# E07 컴포넌트 제작을 위해 알아두어야 할 사항들

## 1.들어가기

컴포넌트란 델파이 프로그래밍에 있어서 기본이 되는 단위이다. 이러한 컴포넌트를 이용하여 개발자는 간단하게 고급스러운 프로그램을 만들어 낼 수 있게 된다. 컴포넌트 생성은 크게 세 가지 방식으로 나누어 생각할 수 있다.

- 내장 컴포넌트에 대해 기본적으로 상속을 받아 파생되는 컴포넌트를 만드는 방법. 이것은 가장 간단하게 컴포넌트를 생성하는 방법이다.
- 내장 컴포넌트의 상위 클래스에서 상속 받아 파생되는 컴포넌트를 만드는 방법. 이것은 기존의 내장 컴포넌트와 같은 레벨의 차원에서 새로운 속성의 추가가 가능하다.
- 기존의 내장 컴포넌트로부터 영향을 받지 않고 처음부터 컴포넌트를 새로 만드는 방법. 이것이 가장 어려운 방법이다.

이와 같은 방법을 사용하여 컴포넌트를 생성하기 위해서는 컴포넌트의 구조를 안다면 도움이 된다. 즉, 어느 클래스 혹은 객체로부터 상속을 받을 것인지를 결정하는 것이 컴포넌트 생성의 시작이므로 가장 중요하다고 할 수 있는 것이다.

## 2.델파이 객체에 대한 이해

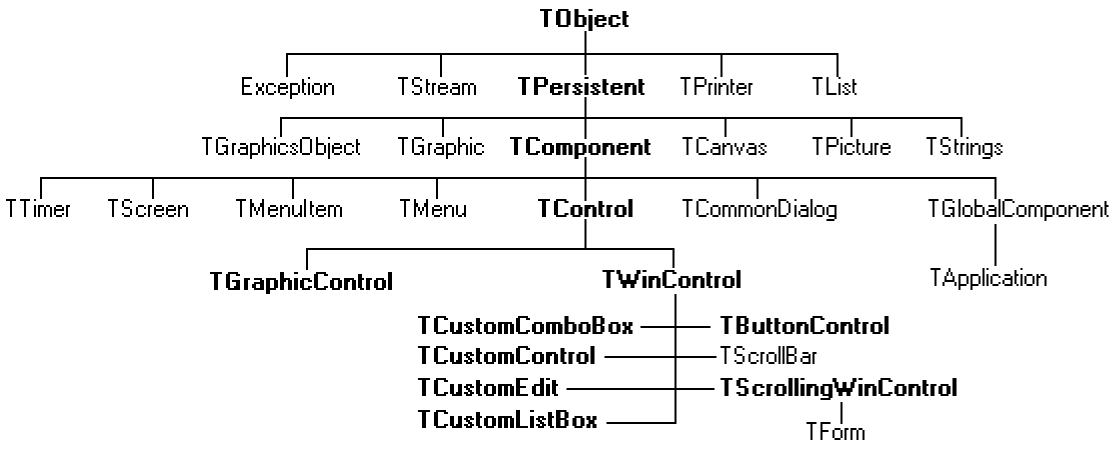
델파이에서 사용하고 있는 VCL 이라는 말은 Visual Component Library 의 약자로 비주얼 컴포넌트의 집합을 뜻하며, 컴포넌트는 객체의 계층적 구조 [그림 1]에서 살펴보면 TComponent 에서부터 그 하위 계층의 객체를 의미한다. 그런데 현재 델파이에서 사용되고 있는 VCL 들을 살펴보면 VCL 의 의미와는 상충되는 비주얼 하지 않은 컴포넌트, 더욱이 컴포넌트가 아닌 객체(Object)도 많이 볼 수 있다. TStrings 나 TCanvas 와 같은 것이 그 예가 될 것이다.

이러한 객체의 계층적 구조는 생물학에서 말하는 동식물의 분류와 그 개념이 비슷하다. 이들의 구조는 결국 하나의 종으로부터 진화하고 발전하면서 다양한 종으로 분류되었다는 것과 이들이 그 조상에게서 물려받은 특성과 자신만의 새로운 특성을 가지고 있다는 것 등이 객체의 계층적 구조와 흡사하다고 할 수 있다.

이제 델파이의 객체에 대한 전반적인 이해와 VCL 에 대한 몇 가지 사실들을 알게 되었으므로 여기서는 각 객체의 계층적인 구조와 특징에 대해서 알아보도록 한다.

[그림 1]은 델파이에 정의되어 있는 객체의 계층적 구조를 보여 주고 있다. 그림에서 볼 수

있듯이 델파이 객체의 가장 상위 계층에는 TObject 가 있고 그 아래로 TObject 로부터 상속 받은 객체들이 트리 구조를 가지면서 구성되어 있다.



[그림 1] 객체의 계층적 구조

이제 각각의 객체에 대해서 좀 더 자세히 알아보도록 하자.

## 2.1 TObject

TObject 는 다른 모든 객체들과 컴포넌트의 최상위 계층의 조상이다. [그림 1]에서 볼 수 있듯이 TObject 에게는 여러 개의 하위 객체가 존재하는데, 이 객체들은 대부분 보여지지 않는 곳에서 제 역할을 하고 있기 때문에 아마도 그리 익숙하지 않은 객체들일 것이다.

이러한 TObject 는 모든 델파이 객체에 대해 공통적인 다음과 같은 기능을 하는 메소드를 위한 인터페이스를 제공한다.

- 객체가 필요로 하는 메모리를 할당, 초기화, 해제하여 객체를 생성, 관리, 제거하는 능력.
- 객체의 클래스 형과 인스턴스 정보, 그리고 실행시간 형 정보(run-time type information).
- 메시지 핸들링 능력.

TObject 의 선언부를 보도록 하자.

```
TObject = class
  constructor Create;
  procedure Free;
  class function InitInstance(Instance: Pointer): TObject;
  procedure CleanupInstance;
  function ClassType: TClass;
  class function ClassName: ShortString;
  class function ClassNameIs(const Name: string): Boolean;
  class function ClassParent: TClass;
  class function ClassInfo: Pointer;
  class function InstanceSize: Longint;
  class function InheritsFrom(AClass: TClass): Boolean;
  class function MethodAddress(const Name: ShortString): Pointer;
  class function MethodName(Address: Pointer): ShortString;
  function FieldAddress(const Name: ShortString): Pointer;
  function GetInterface(const IID: TGUID; out Obj): Boolean;
  class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
  class function GetInterfaceTable: PInterfaceTable;
  function SafeCallException(ExceptObject: TObject;
           ExceptAddr: Pointer): HResult; virtual;
  procedure AfterConstruction; virtual;
  procedure BeforeDestruction; virtual;
  procedure Dispatch(var Message); virtual;
  procedure DefaultHandler(var Message); virtual;
  class function NewInstance: TObject; virtual;
  procedure FreeInstance; virtual;
  destructor Destroy; virtual;
end;
```

위의 선언부에서 알 수 있듯이 객체의 사용에 있어서 가장 중요하다고 할 수 있는 객체의 생성과 소멸을 위해 생성자(Constructor)와 소멸자(Destructor)를 선언하고 있다. 그 외에 클래스 정보나, 객체에 의해 구현되는 인터페이스를 위한 함수 등이 포함되어 있다. private 혹은 public 등으로 지정하지 않는 경우에는 public 으로 설정되는데, 이것은 TObject 의 모든 하위 객체는 TObject 의 모든 함수나 프로시저에 접근할 수 있다는 것을 뜻한다. 예를 들면, 어떤 객체가 생성자나 소멸자를 가지고 있지 않더라도 가장 상위 객체인 TObject 에 생성자와 소멸자가 선언되어 있으므로 하위 객체의 생성과 소멸이 가능하게 된다. 또 한가지 특기할 점은 class 키워드로 선언된 클래스 메소드가 많이 있다는 것이다. 이와 같은 클래스 메소드는 클래스에 직접적으로 적용할 수 있기 때문에 다음과 같은 문장이 성립된다.

TButton.ClassParent;

TObject 는 컴포넌트가 아니고, 스트림 혹은 할당 능력이 필요하지 않는 경우에 기본 클래스로 사용한다. 새로운 객체 형을 정의하는 경우 조상 형을 설정하지 않으면 델파이는 자동적으로 TObject 를 조상으로 사용한다.

델파이 객체가 가지는 강력한 능력들은 TObject 가 가지는 메소드들로부터 만들어지기 때문에 이러한 메소드들의 대부분은 델파이 환경에서 내부적으로 사용되고, 사용자가 직접 호출하여 사용하지 않도록 되어 있다. 나머지 다른 메소드들은 하위 객체나 컴포넌트에서 좀 더 다양한 기능을 수행하도록 하기 위해 오버라이드 되어 사용된다.

## 2.2 TPersistent

TPersistent 는 할당과 스트림 능력을 갖는 모든 객체의 조상이다. 즉, 객체를 저장하거나 읽어오기 위한 기본 클래스이다.

모든 컴포넌트는 DFM 파일에 그 속성이 저장되며, 프로젝트에서 폼 파일을 불러들일 때 폼 파일에 기록되어 있는 속성 값을 이용해 컴포넌트가 이전에 작업된 상태로 나타나도록 한다. 이러한 작업을 하는 가장 기본적인 객체가 바로 TPersistent 인 것이다. 만약에 TPersistent 객체가 없다면 폼 파일은 아무 의미가 없다.

TPersistent 는 다른 객체에 할당될 수 있고, 스트림을 이용하여 속성을 읽고 쓰는 것이 가능한 모든 객체의 공통적인 행동을 캡슐화 한다. 이러한 목적으로 TPersistent 는 다음과 같은 메소드를 정의한다.

- 스트림에 공개되지 않은 데이터를 로딩, 저장하는 프로시저 정의.
- 속성에 값을 할당.
- 한 객체의 내용을 다른 객체로 할당.

TPersistent 의 선언부는 다음과 같다.

```
TPersistent = class(TObject)
  private
    procedure  AssignError(Source:  TPersistent);
  protected
```

```
    procedure AssignTo(Dest: TPersistent); virtual;
    procedure DefineProperties(Filer: TFiler); virtual;
    function  GetOwner: TPersistent; dynamic;
  public
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); virtual;
    function  GetNamePath: string; dynamic;
  end;
```

여기에 정의되어 있는 메소드 중에서 가장 중요한 메소드는 DefineProperties 이다. 이 메소드는 사실 아무 내용도 들어있지 않은 빈 껍데기에 불과하다. 그렇지만 이 메소드가 중요한 이유는 하위 객체들이 속성을 저장하거나 읽어오는데 사용하는 메소드이기 때문이다. 기본적으로는 폼 파일과 같은 스트림에 객체를 쓰면 공개되어 있는 속성의 값만이 쓰여지게 되고 이렇게 되면 스트림에서 객체를 읽어오더라도 그러한 값들만을 가져오게 된다. 그러므로 공개되어 있지 않은 속성 값도 함께 저장하고 읽어올 수 있도록 하위 객체에서는 이 DefineProperty 메소드를 오버라이드하여 사용하는 것이다.

TPersistent 의 인스턴스는 직접 만들어 사용하지는 않는다. TPersistent 는 컴포넌트는 아니지만 스트림 혹은 할당 능력을 필요로 하는 객체를 정의하는 경우에 기본 클래스로 사용한다.

## 2.3 TComponent

TComponent 는 델파이의 모든 컴포넌트 객체의 조상이다. TComponent 는 다음과 같은 기능을 제공하는 메소드와 속성을 정의하여 델파이의 모든 컴포넌트에 대한 공통적인 행동을 캡슐화 한다.

- 컴포넌트 팔레트에 나타나고, 폼 디자이너에서 다룰 수 있는 능력.
- 향상된 스트림과 파일링 능력.

TComponent 는 컴포넌트를 생성하기 위해서는 가장 기본이 되는 객체이다. 따라서 이 객체에는 컴포넌트의 정보에 관련된 속성들이 대부분이다. 이제 TComponent 가 갖는 메소드 중에서 컴포넌트의 생성에 유용한 몇 가지를 살펴보도록 하자.

먼저 다음의 코드를 보자.

```
Var
  Button1 : TButton;
begin
   Button1 := TButton.Create(Self);
  // 필요한 작업을 한다
   Button1.Free;
end;
```

이 코드는 우리가 일반적인 애플리케이션을 작성하는 경우에도 자주 보게 되는 코드이다. 이 코드는 새로운 버튼을 하나 생성하고 필요한 작업을 한 후 소멸시키는 아주 간단한 코드이다. 여기서 버튼을 생성하는 Create 메소드가 선언되어 있는 객체가 바로 TComponent 이고 여기서는 TButton 에서 이 메소드를 상속 받아 사용하고 있는 것이다.

또한 컴포넌트를 작성하는데 있어서 유용하게 사용할 수 있는 속성인 ComponentState 가 이 객체에 선언되어 있는데, 이 속성은 컴포넌트의 상태를 가지고 있게 된다. 보통의 경우 메소드는 실행 시에 사용하게 되는데, DBGrid 처럼 실행시간이 아닌 설계 시에도 메소드를 이용한 작업이 필요한 경우 이 속성을 이용하면 손쉽게 구현할 수 있다. 이 속성이 가질 수 있는 값들에는 다음과 같은 것들이 있다.

```
type TComponentState = set of (csLoading, csReading, csWriting, csDestroying, csDesigning,
csAncestor, csUpdating, csFixups);
property ComponentState: TComponentState;
```

이 값들 중에서 위에 설명한 기능을 갖게 하기 위해 가장 유용한 값이 csDesigning 이다. 설계 시에 어떤 작업을 수행하려면 컴포넌트의 상태를 점검하여 csDesigning 이면 작업을 수행하도록 하면 되는 것이다. 다음의 예를 보자.

```
procedure TToolButton.Paint;
const
  XorColor = $00FFD8CE;
var
  R: TRect;
begin
  if FToolBar = nil then Exit;
  if Style = tbsDivider then
     with Canvas do
```

```
      begin
       R := Rect(Width div 2 - 1, 0, Width, Height);
         DrawEdge(Handle, R, EDGE_ETCHED, BF_LEFT)
      end;
  if csDesigning in ComponentState then
    { Draw separator outline }
    if Style in [tbsSeparator, tbsDivider] then
       with Canvas do
        begin
          Pen.Style := psDot;
          Pen.Mode := pmXor;
          Pen.Color := XorColor;
          Brush.Style := bsClear;
         Rectangle(0, 0, ClientWidth, ClientHeight);
        end
    { Draw Flat button face }
    else if FToolBar.Flat and not Down then
       with Canvas do
        begin
         R := Rect(0, 0, Width, Height);
         DrawEdge(Handle, R, BDR_RAISEDINNER, BF_RECT);
        end;
end;
```

위의 예는 TToolButton을 그리는 코드로 여기에 위에 설명한 ComponentState와 csDesigning을 사용한 것을 볼 수 있다. 이 예제는 여러분도 사용하다 보면 알 수 있겠지만 버튼과 버튼 사이의 분리자가 실행 시에는 그 형태가 나오지 않지만 설계 시에는 나타나는 것을 알 수 있을 것이다. 위의 코드가 바로 그런 기능을 정의하고 있다. 이 외에도 대부분의 컴포넌트들이 이 속성을 사용하고 있다.

다시 TComponent객체에 대한 얘기로 돌아가 보자. 대체로 TComponent의 인스턴스는 만들지 않는다. TComponent는 그 자체보다는 이 객체를 이용하여 다른 객체를 생성해 내는데 사용될 뿐이다. TComponent는 컴포넌트 팔레트에는 보여지고, 폼 디자이너에서 사용되지만 실행 시 보여지지는 않는 컴포넌트를 정의할 때 기본 클래스로 사용한다. TComponent의 속성과 메소드는 하위 클래스들이 자신의 행동을 자신에게 맞도록 설정할 수 있는 기본적인 환경을 제공한다.

