21-1, 21-2
 데이터 저장소 21-2, 21-4
 리소스 디스펜서 39-6
 배포 13-7
 암시적 트랜잭션 21-6 ~ 21-7
 컴포넌트 21-1 ~ 21-20
 개요 21-1 ~ 21-2
 프로바이더 21-3, 21-4
 ADO 객체 21-1
 Connection 객체 21-4 ~ 21-5
 RDS DataSpace 21-17
 Recordset 21-9, 21-10 ~ 21-11
 ADO 데이터셋 21-9 ~ 21-17
 데이터 파일 21-15 ~ 21-16
 명령 실행 21-20
 배치 업데이트 21-12 ~ 21-15
 비동기 페치 21-12
 연결 21-10
 인덱스 기반 검색 18-28
 ADO 명령 21-7, 21-17 ~ 21-20
 데이터 검색 21-19
 매개변수 21-19 ~ 21-20
 반복 17-13
 비동기화 21-19
 실행 21-18
 지정 21-17 ~ 21-18
 취소 21-18 ~ 21-19
 ADO 연결 21-2 ~ 21-9
 데이터 저장소에 연결 21-2 ~ 21-7
 명령 실행 21-6
 비동기화 21-5
 시간 초과 21-5 ~ 21-6
 이벤트 21-8 ~ 21-9
 ADO 페이지
 (컴포넌트 팔레트) 21-1, 3-30, 14-1
 ADOExpress 21-1
 ADT 필드 19-22, 19-23 ~ 19-25
 영구적 필드 19-24
 평평하게 하기 (flattening) 15-22
 표시 15-22, 19-23
 ADTG 파일 21-15
 AfterApplyUpdates 이벤트 23-32, 24-8
 AfterCancel 이벤트 18-21
 AfterClose 이벤트 18-4
 AfterConnect 이벤트 17-3, 25-30
 AfterDelete 이벤트 18-20
 AfterDisconnect 이벤트 17-4, 25-30
 AfterDispatch 이벤트 28-5, 28-8

AfterEdit 이벤트 18-17
 AfterGetRecords 이벤트 24-8
 AfterInsert 이벤트 18-18
 AfterOpen 이벤트 18-4
 AfterPost 이벤트 18-20
 AfterScroll 이벤트 18-5
 AggFields 속성 23-14
 Aggregates 속성 23-12, 23-13
 AliasName 속성 20-14
 Align 속성 3-19, 6-4
 상태 표시줄 3-43
 텍스트 컨트롤 7-7
 패널 6-44
 Alignment 속성 3-35
 데이터 그리드 15-20
 메모 3-32
 상태 표시줄 3-44
 서식있는 텍스트 컨트롤 3-32
 열 헤더 15-20
 의사 결정 그리드 16-12
 필드 19-11
 data-aware 메모 컨트롤 15-9
 AllowAllUp 속성 3-36
 스피드 버튼 6-46
 툴 버튼 6-48
 AllowDelete 속성 15-28
 AllowGrayed 속성 3-36
 AllowInsert 속성 15-28
 alTop 상수 6-44
 Anchor 속성 3-19
 ANSI 문자 집합 4-39, 12-2
 AnsiChar 4-39
 AnsiString 4-41
 Apache 서버 DLL
 생성 29-2
 Apache 애플리케이션 27-7
 디버깅 27-9
 생성 28-2, 29-2
 Apache DLL 13-10
 배포 13-11
 Apache Server DLL 27-7
 ApacheServer DLLs
 생성 28-2
 apartment 스레드 36-8 ~ 36-9
 Append 메소드 18-18, 18-19
 Insert와 비교 18-18
 AppendRecord 메소드 18-21
 Apply 메소드 20-46
 Apply Updates 대화 상자 34-25
 ApplyRange 메소드 18-34
 ApplyUpdates 메소드 10-29, 20-34
 클라이언트 데이터셋 21-13, 23-7, 23-20, 23-21 ~ 23-22, 24-3
 프로바이더 23-21, 24-3, 24-8
 BDE 데이터셋 20-36
 TDatabase 20-36
 TXMLTransformClient 26-10
 AppServer 속성 23-33, 24-3, 25-18, 25-31
 Arc 메소드 8-4
 AS_ApplyUpdates 메소드 24-3
 AS_DataRequest 메소드 24-3
 AS_Execute 메소드 24-3
 AS_GetParams 메소드 24-3
 AS_GetProviderNames 메소드 24-3
 AS_GetRecords 메소드 24-4
 AS_RowRequest 메소드 24-4
 ASCII 테이블 20-5
 ASP 33-10, 33-13, 37-1 ~ 37-8
 수행 제한 37-1
 스크립트 랭귀지 33-13, 37-3
 웹 브로커와 비교 37-1
 페이지 생성 37-3
 ActiveX와 비교 37-7
 HTML 문서 37-1
 UI 디자인 37-1
 ASP intrinsics 37-3 ~ 37-7
 애플리케이션 객체 37-4
 액세스 37-2 ~ 37-3
 Request 객체 37-4 ~ 37-5
 Response 객체 37-5
 Server 객체 37-6 ~ 37-7
 Session 객체 37-6
 Assign 메소드
 문자열 목록 3-54
 Assign Local Data 명령 23-14
 AssignedValues 속성 15-21
 AssignValue 메소드 19-16
 Associate 속성 3-34
 as-soon-as-possible 비활성화 39-4
 Attributes 속성
 매개변수 18-46, 18-53
 TADOConnection 21-6
 AutoCalcFields 속성 18-22
 AutoComplete 속성 25-8
 AutoDisplay 속성 15-9, 15-10
 AutoEdit 속성 15-5
 AutoHotKeys 속성 6-34
 Automation
 우선 바인딩 33-18
 인터페이스 36-12 ~ 36-15
 지연 바인딩 36-14
 최적화 33-18
 타입 설명 33-12
 타입 확인 36-13
 타입 호환성 34-11, 36-15 ~ 36-16
 Active Server Object 37-2
 IDispatch 인터페이스 36-14
 Automation 객체 33-12
 마법사 36-4 ~ 36-9
 컴포넌트 랩퍼 35-7 ~ 35-8
 예제 35-10 ~ 35-12
 COM 객체 참조
 Automation 서버 33-10, 33-12 ~ 33-13
 객체 액세스 36-14
 타입 라이브러리 33-17
 COM 객체 참조
 Automation 컨트롤러 33-12, 35-1, 35-12 ~ 35-16, 36-14
 객체 생성 35-13
 디스패치 인터페이스 35-13 ~ 35-14
 예제 35-10 ~ 35-12
 이벤트 35-14 ~ 35-16
 이중 인터페이스 35-13
 AutoPopup 속성 6-49
 AutoSelect 속성 3-32
 AutoSessionName 속성 20-18, 20-30, 28-18
 AutoSize 속성 3-19, 3-32, 6-5, 13-13, 15-8
 .AVI 파일 8-32
 AVI 클립 3-47, 8-29, 8-32