비주얼 한 컴포넌트를 생성하기 위해서는 TControl이나 그 하위 클래스를 이용한다. 또한 비주얼 한 윈도우 컨트롤을 생성하기 위해서는 TWinControl 혹은 그 하위 클래스를 사용한다.

## 2.4 TControl

TControl은 비주얼 한 모든 컴포넌트의 추상적인 기본 클래스이다. 즉, TControl에는 컴포넌트들이 갖는 속성 중에 보여지는 데 필요한 속성 즉, Caption, Visible, Top, Width, Left, Height, Enabled, Font, Color 등과 같은 속성들을 가지게 되고, 그와 관련된 메소드를 가지고 있다. 또한 컴포넌트가 비주얼 하다는 의미는 그것들을 실행 시에 조작할 수 있음을 의미한다. 이에 따라 이벤트가 처음으로 등장하게 되는데, 대부분이 마우스에 관련된 이벤트이다. OnClick, OnDblClick, OnMouseMove, OnMouseDown, OnMouseUp, OnDragDrop, OnDragOver, OnEndDrag, OnStartDock, OnEndDock, OnCanResize, OnConstrainedResize, OnResize등이 그것이다.

TControl의 선언 중에 한가지 유의해서 보아야 할 점은 이벤트와 속성 중 일부가 protected에 선언되어 있다는 점이다. 이 속성이나 이벤트들은 Object Inspector에도 나타나지 않는다. 많은 컴포넌트가 TControl처럼 중요한 속성이나 이벤트 혹은 메소드를 protected에 선언하는 것을 볼 수 있을 것이다. 이렇게 일부 속성, 이벤트, 메소드를 public이나 published에 선언하지 않고, protected에 선언하는 것은 이들의 사용 범위가 명확하지 않다는 점 때문이다. 좀 더 자세히 설명하자면 그 이유는 다음과 같다. 어떤 컴포넌트의 하위 객체도 그 상위 객체의 속성, 이벤트 그리고 메소드를 재선언하여 사용할 수 있다. 즉 상위 객체에서 protected에 선언했다 하더라도 그 하위 객체가 이것을 public이나 published에 선언하여 사용할 수 있는데, 문제는 상위 객체에서 published에 선언한 것은 하위 객체에서 protected로 선언할 수 없다는 데 있다. 즉 그 가시 범위를 좁혀서 재선언할 수는 없다는 것이다. 따라서 상위 객체에서 어떤 속성을 published로 선언하게 되면, 이 컴포넌트의 모든 하위 객체는 이 속성이 필요하든 혹은 필요하지 않든 간에 published로 사용해야 한다는 점이다. 이런 이유 때문에 하위 객체에서 사용하지 않을 수도 있는 속성이나 이벤트 또는 메소드를 protected에 선언하는 것이다.

## 2.5. TGraphicControl

TGraphicControl은 비윈도우 컴포넌트의 기본 클래스 중 하나이며, 윈도우 핸들을 가지고 있지 않기 때문에 입력 포커스를 받을 수 없고, 다른 컨트롤을 포함할 수도 없다. 그렇지만 이 TGraphicControl은 그래픽에 관한 작업을 할 수 있어 그 하위 객체로 TBevel, TImage, TPaintBox, TShape, TSpeedButton, TSplitter 그리고 TCustomLabel 등을 가지고 있다. 이 객체는 TControl로부터 상속 받은 모든 속성에 캔버스 속성을 추가한다. 캔버스 속성을 이용하면 선 그리기,사각형 그리기, 색칠하기, 텍스트 출력 , 이미지 복사 등과 같은 그래픽에 관련한 작업을 할 수 있다. 그래서 TGraphicControl은 객체가 입력 포커스를 받거나 다른 컴포넌트를 포함할 필요가 없고 그래픽과 관련한 기능을 가져야 할 때 사용한다.

## 2.6 TWinControl

TWinControl 은 모든 윈도우 컴포넌트의 기본 클래스이다. 여기서 말하는 윈도우 컴포넌트의 특징은 다음과 같다.

- 애플리케이션의 실행 중에 포커스를 받을 수 있다. 사용자가 키보드를 이용하여 컴포넌트에 작업을 할 수 있는 컴포넌트는 오직 윈도우 컴포넌트 뿐이다.
- 다른 컴포넌트를 포함할 수 있다. 어떤 컴포넌트 A 를 포함한 컴포넌트 B 를 컴포넌트 A 의 parent 라고 하고 컴포넌트 A 는 컴포넌트 B 의 child 라고 하는데, 오직 윈도우 컴포넌트만이 하나 이상의 child 컴포넌트를 포함할 수 있다.
- 윈도우 핸들을 갖는다.

대부분 새롭게 생성되는 컴포넌트는 TWinControl 보다는 좀 더 특별화 된 컴포넌트를 이용하는데, TCustomControl, TButtonControl, TCustomComboBox, TCustomEdit 또는 TCustomListBox 등이 그것이다.

## 2.7 Custom 컴포넌트

델파이에는 Custom 으로 시작되는 이름을 갖는 객체들이 많다. 이들 객체들의 특징은 이들이 가지고 있는 속성이나 메소드들이 대부분 protected 로 선언되어 있다는 것이다. 이 객체들이 따로 만들어지게 된 이유는 앞에서 설명했던 것과 같이 상위 객체에서 상속을 받아 생성되는 하위 객체에서는 상위 객체의 속성이나 메소드를 가시 범위가 더 좁아지도록 재선언할 수 없기 때문이다. 이렇게 중요한 속성이나 메소드들을 protected 로 선언함으로써 그 하위 객체들이 필요한 속성이나 메소드만을 published 로 선언할 수 있도록 하는 것이다. 일례로 TCustomLabel 에서 상속을 받아 생성된 TLabel 의 경우에는 TCustomLabel 의 모든 속성을 published 로 재선언한 것을 알 수 있다.

## 3.표준 컴포넌트를 이용한 컴포넌트 만들기

그럼 먼저 델파이의 기본 표준 컴포넌트에서 상속을 받아 간단한 컴포넌트를 만들어 보자. 여기서는 TLabel 을 상속 받아 컴포넌트를 생성할 것이고, 이 컴포넌트는 기본값을 새롭게 정의하여 새로운 모양을 가진 컴포넌트로 정의될 것이다.

다음에 보여지는 [코드 1]은 컴포넌트를 생성하기 위한 가장 기본적인 골격이다. 이 코드는 TButton 으로부터 상속을 받아 컴포넌트를 만들기 위한 것으로 TButton 클래스로부터 상속 받은 TButton1을 생성해 낸다.

```
Unit Button1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TButton1 = class(TButton)
   private
     { Private declarations }
   protected
     { Protected declarations }
   public
     { Public declarations }
   published
     { Published declarations }
   end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TButton1]);
```

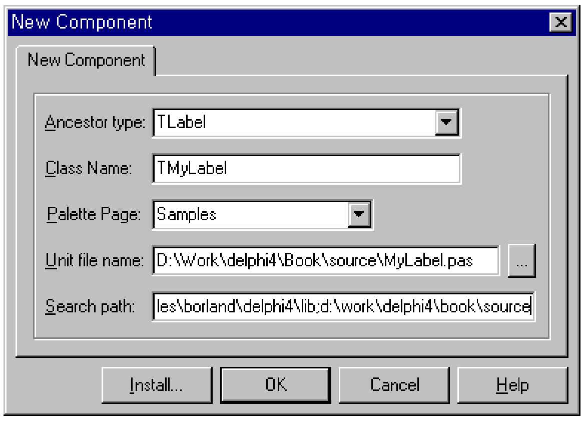
```
end;

end.
```

[코드 1] 컴포넌트 생성을 위한 기본 코드 골격

컴포넌트를 만드는 데 있어서 위와 같은 기본적인 골격을 생성하기 위한 가장 손쉬운 방법은 위저드를 이용하는 방법이다. 이 위저드를 이용하게 되면 델파이가 자동으로 기본적인 코드를 생성해 주기 때문에 작업을 손쉽게 할 수 있다. 이 위저드는 Component|New Component 메뉴를 선택하면 된다. 여기서 우리가 만들어 낼 컴포넌트의 상위 클래스를 정의하고 새로운 컴포넌트의 이름만 설정하면 된다. 물론 위저드를 이용하지 않고 직접 [코드 1]과 같은 모양을 갖도록 코딩을 해도 된다.

이제 TLabel 에서 상속 받은 새로운 컴포넌트를 만들어 보자.



Ancestor type: TLabel
Class Name: TMyLabel
Palette Page: Samples
Unit file name: D:\Work\delphi4\Book\source\MyLabel.pas
Search path: les\borland\delphi4\lib;d:\work\delphi4\book\source

[그림 2] New Component 화면

먼저 델파이 메뉴의 Component|New Component 를 선택한다. 새로운 컴포넌트를 만들기 위한 대화 상자가 나타날 것이다[그림 2]. 여기에서 우리는 TLabel 로부터 상속 받은 컴포넌트를 만들 것이기 때문에 Ancestor type 에는 TLabel 을 선택한다. 이제 우리가 만들 새로운 클래스의 이름을 정의하면 되는데 기본적으로 TLabel1 이라고 되어 있는 것을 TMyLabel 이라고 설정하고 유닛을 둘 폴더도 함께 설정하도록 하면 Search path 에 그 폴더가 포함되어 있는 것을 확인할 수 있다. 또한 팔레트 페이지는 여러분이 만든 컴포넌트를 가질 페이지를 설정하게 된다. 적절한 페이지를 선택하고 OK 버튼을 눌러보자. 다음과 같은 기본 골격이 생성될 것이다[코드 2].

```
Unit MyLabel;
```

```
Interface

Uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
   StdCtrls;

Type
  TMyLabel = class(TLabel)
   Private
     { Private declarations }
   protected
     { Protected declarations }
   public
     { Public declarations }
   published
     { Published declarations }
   end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyLabel]);
End;

End.
```

[코드 2] TLabel 에서 상속 받은 컴포넌트 코드의 골격

델파이 컴포넌트 라이브러리에 이 유닛이 컴파일 될 때 어떤 컴포넌트 팔레트에 등록이 될 것인가는 다음과 같이 RegisterComponents 라는 메소드를 이용한다. 이 메소드에 대해서는 뒤에 좀 더 자세하게 알아보도록 하자.

```
procedure Register;
```

```
begin
  RegisterComponents('Samples', [TMyLabel]);
end;
```

이 프로시저는 이 유닛의 컴파일에 영향을 미치지 않으며, 이 프로시저를 실행 시에 호출하지 않는 한 결코 나타나지 않는다. 물론 실행 시에는 전혀 사용할 필요가 없다.

유닛이 생성되었으므로 먼저 이 유닛을 저장하도록 하자. File|Save 를 선택하여 저장하는데, 유닛의 이름은 이미 정의해 주었기 때문에 주어진 이름으로 저장될 것이다. 우리가 생성할 이 컴포넌트는 폰트의 속성을 변경하여 새로운 기본값을 갖도록 할 것이다. 그러기 위해서는 Create 메소드를 오버라이드하여 폰트의 속성을 설정해야 한다. 메소드를 선언하기 위해 클래스 선언부에 다음과 같은 코드를 작성한다.

```
Type
  TMyLabel = class(TLabel)
   Public
     constructor Create(AOwner: Tcomponent): override;
   end;
```

그러면 이제 Create 메소드를 실제로 구현해 보자.

```
constructor TMyLabel.Create(AOwner: TComponent);
begin
  inherited  Create(AOwner);
  Color := clBlue;
  Font.Color := clYellow;
  Font.Name := '굴림체';
  Font.Size := 12;
  Font.Style := [fsBold, fsItalic];
end;
```

이때 한 가지 주의해야 할 점은 코드의 순서이다. 위의 생성자를 구현하는 것처럼 메소드를 오버라이드하여 사용하는 경우에는 상속을 받을 위치를 확실하게 정의해야 한다. 대개 이런 경우에는 먼저 상속을 받은 이후에 다른 작업을 하게 된다. 반대로 소멸자와 같은 경우는 모든 작업을 끝내고 마지막에 상속을 받는다.

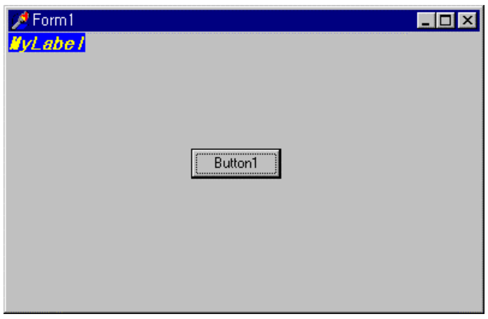
여기가지 됐다면 유닛을 저장하자. 그리고 이 유닛을 컴파일을 하여 소스 코드에 에러가 있는 지를 먼저 확인한다. 에러가 없다면 이제 컴포넌트를 팔레트에 등록하기만 하면 된다. 그러나 컴포넌트를 등록하기 전에 먼저 소스 코드에 구문 오류가 있는지 확인하고 이 컴포넌트가 원하는 동작을 하는지 확인하는 것이 순서일 것이다.

그렇다면 지금부터는 간단한 프로그램을 통해 위에서 만든 TMyLabel 이 정확히 동작하는지 확인하도록 하자.

먼저 File|New Application 을 선택하여 새로운 프로그램을 준비하고 여기에 TButton 을 하나 갖다 놓는다. 그리고 이 버튼의 OnClick 핸들러를 다음과 같이 생성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var MyLabelTest : TMyLabel;
begin
  MyLabelTest := TMyLabel.Create(Self);
  MyLabelTest.Parent := Self;
  MyLabelTest.Caption := 'MyLabel';
   MyLabelTest.Show;
end;
```

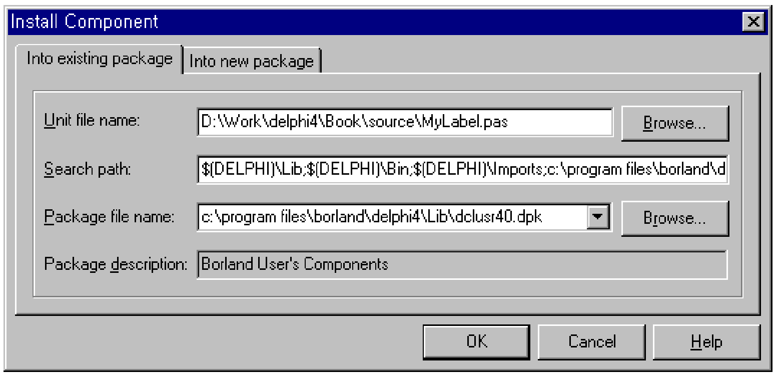
그리고 유닛의 인터페이스 부분에 있는 uses 절에 우리가 만든 컴포넌트 소스 파일의 이름을 추가해야 한다. 우리가 정의한 컴포넌트 클래스가 이 파일에 정의되어 있기 때문이다. 만약 추가하지 않으면 에러를 발생할 것이다. 자 모든 것이 끝났다면 이제 테스트를 해보자. 테스트 프로그램을 컴파일하고 실행하면 버튼이 하나 있는 폼이 생성된다. 폼의 버튼을 눌러보면 [그림 3]과 같은 결과가 나올 것이다. 만약 그림과 같은 결과가 나오지 않았다면 소스 코드를 다시 한 번 확인하도록 한다.



[그림 3] 테스트 결과 화면

테스트가 성공적으로 끝났다면 이제 남은 일은 이 컴포넌트를 컴포넌트 팔레트에 등록하는 일이다. 컴포넌트 팔레트에 등록하려면 델파이 메뉴에서 Component|Install Component ..를 선택한다. 그러면 Install component 창이 나타난다[그림 4].

우리가 작성한 컴포넌트 소스 파일을 Unit file name에 설정한다. 그리고 Package file name에는 어떤 패키지에 이 컴포넌트가 포함될 지를 설정한다. 델파이에서는 기본적으로 dclusr60.dpk 라는 사용자 정의 패키지 파일을 이용하도록 한다. 모든 설정이 끝났다면 OK 버튼을 눌러보자. 그러면 델파이는 Package file name에 정의되어 있는 패키지를 다시 컴파일 할 것인지 물어오게 된다. Yes 를 선택하면 패키지를 컴파일 하게 되고 [그림 5]와 같은 윈도우가 보여지면 성공적으로 컴포넌트가 패키지에 포함된다. 그러면 여러분은 미리 정의해 준 컴포넌트 팔레트인 'Samples'에 컴포넌트가 생성된 것을 확인할 수 있을 것이다.


Install Component

Into existing package | Into new package

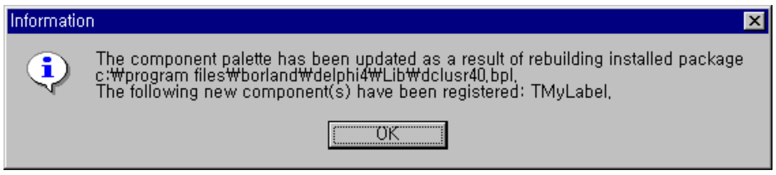
Unit file name: D:\Work\delphi4\Book\source\MyLabel.pas

Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Imports;c:\program files\borland\d

Package file name: c:\program files\borland\delphi4\Lib\dclusr40.dpk

Package description: Borland User's Components

[그림 4] Install Component 화면


Information

The component palette has been updated as a result of rebuilding installed package c:₩program files₩borland₩delphi4₩Lib₩dclusr40.bpl.
The following new component(s) have been registered: TMyLabel.

[그림 5] 컴포넌트 인스톨이 완료된 화면

이제는 실제로 우리가 만든 컴포넌트를 이용하여 프로그램을 작성해 보자. 먼저 File|New Application 을 선택한다. 여기에 TEdit 컴포넌트 하나와 TButton 하나 그리고 우리가 만들어 방금 등록시켜 놓은 TMyLabel 을 가져 다 놓도록 하자. 여기서 TButton 의 OnClick 이벤트에 다음과 같이 코딩 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MyLabel1.Caption := Edit1.Text;
end;
```

자 위와 같이 코딩을 했다면 실행을 해보자. Edit 컴포넌트에 간단한 문자열을 입력하고 버튼을 눌러보면 우리가 생성한 컴포넌트에 그 단어가 나타나는 것을 확인할 수 있을 것이다.

지금까지 우리는 TLabel 컴포넌트에 간단한 속성을 추가해 봄으로써 컴포넌트를 생성하는 기본적인 방법을 알아 보았다. 아마도 컴포넌트를 생성하는 가장 기본적인 이유는 자신만의 속성을 주고자 하는 경우가 많으므로 여기까지만 알고 있어도 간단한 자신만의 컴포넌트를 만들 수 있을 것이다.

## 4. 컴포넌트의 구성 요소

만약 좀 더 다양한 기능을 갖는 컴포넌트를 만들고자 하는 경우에는 어떻게 할까? 예를 들어 이미지 컴포넌트에 그림을 여러 개를 가지고 있도록 하고, 마우스가 이미지 컴포넌트에 들어가게 되면 다른 그림이 나오도록 하고 싶다면 새로운 속성과 이벤트, 그리고 메소드가 필요하게 된다. 이와 같이 컴포넌트를 만드는 데는 속성, 이벤트, 메소드 등을 새롭게 추가하거나 기존의 기능에 부가적인 기능을 더 정의하여 구현한다. 그러므로 컴포넌트를 만들기 위해서는 속성, 이벤트, 메소드 등 다양한 구성 요소들에 대해 먼저 이해를 하도록 해야 한다.

이제 이 구성 요소들에 대해 하나씩 알아보도록 하자. 먼저 컴포넌트의 가장 기본적인 구성 요소라고 할 수 있는 속성에 대해서 알아보자.

### 4.1. 속성

속성은 왜 생성할까? 아마도 이런 생각을 가져 본 독자는 그리 많지 않을 것이다. 먼저 속성을 생성하는 이유를 생각해 보자. 애플리케이션을 만드는 개발자의 입장에서 보면, 속성은 변수와 비슷한 의미를 가진다. 개발자는 마치 변수처럼 속성의 값을 읽고 쓸 수 있는 것이다. 다시 말하면 속성은 컴포넌트를 사용하기 위한 값들을 저장하고 있는 변수라고 할 수 있다. 다만 변수로 할 수 있는 작업 중에 속성으로 할 수 없는 작업은 오직 var 파라미터로 사용하는 것 뿐이다. 컴포넌트의 개발자에게 있어서는 이 속성은 애플리케이션 개발자와 상호 작용을 하기 위한 가장 기본적인 단위가 되는 것이다. 그러므로 속성을 잘 선택하여 생성해야 하고, 이 속성들을 최대한 활용할 수 있도록 설계하는 것이 좋다.

#### 1) 속성의 특성

속성은 다음과 같은 특성을 갖는다.

- 메소드가 실행 시에만 사용 가능한 것에 반하여 속성은 디자인 시에도 사용이 가능하여 애플리케이션이 실행되기 전에 컴포넌트를 조작할 수 있다.
- 속성은 그 구현 내용을 숨길 수 있다.
- 속성은 virtual 이 될 수 있다. 즉, 프로그램 내에서는 선언되어 있는 한 속성이 다른 컴포넌트에서는 다르게 구현될 수 있는 것이다.

#### 2) 속성의 형

속성에는 어떤 형이 있을까? 속성은 어떤 형이든지 가질 수 있다. 각각의 형은 오브젝트 인

스펙터 상에서도 서로 다르게 표현된다[표 1]. 그렇다면 컴포넌트의 모든 속성이 오브젝트 인스펙터 상에 나타날까? 그렇지는 않다. 오브젝트 인스펙터에 나타나는 속성은 published 로 선언된 경우이다. 즉, 속성을 선언하는 경우에 오브젝트 인스펙터 상에 나타낼 속성은 published 에 재선언하는 것이다. 예를 들면, TWinControl 로부터 상속 받은 컴포넌트에는 Ctl3D 라는 속성이 protected 로 선언되어 있다. 이속성을 재선언하여 여러분은 public 이나 published 로 선언할 수 있다.

다음의 코드는 Ctl3D 속성을 published 로 재선언하여 오브젝트 인스펙터 상에 나타나도록 한다.

```
Type
  TMyComponent = class(TWinControl)
   Published
     Property Ctl3D;
   End;
```

| 속성의 형 | 오브젝트 인스펙터에서의 처리 |
|---|---|
| 단순형 | Numeric, character 그리고 string 속성은 숫자, 문자 그리고 문자열로 나타난다. 개발자는 속성의 값을 직접 편집할 수 있다. |
| 열거형 | 열거형의 속성(부울형을 포함)은 편집이 가능한 문자열로 나타난다. 개발자는 속성 컬럼을 더블 클릭하여 다음 값으로 변경할 수 있으며, 드롭 다운 리스트를 이용하여 가능한 모든 값들 중에 선택할 수 있다. |
| 집합형 | 집합형의 속성은 더블 클릭하여 각각의 구성 요소를 설정할 수 있다. 이 구성 요소는 부울형의 값을 갖게 되고 이 집합형에 포함되는 값은 true 로 설정한다. |
| 객체형 | 객체형의 속성은 대개의 경우 속성 에디터를 가진다. 이 객체형의 속성 중에 published 로 선언된 속성들은 객체형 속성을 더블 클릭함으로써 오브젝트 인스펙터 상에 나타나게 되고 수정이 가능하게 된다. 객체형 속성은 TPersistent 의 하위 객체이어야 한다. |
| 배열형 | 배열형 속성은 자신만의 속성 에디터를 가져야 한다. 오브젝트 인스펙터에서는 이 속성값을 수정할 수 없기 때문이다. 속성 에디터는 컴포넌트를 등록하는 과정에 정의해 준다. |

[표 1] 속성 형에 따른 오브젝트 인스펙터 상의 표시법.

여기서 주의해야 할 점은 속성을 재선언하는 경우에는 위의 예와 같이 속성의 이름만 사용한다는 것이다. 속성의 형이나 다른 정보를 전혀 포함하지 않는다.

### 3) 속성의 정의 방법

지금까지 속성에 대한 기본적인 사항을 모두 알아보았다. 이제는 속성을 정의하는 방법에 대해서 알아보도록 하자. 먼저 속성의 정의는 어떻게 할까? 속성을 정의하기 위해서는 속성의 이름, 형, 그리고 속성의 값을 읽고 쓰기 위한 read, write 부분의 선언이 필요하다. 만약 write 부분이 정의되지 않으면 속성은 읽기 전용으로 선언된다.

published 에 선언된 속성은 오브젝트 인스펙터 상에서도 편집이 가능하도록 선언되고, public 으로 선언된 경우에는 실행 시에 코드상으로 읽고 쓰는 것이 가능하도록 선언된다.

다음 예제는 count 라는 속성을 선언한 것이다.

```
Type
  TMyComponent = class (TComponent)
   Private
     Fcount: Integer;   //내부 저장용으로 사용
     Procedure SetCount(Value: Integer);   //쓰기 메소드
   Public
    Property Count: Integer read Fcount write SetCount;
   End;
```

위의 코드에서 보면 앞에서 설명하지 않은 한 가지를 발견할 수 있을 것이다. 바로 내부 저장용으로 사용하는 저장 영역인데, 이 저장 영역을 정의하는데 있어서 델파이는 어떤 제약 사항도 두지 않고 있다. 그러나 보통은 다음과 같은 규칙을 적용한다.

- 속성 데이터는 클래스 필드에 저장한다.
- 속성 데이터를 저장하기 위한 필드는 private 로 선언하여 컴포넌트 안에서만 접근할 수 있도록 한다.
- 이러한 필드는 F 로 시작하는 이름을 갖는다.

다시 한 번 위의 코드를 보자. Count 속성의 정의를 보면 읽을 경우에는 내부 저장용 필드를 바로 사용하였고 쓸 경우에는 메소드를 사용하였다. 전자의 경우를 직접 접근 방식이라고 하는데 속성을 사용하는 가장 간단한 방법이다.

다음에 보여지는 예제는 이러한 직접 접근 방식을 이용한 속성의 선언이다.

```
Type
  TMyComponent = class(TComponent)
   Private
     FMyProperty: Boolean;
   Published
    Property MyProperty: Boolean read FMyProperty write FMyProperty;
   End;
```

위의 예와 같은 경우는 오브젝트 인스펙터 상에 속성이 나타나고 그 속성의 값을 오브젝트 인스펙터에서 변경할 수 있지만 그 속성의 값이 변하더라도 곧바로 적용되지는 않는다. 보통의 경우에는 속성의 값을 읽기 위해서는 이와 같은 직접 접근 방식을 사용하지만 속성의 값을 설정하기 위해서는 메소드를 사용한다. 이러한 선언은 위의 경우와 반대로 속성의 값을 변화 시키면 컴포넌트의 상태가 즉시 영향을 받는다.

4) 속성에 대한 접근 메소드

메소드를 사용하는 경우에 메소드는 보호되어야 하며 보통은 virtual 로 선언해 준다. 이렇게 하면 이 컴포넌트의 하위 객체가 이 속성의 구현을 오버라이드 할 수 있기 때문이다. 메소드의 선언은 public 으로 하지 않는 것이 좋다. 이렇게 함으로써 애플리케이션의 개발자가 이 메소드를 이용하여 속성을 변경하는 것을 막을 수 있기 때문이다.

다음의 예는 세 개의 속성을 정의하는데, index 라는 예약어를 사용하여 세 개의 속성이 모두 같은 메소드를 사용할 수 있도록 한다.

```
Type
  TMyCalendar = 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;
      Procedure SetDateElement(Index: Integer; Value: Integer);
```

위에 정의된 세 개의 속성은 모두 정수형이라는 공통점을 가지고 있다. 또한 날짜를 구성하

고 있다는 공통점을 가지고 있기 때문에 이러한 경우에는 메소드를 공유하는 것이 더 효율적이다. 즉, 단 두 개의 메소드로 세 개의 속성을 읽고 쓸 수 있는 것이다.

다음의 예는 앞에서 설명한 세 개의 속성을 읽기 위한 메소드이다.

```
Function TMyCalendar.GetDateElement(Index: Integer): Integer;
Var
  AYear, AMonth, ADay: Word;
Begin
  //FDate의 값을 년, 월, 일로 구분하여 AYear, AMonth, ADay 에 저장한다.
  DecodeDate(FDate, AYear, AMonth, ADay);
  Case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
```

다음은 속성을 쓰기 위한 메소드이다.

```
Function TMyCalendar.SetDateElement(Index: Integer; Value: Integer);
Var
  AYear, AMonth, ADay: Word;
Begin
  If Value > 0 then //Value 의 값이 유효한지 검사한다. 날짜는 음수를 가질 수 없다.
   begin
     DecodeDate(FDate, AYear, AMonth, ADay);
     Case Index of
        1: Ayear := Value;
       2: Amonth := Value;
       3: Aday := Value;
        else Exit;
      end;
  //AYear, AMonth. ADay 의 값을 조합하여 FDate에 저장한다.
     FDate := EncodeDate(AYear, AMonth, ADay);
```

```
      Refresh;
   End;
End;
```

속성의 읽기 메소드는 파라미터를 갖지 않는 함수로써 속성과 같은 형의 값을 반환한다. 보통의 경우 읽기 메소드의 이름은 Get 으로 시작한다. 예를 들어 Count 속성의 읽기 메소드는 GetCount 가 될 것이다. 읽기 메소드는 대체로 내부 저장 값을 적절한 형의 속성 값으로 만들어 주는 역할을 한다.

앞에서 읽기 메소드가 파라미터를 가지지 않는다고 했다. 그러나 예외 없는 룰이 없다는 말처럼 이 경우에도 예외가 존재한다. 바로 배열형 속성과 앞에서 설명했던 index 를 사용하는 경우이다.

read 부분이 선언되지 않으면 속성은 쓰기 전용으로 설정되는데, 이러한 속성은 거의 사용되지 않는다.

그렇다면 쓰기 메소드의 경우는 어떨까? 쓰기 메소드는 속성과 같은 형인 하나의 파라미터를 갖는다. 이 파라미터는 참조 혹은 값에 의한 전달이 모두 가능하고 이름에 대한 제한은 없다. 보통의 경우, 쓰기 메소드는 Set 으로 시작하는 이름을 갖는다. 앞에서 예를 들었던 Count 속성의 쓰기 메소드는 SetCount 가 될 것이다. 파라미터로 전달된 값은 속성의 새로운 값으로 설정되고 이 쓰기 메소드는 내부 저장 값으로 적절하도록 만드는 역할을 한다.

쓰기 메소드의 경우에도 앞에서 설명한 '하나의 파라미터' 라는 명제에 예외가 존재하는데, 읽기 메소드와 마찬가지로 배열형 속성의 경우와 index 를 사용하는 경우이다. 이 때는 index 가 쓰기 메소드의 두번째 파라미터로 전달된다.

만약 write 부분이 선언되지 않으면 속성은 읽기 전용으로 설정된다. 만약 설계 시에도 속성을 사용하려면 속성은 반드시 읽고 쓸 수 있도록 정의되어야 한다. 다시 말하면 read 부분과 write부분이 모두 정의되어야 하는 것이다.