B

B2B 통신

XML 30-1

Bands 속성 3-37, 6-49

.bashrc 10-15

BatchMove 메소드 20-8

BDE

Borland Database Engine

참조

BDE(Borland Database Engine) 20-1

별칭 20-25 ~ 20-27

삭제 20-26

생성 20-25 ~ 20-26

지정 20-15

유틸리티 20-55

BDE 데이터셋 14-1, 18-2, 20-2 ~ 20-13

데이터베이스 20-3 ~ 20-4

로컬 데이터베이스 지

원 20-5 ~ 20-8

배치 작업 20-49 ~ 20-53

복사 20-51

세션 20-3 ~ 20-4

의사 결정 지원 컴포넌트 16-5

캐시된 업데이트 적용 20-36

타입 20-2

BDE 페이지

(컴포넌트 팔레트) 3-30, 14-1

BDE Administration

유틸리티 20-15, 20-55

BeforeApplyUpdates

이벤트 23-32, 24-8

BeforeCancel 이벤트 18-21

BeforeClose 이벤트 18-4

BeforeConnect 이벤트 17-3, 25-30

BeforeDelete 이벤트 18-19

BeforeDisconnect

이벤트 17-4, 25-30

BeforeDispatch 이벤트 28-5, 28-7

BeforeEdit 이벤트 18-17

BeforeGetRecords

이벤트 24-8

BeforeInsert 이벤트 18-18

BeforeOpen 이벤트 18-4

BeforePost 이벤트 18-20

BeforeScroll 이벤트 18-5

BeforeUpdateRecord

이벤트 20-33, 20-41, 23-22, 24-11

BeginDrag 메소드 7-1

BeginRead 메소드 9-8

BeginTrans 메소드 17-6

BeginWrite 메소드 9-8

Beveled 3-35

BevelKind 속성 3-22

bin 디렉토리 10-17

BinaryOp 메소드 4-30

BLOB

캐시 20-4

BLOB 필드 15-2

값 얻기 20-4

값 표시 15-9

그래픽 보기 15-10

요구 즉시 폐치 24-6

BLOBs 15-9

BlockMode 속성 32-9, 32-10

bmBlocking 32-10

BMPDlg 유닛 8-21

bmThreadBlocking 32-9, 32-10

Bof 속성 18-6, 18-8

Bookmark 속성 18-9

BookmarkValid 메소드 18-9

BorderWidth 속성 3-41

Borland Database Engine

5-10, 14-1, 18-2

기본 연결 속성 20-19

데이터 검색 18-48, 20-2, 20-10

데이터베이스 연결

열기 20-20

데이터베이스에 연결

20-13 ~ 20-16

데이터셋 20-2

드라이버 20-1, 20-14

드라이버 이름 20-14

리소스 디스펜서 39-6

배치 작업 20-49 ~ 20-53

배포 13-8, 13-15

별칭 20-3, 20-14,

20-16

사용 가능성 20-26

이중 쿼리 20-10

지정 20-14

세션 20-17

암시적 트랜잭션 20-31

연결 관리 20-19 ~ 20-22

연결 닫기 20-20

웹 애플리케이션 13-10

이중 쿼리 20-9 ~ 20-10

캐시된 업데이트 20-33 ~ 20-49

업데이트 오류 20-38

테이블 타입 20-5

API 호출 20-1, 20-4

ODBC 드라이버 20-16

.BPL 파일 11-1, 13-3

Brush 속성 3-46, 8-4, 8-8, 45-3

BrushCopy 메소드 45-3, 45-7

ButtonAutoSize 속성 16-10

ButtonStyle 속성

데이터 그리드 15-20,

15-21

ByteType 4-44

C

.CAB 파일 38-16

CacheBlobs 속성 20-4

CachedUpdates 속성 10-29, 20-33

calendar 컴포넌트 3-40

CanBePooled 메소드 39-8

Cancel 메소드 18-18,

18-21, 21-19

Cancel 속성 3-35

CancelBatch 메소드 10-29, 21-13, 21-14

CancelRange 메소드 18-34

CancelUpdates 메소드

10-29, 21-13, 23-6

CanModify 속성

데이터 그리드 15-25

데이터셋 15-5, 18-17, 18-38

쿼리 20-11

Canvas 속성 3-47, 40-7

Caption 속성 3-20

TForm 3-43

그룹 상자와 라디오 그룹 3-41

레이블 3-43

열 헤더 15-20

의사 결정 그리드 16-12

잘못된 입력 6-32

Cast 메소드 4-28

CastTo 메소드 4-29

CDaudio 디스크 8-32

CellDrawState 함수 16-12

CellRect 메소드 3-44

Cells 속성 3-45

Cells 함수 16-12

CellValueArray 함수 16-12

CGI 애플리케이션 13-11

- 생성 29-2
- CGI 프로그램 27-5, 27-6, 27-7
 - 생성 28-2
- Change 메소드 51-11
- ChangeBounds 속성 10-22
- ChangeCount 속성 10-29, 20-33, 23-6
- ChangedTableName 속성 20-53
- CHANGEINDEX 23-8
- ChangeScale 속성 10-22
- Char 데이터 타입 4-39, 12-2
- Chart Editing 대화 상자 16-15 ~ 16-18
- Chart FX 13-5
- CHECK 제한 조건 24-13
- Checked 속성 3-36
- CheckSynchronize 루틴 9-5
- Chord 메소드 8-4
- Clear 메소드
 - 문자열 목록 3-53, 3-54
 - 필드 19-16
- Click 메소드 43-2
 - 오버라이드 43-6, 50-11
- Clipbrd 유닛 7-8
- CloneCursor 메소드 23-15
- Close 메소드
 - 데이터베이스 연결 20-20
 - 데이터셋 18-4
 - 세션 20-18
 - 연결 컴포넌트 17-4
- CloseDatabase 메소드 20-20
- CloseDataSets 메소드 17-12
- CLSID 33-6
 - 라이선스 패키지 파일 38-7
- CLSIDs 33-5, 33-16
- CLX 3-1
 - 애플리케이션 10-1
 - VCL과 비교 10-6
 - 객체 생성자 10-13
 - 배포 13-6
- clx60.bpl 13-6
- CM_EXIT 메시지 51-11
- CMExit 메소드 51-11
- CoClass 33-6
 - 생성 33-6, 35-5, 35-13
 - 선언 35-5
 - 컴포넌트 랩퍼 35-3
 - ActiveX 컨트롤 38-4
 - CLSID 33-6
- CoClasses
 - 생성 34-19, 35-13 ~ ??
 - 업데이트 34-20
- 이름 지정 36-3
- 컴포넌트 랩퍼 35-1
 - 제한 사항 35-2
- Type Library 에디터 34-9, 34-16, 34-22 ~ 34-23
- Code Insight
 - 템플릿 5-3
- ColCount 속성 15-28
- Color 속성 3-19, 3-43, 3-46
 - 데이터 그리드 15-20
 - 브러시 8-8
 - 열 헤더 15-20
 - 의사 결정 그리드 16-12
 - 펜 8-5, 8-6
- ColorChanged 속성 10-22
- Cols 속성 3-45
- Columns 속성 3-38, 15-18
 - 그리드 15-16
 - 라디오 그룹 3-41
- Columns Editor
 - 열 삭제 15-19
 - 열 재정렬 15-19
 - 영구적 열 생성 15-18
- ColWidths 속성 3-45, 7-15
- COM 5-14
 - 개요 33-1 ~ 33-23
 - 기술 33-11
 - 마법사 33-18 ~ 33-23, 36-1
 - 사양 33-1, 33-2
 - 스텝(stub) 33-8
 - 애플리케이션 33-18
 - 구성 요소 33-3 ~ 33-10
 - 분산 5-14
 - 우선 바인딩 33-16
 - 인터페이스 33-3 ~ 33-5, 36-3
 - 구현 33-6, 33-23
 - 디스패치 식별자 36-14
 - 마살링 33-8 ~ 33-9
 - 수정 34-20 ~ 34-22, 36-9 ~ 36-12
 - 이중 인터페이스 36-13 ~ 36-14
 - 인터페이스 포인터 33-4
 - 최적화 33-18
 - 타입 라이브러리에 추가 34-20
 - 타입 정보 33-15
 - Automation 36-12 ~ 36-15
 - IUnknown 33-4
- 정의 33-1 ~ 33-2
- 집합체 33-9
- 컨테이너 33-10, 35-1
- 컨트롤러 33-10, 35-1
- 클라이언트 33-3, 33-9, 34-20, 35-1 ~ 35-16
- 프록시 33-7, 33-8
- 확장 33-2, 33-10 ~ 33-12
 - 종속성 33-11
- COM 객체 33-3, 33-5 ~ 33-9, 36-1 ~ 36-18
 - 등록 36-17
 - 디버깅 36-18
 - 디자인 36-2
 - 마법사 36-2 ~ 36-4, 36-5 ~ 36-9
 - 생성 36-1 ~ 36-16
 - 수명 관리 4-20
 - 스레드 모델 36-6 ~ 36-9
 - 인스턴스 36-5 ~ 36-6
 - 인터페이스 33-3, 36-9 ~ 36-15
 - 집합체 33-9
 - 컴포넌트 랩퍼 35-1, 35-2, 35-3, 35-6 ~ 35-12
 - 타입 확인 33-17
- COM 라이브러리 33-2
- COM 서버 33-3, 33-5 ~ 33-9, 36-1 ~ 36-18
 - 디자인 36-2
 - 원격 33-7
 - 최적화 33-18
 - in-process 33-7
 - out-of-process 33-7
- COM 인터페이스, 예외 발생 34-9
- COM+ 5-15, 25-6, 33-11, 33-14, 39-1
 - 트랜잭션 객체 참조
 - 객체 풀링 39-8
 - 이벤트 35-15 ~ 35-16, 39-18 ~ 39-20
 - 발생 39-20
 - 이벤트 객체 39-19
 - 마법사 39-19
 - 생성 39-19
 - 인터페이스 포인터 33-5
 - 트랜잭션 25-19
 - 트랜잭션 객체 33-14 ~ 33-15
 - 호출 동기화 39-18
 - 활동 구성 39-18
 - in-process 서버 33-7

MTS와 비교 39-1
 COM+ 애플리케이션 39-6,
 39-22
 COM+ Component
 Manager 39-23
 COMCTL32.DLL 6-43
 Command Text 에디터
 18-44
 CommandCount 속성 17-13,
 21-7
 Commands 속성 17-13,
 21-7
 CommandText 속성 18-44,
 21-16, 21-17, 21-19,
 22-6, 22-7, 22-8, 23-32,
 23-33
 CommandTimeout 속성
 21-5, 21-19
 CommandType 속성 21-16,
 21-17, 22-6, 22-7, 22-8,
 23-32
 Commit 메소드 17-8
 CommitTrans 메소드 17-8
 CommitUpdates 메소드
 10-29, 20-34, 20-36
 Compare 메소드 4-32
 CompareBookmarks 메소
 드 18-10
 CompareOp 메소드 4-33
 Component 마법사 40-9
 ComputerName 속성 25-25
 ConfigMode 속성 20-26
 Connected 속성 17-3
 연결 컴포넌트 17-3
 Connection 속성 21-3,
 21-10
 Connection Editor 22-5
 Connection String
 Editor 21-4
 ConnectionBroker 23-26
 ConnectionName 속성 22-4
 ConnectionObject 속성 21-4
 ConnectionString 속성 17-2,
 17-4, 21-3, 21-10
 ConnectionTimeout
 속성 21-5
 ConnectOptions 속성 21-5
 Console Application 5-3
 CONSTRAINT 제약 조건
 24-13
 ConstraintErrorMessage 속
 성 19-11, 19-21, 19-22
 Constraints 속성 6-4, 23-8,
 24-13
 Content 메소드

페이지 프로듀서 28-15
 ContentFromStream 메소드
 페이지 프로듀서 28-15
 ContentFromString 메소드
 페이지 프로듀서 28-15
 ContentStream 속성
 웹 응답 객체 28-12,
 28-13
 ContextHelp 5-30
 ControlType 속성 16-9,
 16-15
 Convert 함수 4-59, 4-60,
 4-61, 4-63, 4-65
 CopyFile 함수 4-53
 CopyFrom 함수 4-58
 CopyMode 속성 45-3
 CopyRect 메소드 8-4, 45-3,
 45-7
 CopyToClipboard 메소드
 7-10
 그래픽 15-10
 data-aware 메모
 컨트롤 15-9
 CORBA 4-17
 다계층 데이터베이스
 애플리케이션 25-11
 애플리케이션 서버에
 연결 25-28
 CORBA 데이터 모듈 25-5
 스프레드 모델 25-17
 인스턴스 25-16
 CORBA 연결 25-11, 25-28
 CORBA Data Module
 마법사 25-16 ~ 25-17
 Count 속성
 문자열 목록 3-52
 TSessionList 20-30
 Create 메소드 3-11
 Create Data Set 명령 18-39
 Create Submenu 명령(메뉴 디
 자이너) 6-35, 6-38
 CREATE TABLE 17-11
 Create Table 명령 18-39
 CreateDataSet 메소드 18-39
 CreateFile 함수 4-53
 CreateObject 메소드 37-3
 CreateParam 메소드 23-29
 CreateSharedPropertyGroup
 39-6
 CreateSuspended
 매개변수 9-11
 CreateTable 메소드 18-39
 CreateTransactionContextEx
 예제 39-12 ~ 39-13
 CreateWidget 속성 10-22

crtldcu 13-6
 Currency 속성
 필드 19-11
 CursorChanged 속성 10-22
 CursorType 속성 21-12,
 21-13
 CurValue 속성 24-12
 Custom 속성 25-43
 CustomConstraint 속성
 19-11, 19-21, 23-7,
 23-8
 CutToClipboard 메소드 7-10
 그래픽 15-10
 data-aware 메모 컨트롤
 15-9

D

data
 records 참조
 Data 속성 23-5, 23-14,
 23-16, 23-35
 Data Bindings 에디터 35-8
 Data Controls 페이지
 (컴포넌트 팔레트) 3-29,
 14-2, 14-15, 15-1, 15-2,
 25-2
 data-aware 컨트롤 14-15,
 15-1 ~ 15-31, 19-17,
 51-1
 공통적인 기능 15-2
 그래픽 표시 15-10
 그리드 15-14
 다시 그리기 사용
 불가능 15-6, 18-8
 데이터 새로 고침 15-7
 데이터 입력 19-14
 데이터 찾아보기 51-1 ~
 51-7
 데이터 편집 51-7 ~
 51-11
 데이터 표시 15-6 ~ 15-7
 그리드에서 15-15,
 15-27
 현재 값 15-8
 데이터셋 관련 15-3 ~
 15-4
 레코드 삽입 18-19
 목록 15-2
 변경 내용에 응답 51-6
 생성 51-1 ~ 51-12
 소멸 51-6
 읽기 전용 15-8
 편집 15-5 ~ 15-6,
 18-17
 필드 표시 15-7

Database 매개변수 22-4
 database engines
 Borland Database Engine 참조
 Database Explorer 20-15
 Database Properties
 에디터 20-14
 연결 매개변수 보기 20-15
 DatabaseCount 속성 20-21
 DatabaseName 속성 17-2,
 20-3, 20-14
 이중 쿼리 20-10
 Databases 속성 20-21
 DataChange 메소드 51-10
 DataCLX 10-6
 DataField 속성 15-11, 51-5
 조회 리스트 박스와 콤보
 박스 15-12
 DataRequest 메소드 23-32,
 24-3
 DataSet 속성
 데이터 그리드 15-16
 프로바이더 24-2
 dataset providers
 providers 참조
 DataSetCount 속성 17-12
 DataSetField 속성 18-37
 DataSets 속성 17-12
 DataSnap 페이지 (컴포넌트
 팔레트) 3-30, 25-2, 25-6
 DataSource 속성
 데이터 그리드 15-16
 데이터 탐색기 15-31
 조회 리스트 박스와 콤보
 박스 15-12
 쿼리 18-47
 ActiveX 컨트롤 35-8
 data-aware 컨트롤 51-5
 DataType 속성
 매개변수 18-46, 18-52
 DateTimePicker 컴포넌트
 3-40
 DAX 33-2, 33-21 ~ 33-23
 Day 속성 50-5
 DB/2 드라이버
 배포 13-9
 dBASE 테이블 20-5
 데이터 액세스 20-9
 로컬 트랜잭션 20-32
 레코드 추가 18-19
 암호 보호 20-22 ~ 20-24
 이름 재지정 20-8
 인덱스 20-6
 DatabaseName 20-3
 DBChart 컴포넌트 14-15
 DBCheckBox 컴포넌트 15-2,
 15-13 ~ 15-14
 DBComboBox 컴포넌트
 15-2, 15-11 ~ 15-12
 DBConnection 속성 23-17
 DBCtrlGrid 컴포넌트 15-2,
 15-27 ~ 15-28
 속성 15-28
 dbDirect 페이지 (컴포넌트
 팔레트) 14-2
 DBEdit 컴포넌트 15-2, 15-8
 dbExpress 10-23 ~ 10-29,
 13-7, 14-2, 22-1 ~ 22-2
 드라이버 22-3 ~ 22-4
 디버깅 22-18 ~ 22-19
 메타데이터 22-13 ~
 22-18
 배포 22-1
 컴포넌트 22-1 ~ 22-19
 dbExpress 애플리케이션
 13-10
 dbExpress 페이지 (컴포넌트 팔
 레트) 3-30, 22-2
 DBGrid 컴포넌트 15-2,
 15-15 ~ 15-27
 속성 15-20
 이벤트 15-26
 DBGridColumn
 컴포넌트 15-16
 DBImage 컴포넌트 15-2,
 15-10
 DBListBox 컴포넌트 15-2,
 15-11 ~ 15-12
 DBLogDlg 유닛 17-4
 DBLookupComboBox
 컴포넌트 15-2, 15-12 ~
 15-13
 DBLookupListBox
 컴포넌트 15-2, 15-12 ~
 15-13
 DBMemo 컴포넌트 15-2,
 15-9
 DBMS 25-1
 DBNavigator 컴포넌트 15-2,
 15-28 ~ 15-31
 DBRadioGroup 컴포넌트
 15-2, 15-14
 DBRichEdit 컴포넌트 15-2,
 15-9 ~ 15-10
 DBSession 속성 20-3
 DBText 컴포넌트 15-2,
 15-8
 dbxconnections.ini 22-4,
 22-5
 dbxdrivers.ini 22-3 ~ 22-4
 DCOM 33-7, 33-8
 다계층 애플리케이션 25-9
 분산 애플리케이션 5-14
 애플리케이션 서버에
 연결 23-26, 25-25
 InternetExpress
 애플리케이션 25-38
 DCOM 연결 25-9, 25-25
 DCOMCnfg.exe 25-38
 .DCP 파일 11-2, 11-12
 .DCR 파일 47-3
 .DCU 파일 11-2, 11-12
 DDL 17-10, 18-43, 18-49,
 20-9, 22-11
 Decision Cube 에디터 16-7
 ~ 16-8
 메모리 제어 16-8
 자원 설정 16-7
 큐브 용량 16-20
 Decision Cube 페이지
 (컴포넌트 팔레트) 16-1,
 14-15
 Decision Query 에디터 16-6
 DECnet 프로토콜
 (Digital) 32-1
 Default 상자 5-3
 Default 속성
 액션 항목 28-7
 DEFAULT_ORDER 23-8
 DefaultColWidth 속성 3-45
 DefaultDatabase 속성 21-4
 DefaultDrawing 속성 7-13,
 15-26
 DefaultExpression 속성
 19-20, 23-7
 DefaultHandler 메소드 46-3
 DefaultPage 속성 29-19
 DefaultRowHeight 속성 3-45
 Delete 메소드 18-19
 문자열 목록 3-53, 3-54
 Delete 명령
 (메뉴 디자이너) 6-38
 DELETE 문 20-41, 20-44,
 24-10
 Delete Table 명령 18-41
 Delete Templates 대화
 상자 6-40
 Delete Templates 명령 (메뉴
 디자이너) 6-38, 6-40
 DeleteAlias 메소드 20-26
 DeleteFile 함수 4-51
 DeleteFontResource
 함수 13-14
 DeleteIndex 메소드 23-9
 DeleteRecords 메소드 18-41