쓰기 메소드의 경우는 보통의 경우 값을 변경하기 전에 새로운 값이 현재의 값과 다른지를 조사하는데 사용한다. 다음은 간단한 쓰기 메소드의 예제이다.

```
Procedure TMyComponent.SetCount(Value: Integer);
Begin
  If Value <> FCount then
```

```
   Begin
     FCount := Value;
      Update;
   End;
End;
```

5) Default, Nodefault 지시자

속성을 선언하는 경우에 Default 지시자를 사용하는 것을 볼 수 있다. 즉, 속성의 기본 값을 설정할 수 있는데 델파이는 이 기본 값을 이용하여 속성을 폼 파일에 저장할 지의 여부를 결정한다. 만약 이 default 값을 설정하지 않으면 델파이는 속성을 항상 폼 파일에 저장한다. Default 값을 설정하는 방법은 다음의 예와 같다.

```
  Property Counting Boolean read GetCounting write SetCounting default True;
```

여기서 주의해야 할 점은 이렇게 기본 값이 설정된 경우 이 값이 속성의 값에는 영향을 미치지 않는다는 것이다. 다시 말하면, 속성에 어떤 값도 할당하지 않고 그 속성을 사용할 때 그 속성의 값이 기본 값과 일치하지 않을 수도 있다는 것이다.
그렇다면 이런 경우에 그 속성은 어떤 값을 가지고 있을까? 대개의 경우는 컴포넌트의 생성자에서 속성값을 초기화하고 있다. 하지만 초기화를 하지 않았다 하더라도 정수형 속성은 0, 문자열 속성은 null, 그리고 부울형 속성은 False 로 초기화 된다.

속성을 재선언하는 경우에 상속 받은 속성이 어떠한 기본 값도 가지지 않도록 설정할 수 있는데, 이 경우는 속성을 선언할 때 nodefault 지시자를 추가하여 사용한다.

```
  Property WhatIsIt string nodefault;
```

6) 배열형 속성 생성

어떤 속성들은 그 자신이 인덱스를 갖고 있다. 예를 들면, TMemo 의 Lines 와 같은 경우인데, 이 속성은 메모의 내용을 문자열의 배열로 저장한다. 배열형 속성의 정의는 다음의 두 가지를 제외하고는 앞에서 설명한 일반 속성을 정의하는 것과 같다.

- 속성의 형 정의와 함께 하나 이상의 인덱스를 가진다. 이 인덱스는 어떤 형의 값이라도 가질 수 있다.
- 만약 속성의 read, write부분을 정의한다면 반드시 메소드를 가져야 한다.

다음의 예는 배열형 속성의 정의와 이 속성의 메소드이다.

```
Type
  TMyComponent = class(TComponent)
   Private
     Function GetMonthSeason(Index: Integer): string;
   Public
     Property MonthSeason[Index: Integer]: string read GetMonthSeason;
   End;

function TMyComponent.GetMonthSeason(Index: Integer): string;
begin
  case Index of
      1, 2, 12: Result :=‘ 추운 겨울’;
     3..5: Result := ‘ 따뜻한 봄’;
     6..8: Result := ‘ 뜨거운 여름’;
     9..11: Result := ‘ 선선한 가을’;
   end;
end;
```

7) 속성의 저장과 로딩

델파이는 폼들과 그들의 컴포넌트를 폼 파일(.DFM)에 이진 형식으로 저장한다. 컴포넌트는 개발자가 자신이 만든 컴포넌트를 폼에 추가할 때, 폼이 저장되면 그 컴포넌트도 반드시 함께 저장되는 능력을 가지고 있어야 한다. 마찬가지로 델파이에서 폼 파일을 읽거나 그 프로젝트가 실행되는 경우에는 폼 파일로부터 다시 불러올 수 있어야 한다.

대부분은 컴포넌트를 상속 받게 되면 이러한 능력을 가지고 있게 되므로 문제가 될 것이 없다. 그러나 혹시라도 저장되는 방법을 바꾸거나 로딩될 때 초기화하는 방법을 바꾸고자 한다면 이러한 내용을 알아야 할 것이다..

폼 파일을 살펴보면 폼 파일 내부에는 그 폼의 속성이 나열되어 있고, 마찬가지로 컴포넌트의 속성도 나열되어 있다. 각각의 컴포넌트는 자신의 정보를 저장, 로딩할 수 있어야 한다. 기본적으로 컴포넌트가 저장될 때는 public 과 published 에 선언된 속성 중에 기본 값과 다른 것들을 선언된 순서에 의해서 저장한다. 반대로 로딩하는 경우에는 먼저 컴포넌트가 생

성되고 모든 속성을 기본 값으로 설정한 후 저장되어 있는 기본 값이 아닌 속성들을 읽어오는 방식을 취한다.

델파이 컴포넌트는 기본 값과 다른 경우에만 저장한다고 했다. 그런데 만약 이 기본 값이 정의되어 있지 않으면 어떻게 될까? 이런 경우에는 그 속성의 값에 상관없이 컴포넌트는 항상 그 속성을 저장한다. 기본 값을 설정하는 방법은 앞에서 설명한 것과 같다.

델파이는 컴포넌트의 어떤 속성을 저장할 지를 결정할 수 있도록 한다. 기본적으로 published 로 정의된 속성은 저장되는데 이 속성들 역시 그 저장 여부를 결정할 수 있다. 속성의 저장 여부를 설정하기 위해서는 stored 라는 지시자를 사용하는데 이 지시자 다음에는 True, False, 그리고 부울 메소드의 이름이 들어갈 수 있다.

다음의 코드는 새로운 세 개의 속성을 정의하는데, 그 중 하나는 항상 저장하도록 또 하나는 항상 저장하지 않도록 그리고 나머지 하나는 부울 메소드의 값에 따라 저장 여부를 결정하도록 하고 있다.

```
Type
  TMyComponent = class(TComponent)
   Protected
     Function StoreIt: Boolean;
   Public
     Property Important: Integer stored True;   //항상 저장
   Published
     Property Unimportant: Integer stored False;  //항상 저장하지 않음
     Property Sometimes: Integer stored StoreIt;  //함수 값에 따라 저장 여부 결정
   End;
```

폼 파일에 저장되어 있는 정보를 읽어 들이게 되면, Loaded 라는 이름을 가진 가상 메소드가 호출된다. 이 메소드는 필요한 초기화를 수행하는데, 이 메소드는 폼과 컴포넌트가 보여지기 전에 호출되므로 초기화로 인한 화면의 깜빡 거림에 대해 걱정할 필요는 없다. 컴포넌트의 속성 값을 불러들인 후 초기화하는 방법을 변경하려면 Loaded 메소드를 오버라이드하여 사용하면 된다.

다음의 코드는 TDatabase에 정의되어 있는 Loaded 메소드이다.

```
procedure TDatabase.Loaded;
```

```
begin
  inherited Loaded;
  try
    if FStreamedConnected then Open
     else CheckSessionName(False);
  except
    if csDesigning in ComponentState then
        Application.HandleException(Self)
     else
        raise;
  end;
end;
```

## 4.2 이벤트 생성

이벤트는 시스템에서 발생한 사건과 그 사건의 응답인 코드의 연결을 말한다. 이 응답 코드를 이벤트 핸들러라고 하며 대부분 애플리케이션 개발자에 의해 쓰여진다. 이벤트는 애플리케이션 개발자가 클래스를 변경하지 않고도 컴포넌트의 행위를 조정할 수 있도록 한다.

이벤트의 생성은 속성의 생성과 거의 유사하다. 독자들은 먼저 속성의 생성에 대한 내용을 알고 있으면 이 부분을 쉽게 이해할 수 있을 것이다.

### 1) 이벤트란 무엇인가?

* 이벤트는 특정 클래스 인스턴스의 메소드를 지칭하는 메소드 포인터이다.
애플리케이션 개발자의 입장에서 본다면 이벤트는 OnClick 과 같이 단지 시스템의 사건에 대한 이름일 뿐이다. 예를 들면, Button1 이라는 이름의 버튼은 OnClick 이라는 메소드를 가지고 있다. 기본적으로 델파이는 Button1Click 이라는 이름의 이벤트 핸들러를 만들게 되고, 버튼에 클릭 이벤트가 발생하게 되면 OnClick 에 할당되어 있는 메소드를 호출하게 되는데 위의 경우에는 Button1Click 이 그것이다.

* 이벤트는 메소드 포인터이다.
델파이에서는 이벤트를 구현하기 위해 메소드 포인터를 이용한다. 메소드 포인터는 인스턴스 객체의 특정 메소드를 지칭하기 위한 특별한 포인터 형이다. 컴포넌트의 코드는 이벤트가 발생한 것을 감지하면 그 이벤트에 대해 사용자가 정의한 메소드를 호출한다. 메소드 포인터는 마치 일반 프로시저와 같이 동작한다. 그러나 이 메소드 포인터는 객체에 대한 감춰

진 포인터를 유지한다. 애플리케이션 개발자가 핸들러를 컴포넌트의 이벤트에 할당할 때 이 할당은 단지 특별한 이름을 가진 메소드가 아니라 특정 인스턴스 객체의 메소드인 것이다. 이 특정 인스턴스 객체는 대개 컴포넌트를 포함한 폼이지만 꼭 그럴 필요는 없다.

다음의 예는 모든 컨트롤이 클릭 이벤트를 핸들링하기 위한 Click 이라는 동적 메소드를 보여 준다.

```
  procedure Click; dynamic;
```

* 이벤트는 속성이다.
컴포넌트는 이벤트를 구현하기 위해 속성을 사용한다. 대부분의 다른 속성과는 달리 이벤트는 read, write 부분에 메소드를 사용하지 않는다. 대신 이벤트 속성은 속성과 같은 형의 클래스 필드를 사용한다. 편의를 위해 필드의 이름은 속성의 이름처럼 F 로 시작한다. 예를 들면, OnClick 메소드 포인터는 TNotifyEvent 형의 FOnClick 이라는 필드에 저장된다. OnClick 이벤트 속성의 선언은 다음과 같은 형태이다.

```
Type
  TControl = class(TComponent)
   Private
     FOnClick: TNotifyEvent;
   Protected
    Property OnClick: TNotifyEvent read FOnClick write FOnClick;
```

다른 속성들과 마찬가지로 실행 시에 이벤트의 값을 설정 혹은 변경할 수 있다. 이벤트를 속성으로 가지는 것의 가장 큰 장점은 컴포넌트를 사용하는 경우 개발 시에 오브젝트 인스펙터를 이용하여 핸들러를 이벤트에 할당할 수 있다는 것이다.

* 이벤트 형은 메소드 포인터 형이다.
이벤트는 이벤트 핸들러를 지시하고 있기 때문에 이벤트 속성의 형은 메소드 포인터 형이어야 한다. 또한 이벤트 핸들러에 사용될 코드는 적당한 형의 메소드이어야 한다. 모든 이벤트 핸들러 메소드는 프로시저이다. 주어진 형의 이벤트와 사용하기 위해 이벤트 핸들러 메소드는 같은 수의 같은 타입인 파라미터를 같은 순서로 같은 방법을 이용하여 전달해야 한다. 델파이는 모든 기본 이벤트에 대해 메소드 형을 선언하고 있는데, 새로운 이벤트를 생성하는 경우에는 적절한 형이 있다면 그 형을 사용할 수 있고, 그렇지 않다면 새로운 형을 선언하여 사용할 수 도 있다.

* 이벤트 핸들러 타입은 프로시저이다.
컴파일러가 함수 형태의 메소드 포인터 형을 선언할 수 있도록 하고 있지만, 이벤트를 핸들링하기 위해서는 그렇게 사용하지 말아야 한다. 함수는 정의되지 않은 형태의 결과를 돌려주거나 혹은 함수 자체가 유효하지 못 한 경우가 발생할 수 있기 때문이다. 이러한 이유 때문에 모든 이벤트와 이 이벤트들에 할당된 이벤트 핸들러는 프로시저로 선언되어야 한다. 이벤트 핸들러는 함수로 선언될 수 없지만, var 파라미터를 사용하면 애플리케이션으로부터 정보를 받을 수 있다. 이렇게 하는 경우 핸들러를 호출하기 전에 파라미터를 반드시 유효한 값으로 할당해야 한다.

Var 파라미터를 사용한 예는 OnKeyPress 이벤트의 이벤트 핸들러에서 볼 수 있다.

```
Type
  TKeyPressEvent = procedure(Sender: Tobject; var Key: Char) of object;
```

여기서 Key 파라미터는 사용자가 키보드를 이용하여 입력한 문자를 가지고 있다. 그런데 어떤 경우에는 그 문자를 다른 문자로 바꾸기를 원할 수 있다. 이런 경우 핸들러를 다음과 같이 정의하여 사용할 수 있다. 이 예는 들어온 문자를 모두 대문자로 바꾸는 코드이다.

```
Procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
Begin
  Key := UpCase(Key);
End;
```

* 이벤트 핸들러는 선택적이다.
이벤트를 생성할 때, 컴포넌트를 이용하는 개발자가 이벤트 핸들러를 정의하지 않을 수도 있다. 이것은 컴포넌트가 특정 이벤트에 핸들러가 없어도 문제가 생겨서는 안됨을 뜻한다. 또한 컴포넌트에서 필요로 하지 않는 이벤트는 포함하지 않아도 된다.

## 2) 표준 이벤트 구현

델파이의 컨트롤들은 윈도우에서 보통 발생하는 이벤트들을 상속 받았다. 이 이벤트들을 표준 이벤트라고 하는데, 이 이벤트들이 모두 컨트롤 안에 구현되어 있지만 이것들에 핸들러를 할당할 수 없는 경우도 많이 있다. 컨트롤을 생성할 때, 이벤트를 컨트롤의 사용자에게 보여 줄지를 결정할 수 있다.

* 표준 이벤트
표준 이벤트들은 두 가지로 구분할 수 있는데, 그것은 모든 컨트롤에 정의된 것들과 표준 윈도우 컨트롤에만 구현된 것들이다.

모든 컨트롤에 구현된 것들에는 OnClick, OnDragDrop, OnEndDrag, OnMouseMove, OnDblClick, OnDragOver, OnMouseDown, 그리고 OnMouseUp 이다. 이 이벤트들은 TControl 클래스에 선언되어 있고, 이들은 각각 이벤트 이름과 연관된 이름을 가진 가상 메소드를 갖는다. 예를 들면, OnClick 은 Click 메소드를 호출하고 OnEndDrag 는 DoEndDrag 메소드를 호출한다.

표준 윈도우 컨트롤에만 선언되어 있는 이벤트는 OnEnter, OnKeyDown, OnKeyPress, OnKeyUp, OnExit 등으로 이 이벤트들도 위에 설명한 것과 마찬가지로 이 이벤트들에 연결되어 있는 가상 메소드를 갖는다.

* 이벤트를 볼 수 있도록 만들자.
표준 이벤트들과 그 메소드들의 선언이 protected 이므로, 만약 이들 추상 클래스 중 하나에서 상속 받은 이벤트를 실행 시나 개발 시에 사용하려면 그 이벤트를 public 이나 published 로 재선언해 주어야 한다.

다음의 예는 OnClick 이벤트를 개발 시에 사용하기 위해 재선언한 경우이다.

```
Type
  TMyControl = class(TCustomControl)
   :
   published
     property OnClick;
   end;
```

* 표준 이벤트 핸들링을 바꿔 보자.
만약 컴포넌트가 어떤 이벤트에 대해 응답하는 방법을 바꾸고자 한다면 이것을 코드로 작성하여 이벤트에 할당하는 방법을 사용할 수 있다. 또한 메소드의 구현을 오버라이드 함으로써 내부 이벤트 핸들러를 수정하거나, 이 수정된 핸들러를 상속 받아 호출하는 것이 가능하다.

다음의 코드는 Click 메소드를 오버라이드하는 것을 보여 준다.

```
Procedure Click override  //선언부분
:
procedure TMyComponent.Click;
begin
   inherited Click;
  // 원하는 작업을 구현한다.
End;
```

3) 새로운 이벤트 정의하기.

완전히 새로운 이벤트를 정의하는 경우는 거의 없다. 그렇지만 기존에 갖고 있는 이벤트가 아니라면 새로운 이벤트를 정의해서 사용할 수 있다. 다른 컴포넌트에서 찾아볼 수 없는 동작을 구현하려면 많은 시간을 투자해야만 할 것이다. 혹시라도 그런 경우가 발생하면 다음에 설명하는 것들을 고려해야 한다.