DeleteSQL 속성 20-41
DeleteTable 메소드 18-41
Delphi
 ActiveX 프레임워크
 (DAX) 33-21 ~ 33-23,
 33-2
Delta 속성 23-5, 23-20
\$DENYPACKAGEUNIT 컴파
일러 지시어 11-10
DEPLOY 13-8, 13-9,
13-15
DESIGNONLY 컴파일러
지시어 11-10
Destroy 메소드 3-11
DeviceType 속성 8-32
 생성 3-7
.DFM 파일 3-7, 10-2, 12-
10, 42-11
 생성 12-14
Dialogs 페이지
(컴포넌트 팔레트) 3-30
DimensionMap 속성 16-5,
16-7
Dimensions 속성 16-12
Direction 속성
 매개변수 18-46, 18-53
Directory 지시어 13-11
DirtyRead 17-9
DisableCommit 메소드
39-12
DisableConstraints
메소드 23-31
DisableControls 메소드 15-6
DisabledImages 속성 6-47
Dispatch 메소드 46-3, 46-4
dispID 36-14
 바인딩 36-15
dispIDs 33-16
dispinterfaces 25-31,
36-12, 36-13, 36-14
 동적 바인딩 34-9
 타입 라이브러리 34-9
DisplayFormat 속성 15-26,
19-11, 19-15
DisplayLabel 속성 15-17,
19-11
DisplayWidth 속성 15-16,
19-11
DLL 10-15
 국제화 12-12, 12-14
 배포 13-10
 생성 5-9
 설치 13-5
 패키지 11-1, 11-2
 Apache 13-11
 COM 서버 33-7
 스레드 모델 36-7
 HTML에 포함 28-14
 HTTP 서버 27-5, 27-6
 MTS 39-2
DllGetClassObject 39-3
DllRegisterServer 39-3
DML 17-10, 18-43, 18-49,
20-9
.DMT 파일 6-39, 6-40
Document Object Model
 DOM [참조](#)
Document Type Definition
 file
 DTD file [참조](#)
DocumentElement 속성
30-3
DoExit 메소드 51-12
DOM 30-2, 30-2
 구현 30-2
 사용 30-3
Down 속성 3-36
 스피드 버튼 6-45
.DPK 파일 11-2, 11-6
.DPL 파일 11-2, 11-12
DragCursor 속성 3-20
DragMode 속성 3-20, 7-1
 그리드 15-19
Draw 메소드 8-4, 45-3,
45-7
DrawShape 8-15
dprintf 유틸 20-54
DriverName 속성 20-14,
22-3
DropConnections 메소드
20-14, 20-21
DropDownCount 속성 3-39,
15-11
DropDownMenu 속성 6-49
DropDownRows 속성
 데이터 그리드 15-20,
 15-21
 조회 콤보 박스 15-13
DTD 파일 30-1
dynamic 지시어 41-9

E

EAbort 4-15
EBX 레지스터 10-9, 10-21
Edit 메소드 18-17, 47-10,
47-11
Edit 컨트롤 3-31
EditFormat 속성 15-26,
19-11, 19-15
EditKey 메소드 18-28,
18-30
EditMask 속성 19-14
 필드 19-11
EditRangeEnd 메소드 18-33,
18-34
EditRangeStart 메소드
18-33, 18-34
Ellipse 메소드 8-4, 8-11,
45-3
\$ELSEIF 지시어 10-19
EmptyDataSet 메소드
18-41, 23-27
EmptyStr variable 4-46
EmptyTable 메소드 18-41
EnableCommit 메소드 39-12
EnableConstraints 메소
드 23-31
EnableControls 메소드 15-6
Enabled 속성
 레이터 소스 15-4, 15-5
 메뉴 6-41, 7-10
 스피드 버튼 6-45
 액션 항목 28-7
 data-aware 컨트롤 15-7
EnabledChanged 속성 10-22
EndRead 메소드 9-8
EndWrite 메소드 9-8
Eof 속성 18-6, 18-7
EPasswordInvalid 4-17
EReadError 4-57
ERemovableException 31-7
ErrorAddr 변수 4-17
EventFilter 6-5, 46-5
EWriteError 4-57
Exception 4-16
 exception handling
 exceptions [참조](#)
 exceptions
 exception handling [참조](#)
Exclusive 속성 20-6
ExecProc 메소드 18-54,
22-11
ExecSQL 메소드 18-48,
18-49, 22-11
 업데이트 객체 20-47
Execute 9-6
Execute 메소드
 대화 상자 3-47, 52-4
 스레드 9-4
 연결 컴포넌트 17-10 ~
 17-11
 클라이언트 데이터셋
 23-28, 24-3
 프로바이더 24-3

ADO 명령 21-18, 21-20
 TBatchMove 20-52
 ExecuteOptions 속성 21-12
 ExecuteTarget 메소드 6-28
 Expandable 속성 15-23
 Expanded 속성
 데이터 그리드 15-20
 열 15-22, 15-23
 Expression 속성 23-12
 ExprText 속성 19-10
 Extensible Markup Language
 XML 참조

F

FastNet 페이지
 (컴포넌트 팔레트) 3-30
 Fetch Params 명령 23-28
 FetchAll 메소드 10-29,
 20-34
 FetchBlobs 메소드 23-28,
 24-4
 FetchDetails 메소드 23-28,
 24-4
 FetchOnDemand 속성 23-27
 FetchParams 메소드 23-28,
 24-3
 Field Link 디자이너 18-36
 FieldByName 메소드 18-32,
 19-20
 FieldCount 속성
 영구적 필드 15-17
 FieldDefs 속성 18-39
 FieldKind 속성 19-11
 FieldName 속성 19-5,
 19-11, 25-42
 데이터 그리드 15-20,
 15-21
 영구적 필드 15-17
 의사 결정 그리드 16-12
 Fields 속성 19-19
 Fields Editor 5-19, 19-3
 속성 집합 정의 19-13
 속성 집합 제거 19-14
 열 재정렬 15-19
 영구적 필드 삭제 19-10
 영구적 필드 생성 19-4 ~
 19-5, 19-5 ~ 19-10
 제목 표시줄 19-4
 탐색 버튼 19-4
 필드 목록 19-4
 필드 속성 적용 19-13
 FieldValues 속성 19-19
 FileAge 함수 4-53
 FileExists 함수 4-51
 FileGetDate 함수 4-53

FileName 속성
 클라이언트 데이터셋
 14-10, 23-34, 23-35
 files
 file streams 참조
 FileSetDate 함수 4-53
 FillRect 메소드 8-4, 45-3
 Filter 속성 18-13, 18-14 ~
 18-15
 Filtered 속성 18-13
 FilterGroup 속성 21-13,
 21-14
 FilterOnBookmarks
 메소드 21-11
 FilterOptions 속성 18-15
 finally 예약어 45-6, 52-5
 FindClose 프로시저 4-51
 FindDatabase 메소드 20-21
 FindFirst 메소드 18-16
 FindKey 메소드 18-28,
 18-29
 EditKey와 비교 18-30
 FindLast 메소드 18-16
 FindNearest 메소드 18-28,
 18-29
 FindNext 메소드 18-16
 FindNext 함수 4-51
 FindPrior 메소드 18-16
 FindResourceHInstance
 함수 12-13
 First Impression 13-5
 FindSession 메소드 20-30
 First 메소드 18-6
 FixedColor 속성 3-45
 FixedCols 속성 3-45
 FixedOrder 속성 3-37, 6-
 49
 FixedRows 속성 3-45
 FixedSize 속성 3-37
 FlipChildren 메소드 12-8
 FloodFill 메소드 8-4, 45-3
 FocusControl 메소드 19-16
 FocusControl 속성 3-43
 Font 속성 3-19, 3-32, 3-
 43, 8-4, 45-3
 데이터 그리드 15-20
 열 헤더 15-20
 data-aware 메모
 컨트롤 15-9
 FontChanged 속성 10-22
 Footer 속성 28-20
 FOREIGN KEY 제약 조
 건 24-13
 Format 속성 16-12
 Formula One 13-5

Found 속성 18-16
 FoxPro 테이블
 로컬 트랜잭션 20-32
 FrameRect 메소드 8-4
 FReadOnly 51-8
 Free 메소드 3-11, 10-13
 Free 스레드 36-7 ~ 36-8
 Free 스레드 마샬러 36-8
 FreeBookmark 메소드 18-10
 FromCommon 4-64

G

\$G 컴파일러 지시어 11-10,
 11-12
 GDI 애플리케이션 40-7,
 45-1
 Generate event support
 code 36-11
 GetAliasDriverName
 메소드 20-27
 GetAliasNames 메소드
 20-27
 GetAliasParams 메소드
 20-27
 GetAttributes 메소드 47-10
 GetBookmark 메소드 18-9
 GetConfigParams 메소드
 20-27
 GetData 메소드
 필드 19-16
 GetDatabaseNames
 메소드 20-27
 GetDriverNames 메소드
 20-27
 GetDriverParams 메소드
 20-27
 GetFieldByName 메소드
 28-9
 GetFieldNames 메소드
 17-14, 20-27
 GetFloatValue 메소드 47-8
 GetGroupState 메소드 23-10
 GetHandle 5-25
 GetHelpFile 5-25
 GetHelpStrings 5-26
 GetIDsOfNames 메소드
 36-14
 GetIndexNames 메소드
 17-14, 18-26
 GetMethodValue 메소드
 47-8
 GetNextPacket 메소드
 10-29, 20-34, 23-27,
 24-4

GetOptionalParam
 메소드 23-16, 24-7
 GetOrdValue 메소드 47-8
 GetPalette 메소드 45-5
 GetParams 메소드 24-3
 GetPassword 메소드 20-23
 GetProcedureNames
 메소드 17-14
 GetProcedureParams
 메소드 17-14
 GetProperties 메소드 47-11
 GetRecords 메소드 24-4,
 24-7
 GetSessionNames
 메소드 20-30
 GetStoredProcNames
 메소드 20-27
 GetStrValue 메소드 47-8
 GetTableNames 메소드
 17-13, 20-27
 GetValue 메소드 47-8
 GetVersionEx 함수 13-15
 GetViewerName 5-24
 GetXML 메소드 26-10
 Glyph 속성 3-36, 6-45
 GNU 어셈블러 10-18
 GNU make 유틸리티 10-16
 GotoCurrent 메소드 18-42
 GotoKey 메소드 18-28,
 18-29
 GotoNearest 메소드 18-28,
 18-29
 Graph Custom Control 13-5
 Graphic 속성 8-18, 8-21,
 45-4
 graphics methods
 팔레트 45-5
 GridLineWidth 속성 3-45
 grids
 decision grids 참조
 Grouped 속성
 툴 버튼 6-48
 GroupIndex 속성 3-36
 메뉴 6-42
 스피드 버튼 6-45, 6-46
 GroupLayout 속성 16-10
 Groups 속성 16-10
 GUI 애플리케이션 3-23
 GUID 4-22, 33-4, 34-8
 생성 4-22

H

\$H 컴파일러 지시어 4-41,
 4-49

Handle 속성 4-56, 10-22,
 32-6, 40-3, 40-4, 40-5,
 45-3
 장치 컨텍스트 8-1
 HandleException 4-14
 HandleException 메소드
 46-3
 HandleShared 속성 20-16
 HandlesTarget 메소드 6-28
 HasConstraints 속성 19-11
 HasFormat 메소드 7-11,
 8-23
 Header 속성 28-20
 Height 속성 3-19, 3-22,
 6-4
 리스트 박스 15-11
 TScreen 13-13
 Help Hints 15-30
 Help Manager 5-22, 5-23 ~
 5-32
 HelpContext 5-29, 5-30
 HelpContext 속성 3-44
 HelpFile 5-30
 HelpFile 속성 3-44
 HelpIntfs.pas 5-23
 HelpKeyword 5-29, 5-30
 HelpSystem 5-29, 5-30
 HelpType 5-29, 5-30
 Hint 속성 3-44
 Hints 속성 15-30
 HorzScrollBar 3-33
 Host 속성
 TSocketConnection
 25-26
 HostName 속성
 TCorbaConnection 25-28
 HotImages 속성 6-47
 HotKey 속성 3-34
 HTML 명령 28-14
 데이터베이스 정보 28-18
 생성 28-15
 HTML 문서 27-5
 데이터베이스 28-17
 데이터셋 28-20
 데이터셋 페이지
 프로듀서 28-18
 스타일 시트 25-42
 테이블 포함시키기 28-20
 테이블 프로듀서 28-19 ~
 28-21
 템플릿 25-41, 25-43 ~
 25-44, 28-14 ~ 28-15
 페이지 프로듀서 28-14 ~
 28-17

포함된 ActiveX 컨트롤 38-1
 ActiveForm에 대해
 생성 38-6
 ASP 37-1
 HTTP 응답 메시지 27-6
 InternetExpress
 애플리케이션 25-35
 HTML 테이블 28-14, 28-20
 생성 28-19 ~ 28-21
 속성 설정 28-19
 캡션 28-20
 HTML 템플릿 25-43 ~
 25-44, 28-14 ~ 28-17
 기본 25-41, 25-43
 WebSnap 페이지
 프로듀서 29-9
 HTML 투명 태그
 구문 28-14
 매개변수 28-14
 변환 28-14, 28-15
 이미 정의된 28-14
 HTML 폼 25-41
 HTML Result 뷰 29-1
 HTMLDoc 속성 25-41,
 28-15
 HTMLFile 속성 28-15
 HTTP 27-3
 개요 27-4 ~ 27-6
 다계층 애플리케이션
 25-10
 메시지 헤더 27-3
 상태 코드 28-11
 애플리케이션 서버에
 연결 25-27
 요청 메시지 요청 메시지
 참조
 요청 헤더 27-4, 28-9,
 37-4
 응답 메시지 응답 메시지
 참조
 응답 헤더 28-12, 37-5
 SOAP 31-1
 HTTP 요청 메시지 29-13
 HTTP 응답
 액션 29-16
 httpd.conf 13-11
 httpsvr.dll 25-10, 25-13,
 25-27
 HyperHelp 뷰어 5-22

I

IApplicationObject
 인터페이스 37-4

IAppServer 인터페이스
 23-32, 23-33, 24-3 ~
 24-4, 25-5
 상태 정보 25-21
 원격 프로바이더 24-3
 지역 프로바이더 24-3
 트랜잭션 25-19
 확장 25-17
 호출 25-30
 XML 브로커 25-36
 IConnectionPoint
 인터페이스 36-12
 IConnectionPointContainer
 인터페이스 36-12
 ICustomHelpViewer 5-22,
 5-23, 5-24, 5-25
 구현 5-24
 IDataIntercept 인터페이
 스 25-26
 IDefaultPageFileName 29-7
 IDispatch 인터페이스 33-9,
 33-19, 36-12, 36-14
 식별자 36-14, 36-15
 Automation 33-12
 IDL 컴파일러 33-18
 IDL (Interface Definition
 Language) 33-16, 33-18
 IDL (Interface Definition
 Language) 34-1
 Type Library 에디터 34-7
 IDOMImplementation 30-3
 IETF 프로토콜 및 표준 27-2
 IExtendedHelpViewer 5-23,
 5-27
 \$IFDEF 지시어 10-18
 \$IFEND 지시어 10-19
 \$IFNDEF 지시어 10-19
 IGetDefaultAction 29-7
 IGetProducerComponent
 29-7
 IGetScriptObject 29-6
 IGetWebAppComponents
 29-7
 IGetWebAppServices 29-7
 IHelpManager 5-23, 5-31
 IHelpSelector 5-23, 5-27
 IHelpSystem 5-23, 5-31
 IID 33-4
 IInterface 인터페이스 4-20,
 4-23, 4-25
 TInterfacedObject에서
 구현 4-20
 IInvokable 4-26, 31-3
 IIS 37-1
 버전 37-2
 IteratorObjectSupport 29-6
 Image HTML 태그
 () 28-14
 ImageIndex 속성 6-47,
 6-49
 ImageList 6-19
 ImageMap HTML 태그
 (<MAP>) 28-14
 Images 속성
 툴 버튼 6-47
 IMalloc 인터페이스 4-17
 IMarshal 인터페이스 36-15,
 36-16
 IME 12-9
 ImeMode 속성 12-9
 ImeName 속성 12-9
 implements 키워드 4-22,
 4-23
 \$IMPLICITBUILD 컴파일러
 지시어 11-10
 Import ActiveX 컨트롤
 명령 35-2, 35-4
 Import Type Library
 명령 35-2, 35-3
 ImportedConstraint 속성
 19-11, 19-22
 \$IMPORTEDDATA 컴파일러
 지시어 11-10
 Increment 속성 3-34
 Indent 속성 3-39, 6-45,
 6-47, 6-49
 Index 속성
 필드 19-11
 index 예약어 50-7
 Index Files 에디터 20-7
 IndexDefs 속성 18-39
 IndexFieldCount 속성 18-26
 IndexFieldNames 속성
 18-27, 22-7
 IndexName과 비교 18-27
 IndexFields 속성 18-26
 IndexFiles 속성 20-6
 IndexName 속성 20-6,
 22-7, 23-9
 IndexFieldNames와
 비교 18-27
 IndexOf 메소드 3-52, 3-53
 Indy Clients 페이지
 (컴포넌트 팔레트) 3-30
 Indy Misc 페이지
 (컴포넌트 팔레트) 3-31
 Indy Servers 페이지
 (컴포넌트 팔레트) 3-31
 INFINITE 상수 9-10
 Informix 드라이버
 배포 13-9
 INI 파일
 Win-CGI 프로그램 27-7
 ini 파일 10-8
 InitWidget 속성 10-22
 INotifyWebActivate 29-6
 in-process 서버 33-7
 ActiveX 33-13
 ASP 37-7
 MTS 39-2
 Input Mask 에디터 19-14
 INSERT 17-11
 Insert 메소드 18-18, 18-19
 Append와 비교 18-18
 메뉴 6-41
 문자열 3-53
 Insert 명령
 (메뉴 디자이너) 6-38
 INSERT 문 20-41, 20-44,
 24-10
 Insert from Resource
 대화 상자 6-43
 Insert From Resource 명령
 (메뉴 디자이너) 6-38,
 6-43
 Insert From Template 명령
 (메뉴 디자이너) 6-38,
 6-39
 Insert Template 대화
 상자 6-39
 InsertObject 메소드 3-54
 InsertRecord 메소드 18-21
 InsertSQL 속성 20-41
 Install COM+ objects
 명령 39-22
 Install MTS objects
 명령 39-22
 InstallShield Express 2-5,
 13-1
 배포
 애플리케이션 13-2
 패키지 13-3
 BDE 13-8
 SQL 연결 13-9
 IntegralHeight 속성 3-38,
 15-11
 InterBase Express
 배포 13-7
 InterBase 드라이버
 배포 13-9
 InterBase 테이블 20-9
 InterBase 페이지
 (컴포넌트 팔레트) 3-30
 InterBase 페이지
 (컴포넌트 팔레트) 14-2

InterBaseExpress 10-25
interface
 COM 34-9
 Type Library 에디터 34-8
 ~ 34-9
InternalCalc 필드 19-6,
 23-11
 인텍스 23-9
Internet 페이지
 (컴포넌트 팔레트) 3-30
Internet Engineering Task
 Force 27-2
Internet Information Server
 (IIS)
 버전 37-2
Internet Information
 Server (IIS) 37-1
InternetExpress 5-13, 25-
 44
 ActiveForms와 비교
 25-33 ~ 25-34
InternetExpress 페이지
 (컴포넌트 팔레트) 3-30
InTransaction 속성 17-7
Invalidate 메소드 49-9
Invoke 메소드 36-14
IObjectContext
 인터페이스 33-14, 37-3,
 39-4
 트랜잭션을 끝내는
 메소드 39-11
IObjectContext
 인터페이스 33-14, 39-2
IOleClientSite
 인터페이스 35-16
IOleDocumentSite
 인터페이스 35-16
IP 주소 32-4, 32-6
 호스트 32-4
 호스트 이름 32-4
 호스트 이름과 비교 32-4
IPageResult 29-7
IPaint 인터페이스 4-19
IPersist 인터페이스 4-17
IProducerEditorViewSupport
 29-7
IProvideClassInfo 33-17
IProviderSupport
 인터페이스 24-2
IPX/SPX 프로토콜 32-1
IRequest 인터페이스 37-4
IResponse 인터페이스 37-5
ISAPI 애플리케이션 27-6,
 27-7
 디버깅 27-9

 생성 28-1, 29-2
 요청 메시지 28-3
ISAPI DLL 13-10
IsCallerInRole 메소드 25-7,
 39-14
IScriptingContext
 인터페이스 37-3
ISecurityProperty
 인터페이스 39-15
IServer 인터페이스 37-6
ISessionObject
 인터페이스 37-6
ISetWebContentOptions
 29-7
ISpecialWinHelpViewer
 5-23
IsSecurityEnabled 39-14
IsValidChar 메소드 19-17
ItemHeight 속성 3-38
 리스트 박스 15-11
 콤보 박스 15-12
ItemIndex 속성 3-38
 라디오 그룹 3-41
Items 속성
 라디오 그룹 3-41
 라디오 컨트롤 15-14
 리스트 박스 3-38
ITypeComp 33-17
ITypeInfo 33-17
ITypeInfo2 33-17
ITypeLib 33-17
ITypeLib2 33-17
IUnknown 인터페이스 4-26,
 33-3, 33-4, 33-19
 Automation 컨트롤러
 36-14
IVarStreamable 4-35 ~
 4-36
IWebVariablesContainer
 29-6
IXMLNode 30-4 ~ 30-5,
 30-6

J

just-in-time 활성화 25-7,
 39-4 ~ 39-5
 사용 가능 39-5

K

K 각주(도움말 시스템) 47-5
KeepConnection 속성 17-3,
 17-12, 20-19
KeepConnections 속성
 20-13, 20-19

KeyDown 메소드 51-9
KeyExclusive 속성 18-29,
 18-33
KeyField 속성 15-13
KeyFieldCount 속성 18-30
KeyViolTableName 속
 성 20-53
KeywordHelp 5-30
Kind 속성
 비트맵 버튼 3-36
Kylix 10-1

L

Last 메소드 18-6
Layout 속성 3-36
 -LEpath 컴파일러
 지시어 11-12
Left 속성 3-19, 3-21,
 3-22, 6-4
LeftCol 속성 3-45
LeftPromotion 메소드 4-31,
 4-33
Length 함수 4-47
libmidas.dcu 13-6
\$LIBPREFIX 지시어 5-9
LibraryName 속성 22-4
\$LIBSUFFIX 지시어 5-9
\$LIBVERSION 지시어 5-9
.LIC 파일 38-7
Lines 속성 3-32, 42-8
LineSize 속성 3-34
LineTo 메소드 8-4, 8-7,
 8-10, 45-3
Link HTML 태그(<A>)
 28-14
Linux
 디렉토리 10-17
 운영 환경 10-14
List 속성 20-30
ListField 속성 15-13
ListSource 속성 15-12
 -LNpath 컴파일러 지시
 어 11-12
Loaded 메소드 42-13
LoadFromFile 메소드
 그래픽 8-19, 45-4
 문자열 3-50
 클라이언트 데이터셋 14-9,
 23-34
 ADO 데이터셋 21-15
LoadFromStream 메소드
 클라이언트 데이터셋
 23-34
LoadPackage 함수 11-4

- LoadParamListItems
 - 프로시저 17-14
- LoadParamsFromIniFile
 - 메소드 22-5
- LoadParamsOnConnect
 - 속성 22-5
- LocalHost 속성
 - 클라이언트 소켓 32-6
- LocalPort 속성
 - 클라이언트 소켓 32-6
- Locate 메소드 18-11
- Lock 메소드 9-7
- LockList 메소드 9-7
- LockType 속성 21-12, 21-13
- LogChanges 속성 23-5, 23-35
- Login 대화 상자 17-4
- LoginPrompt 속성 17-4
- Lookup 메소드 18-11
- LookupCache 속성 19-9
- LookupDataSet 속성 19-9, 19-11
- LookupKeyFields 속성 19-9, 19-11
- LookupResultField 속성 19-11
- lParam 매개변수 46-2
- .LPK 파일 38-7
- LPK_TOOL.EXE 38-7
- 지시어 11-12
- LUpackage 컴파일러