* 이벤트 조작
먼저 무엇이 이벤트를 조작하는지 알아야 한다. 어떤 이벤트들에게서는 상당히 명확한 답을 얻을 수 있다. 예를 들어, MouseDown 이벤트의 경우는 사용자가 마우스의 왼쪽 버튼을 누르면 윈도우는 WM_LBUTTONDOWN 이라는 메시지를 애플리케이션에 보내게 되고 애플리케이션이 이 메시지를 받게 되면 MouseDown 이벤트를 발생한다. 그러나 모든 이벤트가 이렇듯 확실하지는 않다. 스크롤 바가 갖는 OnChange 이벤트의 경우는 이 이벤트를 발생시키는 사건이 여러 종류이다. 키보드를 누른 경우, 마우스를 클릭한 경우, 그리고 다른 컨트롤의 변화에 의한 경우 등이 모두 가능하기 때문이다. 그러므로 새로운 이벤트를 선언하는 경우에는 이 이벤트를 발생시키는 적절한 시스템 사건들을 먼저 확실하게 알아야 한다.

이벤트에 제공되는 시스템 사건의 종류로는 두 가지가 있는데, 그 중의 하나는 사용자에 의한 것과 다른 하나는 상태의 변화에 의한 것이다. 먼저 사용자에 의한 이벤트는 대부분이 사용자가 컴포넌트에 어떤 작업을 했음을 의미하는 윈도우 메시지에 의해 조작된다. 상태 변화 이벤트도 포커스 전환과 같은 경우 윈도우 메시지와 관련이 있지만, 속성의 변화나 코드에 의해서도 발생할 수 있다. 이벤트의 선언은 애플리케이션 개발자가 이해하고 사용할 수 있도록 신중하게 이루어져야 한다.

* 핸들러 형 선언
이벤트의 발생 시점에 대한 내용이 정리되었다면 이제 이벤트가 어떻게 핸들링 될 지를 선언해야 한다. 이것은 이벤트의 형을 결정하는 것을 의미한다.

- 단순 통지형(Simple notifications) : 이런 형은 단순히 어떤 이벤트가 발생했는지를 알려주는 것으로, 언제, 어디서 발생했는지에 대한 추가 정보가 필요하지 않다. 이와 같은 경우는 Sender 라는 하나의 파라미터를 사용하는 TNotifyEvent 형을 사용한다. 핸들러가 이벤트에 대해서 알아야 할 것은 '어떤 종류의 이벤트인가' 그리고 '어떤 컴포넌트에서 발생했는가' 라는 것이다. 예를 들면, Click 이벤트가 통지형 핸들러로써 클릭 이벤트의 핸들러를 작성하면 알 수 있는 것은 클릭 이벤트가 발생했다는 것과 어느 컴포넌트가 클릭 되었다는 것이다.

- 이벤트별 핸들러 : 어떤 경우에는 위에 설명한 정보들만으로는 부족한 경우가 있다. 예를 들어 KeyPressed 이벤트의 경우에는 앞에서 설명한 정보들 이외에도 어떤 키가 눌렸는지 알아야 하기 때문이다. 이 경우에는 핸들러 형이 추가적인 정보를 위한 파라미터를 가져야 한다.

- 핸들러로부터 정보 반환 : 모든 이벤트 핸들러가 프로시저이기 때문에 핸들러로부터 정보를 돌려 받으려면 var 파라미터를 사용해야 한다. 컴포넌트는 그 정보를 이용하여 사용자의 핸들러가 실행된 후에 이벤트를 처리할 지, 혹은 어떤 방법으로 처리할 지를 결정하는 데 사용할 수도 있다.

* 이벤트 선언
이벤트 핸들러의 형을 결정했다면 메소드 포인터와 이벤트 속성을 선언할 준비가 된 것이다. 이벤트에는 사용자가 그 이벤트가 무엇인지 알 수 있도록 의미 있고 그 이벤트를 설명할 수 있는 이름을 주도록 한다. 델파이의 대부분 이벤트는 On 으로 시작하는 이름을 갖는다. 이것은 단지 편의를 위한 것이고 강제성이 있는 것은 아니다. 오브젝트 인스펙터는 속성의 형이 무엇인지를 확인하여 이 속성이 이벤트 속성인지를 구별한다. 즉, 모든 메소드 포인터 속성은 이벤트 페이지에 나타나게 되는 것이다.

* 이벤트 호출
이벤트의 호출은 하나로 통합시켜 놓는 것이 좋다. 즉, 컴포넌트에 애플리케이션의 이벤트 핸들러들이 호출하는 가상 메소드를 생성하고 이 메소드에서 기본 핸들링을 제공하도록 해야 한다. 모든 이벤트 호출을 한 곳에서 이루어지도록 하면 다른 개발자가 이 컴포넌트로부터 상속 받는 컴포넌트를 생성하는 경우, 하나의 메소드만 오버라이드 하면 이벤트 핸들링을 새롭게 설정할 수 있다.

- 핸들러가 없어도 유효해야 한다. : 이벤트 핸들러가 없어도 에러를 발생하지 않아야 하며, 애플리케이션의 이벤트 핸들링 코드의 특정한 응답에 의해서만 컴포넌트가 정확한 동작을 하도록 만들어서는 안 된다. 애플리케이션의 이벤트 핸들러를 호출하는 코드는 다음의 예와 같은 모습을 갖는다.

```
if Assigned(OnClick) then OnClick(Self);
… //기본 핸들링
```

다음과 같은 모습을 가지면 안 된다.

```
if Assigned(OnClick) then OnClick(Self)
else  …//기본 핸들링
```

- 사용자가 기본 핸들링을 오버라이드 할 수 있다 : 어떤 종류의 이벤트들에 대해서는 애플리케이션 개발자가 기본 핸들링을 바꾸거나 삭제하기를 원할 수 있다. 이러한 작업이 가능하게 하려면 핸들러로 참조에 의해 인수를 전달하고 핸들러가 반환될 때 그 값을 검사하도록 해야 한다. 예를 들면, KeyPressed 이벤트의 경우 애플리케이션 개발자가 var 파라미터인 Key 를 null 문자로 설정하여 기본 핸들링을 삭제할 수 있다.

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then  …//기본 핸들링 구현
```

실제의 코드는 윈도우 메시지와 관련되어 아마도 조금 다른 모습을 갖게 될 것이다. 그러나 같은 원리를 갖는다. 기본적으로 이 컴포넌트는 사용자가 할당한 핸들러를 호출하고, 표준 핸들링을 수행한다. 만약 사용자의 핸들러가 Key 의 값을 null 문자로 설정하게 되면, 컴포넌트는 기본 핸들링을 수행하지 않는다.

## 4.3. 메소드 생성

컴포넌트 메소드들은 클래스의 구조 내에 프로시저나 함수로 선언된다. 컴포넌트의 메소드에서 무슨 작업을 하든지 제약 사항을 가지고 있지는 않지만, 델파이에서는 몇 가지 표준안을 가지고 있다. 이 표준 안들에 대해서 이제부터 알아보도록 한다.

1) 의존성을 가지지 않는다.
컴포넌트를 작성할 때는 전제 조건을 최소화해야 한다. 최대한 애플리케이션 개발자들이 이 컴포넌트로 그들이 하고자 하는 모든 일을 언제든지 원하기만 하면 할 수 있도록 해야 하는 것이다. 물론 이러한 작업이 힘든 것은 사실이지만 최대한 가능하도록 해야 한다. 다음은 피해야 할 의존성들이다.

- 컴포넌트를 사용하기 위해 반드시 사용자가 호출해야 하는 메소드.

- 특정한 순서로 실행되어야 하는 메소드.
- 어떤 메소드 혹은 이벤트가 발생할 수 없는 상태로 컴포넌트를 전환시키는 메소드.

위에 열거한 메소드들은 반드시 피해야 한다. 그러기 위해서는 이러한 상황을 피하기 위한 방법을 제공해야 하는데, 최소한 예외(exception)를 발생시킬 수는 있어야 할 것이다. 다시 말하면, 만약 어떤 코드들이 서로 의존하는 상황을 만들게 되었다면 이 코드들이 문제를 발생하지 않도록 하는 책임은 컴포넌트의 개발자에게 있는 것이다.

2) 메소드의 이름

델파이는 메소드나 그 메소드가 갖는 파라미터의 이름에 대해서는 어떤 제약 사항도 가지고 있지 않다. 하지만 여기에도 애플리케이션 개발자의 편의를 위한 몇 가지 사항이 있다. 여러분이 만들게 될 컴포넌트를 수 많은 사람들이 사용할 수 있다는 것을 명심하자. 만약 여러분이 여러분 자신이나 몇몇의 프로그래머만이 사용하도록 작성하고 있다면, 이 메소드들의 이름을 무엇으로 지을 것인가에 대해 그리 깊게 생각할 필요는 없다. 그러나 메소드의 이름을 여러분이 작성하는 컴포넌트의 구조에 대해 잘 모르는 다른 개발자 혹은 코딩 자체에 익숙하지 못한 다른 개발자들을 위해 이름 만으로도 그 메소드가 어떤 작업을 할 지 알 수 있도록 하는 것이 더 좋지 않을까? 메소드의 이름을 만드는 몇 가지 제안 사항은 다음과 같다.

- 이름 자체로 설명이 가능하도록 의미 있는 단어들을 사용한다. – 우리가 흔히 사용하게 되는 Copy 라는 단어를 이름으로 정하는 것 보다는 CopyToClipboard 라고 사용하는 것이 그 의미가 명확하다. 즉, 간단하고 짧은 이름보다는 의미를 갖도록 하는 것이 중요하다.
- 함수의 이름은 그 함수가 무엇을 반환하는지를 파악할 수 있도록 한다. – 만약 X 라는 함수가 있다고 가정하자. 이 이름만을 가지고는 이 함수가 어떤 작업을 하고, 어떤 값을 반환하는지 알기 어렵다. 그렇지만 이 함수의 이름을 GetHorizontalPosition 이라고 명명하면 어떨까? 아마도 영어만 이해할 수 있다면 이 함수가 어떤 값을 반환하는지 명확히 알 수 있을 것이다.

마지막으로 한 가지 더 고려해야 할 점은 컴포넌트에 선언된 메소드들이 과연 메소드로써 필요한 것인지의 여부이다. 만약 어떤 메소드가 굳이 메소드로 사용되어야 할 필요가 없는 경우라면 그 메소드 대신에 속성으로 사용하는 것이 더 효율적일 것이다.

3) 메소드 보호

필드, 메소드, 그리고 속성을 포함한 클래스의 모든 부분에서 보호 또는 가시범위에 대해 선언되어 있다. 메소드에 적절한 가시범위를 설정하는 것은 아주 간단하다. 여러분이 작성하는 대부분의 메소드는 public 혹은 protected 로 선언된다. 만약 그 컴포넌트에서만 사용할 수 있거나, 이 컴포넌트로부터 상속을 받은 다른 컴포넌트에서 조차 사용할 수 없는 메소드라면 private로 선언하게 되는 데, 이러한 경우는 극히 드물다.

* public 으로 선언되어야 할 메소드.
애플리케이션 개발자가 호출해야 하는 모든 메소드는 public 으로 선언 되야 한다. 대부분의 메소드가 이벤트 핸들러에서 호출된다는 것을 명심하자. 이것은 메소드가 시스템 자원을 낭비하거나 윈도우가 사용자에게 응답을 보낼 수 없는 상태를 예방하기 위함이다. 그 예로 생성자와 소멸자는 항상 public 으로 선언된다.

* protected 로 선언되어야 할 메소드.
애플리케이션이 언제든지 호출할 수 있는 메소드가 아니라면 이 메소드는 protected 로 선언되어야 한다. 또한 애플리케이션에서는 호출하지 않지만, 이 컴포넌트에서 상속 받은 컴포넌트가 사용해야 할 메소드라면 역시 protected 로 선언한다. 예를 들면, 특정한 데이터 셋을 가지고 작업을 하는 메소드가 있는데, 이 메소드를 public 으로 선언한다면 애플리케이션이 데이터가 준비되기 전에 이 메소드를 호출할 수 있어 잘못된 결과를 가져올 수 있다. 그러나, 만약 이 메소드를 protected 로 선언하면 이 메소드는 애플리케이션에 의해 직접 호출될 수 없게 되고, 먼저 public 으로 선언된 메소드를 이용하여 데이터를 준비한 후 protected 로 선언된 메소드를 호출하면 앞에서 설명한 문제는 예방할 수 있게 된다. 또한 속성을 위한 메소드는 가상 protected 로 선언된다. 이 메소드들은 애플리케이션 개발자들이 속성의 구현을 오버라이드하여 사용할 수 있도록 하고, 이 메소드들이 직접적으로 호출되는 것을 막기 위해 가상 protected 로 선언한다.

* 추상 메소드
델파이 컴포넌트에서 어떤 메소드들은 abstract 로 선언된다. VCL 에서는 이러한 추상 메소드들이 'custom' 이라는 이름으로 시작하는 클래스에서만 존재한다. 이러한 클래스들은 그들 자신이 추상 클래스이고 이 클래스들은 오직 다른 하위 클래스들의 상속을 위해서만 존재한다. 이러한 추상적인 구성 요소를 가진 클래스는 인스턴스 객체를 생성할 수 없다.

### 4) 가상 메소드 생성

가상 메소드는 같은 메소드 호출에 대해 서로 다른 형이 서로 다른 코드를 수행하도록 하기를 원하는 경우 선언한다. 만약 애플리케이션 개발자에 의해 직접적으로 사용될 컴포넌트를 생성한다면 어떠한 메소드도 가상으로 만들 필요는 없다. 그러나 만약 이 컴포넌트로부터

다른 컴포넌트가 상속 받아 생성될 수 있도록 하려면 가상 메소드를 추가해야 한다.

5) 메소드 선언

컴포넌트에 메소드를 선언하는 것은 클래스 메소드를 선언하는 것과 동일하다. 메소드를 컴포넌트에 새롭게 선언하려면 다음과 같은 두 가지 작업이 필요하다.

- 컴포넌트의 객체 형 선언 부분에 선언을 추가한다.
- 컴포넌트 유닛의 implementation 부분에 메소드를 구현한다.

다음의 코드는 두 개의 새로운 메소드를 선언하는데, 그 중 하나는 protected 로 선언되어 있고, 다른 하나는 public 으로 선언된 가상 메소드이다.

```
Type
  TMyComponent = class(TControl)
   Protected
     Procedure MakeBigger;
   Public
     Function CalculateArea: Integer; virtual;
   End;
:
implementation
:
procedure TMyComponent.MakeBigger;
begin
  Height := Height + 5;
  Width := Width + 5;
End;

Function TMyComponent.CalculateArea: Integer;
Begin
  Result := Width * Height;
End;
```

4.4 컴포넌트에 그래픽 사용하기.

윈도우는 디바이스에 독립적인 그래픽을 그릴 수 있도록 강력한 그래픽 디바이스 인터페이스(GDI)를 제공한다. 이러한 그래픽 디바이스 인터페이스는 개발자에게 그래픽 자원을 관리해야 하는 등의 추가적인 요구사항을 가지고 있다. 델파이는 이러한 그래픽 디바이스 인터페이스를 관리하여 개발자로 하여금 특별한 신경을 쓰기 않아도 되도록 하고 있다. 물론 다른 API 들과 마찬가지로 이러한 그래픽 디바이스 인터페이스 함수도 직접적으로 호출하여 사용할 수 있다. 그러나 델파이의 그래픽 함수를 이용하는 것이 더 쉽고 빠르다.

1) 그래픽이란?

델파이는 윈도우의 GDI 함수를 몇 단계로 나누어서 캡슐화 한다. 컴포넌트 제작자로써 가장 중요한 것은 컴포넌트가 화면에 보여지는 방법일 것이다. 델파이는 이를 위해 간단하고 완전한 인터페이스를 준비해 놓았다. 바로 여러분의 컴포넌트에 있는 캔버스 속성이 그것이다. 캔버스는 그래픽에 관련된 모든 자원을 관리해 주기 때문에 pen 혹은 brush 등의 핸들을 생성, 선택, 또는 해제하는 등의 작업에 신경 쓸 필요가 없다. 여러분은 오직 캔버스에게 어떤 종류의 pen 을 사용할 지만 알려 주면 된다. 그 뒷 일은 캔버스가 깨끗이 해결한다.

델파이가 그래픽 자원을 관리하면 캐쉬를 이용하기 때문에 다음에 일어나는 비슷한 작업의 속도 향상을 가져올 수 있다는 장점이 있다. 예를 들어 당신의 애플리케이션이 어떤 종류의 pen 을 생성하여 사용하고 해제하는 작업을 반복적으로 해야 한다면 생성, 사용, 그리고 해제하는 작업을 매번 해 주어야 한다. 그러나 델파이는 한 번 사용했던 자원에 대해 캐쉬를 이용하므로 다음 번 작업 시에는 새로운 자원을 생성하지 않고 캐쉬에 있는 자원을 이용하게 된다.

다음에 보여질 두 코드는 파란색 테두리를 가진 노란색 타원을 그린다. 첫번째 코드는 표준 GDI 함수를 사용한 것이고, 두번째는 캔버스를 사용한 것이다.