M

- MainMenu 컴포넌트 6-30
- MainWndProc 메소드 46-3
- make 유틸리티 10-16
- Man 페이지 5-22
- Mappings 속성 20-52
- Margin 속성 3-36
- masks 19-14
- MasterFields 속성 18-35, 22-13
- MasterSource 속성 18-35, 22-13
- Max 속성
 - 진행 표시줄 3-44
 - 트랙 표시줄 3-33
- MaxDimensions 속성 16-19
- MaxLength 속성 3-32
 - data-aware 메모 컨트롤 15-9
 - data-aware rich edit 컨트롤 15-9
- MaxRecords 속성 25-39

- MaxRows 속성 28-20
- MaxStmtsPerConn 속성 22-3
- MaxSummaries 속성 16-19
- MaxTitleRows 속성 15-23
- MaxValue 속성 19-11
- MBCS 4-43
- MDAC 13-7
- MDI 애플리케이션 5-1 ~ 5-2
 - 메뉴
 - 병합 6-41 ~ 6-42
 - 활성 지정 6-42
 - 생성 5-2
- Memo 컨트롤 3-31
- Menu 속성 6-42
- Merge 모듈 13-3
- MergeChangeLog
 - 메소드 23-7, 23-35
- \$MESSAGE 지시어 10-20
- Method 속성 28-10
- MethodType 속성 28-6, 28-10
- Microsoft Server DLL 27-6, 27-7
 - 생성 28-1, 29-2
 - 요청 메시지 28-3
- Microsoft SQL Server
 - 드라이버 배포 13-9
- Microsoft Transaction Server 5-15, 33-14, 39-1
- midas.dll 23-1, 25-3
- midaslib.dcu 25-3
- MIDI 파일 8-33
- MIDL 33-18
 - IDL 참조 항목
- mime 8-22
- MIME 메시지 27-6
- Min 속성
 - 진행 표시줄 3-44
 - 트랙 표시줄 3-33
- MinSize 속성 3-35
- MinValue 속성 19-12
- MM 필름 8-32
- Mode 속성 20-50
 - 펜 8-5
- Modified 메소드 51-11
- Modified 속성 3-32
- Modifiers 속성 3-34
- ModifyAlias 메소드 20-26
- ModifySQL 속성 20-41
- Month 속성 50-5
- MonthCalendar 컴포넌트 3-40

- MouseDown 메소드 51-8
- MouseToCell 메소드 3-44
- .MOV 파일 8-32
- Move 메소드
 - 문자열 목록 3-53, 3-54
- MoveBy 메소드 18-7
- MoveCount 속성 20-52
- MoveFile 함수 4-53
- MovePt 8-28
- MoveTo 메소드 8-4, 8-7, 45-3
- .MPG 파일 8-32
- Msg 매개변수 46-3
- MSI 기술 13-3
- MTS 5-15, 25-6, 33-11, 33-14, 39-1
 - 트랜잭션 객체 참조
 - 객체 참조 39-20 ~ 39-21
 - 런타임 환경 39-2
 - 요구 사항 39-3
 - 트랜잭션 25-19
 - 트랜잭션 객체 33-14 ~ 33-15
 - COM+와 비교 39-1
 - in-process 서버 39-2
- MTS 실행 파일 39-2
- MTS 패키지 39-6, 39-22
- MTS Explorer 39-23
- MultiSelect 속성 3-38
- Multitier 페이지 (New Items 대화 상자) 25-2
- MyBase 23-33

N

- Name 속성
 - 매개변수 18-52
 - 메뉴 항목 3-28
 - 필드 19-12
- NDX 인덱스 20-7
- NetCLX 5-11, 10-6
- NetFileDir 속성 20-24
- Netscape Server DLL 27-6
 - 생성 28-1, 28-2, 29-2
 - 요청 메시지 28-3
- neutral 스레드 36-9
- New 명령 40-11
- New Field 대화 상자 19-5
 - 타입 19-6
 - 필드 정의 19-6, 19-7, 19-8, 19-10
 - 필드 타입 19-6
 - Field properties 19-5
 - Lookup definition 19-6
 - 키 필드 (Key Fields) 19-9

- Dataset 19-9
- Lookup Keys 19-9
- Result Field 19-9
- New Items 대화 상자 5-20, 5-21
- New Thread Object 대화 상자 9-2
- NewValue 속성 20-39, 24-12
- Next 메소드 18-6
- NextRecordSet 메소드 18-54, 22-9
- NOT NULL 제약 조건 24-13
- NOT NULL UNIQUE 제약 조건 24-13
- NotifyID 5-24
- NSAPI 애플리케이션 27-6
- 디버깅 27-9
- 생성 28-1, 28-2, 29-2
- 요청 메시지 28-3
- Null 값
- 범위 18-32
- Null 종료
- 와이드 문자열 4-42
- NumericScale 속성 18-46, 18-52
- NumGlyphs 속성 3-36

O

- Object Broker 25-28
- Object HTML tag (<OBJECT>) 28-14
- Object Inspector 3-7, 3-25, 42-2, 47-6
- 도움말 47-4
- 메뉴 선택 6-38
- 배열 속성 편집 42-2
- Object Repository 5-20 ~ 5-22, 6-12
- 공유 디렉토리 지정 5-20
- 항목 사용 5-21
- 항목 추가 5-20
- Object Repository
- 데이터베이스 컴포넌트 20-16
- 세션 20-18
- ObjectBroker 속성 25-25, 25-26, 25-27, 25-28
- ObjectContext 속성
- 예제 39-13
- ObjectName 속성
- TCorbaConnection 25-28
- Objects 속성 3-45
- 문자열 목록 3-54, 7-16
- ObjectView 속성 15-22, 18-37, 19-23
- .OCX 파일 13-5
- ODBC 드라이버
- ADO 사용 21-1, 21-2
- BDE 사용 20-1, 20-15, 20-16
- ODL (Object Description Language) 33-16, 34-1
- OEM 문자 집합 12-2
- OEMConvert 속성 3-32
- OldValue 속성 20-39, 24-12
- OLE
- 메뉴 병합 6-41
- 컨테이너 3-24
- OLE Automation Automation 참조
- OLE DB 21-1, 21-2
- OleObject 속성 38-13, 38-14
- OLEView 33-18
- OnAccept 이벤트
- 서버 소켓 32-9
- OnAction 이벤트 28-8
- OnAfterPivot 이벤트 16-9
- OnBeforePivot 이벤트 16-9
- OnBeginTransComplete 이벤트 17-7, 21-8
- OnCalcFields 이벤트 18-22, 19-7, 19-8, 23-11
- OnCellClick 이벤트 15-26
- OnChange 이벤트 19-15, 45-7, 49-7, 50-11, 51-11
- OnClick 이벤트 3-35, 43-1, 43-2, 43-5
- 메뉴 3-28
- 버튼 3-6
- OnClientConnect 이벤트 32-7
- OnClientDisconnect 이벤트 32-7
- OnColEnter 이벤트 15-26
- OnColExit 이벤트 15-26
- OnColumnMoved 이벤트 15-19, 15-26
- OnCommitTransComplete 이벤트 17-8, 21-8
- OnConnect 이벤트
- 클라이언트 소켓 32-8
- OnConnectComplete 이벤트 21-8
- OnConnecting 이벤트
- 서버 소켓 32-9
- OnConstrainedResize 이벤트 6-4
- OnCreate 이벤트 40-13
- OnDataChange 이벤트 15-4, 51-6, 51-10
- OnDataRequest 이벤트 23-32, 24-3, 24-12
- OnDbClick 이벤트 15-26, 43-5
- OnDecisionDrawCell 이벤트 16-12
- OnDecisionExamineCell 이벤트 16-13
- OnDeleteError 이벤트 18-20
- OnDisconnect 이벤트 21-8
- 클라이언트 소켓 32-7
- OnDragDrop 이벤트 7-2, 15-26, 43-5
- OnDragOver 이벤트 7-2, 15-26, 43-5
- OnDrawCell 이벤트 3-44
- OnDrawColumnCell 이벤트 15-26
- OnDrawDataCell 이벤트 15-26
- OnDrawItem 이벤트 7-16
- OnEditButtonClick 이벤트 15-21, 15-26
- OnEditError 이벤트 18-17
- OnEndDrag 이벤트 7-3, 15-26, 43-5
- OnEnterPage 메소드 37-2
- OnEnter 이벤트 15-26, 43-5
- OnError 이벤트
- 소켓 32-8
- OnExecuteComplete 이벤트 21-9
- OnExit 이벤트 15-27, 51-12
- OnFilterRecord 이벤트 18-13, 18-15
- OnGetData 이벤트 24-8
- OnGetDataSetProperties 이벤트 24-7
- OnGetTableName 이벤트
- 트 20-11, 23-22, 24-12
- OnGetText 이벤트 19-15, 19-16
- OnGetThread 이벤트 32-9
- OnHandleActive 이벤트
- 클라이언트 소켓 32-8
- OnHTMLTag 이벤트 25-44, 28-15, 28-16, 28-17
- OnIdle 이벤트 핸들러 9-5

- OnInfoMessage 이벤트 21-9
- OnKeyDown 이벤트 15-27, 43-5, 51-9
- OnKeyPress 이벤트 15-27, 43-5
- OnKeyUp 이벤트 15-27, 43-5
- OnLayoutChange 이벤트 16-9
- OnLogin 이벤트 17-5
- OnMeasureItem 이벤트 7-15
- OnMouseDown 이벤트 8-24, 8-25, 43-5, 51-8
전달된 매개변수 8-24
- OnMouseMove 이벤트 8-24, 8-26, 43-5
전달된 매개변수 8-24
- OnMouseUp 이벤트 8-14, 8-24, 8-25, 43-5
전달된 매개변수 8-24
- OnNewDimensions 이벤트 16-9
- OnNewRecord 이벤트 18-18
- OnPaint 이벤트 3-47, 8-2
- OnPassword 이벤트 20-13, 20-23
- OnPopup 이벤트 7-12
- OnPostError 이벤트 18-20
- OnReceive 32-10
- OnReceive 이벤트 32-8
- OnReconcileError 이벤트 10-29, 20-33, 23-21, 23-24
- OnRefresh 이벤트 16-7
- OnRequestRecords 이벤트 25-39
- OnResize 이벤트 8-2
- OnRollbackTransComplete 이벤트 17-9, 21-8
- OnScroll 이벤트 3-33
- OnSend 32-10
- OnSend 이벤트 32-8
- OnSetText 이벤트 19-15, 19-16
- OnStartDrag 이벤트 15-27
- OnStartPage 메소드 37-2
- OnStartup 이벤트 20-18
- OnStateChange 이벤트 15-4, 16-9, 18-4
- OnSummaryChange 이벤트 16-9
- OnTerminate 이벤트 9-6
- OnTitleClick 이벤트 15-27
- OnTranslate 이벤트 26-7
- OnUpdateData 이벤트 15-4, 24-8, 24-9
- OnUpdateError 이벤트 10-29, 20-33, 20-38 ~ 20-40, 23-23, 24-11
- OnUpdateRecord 이벤트 20-33, 20-37 ~ 20-38, 20-41, 20-47
- OnValidate 이벤트 19-15
- OnWillConnect 이벤트 17-5, 21-8
- Open 메소드
데이터셋 18-4
서버 소켓 32-7
세션 20-18
연결 컴포넌트 17-3
쿼리 18-48
- OpenDatabase 메소드 20-18, 20-20
- OpenSession 메소드 20-29, 20-30
- Options 속성 3-45
데이터 그리드 15-24
의사 결정 그리드 16-12
프로바이더 24-5 ~ 24-6
TSQLClientDataSet 23-17
- OpenString 4-42
- Oracle 드라이버
배포 13-9
- Oracle 테이블 20-12
- Oracle8
테이블 생성 시의 제한 사항 18-40
- ORDER BY 절 18-26
- Orientation 속성
데이터 그리드 15-28
트랙 표시줄 3-33
- Origin 속성 8-28, 19-12
- out-of-process 서버 33-7
ASP 37-7
- Overload 속성 20-12
- override 지시어 41-8, 46-4
- Owner 속성 3-12, 40-13
- OwnerDraw 속성 7-13
- owner-draw 컨트롤 3-54, 7-12
그리기 7-15, 7-16
리스트 박스 3-38, 3-39
선언 7-13
크기 조정 7-15
- 동적 로딩 11-4
- Package Collection
에디터 11-13
- PacketRecords 속성 10-29, 20-34, 23-27
- PageSize 속성 3-34
- Paint 메소드 45-6, 49-8, 49-9
- PaletteChanged 메소드 45-5
- PaletteChanged 속성 10-22
- PanelHeight 속성 15-28
- Panels 속성 3-44
- PanelWidth 속성 15-28
- PAnsiChar 4-42
- PAnsiString 4-47
- Paradox 테이블 20-3, 20-5
네트워크 제어 파일 20-24
데이터 액세스 20-9
디렉토리 20-24 ~ 20-25
로컬 트랜잭션 20-32
레코드 추가 18-19
배치 이동 20-53
암호 보호 20-22 ~ 20-24
이름 재지정 20-8
인덱스 검색 18-26
DatabaseName 20-3
- ParamBindMode 속성 20-12
- ParamByName 메소드
내장 프로시저 18-53
쿼리 18-46
- ParamCheck 속성 18-45, 22-12
- Parameters 속성 21-20
TADOCommand 21-20
TADOQuery 18-45
TADOStoredProc 18-51
- ParamName 속성 25-42
- Params 속성
내장 프로시저 18-51
쿼리 18-45, 18-47
클라이언트 데이터셋 23-28, 23-29
TDatabase 20-15
TSQLConnection 22-4
XML 브로커 25-39
- ParamType 속성 18-46, 18-52
- ParamValues 속성 18-47
- Parent 속성 40-13
- ParentColumn 속성 15-23
- ParentShowHint 속성 3-44
- passthrough SQL 20-31, 20-31 ~ 20-32
- PasteFromClipboard
메소드 7-10

P

- \$P 컴파일러 지시어 4-49
- package

- 그래픽 15-10
 - data-aware 메모 컨트롤 15-9
- PathInfo 속성 28-6
- .PCE 파일 11-13
- PChar 4-42
 - 문자열 변환 4-47
- PDOXUSRS.NET 20-24
- Pen 속성 8-4, 8-5, 45-3
- PenPos 속성 8-4, 8-7
- PENWIN.DLL 11-11
- PickList 속성 15-20, 15-21
- picture 객체 8-3, 45-4
- Picture 속성 3-46, 8-17
 - 프레임 6-15
- Pie 메소드 8-4
- Pixel 속성 8-4, 45-3
- Pixels 속성 8-5, 8-9
- pmCopy 상수 8-29
- pmNotXor 상수 8-29
- Polygon 메소드 8-4, 8-12
- PolyLine 메소드 8-4, 8-10
- PopupMenu 속성 7-11
- PopupMenu 컴포넌트 6-30
- Port 속성
 - 서버 소켓 32-7
 - TSocketConnection 25-26
- Position 속성 3-34, 3-44
- Post 메소드 18-20
 - Edit 메소드 18-18
- Precision 속성
 - 매개변수 18-46, 18-52
 - 필드 19-12
- Prepared 속성
 - 단방향 데이터셋 22-9
 - 쿼리 18-48
- PRIMARY KEY 제약 조건 24-13
- Prior 메소드 18-7
- Priority 속성 9-3
- private 3-9
- private 섹션 4-2
- private 속성 42-5
- PrivateDir 속성 20-25
- ProblemCount 속성 20-53
- ProblemTableName 속성 20-53
- ProcedureName 속성 18-50
- Project Manager 6-2
- Project Options 대화 상자 5-3
- Property Page 마법사 ?? ~ 38-12
- Proportional 속성 8-2

- protected 3-9
 - 이벤트 43-5
 - 지시어 43-5
 - 클래스의 부분 41-5
 - 키워드 42-3, 43-5
- protected 섹션 4-2
- ProviderFlags 속성 24-5, 24-10
- ProviderName 속성 14-12, 23-26, 24-3, 25-24, 25-39, 26-9
- PString 4-47
- public 3-9
 - 속성 42-11
 - 지시어 43-5
 - 클래스의 부분 41-6
 - 키워드 43-5
- public 섹션 4-2
- published 3-9, 42-3
 - 속성 42-11, 42-12
 - 예제 49-2, 50-2
 - 지시어 42-3, 43-5, 52-3
 - 클래스의 부분 41-6
 - 키워드 43-5
- PVCS Version Manager 2-5
- PWideChar 4-42
- PWideString 4-47

Q

- QReport 페이지 (컴포넌트 팔레트) 3-30
- Qt 라이브러리 10-22
- Qt widget 10-13
- Query 속성
 - 업데이트 객체 20-48
- QueryBuilder 18-44
- QueryInterface 메소드 4-20, 4-25, 33-4
 - 집합체 33-9
- query-type datasets
 - queries 참조

R

- .RC 파일 6-42
- RDBMS 14-3, 25-1
- RDSCConnection 속성 21-17
- Read 메소드
 - TFileStream 4-56
- read 메소드 42-6
- read 예약어 42-8
- ReadBuffer 메소드
 - TFileStream 4-57
- ReadCommitted 17-9
- README 13-15

- ReadOnly 속성 3-32, 51-3, 51-8, 51-9
 - 데이터 그리드 15-20, 15-25
 - 테이블 18-38
 - 필드 19-12
 - data-aware 메모 컨트롤 15-9
 - data-aware 컨트롤 15-5
 - data-aware rich edit 컨트롤 15-9
- ReasonString 속성 28-12
- rebars 6-43, 6-48
- ReceiveBuf 메소드 32-8
- ReceiveIn 메소드 32-8
- RecNo 속성
 - 클라이언트 데이터셋 23-2
- Reconcile 메소드 10-29, 20-34
- RecordCount 속성
 - TBatchMove 20-52
- RecordSet 속성 21-19
- Recordset 속성 21-10
- RecordsetState 속성 21-11
- RecordStatus 속성 21-12, 21-14
- Rectangle 메소드 8-5, 8-11, 45-3
- Reduced XML Data file
 - XDR file 참조
- Refresh 메소드 15-7, 23-31
- RefreshLookupList 속성 19-10
- RefreshRecord 메소드 23-31, 24-4
- Register 메소드 8-3
- Register 프로시저 40-12, 47-2
- RegisterComponents
 - 프로시저 11-6, 40-12, 47-2
- RegisterConversionType
 - 함수 4-60, 4-61
- RegisterHelpViewer 5-32
- RegisterNonActiveX
 - 프로시저 38-3
- RegisterPooled 프로시저 25-8
- RegisterPropertyEditor
 - 프로시저 47-12
- RegisterTypeLib 함수 33-18
- RegisterViewer 함수 5-28
- Registry 12-10
- REGSERV32.EXE 13-5

Release 메소드 4-20, 4-23,
4-24, 4-25, 33-4
 TCriticalSection 9-7
RemoteHost 속성
 클라이언트 소켓 32-6
RemotePort 속성
 클라이언트 소켓 32-6
RemoteServer 속성 23-26,
25-24, 25-29, 25-36,
25-39, 26-9
RemoveAllPasswords
 메소드 20-23
RemovePassword
 메소드 20-23
RenameFile 함수 4-53,
4-54
RepeatableRead 17-9
Repository 대화 상자 5-20
RepositoryID 속성 25-25,
25-28
RequestLive 속성 20-10
RequestRecords 메소드
 25-39
ResetEvent 메소드 9-10
ResolveToDataSet 속성
 24-4
resourcestring 예약어 12-10
RestoreDefaults 메소드
 15-22
Result 매개변수 46-6
Resume 메소드 9-11
ReturnValue 속성 9-9
RevertRecord 메소드 10-29,
20-34, 23-6
RFC 문서 27-2
rich text edit 컨트롤 3-31
rich-text memo fields 15-2
RightPromotion 메소드 4-31,
4-33
Rollback 메소드 17-8
RollbackTrans 메소드 17-9
RoundRect 메소드 8-5,
8-11
RowAttributes 속성 28-19
RowCount 속성 15-13,
15-28
RowHeights 속성 3-45,
7-15
RowRequest 메소드 24-4
Rows 속성 3-45
RowsAffected 속성 18-49
RPC 33-9
RTL 10-6
RTTI 41-6

호출 가능한
 인터페이스 31-2
\$RUNONLY 컴파일
 지시어 11-10

S

safe reference 39-20
SafeArray 34-13
safecall 호출 규칙 34-9
SafeRef 메소드 39-20
Samples 페이지
 (컴포넌트 팔레트) 3-30, 3-31
Save as Template 명령(메뉴
 디자이너) 6-38, 6-40
Save Template 대화 상자
 6-40
SaveConfigFile 메소드
 20-26
SavePoint 속성 23-6
SaveToFile 메소드 8-20
 그래픽 45-4
 문자열 3-50
 클라이언트 데이터셋 14-9,
 23-35, 23-36
 ADO 데이터셋 21-15
SaveToStream 메소드
 클라이언트 데이터셋
 23-36
ScaleBy 속성
 TCustomForm 13-13
Scaled 속성
 TCustomForm 13-13
ScanLine 속성
 비트맵 8-9
 비트맵 예제 8-18
ScktSrvr.exe 25-9, 25-13,
25-26
SCM 5-4
Screen 변수 6-3, 12-9
ScriptAlias 지시어 13-11
ScrollBars 속성 3-45, 7-8
 data-aware 메모 컨트롤
 롤 15-9
SDI 애플리케이션 5-1 ~ 5-2
Sections 속성 3-42
Seek 메소드
 ADO 데이터셋 18-28
Select Menu 대화 상자 6-38
Select Menu 명령(메뉴
 디자이너) 6-38
SELECT 문 18-43
SelectAll 메소드 3-32
SelectCell 메소드 50-12,
51-3

Selection 속성 3-44
SelectKeyword 5-27
SelEnd 속성 3-33
Self 매개변수 40-13
SelLength 속성 3-32, 7-9
SelStart 속성 3-32, 3-33,
7-9
SelText 속성 3-32, 7-9
SendBuf 메소드 32-8
Sender 매개변수 3-27
 예제 8-7
Sendln 메소드 32-8
SendStream 메소드 32-8
ServerGUID 속성 25-24
ServerName 속성 25-24
Servers 페이지(컴포넌트
 팔레트) 3-30, 3-31
Service Start 이름 5-8
Session 변수 20-3, 20-17
SessionName 속성 20-3,
20-13, 20-29, 28-18
Sessions 변수 20-18, 20-29
Sessions 속성 20-30
SetAbort 메소드 39-4,
39-8, 39-11
SetBrushStyle 메소드 8-8
SetComplete 메소드 25-18,
39-4, 39-8, 39-11
SetData 메소드 19-17
SetEvent 메소드 9-10
SetFields 메소드 18-21
SetFloatValue 메소드 47-8
SetKey 메소드 18-28
 EditKey와 비교 18-30
SetLength 프로시저 4-47
SetMethodValue 메소드
 47-8
SetOptionalParam
 메소드 23-16
SetOrdValue 메소드 47-8
SetPenStyle 메소드 8-7
SetProvider 메소드 23-26
SetRange 메소드 18-32,
18-33
SetRangeEnd 메소드 18-32
 SetRange와 비교 18-32
SetRangeStart 메소드 18-31
 SetRange와 비교 18-32
SetSchemaInfo 메소드
 22-13
SetStrValue 메소드 47-8
SetValue 메소드 47-8, 47-9
Shape 속성 3-46

Shared Property
 Manager 39-6 ~ 39-7
 예제 39-7
 Shift 상태 8-24
 ShortCut 속성 6-34
 ShortString 4-41
 Show 메소드 6-7, 6-8
 ShowAccelChar 속성 3-43
 ShowButtons 속성 3-39
 ShowColumnHeaders
 속성 3-40
 ShowFocus 속성 15-28
 ShowHint 속성 3-44, 15-30
 ShowHintChanged 속성
 10-22
 ShowLines 속성 3-39
 ShowModal 메소드 6-6
 ShowRoot 속성 3-39
 ShutDown 5-24
 Simple Object Access
 Protocol SOAP 참조
 Size 속성
 매개변수 18-46, 18-52
 필드 19-12
 SOAP 31-1
 다계층 애플리케이션
 25-10
 애플리케이션 서버에
 연결 25-27
 SOAP 데이터 모듈 25-6
 SOAP 연결 25-10, 25-27
 SOAP 포트 팩킷 31-7
 SOAP Data Module
 마법사 25-16
 SoftShutDown 5-24
 Sorted 속성 3-38, 15-12
 SortFieldNames 속성 22-7
 SourceXml 속성 26-6
 SourceXmlDocument
 속성 26-6
 SourceXmlFile 속성 26-6
 Spacing 속성 3-36
 SparseCols 속성 16-9
 SparseRows 속성 16-9
 SPX/IPX 20-16
 SQL 14-3, 20-9
 명령 실행 17-10 ~ 17-12
 지역 20-9
 표준 24-13
 Decision Query
 에디터 16-6
 SQL 모니터 20-55
 SQL 문
 매개변수 17-11
 생성
 프로바이더 24-4,
 24-10 ~ 24-11
 TSQLDataSet 22-9
 실행 22-10 ~ 22-11
 업데이트 객체 20-41 ~
 20-45
 의사 결정 데이터셋 16-4,
 16-5
 클라이언트 제공 23-32,
 24-6
 프로바이더 생성 24-12
 passthrough SQL 20-31
 SQL 서버
 로그인 14-4
 SQL 속성 18-44
 변경 18-48
 SQL 연결 13-8, 20-1
 드라이버 20-9, 20-15,
 20-32
 드라이버 파일 13-9
 배포 13-9, 13-15
 SQL 쿼리 18-43 ~ 18-44
 결과 집합 18-49
 매개변수 18-45 ~ 18-47,
 20-43
 디자인 타임 시 설정
 18-45
 런타임 시 설정 18-46
 마스터/디테일 관계 18-
 47 ~ 18-48
 바인딩 18-45
 복사 18-44
 수정 18-44
 실행 18-49
 업데이트 객체 20-47
 준비 18-48
 최적화 18-49
 파일에서 로드 18-44
 SQL 탐색기 25-3
 속성 집합 정의 19-13
 SQL Builder 18-44
 SQLConnection 속성 22-3,
 22-18
 SQLPASSTHRUMODE
 20-32
 Standard 페이지
 (컴포넌트 팔레트) 3-29
 StartTransaction 메소드
 17-7
 State 속성 3-36
 그리드 15-16, 15-18
 그리드 열 15-16
 데이터셋 18-3, 19-8
 StatusCode 속성 28-11
 StatusFilter 속성 10-29,
 20-33, 21-13, 23-6,
 23-20, 24-9
 StdConvts 유닛 4-60, 4-61
 Step 속성 3-44
 stored 지시어 42-12
 stored procedure-type
 datasets
 stored procedures 참조
 StoredProcName 속성
 18-50
 StrByteType 4-44
 Stretch 속성 15-10
 StretchDraw 메소드 8-5,
 45-3, 45-7
 String List Editor
 표시 15-11
 Strings 속성 3-52
 StrNextChar 함수 10-18
 Structured Query Language
 SQL 참조
 Style 속성 3-38, 7-13
 리스트 박스 3-38
 브러시 3-46, 8-8
 웹 항목 25-42
 콤보 박스 3-39, 15-11
 툴 버튼 6-48
 펜 8-5
 StyleChanged 속성 10-22
 StyleRule 속성 25-42
 Styles 속성 25-42
 StylesFile 속성 25-42
 subscriber 객체 35-15 ~
 35-16
 사용자별 예약 35-16
 일시적 예약 35-15
 지속적 예약 35-15
 Subtotals 속성 16-12
 SupportCallbacks 속성
 25-18
 Suspend 메소드 9-11
 Sybase 드라이버
 배포 13-9
 Synchronize 메소드 9-4
 System 페이지
 (컴포넌트 팔레트) 3-29