```
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)); //파란색 펜 생성
  OldPenHandle := SelectObject(PaintDC, PenHandle); //DC에게 파란색 펜을 사용 알림
  BrushHandle := CreateSolidBrush(RGB(255, 255, 0)); //노란색 붓 사용
  OldBrushHandle := SelectObject(PaintDC, BrushHandle); //DC에게 노란색 붓 사용 알림
  Ellipse(HDC, 10, 10, 50, 50); //타원 그림
  SelectObject(OldBrushHandle); //원래의 붓으로 돌려줌
  DeleteObject(BrushHandle); //노란색 붓 삭제
  SelectObject(OldPenHandle); //원래의 펜으로 돌려줌
  DeleteObject(PenHandle); //파란색 펜 삭제
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
    Pen.Color := clBlue; //펜 색을 파란색으로 설정
    Brush.Color := clYellow; //붓을 노란색으로 설정
    Ellipse(10, 10, 50, 50); //타원을 그림
  end;
end;
```

2) 캔버스 사용

캔버스 클래스는 윈도우 그래픽을 여러 단계로 캡슐화 한다. [표 2]는 캔버스를 가지고 할 수 있는 일들을 보여 주고 있다.

| 단계 | 작업 | 툴 |
|---|---|---|
| 높은 단계 | 선이나 모양을 그린다. | MoveTo, LineTo, Rectangle, Ellipse 와 같은 메소드 |
| | 문자열을 보여 준다. | TextOut, TextHeight, TextWidth, TextRect 메소드 |
| | 구역을 칠한다. | FillRect, FloodFill 메소드 |
| 중간 단계 | 문자열, 그래픽을 설정 | Pen, Brush, Font 속성 |
| | 픽셀 계산 | Pixels 속성 |
| | 그림을 복사 및 붙이기 | Draw, StretchDraw, BrushCopy, CopyRect 메소드; CopyMode 속성 |
| 낮은 단계 | 윈도우 GDI 함수 호출 | Handle 속성 |

[표 2] 캔버스 기능

캔버스의 속성 및 메소드에 대한 자세한 정보는 델파이 도움말을 참조하면 된다.

3) 그림 다루기

여러분이 델파이에서 하게 되는 대부분의 그래픽 작업은 컴포넌트나 폼의 캔버스에 직접 그리는 것만으로는 부족하다. 그래서 델파이에서는 비트맵, metafile, 그리고 아이콘과 같은 이미지를 다룰 수 있는 기능을 제공한다.

* 그림(Picture), 그래픽 또는 캔버스 사용하기
델파이에는 그래픽과 관련된 세 가지 클래스가 있다.

- 캔버스(TCanvas)는 폼, 그래픽 컨트롤, 프린터 또는 비트맵에 비트맵 그림을 보여준다. 캔버스는 다른 클래스의 속성으로 사용되고, 그 자체가 따로 사용되지는 않는다.
- 그래픽(TGraphic)은 비트맵, 아이콘 또는 metafile 등과 같은 파일이나 자원의 그래픽 이미지를 보여 준다. 델파이는 TImage, TIcon 그리고 TMetafile 클래스를 정의하고 있고, 이것들은 모두 TGraphic 의 하위 클래스이다. 여러분은 물론 여러분만의 그래픽 클래스를 정의할 수 있다. TGraphic 은 모든 그래픽을 위한 최소한의 표준 인터페이스만을 정의하여 다른 모든 종류의 그래픽을 이용하는 애플리케이션을 쉽게 만들 수 있도록 단순한 구조를 제공한다.
- 그림(TPicture)은 모든 그래픽 클래스를 포함할 수 있다. TPicture 형은 비트맵, 아이콘, metafile, 또는 사용자 정의 그래픽 형을 포함할 수 있고, 애플리케이션은 picture 클래스를 이용하여 같은 방법으로 이들을 접근할 수 있다.

* 그래픽 로딩 및 저장
델파이의 모든 그림과 그래픽들은 그들의 이미지를 파일로부터 읽어 들이거나 다시 저장 혹은 다른 파일로 저장할 수 있다.

- 파일로부터 이미지를 로드 하려면 LoadFromFile 메소드를 사용한다.
- 파일로 이미지를 저장하려면 SaveToFile 메소드를 사용한다.

이 두 메소드는 파라미터로 오직 파일의 이름만을 사용한다. LoadFromFile 은 어떤 종류의 그래픽 객체를 생성하고 로드할 것인지를 결정하는데 파일 이름의 확장자를 사용한다. SaveToFile 은 파일의 형이 저장할 그래픽 객체의 형에 적합하든 그렇지 않든 간에 저장한다. 예를 들어 비트맵을 이미지 컨트롤에 로드하는 경우 LoadFromFile 메소드에 비트맵 파일의 이름을 넘겨 주면 된다.

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile(' 물방울.BMP' );
```

End;

여기서 확장자가 .BMP 라는 표준 비트맵 확장자이므로 이것을 TBitmap 으로 그래픽을 생성한다. 그리고 LoadFromFile 을 호출한다.

* 팔레트 핸들링

팔레트 기반의 디바이스에서 실행되는 경우(보통 256 색상 비디오 모드), 델파이는 팔레트를 자동으로 제어하는데, 만약 여러분이 팔레트를 가진 컨트롤을 가지고 있다면, 여러분은 TControl 로부터 상속 받은 두 개의 메소드를 사용할 수 있는데 이 메소드는 윈도우가 팔레트를 제어하는 방법에 대해 구현되어 있다.

대부분의 컴포넌트에서는 팔레트가 필요하지는 않지만, 이미지 컨트롤과 같은 컴포넌트는 윈도우나 스크린 디바이스 드라이버와 상호작용이 필요하므로 팔레트를 사용하게 된다. 활성 윈도우는 자신의 팔레트를 완전하게 사용하고, 비활성 윈도우들은 그들이 사용할 수 있는 만큼의 팔레트만을 사용한다. 그리고 현재의 색을 그 팔레트에서 가장 가까운 색으로 설정하는 것이다. 활성 윈도우가 바뀌게 되면 위의 작업을 그 윈도우에 대해 재실행한다.

델파이는 비트맵과는 달리 팔레트를 생성하거나 관리하는 것을 특별히 지원하지는 않는다. 그러나 만약 팔레트 핸들을 가지고 있다면, 델파이는 여러분 대신 팔레트를 관리해 준다.

4) 오프 스크린 비트맵
복잡한 그래픽 이미지를 그리는 경우, 윈도우 프로그래밍에서 가장 보편적으로 사용되는 방법이 오프 스크린 비트맵을 생성하고, 이미지를 이 비트맵에 그린 후 이 비트맵을 화면상으로 복사하는 것이다. 이 방법은 반복적으로 화면상에 직접 그리게 되면 발생하는 깜빡 거림을 줄일 수 있다. 자원과 파일의 비트맵 이미지를 보여 주는 델파이의 비트맵 클래스 역시 오프 스크린 이미지처럼 작업할 수 있다.

* 오프 스크린 비트맵의 생성과 관리
복잡한 그래픽 이미지를 생성하는 경우에는 화면상에 보여진 캔버스에 직접 그리지 않는 것이 좋다. 폼이나 컨트롤의 캔버스에 직접 그리는 대신에 비트맵 객체를 생성하여 이 비트맵 객체의 캔버스에 그리고 이 이미지를 화면 캔버스에 복사하는 방법을 쓰는 것이 좋다. 오프 스크린 비트맵을 가장 보편적으로 사용하고 있는 것은 그래픽 컨트롤의 Paint 메소드이다. 다른 임시 객체와 마찬가지로 비트맵은 try..finally 의 내부에 사용되어야 한다.

Type

```
  TFancyControl = class(TGraphicControl)
   Protected
     Procedure  Paint;  override;
   End;

Procedure TFancyControl.Paint;
Var
  Bitmap:  TBitmap;
Begin
  Bitmap  :=  TBitmap.Create;
   Try
     //비트맵에 그림을 그린 후 결과를 컨트롤의 캔버스에 복사한다.
   Finally
     Bitmap.Free;
   End;
End;
```

* 비트맵 이미지 복사
델파이에서는 캔버스에서 다른 캔버스로 복사하는 4 가지 방법을 제공한다. 사용하는 메소드에 따라 서로 다른 결과를 가져온다.[표 3]

| 작업 | 메소드 |
|---|---|
| 전체 그래픽을 복사 | Draw |
| 크기를 변경하여 복사 | StretchDraw |
| 캔버스의 일부를 복사 | CopyRect |
| 하나의 색을 변경하면서 복사 | BrushCopy |

[표 3] 이미지 복사 메소드

* 객체의 변경에 대한 응답
캔버스와 캔버스가 가지고 있는 펜, 브러쉬, 그리고 폰트와 같은 객체들은 객체의 변경에 대한 응답을 위한 이벤트들을 가지고 있다. 이 이벤트들을 이용하면 컴포넌트 혹은 이 컴포넌트를 이용하는 애플리케이션이 이미지가 다시 그려짐으로 인한 응답을 할 수 있도록 한다.
그래픽 객체의 변경에 대한 응답은 이 응답을 애플리케이션의 설계 시에 사용하려 할 때 특히 중요하다. 컴포넌트의 설계 시 모습이 오브젝트 인스펙터 상의 속성 값과 일치하도록 하려면 객체의 변경에 대한 응답이 유일한 방법이기 때문이다. 그래픽 객체의 변경에 응답하려면 클래스의 OnChange 이벤트에 메소드를 할당하면 된다. Shape 컴포넌트의 코드를 살

펴보면, 컴포넌트의 생성자는 펜이나 브러쉬가 변경되면 이미지를 다시 그리도록 각각의 OnChange 이벤트에 메소드를 할당하고 있다.

```
Type
  TShape = class(TGraphicControl)
   Public
     Procedure StyleChanged(Sender: TObject);
   End;
:
implementation
:
constructor TShape.Create(AOwner: TComponent);
begin
   inherited Create(AOwner);
  width := 65;
  Height := 65;
  FPen := TPen.Create;
  FPen.OnChange := StyleChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := StyleChanged;
End;

Procedure TShape.StyleChanged(Sender: TObject);
Begin
   Invalidate();
End;
```

## 4.5 메시지 핸들링

전형적인 윈도우 프로그래밍의 특징중의 하나는 윈도우로부터 애플리케이션에 보내지는 메시지를 핸들링하는 것이다. 델파이는 대부분의 메시지를 핸들링 해 준다. 또한 델파이가 핸들링하지 않거나 개발자가 직접 만든 메시지를 처리하는 것도 가능하다.

### 1) 메시지 핸들링 시스템의 이해

모든 델파이 클래스는 메시지 핸들링 메소드 혹은 메시지 핸들러라고 불리는 구조를 가지고

있다. 메시지 핸들러의 기본적인 아이디어는 클래스가 어떤 메시지를 받게 되면 이 메시지에 관련된 메소드들 중에서 하나를 호출하여 이 메시지를 처리하는 것이다.

다음의 순서도는 이러한 메시지 처리 시스템을 보여 준다.

이벤트 발생 -> MainWndProc -> WndProc -> Dispatch -> Handler

VCL 은 모든 윈도우 메시지를 번역하는 메시지 처리 시스템을 선언하고 있다. 여러분은 이 메시지 처리 구조를 덧붙일 필요가 없이 단지 메시지 핸들링 메소드를 생성하기만 하면 된다.

* 윈도우 메시지의 구조
윈도우 메시지는 몇 가지의 필드를 가진 데이터 레코드이다. 이 중에 가장 중요한 것은 메시지가 어떤 것인지를 알려 주는 정수형 값이다. 윈도우는 많은 메시지를 선언하고 있고, Messages 유닛에서 모든 메시지의 구분자를 정의하고 있다. 메시지에서 유용한 다른 정보는 두개의 파라미터 필드와 결과 필드이다. 하나의 파라미터는 16 비트의 크기이고 다른 것들은 모두 32 비트이다. 아마도 윈도우의 코드에서 ' word parameter' 와 ' long parameter' 를 뜻하는 wParam 과 lParam 을 포함하는 것을 보았을 것이다. 각각의 파라미터는 하나 이상의 정보를 포함하고 있어서 lParamHi 와 같은 이름을 사용하기도 한다.
예전에는 윈도우 프로그램을 작성하기 위해서 각각의 파라미터가 어떤 값을 갖게 되는지를 기억하고 있거나 윈도우 API 를 찾아보아야 했었다. 그렇지만 요즈음에는 Microsoft 에서 파라미터에 이름을 지었다. 메시지 크랙킹(message cracking)이라고 불리 우는 이 작업은 각 메시지에 어떤 정보가 있는지 이해하는 것을 훨씬 쉽도록 만들었다. 예를 들면, WM_KEYDOWN 메시지에 대한 파라미터의 이름을 nVirtKey 와 lKeyData 라고 지음으로써 wParam 과 lParam 보다 훨씬 이해하기 쉽게 되었다.

메시지의 모든 형에 대해 델파이는 각각의 파라미터의 의미 있는 이름을 줄 수 있도록 레코드 형을 정의해 놓았다. 예를 들어 마우스 메시지가 마우스 이벤트가 발생한 x 와 y 좌표를 lParam 을 통해 하나는 Hi, 다른 하나는 lo 로 전달하면, 개발자는 lParamHi 나 lParamLo 를 이용하는 대신에 XPos 와 YPos 라는 이름으로 파라미터를 받게 되므로 어느 영역에 어떤 값이 들어가는 지를 신경 쓰지 않아도 되는 것이다.

* 메시지 처리
애플리케이션이 윈도우를 생성하면 윈도우 커널을 통해 윈도우 프로시저를 등록한다. 윈도우 프로시저는 윈도우를 위한 메시지를 핸들하기 위한 것으로 윈도우가 다루어야 하는 수많은 메시지를 case 문으로 가지고 있다. 여기서 윈도우라고 함은 윈도우, 컨트롤, 그리고 화

면에 보이는 모든 것을 의미한다. 새로운 형의 윈도우를 생성할 때마다 완전한 윈도우 프로시저가 생성되어야 한다. 델파이는 메시지의 처리를 몇 가지로 간략화 했다.

- 각각의 컴포넌트는 완전한 메시지 처리 시스템을 상속 받는다.
- 처리 시스템은 기본 핸들링을 갖는다. 특별한 응답이 필요한 메시지에 대해서만 따로 정의한다.
- 메시지 핸들링의 아주 작은 부분만 수정할 수 있고 대부분은 상속 받은 메소드를 이용한다.