T

Table HTML 태그
 (<TABLE>) 28-14
 TableAttributes 속성 28-19
 TableName 속성 18-25,
 18-39, 22-7
 TableOfContents 5-27
 table-type datasets

tables 참조
 TableType 속성 18-39, 20-5 ~ 20-6
 TabOrder 속성 3-22
 Tabs 속성 3-42
 TabStop 속성 3-22
 TabStopChanged 속성 10-22
 TAction 6-20
 TActionClientItem 6-22
 TActionList 6-18, 6-19
 TActionMainMenuBar 6-16, 6-18, 6-19, 6-21
 TActionManager 6-16, 6-18, 6-19
 TActionToolBar 6-16, 6-18, 6-19, 6-21
 TActiveForm 38-3, 38-6
 TAdapterDispatcher 29-13
 TAdapterPageProducer 29-10
 TADOCCommand 21-2, 21-7, 21-10, 21-17 ~ 21-20
 TADOConnection 14-8, 17-1, 21-2, 21-2 ~ 21-9, 21-10
 데이터 저장소에 연결 21-3 ~ 21-5
 TADODataSet 21-2, 21-9, 21-16 ~ 21-17
 TADOQuery 21-2, 21-9
 SQL 명령 21-17
 TADOStoredProc 21-2, 21-9
 TADOTable 21-2, 21-9
 Tag 속성 19-12
 TApacheApplication 27-7
 TApacheRequest 27-7
 TApacheResponse 27-7
 TApplication 5-23, 5-30, 10-7
 TApplicationEvents 6-3
 TASM 코드 10-18
 TASPObject 37-2
 TBatchMove 20-49 ~ 20-53
 오류 처리 20-53
 TBCDField
 기본 서식 19-15
 TBDEClientDataSet 20-2
 TBDEDataSet 18-2
 TBevel 3-46
 TBitmap 45-4
 tbsCheck 상수 6-48
 TCalendar 50-1
 TCanvas
 사용 3-56
 TCGIApplication 27-7
 TCGIRequest 27-7
 TCGIResponse 27-7
 TcharProperty 타입 47-7
 TclassProperty 타입 47-7
 TClientDataSet 23-18
 TClientDataset 5-19
 TClientSocket 32-6
 TcolorProperty 타입 47-7
 TComObject
 추상화(aggregation) 4-24
 TComponent 3-12, 3-15, 40-5
 TcomponentProperty
 타입 47-7
 TControl 3-16, 3-18, 40-4, 43-5
 공통 속성 3-18
 공통 이벤트 3-20
 TConvType 값 4-60
 TConvTypeInfo 4-64
 TCoolBand 3-37
 TCoolBar 6-43
 TCorbaConnection 25-28
 TCorbaDataModule 25-5
 TCP/IP 20-16, 32-1
 다계층 애플리케이션 25-9 ~ 25-10
 서버 32-7
 애플리케이션 서버에 연결 25-25
 클라이언트 32-6
 TCurrencyField
 기본 서식 19-15
 TCustomADODataset 18-2
 TCustomClientDataSet 18-2
 TCustomContentProducer 28-13
 TCustomControl 40-4
 TCustomEdit 10-8
 TCustomGrid 50-1, 50-2
 TCustomIniFile 3-55
 TCustomizedDlg 6-22
 TCustomListBox 40-3
 TCustomVariantType 4-28 ~ 4-36
 TDatabase 14-8, 17-1, 20-3, 20-13 ~ 20-16
 임시 인스턴스 20-20
 끊기 20-21
 DatabaseName 속성 20-3
 TDataSet 18-1
 자손 18-2 ~ 18-3
 TDataSetProvider 24-1, 24-2
 TDataSetTableProducer 28-20
 TDataSource 15-3 ~ 15-4
 TDateField
 기본 서식 19-15
 TDateTime 타입 50-5
 TDateTimeField
 기본 서식 19-15
 TDBChart 14-15
 TDBCheckBox 15-2, 15-13 ~ 15-14
 TDBComboBox 15-2, 15-10, 15-11 ~ 15-12
 TDBCtrlGrid 15-2, 15-27 ~ 15-28
 속성 15-28
 TDBEdit 15-2, 15-8
 TDBGrid 15-2, 15-15 ~ 15-27
 속성 15-20
 이벤트 15-26
 TDBGridColumn 15-16
 TDBImage 15-2, 15-10
 TDBListBox 15-2, 15-10, 15-11 ~ 15-12
 TDBLookupComboBox 15-2, 15-10, 15-12 ~ 15-13
 TDBLookupListBox 15-2, 15-10, 15-12 ~ 15-13
 TDBMemo 15-2, 15-9
 TDBNavigator 15-2, 15-28 ~ 15-31, 18-5, 18-6
 TDBRadioGroup 15-2, 15-14
 TDBRichEdit 15-2, 15-9 ~ 15-10
 TDBText 15-2, 15-8
 TDCOMConnection 25-25
 TDecisionCube 16-1, 16-4, 16-7 ~ 16-8
 이벤트 16-7
 TDecisionDrawState 16-12
 TDecisionGraph 16-1, 16-2, 16-13 ~ 16-18
 TDecisionGrid 16-1, 16-2, 16-10 ~ 16-13
 속성 16-12
 이벤트 16-12
 TDecisionPivot 16-1, 16-2, 16-9 ~ 16-10

속성 16-10
 TDecisionQuery 16-1,
 16-5, 16-6
 TDecisionSource 16-1,
 16-8 ~ 16-9
 속성 16-9
 이벤트 16-9
 TDefaultEditor 47-16
 TDependency_object 5-8
 TDragObject 7-3
 TDragObjectEx 7-3
 TenumProperty 타입 47-7
 Terminate 메소드 9-6
 Terminated 속성 9-6
 TEvent 9-9
 Text 속성 3-32, 3-39,
 3-44
 TextHeight 메소드 8-5,
 45-3
 TextOut 메소드 8-5, 45-3
 TextRect 메소드 8-5, 45-3
 TextWidth 메소드 8-5,
 45-3
 TField 18-1, 19-1 ~
 19-28
 메소드 19-16
 속성 19-1, 19-11 ~
 19-15
 런타임 19-12
 이벤트 19-15 ~ 19-16
 TFieldDataLink 51-4
 TFile 4-54
 TFileStream 3-58, 4-54
 파일 I/O 4-54 ~ 4-58
 TFloatField
 기본 서식 19-15
 TfontProperty 타입 47-7
 TFMTBcdField
 기본 서식 19-15
 TfontNameProperty
 타입 47-8
 TfontProperty 타입 47-8
 TForm
 스크롤 막대 속성 3-33
 TForm 컴포넌트 3-5
 TFrame 6-14
 TGraphic 45-4
 TGraphicControl 40-4,
 49-2
 THeaderControl 3-42
 Thread Status 상자 9-12
 thread-aware 객체 9-4
 ThreadID 속성 9-12
 threadvar 9-5
 THTMLTableAttributes
 28-19
 THTMLTableColumn 28-20
 THTTTPrio 31-9
 THTTTPsoapDispatcher
 31-2, 31-3
 THTTTPSOAPPascalInvoker
 31-3
 THTTTPsoapPascalInvoker
 31-2
 TIBCustomDataSet 18-2
 TIBDatabase 14-9, 17-1
 TickMarks 속성 3-33
 TickStyle 속성 3-33
 TIcon 45-4
 TiledDraw 메소드 45-7
 TImage
 프레임 6-15
 TImageList 6-47
 TIniFile 3-54
 TintegerProperty 타입
 47-7, 47-9
 TInterfacedObject 4-24
 인터페이스 구현 4-20
 파생 4-21
 TInvokableClass 31-6
 TInvokeableVariantType
 4-37 ~ 4-38
 TISAPIApplication 27-6
 TISAPIRequest 27-6
 TISAPIResponse 27-6
 Title 속성
 데이터 그리드 15-20
 TKeyPressEvent 타입 43-3
 TLabel 3-43, 40-4
 .TLB 파일 33-17, 34-2,
 34-26
 TLIBIMP 33-18, 35-5,
 36-15
 tlibimp.exe 35-2
 TListBox 40-3
 TLocalConnection 23-26
 TMemIniFile 3-54, 10-8
 TMemoryStream 3-58
 TMessage 46-4, 46-6
 TMetafile 45-4
 TmethodProperty 타입 47-7
 TMsg 6-5
 TMTSDDataModule 25-5
 TMultiReadExclusiveWrite
 Synchronizer 9-8
 TNestedDataSet 18-37
 TNotifyEvent 43-7
 TObject 3-12, 4-1, 41-4
 ToCommon 4-64
 ToggleButton 10-8
 TOleContainer 35-16
 활성 문서 33-14
 TOleControl 35-6
 TOleServer 35-6
 Top 속성 3-19, 3-21,
 3-22, 6-4, 6-45
 TopRow 속성 3-45
 TordinalProperty 타입 47-7
 TPageControl 3-42
 TPageDispatcher 29-13
 TPageProducer 28-14
 TPaintBox 3-47
 TPanel 3-41, 6-43
 tpHigher 상수 9-3
 tpHighest 상수 9-3
 TPicture 타입 45-4
 tpIdle 상수 9-3
 tpLower 상수 9-3
 tpLowest 상수 9-3
 tpNormal 상수 9-3
 TPopupMenu 6-49
 TPrinter 3-57
 사용 3-57
 TPropertyAttributes 47-11
 TPropertyEditor 클래스
 47-7
 TPropertyPage 38-12
 tpTimeCritical 상수 9-3
 TPublishableVariantType
 4-39
 TQuery 20-2, 20-9 ~
 20-11
 의사 결정 데이터셋 16-5
 TQueryTableProducer
 28-21
 Transactional Data Module
 마법사 25-15
 Transactional Object 마법
 사 39-15 ~ 39-18
 TransformGetData 속성
 26-9
 TransformRead 속성 26-8
 TransformSetParams
 속성 26-10
 TransformWrite 속성 26-8
 TransIsolation 속성 17-10
 로컬 트랜잭션 20-32
 Translation Tools 12-1
 Transliterate 속성 19-12,
 20-50
 Transparent 속성 3-43
 TReader 4-54
 TRegIniFile 10-8
 TRegistry 3-54

TRegistryIniFile 3-55
 TRegSvr 13-5, 33-18
 TRemotable 31-5
 TRemoteDataModule 25-5
 try 예약어 45-6, 52-5
 TScrollBar 3-33, 3-41
 TSearchRec 4-51
 TServerSocket 32-7
 TService_object 5-8
 TSession 20-17 ~ 20-30
 추가 20-28, 20-29
 TsetElementProperty
 타입 47-7
 TsetProperty 타입 47-7
 TSharedConnection 25-32
 TSoapDataModule 25-6
 TSocketConnection 25-25
 TSpinEdit 컨트롤 3-34
 TSQLClientDataSet 22-2
 TSQLConnection 14-8,
 17-1, 22-2 ~ 22-5
 메시지 모니터링 22-18
 바인딩 22-3 ~ 22-5
 TSQLDataSet 22-2, 22-6,
 22-7, 22-8
 TSQLMonitor 22-18 ~
 22-19
 TSQLQuery 22-2, 22-6
 TSQLStoredProc 22-2,
 22-8
 TSQLTable 22-2, 22-7
 TSQLTimeStampField
 기본 서식 19-15
 TStoredProc 20-2, 20-12
 ~ 20-13
 TStream 3-58
 TStringList 3-49 ~ 3-54,
 5-26
 TstringProperty 타입 47-7
 TStringStrings 3-49 ~ 3-54
 TStringStream 3-58
 TTabControl 3-42
 TTable 20-2, 20-4 ~ 20-9
 의사 결정 데이터셋 16-5
 TThread 9-2
 TThreadList 9-5, 9-7
 TTimeField
 기본 서식 19-15
 TToolBar 6-19, 6-43,
 6-46
 TToolButton 6-43
 TTreeView 3-39
 TTypedComObject
 타입 라이브러리 요구
 사항 33-17
 TUpdateSQL 20-40 ~
 20-49
 프로바이더 20-11
 TVarData 레코드 4-27
 TWebActionItem 28-3
 TWebAppDataModule 29-5
 TWebApplication 27-6
 TWebAppPageModule 29-5
 TWebConnection 25-27
 TWebContext 29-13
 TWebDataModule 29-5,
 29-7
 TWebDispatcher 29-13
 TWebPageModule 29-5,
 29-7
 TWebRequest 27-6
 TWebResponse 27-6, 28-3
 TWidgetControl 10-6
 TWinCGIRequest 27-7
 TWinCGIResponse 27-7
 TWinControl 3-17, 10-6,
 12-9, 40-4, 43-5
 공통 속성 3-21
 공통 이벤트 3-22
 TWriter 4-54
 TWSDLHTMLPublish 31-7
 TXMLDocument 30-3,
 30-8
 TXMLTransform 26-6 ~
 26-8
 소스 문서 26-6
 TXMLTransformClient 26-9
 ~ 26-11
 매개변수 26-9
 TXMLTransformProvider
 24-1, 24-2, 26-8 ~ 26-9
 Type Library 에디터 34-2 ~
 34-26
 객체 목록 창 34-5
 구문 34-11, 34-13 ~
 34-18
 디스패치 인터페이스
 34-15
 라이브러리 열기 34-19 ~
 34-20
 레코드 34-17
 레코드 및 유니온 34-10
 추가 34-23 ~ 34-24
 메소드
 추가 34-21 ~ 34-22
 모듈 34-10, 34-18
 추가 34-24
 바인딩 속성 38-11
 별칭 34-10, 34-17
 추가 34-23
 부분 34-3 ~ 34-8
 상태 표시줄 34-5
 속성
 추가 34-21 ~ 34-22
 업데이트 34-25
 열거 타입 34-9 ~ 34-10,
 34-16
 추가 34-23
 오류 메시지 34-5, 34-8,
 34-25
 오브젝트 파스칼과 IDL
 비교 34-11, 34-13 ~
 34-18
 요소 34-8 ~ 34-10
 공통 특징 34-8
 요소 선택 34-5
 유니온 34-18
 인터페이스 34-8, 34-15
 수정 34-20 ~ 34-22
 추가 34-20
 타입 정보 페이지 34-6 ~
 34-8
 메소드용 34-7
 모듈용 34-7
 별칭용 34-6
 상수용 34-7
 속성용 34-7
 열거용 34-6
 유니온용 34-6
 인터페이스용 34-6
 타입 라이브러리용 34-6
 필드용 34-7
 CoClasses용 34-6
 dispinterface용 34-6
 타입 정보의 저장 및
 등록 34-24 ~ 34-26
 타입 정의 34-9 ~ 34-10
 텍스트 페이지 34-7,
 34-21
 툴바 34-3 ~ 34-5
 CoClasses 34-9, 34-16
 추가 34-22 ~ 34-23
 COM+ 페이지 39-5, 39-8
 dispinterfaces 34-9
 interface 34-8 ~ 34-9
 Type Library 에디터 33-16
 애플리케이션 서버 25-17

U

UCS 표준 10-23
 UDP 프로토콜 32-1
 UnaryOp 메소드 4-33
 Unassociate Attributes
 명령 19-14

- UndoLastChange 메소드 23-6
- UniDirectional 속성 18-49
- Unknown 제어 4-23, 4-25
- Unlock 메소드 9-7
- UnlockList 메소드 9-7
- UnregisterPooled 프로시저 25-8
- UnRegisterTypeLib 함수 33-18
- UPDATE 문 20-41, 20-44, 24-10
- Update SQL 에디터 20-42 ~ 20-43
 - Options 페이지 20-42
 - SQL 페이지 20-42
- UpdateBatch 메소드 10-29, 21-13, 21-14
- UpdateCalendar 메소드 51-3
- UpdateMode 속성 24-10
 - 클라이언트 데이터셋 23-22
- UpdateObject 메소드 38-13
- UpdateObject 속성 20-11, 20-33, 20-41, 20-45
- UpdatePropertyPage 메소드 38-13
- UpdateRecordTypes 속성 10-29, 20-33, 23-19
- UpdatesPending 속성 10-29, 20-33
- UpdateStatus 속성 10-29, 20-33, 21-12, 23-19, 24-9
- UpdateTarget 메소드 6-28
- URI
 - URL과 비교 27-4
- URL 27-3
 - URI와 비교 27-4
 - 웹 브라우저 27-5
 - 웹 연결 25-27
 - 자바스크립트
 - 라이브러리 25-36, 25-37
 - 호스트 이름 32-4
 - IP 주소 32-4
 - SOAP 연결 25-27
- URL 속성 25-27, 28-9, 31-9
- Use Unit 명령 5-19, 6-2
- USEPACKAGE 매크로 11-7
- uses 절 3-9, 10-6
 - 데이터 모듈 추가 5-18
 - 순환 참조 방지 6-2
 - 패키지 포함 11-3

UTF-8 문자 집합 10-18

V

- \$V 컴파일러 지시어 4-50
- Value 속성
 - 매개변수 18-46, 18-52, 18-53
 - 집계 23-14
 - 필드 19-17
- ValueChecked 속성 15-13
- Values 속성
 - 라디오 그룹 15-14
- ValueUnchecked 속성 15-13
- var 예약어
 - 이벤트 핸들러 43-3
- VCL 40-1 ~ 40-2
 - 개요 3-1 ~ 3-23
 - 객체 3-1
 - 메인 스레드 9-4
 - TComponent 분기 3-15
 - TControl 분기 3-16
 - TObject 분기 3-14
 - TPersistent 분기 3-14
 - TWinControl 분기 3-17
- VCL 애플리케이션
 - 포팅 10-2 ~ 10-16
- VCL40 패키지 11-1, 11-9
- PENWIN.DLL 11-11
- vcl60.bpl 13-6
- VendorLib 속성 22-4
- VertScrollBar 3-33
- VisualStyle 속성 3-40
- Visible 속성 3-2
 - 메뉴 6-41
 - 콜바 6-50
 - 툴바 6-50
 - 필드 19-12
- VisibleButtons 속성 15-29, 15-30
- VisibleChanged 속성 10-22
- VisibleColCount 속성 3-45
- VisibleRowCount 속성 3-45
- VisiBroker ORB 25-13
- VisualCLX 10-6
- VisualSpeller Control 13-5
- vtable 33-4
 - COM 인터페이스 포인터 33-4
 - 생성자 클래스 35-5, 35-13
 - 이중 인터페이스 36-13
 - 컴포넌트 랩퍼 35-6
- vtables
 - 대 dispinterfaces 34-9
 - 타입 라이브러리와 33-16

W

- W3C 30-2
- WaitFor 메소드 9-9, 9-10
- WantReturns 속성 3-32
- WantTabs 속성 3-32
 - data-aware 메모 컨트롤 15-9
 - data-aware rich edit 컨트롤 15-9
- .WAV 파일 8-33
- wchar_t widechar 10-23
- \$WEAKPACKAGEUNIT
 - 컴파일러 지시어 11-10
- Web Broker 27-1 ~ 27-2
- Web Broker 서버
 - 애플리케이션 28-1 ~ 28-21
 - 개요 28-1 ~ 28-4
 - 데이터 포스트 대상 28-10
 - 데이터베이스 액세스 28-17
 - 데이터베이스 연결
 - 관리 28-18
 - 생성 28-1 ~ 28-3
 - 아키텍처 28-3
 - 웹 디스패처 28-4
 - 응답 만들기 28-8
 - 응답 템플릿 28-14
 - 이벤트 처리 28-5, 28-7, 28-8
 - 테이블 쿼리 28-21
 - 템플릿 28-3
 - 파일 보내기 28-13
 - 프로젝트에 추가 28-3
- Web Deployment Options
 - 대화 상자 38-16
- Web Services Definition Language WSDL 참조
- WebContext 29-13
- WebDispatch 속성 25-39
- WebPageItems 속성 25-41
- WebSnap 27-1 ~ 27-2
- WebSnap 애플리케이션
 - 개요 29-1
- WebSnap 자습서 29-19
- WideChar 4-39, 4-41
- widechar 10-23
- WideString 4-41 ~ 4-42
- widestrings 10-23
- widget 3-17, 10-6, 10-22
- WidgetDestroyed 속성 10-22
- Width 속성 3-19, 3-44, 6-4
 - 데이터 그리드 15-20

- 데이터 그리드 열 15-16
- 펜 8-5, 8-6
- TScreen 13-13
- Win 3.1 페이지(컴포넌트 팔레트) 3-30
- WIN32 10-19
- Win32 페이지(컴포넌트 팔레트) 3-29
- WIN64 10-19
- Win-CGI 프로그램 27-5, 27-6, 27-7
 - 생성 28-2, 29-2
 - INI 파일 27-7
- window
 - 클래스 40-4
- Windows
 - 공통 대화 상자 52-1
 - 생성 52-2
 - 실행 52-4
 - 메시지 46-2
 - 이벤트 43-4
 - 장치 컨텍스트 40-7, 45-1
 - 컨트롤, 하위 분류 40-4
 - 펜 너비 지원 8-7
 - API 함수 45-1
 - Graphics Device Interface (GDI) 8-1
- Windows NT
 - 웹 서버 애플리케이션 디버깅 27-9
- Windows 메시징 10-17
- Windows 소켓 객체 32-5
 - 서버 소켓 32-7
 - 클라이언트 32-5
 - 클라이언트 소켓 32-6
- Windows 컨트롤 하위 분류 40-4
- wininet.dll 25-27
- WM_APP 상수 46-6
- WM_KEYDOWN 메시지 51-8
- WM_LBUTTONDOWN 메시지 51-8
- WM_MBUTTONDOWN 메시지 51-8
- WM_PAINT 메시지 8-2, 46-4
- WM_RBUTTONDOWN 메시지 51-8
- WM_SIZE 메시지 50-4
- WndProc 메소드 46-3, 46-4
- WordWrap 속성 3-32, 7-7, 48-1
 - data-aware 메모 컨트롤 15-9

- wParam 매개변수 46-2
- Wrap 속성 6-47
- Wrapable 속성 6-48
- Write 메소드
 - TFileStream 4-56
- write 메소드 42-6
- write 예약어 42-8
- WriteBuffer 메소드
 - TFileStream 4-57
- WSDL
 - 게시 31-7 ~ 31-8
 - 파일 31-8
 - import하기 31-8 ~ 31-9

X

- \$X 컴파일러 지시어 4-50
- XDR 파일 30-1
- Xerox Network System (XNS) 32-1
- .xfrm 파일 10-2, 3-7
- XML 26-1, 30-1
 - 데이터베이스 애플리케이션 26-1 ~ 26-11
 - 매핑 26-2 ~ 26-3
 - 정의 26-4
 - 문서 타입 선언 30-1
 - 처리 명령 30-1
 - 파서 30-2
 - SOAP 31-1
- XML 데이터 바인딩
 - 마법사 30-5 ~ 30-8
- XML 문서 26-1, 30-1 ~ 30-8
 - 노드 30-2, 30-4 ~ 30-5
 - 값 30-4
 - 변환 파일 26-1
 - 속성 26-5, 30-5, 30-6
 - 자식 30-5
 - 필드로 매핑 26-2
 - 데이터 패킷으로 변환 26-1 ~ 26-8
 - 데이터베이스 정보 게시 26-9
 - 루트 노드 30-3, 30-6, 30-8
 - 인터페이스 생성 30-6
 - 컴포넌트 30-3, 30-8
- XML 브로커 25-36, 25-38 ~ 25-40
 - HTTP 메시지 25-39
- XML 스키마 30-1
- XML 트리 29-1
- XML 파일 21-15

- XML Schema Data file
 - XSD file 참조
- XMLBroker 속성 25-42
- XMLDataFile 속성 24-2, 26-8
- XMLDataSetField 속성 25-42
- XMLMapper 26-2, 26-4 ~ 26-6
- XSD 파일 30-1
- XSL 트리 29-1

Y

- Year 속성 50-5

Z

- Z 컴파일러 지시어 11-12