이러한 메시지 처리 시스템의 가장 큰 이점은 언제든지 어느 컴포넌트에게도 어떤 메시지든 안전하게 보낼 수 있다는 것이다. 컴포넌트가 어떤 메시지를 위한 정의된 핸들러를 가지고 있지 않다면 기본 핸들러가 이 메시지를 처리하는데 보통의 경우 이 메시지는 무시된다.

델파이는 애플리케이션에서 사용되는 각 컴포넌트의 형에 MainWndProc 이라는 윈도우 프로시저 메소드를 등록한다. MainWndProc 은 WndProc 이라는 가상 메소드로 윈도우에서 받은 메시지 구조를 전달하는 부분, 그리고 애플리케이션 클래스의 HandleException 메소드를 호출하여 예외를 처리하는 부분인 예외 처리 블록을 포함하고 있다.

MainWndProc 은 어떤 메시지에 대해서도 특별한 핸들링을 가지지 않는 메소드이고, 각 컴포넌트 형은 WndProc 메소드를 오버라이드하여 필요한 작업을 구현한다. WndProc 메소드는 작업의 처리에 영향을 미칠 수 있는 상태를 확인하여 원치 않는 메시지를 가로챈다. 예를 들면, 컴포넌트가 드래그 되고 있을 때는 키보드 이벤트는 무시된다. 즉, TWinControl 의 WndProc 메소드는 컴포넌트가 드래그 되지 않을 때만 키보드 이벤트를 통과시키는 것이다. 마지막으로 WndProc 은 메시지를 처리하기 위해 어떤 메소드를 사용할 지 결정하는 Dispatch 메소드를 호출한다. Dispatch 메소드는 메시지 구조의 Msg 필드를 사용하여 메시지를 어떻게 처리할 지 결정한다. 컴포넌트가 그 메시지에 대해 핸들러를 선언하였다면 Dispatch 는 그 메소드를 호출한다. 만약 그렇지 않았다면 Dispatch 는 DefaultHandler 를 호출한다.

## 2) 메시지 핸들링 교체

컴포넌트의 메시지 핸들링을 교체하기 전에 어떤 작업을 하고 싶은 지 먼저 확실하게 정하는 것이 좋다. 델파이는 대부분의 윈도우 메시지를 컴포넌트의 개발자와 컴포넌트를 사용하는 사용자가 모두 핸들할 수 있는 이벤트로 바꾼다. 그러므로 메시지 핸들링을 교체하는 대신에 이벤트 핸들링을 교체하는 것이 더 좋다. 메시지 핸들링을 교체하려면 메시지 핸들링 메소드를 오버라이드 한다. 개발자는 특별한 메시지를 가로 채서 메시지를 특정한 환경에서

핸들링하지 않도록 할 수 있다.

* 핸들러 메소드 오버라이딩

컴포넌트가 특정 메시지를 핸들하는 방법을 교체하려면 그 메시지의 핸들링 메소드를 오버라이드한다. 컴포넌트가 특정 메시지를 핸들하지 않는다면, 새로운 메시지 핸들링 메소드를 선언할 필요가 있다. 메시지 핸들링 메소드를 오버라이드 하려면 오버라이드하는 메소드와 같도록 메시지 인덱스를 가지고 새로운 메소드를 선언해야 한다. 이 때, override 지시자를 사용하지 않고 message 지시자를 사용한다. 메소드의 이름이나 하나의 var 파라미터는 오버라이드 된 메소드와 일치할 필요는 없으며 단지 메시지 인덱스만으로도 충분하다. 다음 예제는 WM_PAINT 메시지에 대한 핸들링을 오버라이드 한 것으로 WMPaint 메소드를 다시 선언한 것이다.

```
Type
  TMyComponent = class( ..)
      :
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
  end;
```

* 메시지 파라미터 사용

메시지 핸들링 메소드에서는 메시지 구조의 모든 파라미터에 접근한다. 메시지 핸들러에게 전달된 파라미터가 var 파라미터이기 때문이며, 핸들러는 필요한 경우 파라미터의 값을 변경할 수 있다. 자주 그 값이 변경되는 메시지의 파라미터는 Result 필드인데 이렇게 변경된 값은 메시지를 보내는 SendMessage 를 호출하여 반환한다. 메시지 핸들링 메소드에서 사용되는 message 파라미터의 형이 메시지의 형과 다르므로 윈도우 메시지의 이름과 각 파라미터의 의미를 참조해야 한다. 만약 이전에 사용하던 메시지 파라미터의 이름(wParam, lParam 등)을 사용해야 한다면 Message 를 TMessage 형으로 형 전환하면 사용할 수 있다.

* 메시지 가로채기

어떤 환경에서는 컴포넌트가 메시지를 무시하도록 만들 수 있다. 이것은 컴포넌트가 메시지를 핸들러에게 연결되지 않도록 함으로써 구현이 가능한데, 이를 위해 WndProc 메소드를 오버라이드하여 사용한다. WndProc 메소드는 메시지를 핸들하기 위해 어떤 메소드를 사용할지 결정하는 Dispatch 메소드에 이 메시지를 전달하기 전에 수행된다. 즉, 이 WndProc 메소드를 오버라이드 함으로써 Dispatch 메소드를 수행하기 전에 메시지를 걸러낼 수 있다. WndProc 를 오버라이드하는 형식은 다음과 같다.

```
Procedure TMyControl.WndProc(var Message: TMessage);
```

```
Begin
  //처리를 할 것인가를 결정하는 코드가 들어간다.
  Inherited WndProc(Message);
End;
```

다음의 예는 TControl의 WndProc 메소드의 일부이다.

```
Procedure TControl.WndProc(var Message: TMessage);
Begin
  :
  //메시지의 범위 지정(마우스에 의한 메시지)
 if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then  //드래깅에 대해서는
       DragMouseMsg(TWMMouse(Message))  //특별한 처리를 하고 있다.
     Else  //그 외의 경우에는
       :  //일반적인 처리를 한다.
     end;
:
end;
```

3) 새로운 메시지 핸들러 생성

대부분의 윈도우 메시지에 대해 델파이는 기본적인 핸들러를 제공하므로 여기서는 사용자가 생성한 메시지에 대한 메시지 핸들러를 선언하는 방법을 알아보자.

* 메시지 선언
많은 표준 컴포넌트들이 내부적으로 사용하기 위해 메시지를 선언한다. 메시지의 선언은 두 가지로 구분할 수 있다. 메시지 구분자의 선언과 메시지 레코드 형 선언이 그것이다. 이제 각각에 대해서 알아보도록 하자.

- 메시지 구분자 선언 : 메시지 구분자는 정수형의 상수이다. 윈도우는 1024 이하의 영역을 자신의 메시지를 위해 가지고 있기 때문에 사용자 메시지를 선언하는 경우에는 이 보다 큰 수를 메시지 구분자로 이용해야 한다. WM_APP 상수는 사용자 정의 메시지가 사용할 수 있는 가장 작은 수를 나타낸다. 그러므로 새로운 메시지 구분자를 선언하는 경우에는 WM_APP 를 기본으로 해야 한다. 그러나 몇몇 표준 윈도우 컨트롤은 이 사용자 정의 영역의 상수를 메시지 구분자로 사용하는 메시지를 사용한다. 이러한 컨트롤로는 TListBox, TComboBox, TEdit, 그리고 TButton 등이 있다. 만약 이러한 컴포넌트로부터 상속 받는 컴포넌트를 생성하는 경우이고 이 컴포넌트를 위해 새로운 메시지를 선언하고자 한다면, 먼저 어떤 메시지가 선언되어 있는지 확인할 필요가 있다. 다음의 코드는 사용자 정의 메시지의 메시지 구분자를 선언하는 예를 보여 준다.

```
Const
  WM_MYFIRSTMESSAGE = WM_APP + 400;
  WM_MYSECONDMESSAGE = WM_APP + 401;
```

- 메시지 레코드 형 선언 : 메시지의 파라미터에 기본적으로 주어진 이름 대신에 의미가 있는 이름을 새롭게 주고 싶다면 메시지 레코드 형을 선언해야 한다. 메시지 레코드는 메시지 핸들링 메소드로 전달되는 파라미터의 형이다. 만약 파라미터를 사용하지 않거나, 이전에 사용하던 파라미터의 이름을 사용하려면 기본 메시지 레코드인 TMessage 를 사용하면 된다. 메시지 레코드 형은 편의를 위해 다음과 같이 선언한다.

1. 레코드 형의 이름은 메시지의 이름의 앞에 T를 붙인다.
2. 첫번째 필드로 TMsgParam 형인 Msg 를 선언한다.
3. 2 바이트를 word 형 파라미터로 선언하고 나머지 2 바이트를 사용하지 않거나, 4 바이트를 Longint 파라미터로 선언한다.
4. 마지막으로 Result 라고 불리는 Longint 형의 필드를 선언한다.

다음의 예는 모든 마우스 메시지에 대한 메시지 레코드를 보여 준다.

```
Type
  TWMMouse = record
    Msg: TMsgParam; //메시지 구분자
    Keys: Word; //wParam
    Case Integer of //Result 필드로 두 가지 형식을 사용한다.
      0:  XPos: Smallint;
          YPos: Smallint);
      1: (
          Pos: TSmallPoint;
          Result: Longint);
  end;
```

* 새로운 메시지 핸들링 메소드 선언

대개의 경우 다음의 두 가지 상황에서는 새로운 메시지 핸들링 메소드를 정의하게 된다.

- 표준 컴포넌트에 의해 핸들 되지 않는 윈도우 메시지를 핸들할 필요가 있을 때
- 컴포넌트에서 사용자 정의 메시지를 선언한 경우

메시지 핸들링 메소드를 선언하는 방법은 다음과 같다.

1> 클래스 선언의 protected 부분에 메소드를 선언한다.

2> 메소드는 프로시저로 한다.

3> 메소드의 이름은 핸들할 메시지의 이름으로 한다. 단지 언더라인 문자는 삭제한다.

4> 메시지 레코드 형인 Message 라는 var 파라미터를 하나 사용한다.

5> 메소드의 구현 부분에 원하는 코드를 작성한다.

6> inherited 를 추가한다.

사용자가 CM_CHANGECOLOR 이라는 메시지를 선언한 경우 이 메시지의 핸들러는 아래의 예와 같은 모습을 갖게 될 것이다.

```
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;
```

4.6 설계 시에 컴포넌트를 사용 가능하게 만들자.

이제 컴포넌트를 IDE 에서 사용이 가능하도록 하는 방법에 대해서 알아보도록 한다.

1) 컴포넌트 등록

등록 작업은 유닛을 컴파일하는 동안 이루어지게 되므로 만약 하나의 유닛에서 여러 컴포넌

트를 생성한다면 그 컴포넌트를 한 번에 모두 등록할 수 있다. 컴포넌트를 등록하기 위해서는 Register 프로시저를 추가하는데, 이 프로시저는 컴포넌트를 등록하고 어떤 컴포넌트 팔레트에 인스톨할 것인지를 결정한다.

2) 팔레트 비트맵 추가

모든 컴포넌트는 컴포넌트 팔레트에 표시될 비트맵을 필요로 한다. 만약 이 비트맵을 정의하지 않는다면 델파이는 기본 비트맵을 사용한다. 팔레트 비트맵은 컴포넌트 설계 시에만 필요하므로 유닛에 포함하여 컴파일 할 필요는 없다. 대신에 유닛과 같은 이름으로 .DCR(Dynamic Component Resource)확장자를 갖는 윈도우 리소스 파일로 제공하면 된다. 이 리소스 파일은 델파이의 이미지 에디터로 생성할 수 있다. 이 비트맵은 24 픽셀 크기를 가져야 한다. 인스톨하고자 하는 각 컴포넌트에 대해 팔레트 비트맵 파일을 가져야 하고, 각 팔레트 비트맵 파일은 각 컴포넌트를 위한 비트맵을 가져야 한다. 비트맵 이미지는 컴포넌트와 같은 이름을 갖고, 델파이가 컴포넌트를 컴포넌트 팔레트에 인스톨할 때 비트맵을 찾을 수 있도록 팔레트 비트맵 파일은 컴파일 된 파일과 같은 디렉토리에 있어야 한다. 예를 들어 TMyControl 이라는 컴포넌트를 ToolBox 라는 이름의 유닛에 생성했다면 리소스 파일의 이름은 TOOLBOX.DCR 이라는 이름을 가져야 하며, 이 리소스 파일에 포함된 비트맵은 TMYCONTROL 이라는 이름을 가져야 한다. 리소스의 이름은 대소문자를 가리지는 않지만 편의를 위해서 대문자로 작성하는 것이 보통이다.

3) 컴포넌트에 도움말 추가

폼에서 표준 컴포넌트를 선택하거나 오브젝트 인스펙터에서 속성 혹은 이벤트를 선택하는 경우에 F1 키를 누르면 도움말을 볼 수 있다. 컴포넌트를 생성하는 경우에도 이 도움말 파일을 생성하면 컴포넌트가 이러한 기능을 가지도록 할 수 있다. 우선 윈도우 도움말 파일의 소스 파일(.RTF)을 생성해야 하는데, 이 때는 사용할 수 있는 툴은 어느 것이나 사용해도 된다. 델파이는 Microsoft Help Workshop 이라는 툴을 포함하고 있어서, 도움말 파일을 컴파일 할 수 있도록 하고 있다. 이 Help Workshop 의 도움말을 이용하여 도움말 파일을 생성하는 자세한 정보를 얻을 수 있다.

4) 속성 에디터 추가

오브젝트 인스펙터는 모든 형의 속성을 위한 기본 에디터를 제공한다. 그러나 속성 에디터를 사용하면 특정 속성에 대한 에디터를 제공할 수 있다. 이 때 새로 생성한 컴포넌트의 속성에만 적용되는 속성 에디터 뿐만 아니라 특정 형의 속성에 모두 적용되는 에디터를 생성할 수도 있다.

* 속성 에디터 클래스 상속

DsgnIntf.pas 파일에는 여러 가지 종류의 속성 에디터를 선언하고 있다. 이것들은 모두 TPropertyEditor 에서 상속 받은 것으로 속성 에디터를 생성할 때, TPropertyEditor 에서 직접

상속을 받거나 [표 4]에서 볼 수 있는 클래스로부터 상속을 받아 생성한다.

| 형 | 설명 |
|---|---|
| TOrdinalProperty | TOrdinalProperty 에서 상속 받은 모든 속성 에디터 |
| TIntegerProperty | 모든 정수형 |
| TCharProperty | Char 형 |
| TEnumProperty | 모든 열거형 |
| TFloatProperty | 모든 실수형 |
| TStringProperty | 문자열 |
| TSetElementProperty | 부울형으로 나타내는 집합형의 구성 요소 |
| TSetProperty | 모든 집합형 |
| TClassProperty | 클래스 |
| TMethodProperty | 메소드 포인터(이벤트) |
| TComponentProperty | 같은 폼 내의 컴포넌트 |
| TColorProperty | 컴포넌트 색 |
| TFontNameProperty | 폰트 이름 |
| TFontProperty | 폰트 |

[표 4] 속성 에디터 형

다음의 예는 DDEREG.PAS 의 코드 중 일부로 TPropertyEditor 로부터 상속 받은 속성 에디터이다.

```
type
  TDdeLinkInfoProperty = class(TPropertyEditor)
  public
    function GetAttributes: TPropertyAttributes; override;
     procedure Edit; override;
    function GetValue: string; override;
  end;
```

* 속성을 문자열처럼 수정

모든 속성은 그들의 실제 값을 오브젝트 인스펙터 상에 나타낼 때 문자열처럼 보여 줄 수 있어야 한다. 대부분의 속성은 그 속성에 새로운 값을 사용자가 타이핑으로 입력할 수 있도록 한다. 그러므로 속성 에디터 클래스는 그들의 실제 값과 문자열 형태의 전환을 가능하도록 오버라이드하여 사용할 수 있는 가상 메소드를 제공한다. 이 메소드들의 이름은 GetValue 와 SetValue 이다. 새롭게 생성하는 속성 에디터도 값을 할당하고 읽을 수 있도록

메소드를 상속 받아야 한다.[표 5]

| 속성 형 | 읽기 메소드 | 쓰기 메소드 |
|---|---|---|
| 실수형 | GetFloatValue | SetFloatValue |
| 메소드 포인터(이벤트) | GetMethodValue | SetMethodValue |
| 서수형 | GetOrdValue | SetOrdValue |
| 문자열 | GetStrValue | SetStrValue |

[표 5] 속성 값을 읽고 쓰는 메소드들

GetValue 메소드를 오버라이드하게 되면 [표 5]의 읽기 메소드 중 하나를 호출하게 되고, SetValue 를 오버라이드하면 쓰기 메소드 중 하나를 호출한다.

- 속성 에디터의 GetValue 메소드는 속성의 현재 값을 나타내는 문자열을 반환한다. 오브젝트 인스펙터는 이 문자열 값을 속성의 값을 나타내는데 사용한다. 기본적으로 GetValue 는 unknown 을 반환한다.
- 속성이 그 값의 문자열 표현방식을 지원하기 위해서는 GetValue 메소드를 오버라이드 하여 사용해야 한다.
- 만약 속성의 값이 문자열 형이 아니라면 GetValue 는 그 값을 문자열 형으로 변환하는 기능을 가져야 한다.
- 속성 에디터의 SetValue 메소드는 오브젝트 인스펙터에 사용자가 타이핑한 값을 가져온다. 그리고 이 값을 적당한 값으로 변환하여 그 속성의 값으로 설정하는 것이다. 만약 사용자가 입력한 문자열 값이 적당한 값이 아니라면 SetValue 메소드는 예외 처리를 하고 그 값을 사용하지 않아야 한다.
- 문자열 값을 입력 받아 속성의 값으로 설정하려면, SetValue 메소드를 오버라이드 하여 사용한다.
- SetValue 메소드는 위에 설명한 쓰기 메소드 중 하나를 호출하기 전에 먼저 문자열 값을 적정한 값으로 변환해야 한다.

이제 여기서 TIntegerProperty 를 위한 SetValue 와 GetValue 메소드를 살펴보자. 정수형은 서수형이므로 GetValue 메소드는 GetOrdValue 형을 호출하고 그 값을 문자열형으로 변환한다. SetValue 는 문자열을 정수형으로 변환하여 그 값이 적절한지 확인하고 SetOrdValue 메소드를 호출한다.

```
Function TIntegerProperty.GetValue: string;
Begin
  Result := IntToStr(GetOrdValue);
```

```
End;

Procedure TIntegerProperty.SetValue(const Value: string);
Var
  L: Longint;
Begin
  L := StrToInt(Value);
  With GetTypeData(GetpropType)^ do
    If (L < MinValue) or (L > MaxValue) then
        Raise EPropertyError.Create(FmtLoadStr(SOutOfRange, [MinValue, MaxValue]));
   SetOrdValue(L);
End;
```

*** 속성을 전체적으로 변경**

**사용자가 비주얼 하게 속성을 변경할 수 있도록 대화 상자를 제공할 수 있다. 보통은 그 속성 자체가 클래스인 경우에 많이 사용하게 되는데 그 대표적인 예가 Font 속성이다. 이러한 대화 상자를 제공은 속성 에디터 클래스의 Edit 메소드를 오버라이드 함으로써 가능하다. Edit 메소드는 GetValue 와 SetValue 메소드에서 사용하는 읽기, 쓰기 메소드를 역시 사용한다. 이 Edit 메소드는 읽기, 쓰기 메소드를 함께 사용한다. 사용자가 '...' 버튼을 누르거나 속성의 값이 나타나는 부분을 더블 클릭하게 되면, 오브젝트 인스펙터는 속성 에디터의 Edit 메소드를 호출한다. 다음 예제는 TColorProperty 의 Edit 메소드로 윈도우의 색상 대화 상자를 사용하고 있다.**

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
Begin
  ColorDialog := TColorDialog.Create(Application);
   Try
     ColorDialog.Color := GetOrdValue;
     If ColorDialog.Execute then
         SetOrdValue(ColorDialog.Color);
   Finally
      ColorDialog.Free;
   End;
End;
```

* 특정한 에디터 특성

속성 에디터는 오브젝트 인스펙터가 어떻게 속성을 나타낼지 알 수 있도록 정보를 제공해야 한다. 예를 들면, 오브젝트 인스펙터는 속성이 부가적인 프로퍼티를 갖는지 혹은 사용 가능한 값의 리스트를 갖는지 등을 알아야 한다. 이러한 에디터 속성을 정의하려면 속성 에디터의 GetAttributes 메소드를 오버라이드 하면 된다. GetAttributes 메소드는 TPropertyAttributes 형인 값의 집합을 반환한다. TPropertyAttributes 형의 값은 [표 6]에서 확인할 수 있다.

| 플래그 | 관련된 메소드 | 의미 |
|---|---|---|
| paValueList | GetValues | 열거형 값의 리스트를 제공한다. |
| paSubProperties | GetProperties | 부가적인 속성을 가지고 있다. |
| padialog | Edit | 속성을 전체적으로 변경하기 위한 대화 상자를 갖는다. |
| paMultiSelect | 없음 | 하나 이상의 컴포넌트가 선택되면 보여진다. |
| paAutoUpdate | SetValue | 변화가 일어나면 컴포넌트를 바로 갱신한다. |
| paSortList | 없음 | 값의 리스트를 정렬한다. |
| paReadOnly | 없음 | 속성 값을 사용자가 변경할 수 없다. |
| paRevertable | 없음 | 속성의 현재 값은 무시하고 기본 혹은 표준 값으로 전환하는 것을 가능하게 한다. |

[표 6] 속성 에디터 특성 플래그

Color 속성은 가장 융통성 있는 입력 방법을 제공하는데, 타이핑, 리스트에서 선택, 그리고 에디터를 통해 가능하다. TColorProperty 의 GetAttributes 메소드는 다음의 예와 같은 값을 반환한다.

```
Function TColorProperty.GetAttributes: TPropertyAttributes;
Begin
  Result := [paMultiSelect, paDialog, paValueList];
End;
```

* 속성 에디터 등록

속성 에디터를 생성하면 델파이에 등록해야 사용할 수 있다. 개발자는 생성된 에디터를 특정 속성 형의 모든 속성에 사용하도록 등록하거나 특정 형인 컴포넌트의 특정 속성에만 사용하도록 등록하는 것이 가능하다. 속성 에디터를 등록하려면 RegisterpropertyEditor 프로시저를 호출한다.

RegisterPropertyEditor 프로시저는 4 개의 파라미터를 갖는다.

- 변경할 속성 형을 위한 형 정보 포인터
  - 항상 내장 함수인 TypeInfo를 사용한다.
  - TypeInfo(TMyComponent)
- 에디터가 적용 될 컴포넌트의 형. 이 값이 nil 이면 주어진 형의 모든 속성에 적용된다.
- 속성의 이름.
  - 이 파라미터는 앞의 파라미터가 특정한 컴포넌트의 형을 가진 경우에만 적용된다.
- 특정한 속성을 변경하기 위한 속성 에디터의 형.

다음은 Register 프로시저에서 RegisterpropertyEditor 를 사용한 예를 보여 준다.

```
Procedure Register;
Begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '' , TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), Tcomponent, ' Name' ,
                              TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TmenuItem), TMenu, '', TmenuItemProperty);
End;
```

이 프로시저에서는 RegisterPropertyEditor 의 세 가지의 사용 방법을 보여 주고 있다.

- 첫번째 문장이 가장 보편적으로 사용되는 방법이다. TComponent 형의 모든 속성을 위한 TComponentProperty 속성 에디터를 등록한다. 속성 에디터를 등록하는 경우, 특정한 형의 모든 속성에 사용하기 위해 속성 에디터를 등록하는 경우가 대부분이므로 그 형을 위한 에디터를 생성하고 등록할 때 RegisterPropertyEditor 의 두번째와 세번째 파라미터를 nil과 값이 없는 문자열을 사용하면 된다.
- 두번째 문장은 가장 사용이 드문 형태의 등록 작업으로 특정한 형인 컴포넌트의 특정 속성에 적용되는 속성 에디터를 등록한다. 위의 예에서는 모든 컴포넌트의 Name속성을 위한 에디터를 등록한다.
- 세번째 문장은 위의 두 문장의 중간 정도의 제한을 가지고 있다. TMenu 형인 컴포넌트의 TMenuItem 형인 모든 속성에 대한 에디터를 등록한다.

5) 컴포넌트 에디터 추가

컴포넌트 에디터는 설계 시에 컴포넌트를 더블 클릭한 경우 어떤 작업을 할 지를 결정하고

컴포넌트에서 오른쪽 마우스 버튼이 눌렸을 경우 나오게 되는 메뉴에 명령어를 추가한다. 또한 컴포넌트 에디터는 컴포넌트를 윈도우 클립보드에 복사하는 기능도 갖는다.

만약 컴포넌트에 컴포넌트 에디터를 추가하지 않으면, 델파이는 기본 컴포넌트 에디터를 사용한다. 기본 컴포넌트 에디터는 TDefaultEditor 클래스에 의해 구현되는데, 이 TDefaultEditor 클래스는 컴포넌트의 메뉴에 새로운 아이템을 추가하지 않고 컴포넌트를 더블 클릭하면 컴포넌트의 속성을 검색하여 처음으로 나오는 이벤트 핸들러를 생성한다. 예를 들어, 버튼을 폼에 추가하고 버튼을 더블 클릭해 보면 버튼의 클릭에 대한 이벤트 핸들러를 생성한다.

메뉴에 새로운 아이템을 추가하거나, 더블 클릭 되었을 때 행동을 변경하려면, 또는 새로운 클립보드 형식을 추가하려면 TComponentEditor 에서 새로운 클래스를 생성하고 이 클래스를 컴포넌트와 함께 등록해야 한다. 오버라이드 된 메소드에서는 편집될 컴포넌트에 접근하기 위해 TComponentEditor 의 Component 속성을 사용할 수 있다.

* 메뉴에 아이템 추가
사용자가 컴포넌트에서 오른쪽 마우스 버튼을 누르면 GetVerbCount 와 GetVerb 메소드가 메뉴를 구성하기 위해 호출된다. 그러므로 이 메소드를 오버라이드하면 원하는 메뉴를 추가할 수 있다. GetVerbCount 메소드는 메뉴에 추가한 명령어의 수를 돌려주기 위해 오버라이드한다. GetVerb 메소드는 이 명령어에 추가될 문자열을 돌려준다. GetVerb 를 오버라이드 할 때는 메뉴 아이템을 선택하기 위한 shortcut 키로 사용하기 위해 '&' 를 사용하는데, '&' 문자에 바로 이어져서 나오는 문자에는 밑줄이 그어지고 그 문자가 shortcut 키가 된다. 만약 메뉴를 선택했을 때 대화 상자가 표시되는 경우에는 '…' 를 추가하도록 한다. GetVerb 는 명령어의 인덱스를 나타내는 파라미터를 하나 사용한다. 다음의 예제는 GetVerbCount 와 GetVerb 메소드를 오버라이드 하여 두 개의 명령어를 추가한 것이다.

```
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;
```

이 때, GetVerbCount 에서 돌려준 값만큼 GetVerb 에서는 그 값을 돌려줘야 한다. 위의 예에서 보면 GetVerbCount 에서 결과로 2 를 돌려주었기 때문에 GetVerb 메소드에서 보면 두 가지의 결과 값이 나와야 하는 것이다.

GetVerb 에서 돌려주는 명령어가 설계 시에 선택된 경우에는 ExecuteVerb 메소드가 호출된다. GetVerb 에서 돌려주는 명령어들에 대해서는 ExecuteVerb 메소드에서 각각을 구현한다.

다음의 ExecuteVerb 메소드는 위의 예에서 나온 명령어들에 대해 구현한 것이다.

```
Procedure TMyEditor.ExecuteVerb(Index: Integer);
Var
  MySpecialDialog: TMyDialog;
Begin
  Case Index of
    0: begin
         MySpecialDialog := TMyDialog.Create(Application);   //대화 상자 생성
         If MySpecialDialog.Execute then   //OK 버튼이 눌렸을 경우 속성에 값을 할당
           MyComponent.FThisProperty := MySpecialDialog.ReturnValue;
          MySpecialDialog.Free;
       End;
    1: That;   //That 메소드 호출
  end;
end;
```

이제 컴포넌트를 더블 클릭 하였을 때 어떤 동작을 할지 결정해보자. 컴포넌트가 더블 클릭되면 컴포넌트 에디터의 Edit 메소드가 호출된다. 기본적으로 Edit 메소드는 메뉴의 첫번째 명령어를 실행한다. 그러므로 위의 예에서는 DoThis 명령어가 실행된다. 만약 이러한 기본적인 동작을 바꾸고 싶다면 어떻게 할까? 예를 들어 메뉴에 명령어를 추가하지 않았다거나, 몇 개의 명령어들을 포함하고 있는 대화 상자를 표시하고 싶다면 이와 같은 기본적인 동작을 바꿔야만 할 것이다. Edit 메소드를 오버라이드하면 새로운 동작을 정의할 수 있는데, 다음의 예는 컴포넌트가 더블 클릭 되면 Font 대화 상자를 표시하도록 하고 있다.

```
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;
```

만약 컴포넌트를 더블 클릭 했을 때 코드 에디터의 이벤트 핸들러를 보여 주도록 하려면 TComponentEditor 대신에 컴포넌트 에디터의 기본 클래스로 TDefaultEditor 를 사용해야 한다. 그리고 Edit 메소드를 오버라이드 하는 대신에 TDefaultEditor.EditProperty 메소드를 오버라이드하여 사용한다. EditProperty 는 컴포넌트의 이벤트 핸들러를 검색하여 첫번째 발견한 이벤트 핸들러를 보여 준다. 이 때 특정 이벤트를 설정하면 그 이벤트 핸들러를 대신 보여 주도록 할 수 있다.

```
Procedure TMyEditor.Editproperty(PropertyEditor: TPropertyEditor;
                                          Continue,  FreeEditor:  Boolean)
Begin
  If (PropertyEditor.ClassName = ‘ TMethodProperty’ ) and
      (PropertyEditor.GetName = ‘ OnSpecialEvent’ ) then
       DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
End;
```

다음에는 클립보드 형식을 추가해 보자. 기본적으로는 사용자가 IDE 에서 컴포넌트가 선택되어 있을 때 복사 명령어를 선택하면 컴포넌트는 델파이의 내부 형식으로 복사된다. 이렇게 복사된 후에는 다른 폼이나 데이터 모듈로 붙여 넣기가 가능하다. Copy 메소드를 오버라이드 하면 추가된 클립보드 형식을 사용할 수 있다. 예를 들어 다음의 Copy 메소드는 TImage 컴포넌트가 picture 속성의 값을 클립보드로 복사할 수 있도록 한다.

```
Procedure TMyComponent.Copy;
Var
  MyFormat: Word;
  AData, APalette: THandle;
Begin
  TImage(Component).Picture.Bitmap.SaveToClipBoardFormat(MyFormat, AData, APalette);
  ClipBoard.SetAsHandle(MyFormat, AData);
End;
```

이제 컴포넌트 에디터의 모든 것을 정의했다. 이렇게 만든 컴포넌트 에디터를 등록하는 작업만이 남은 것이다. 컴포넌트 에디터를 특정 컴포넌트 클래스에 대해 등록하지 않으면 사용할 수 없다. 이렇게 등록된 컴포넌트 에디터는 폼에 있는 그 클래스의 컴포넌트가 선택되었을 때 생성된다. 컴포넌트 클래스와 컴포넌트 에디터를 연결하는 작업은 RegisterComponentEditor 를 호출함으로써 이루어진다. RegisterComponentEditor 는 에디터를 사용할 컴포넌트 클래스의 이름과 선언한 컴포넌트 에디터 클래스의 이름을 갖는다.

다음의 예는 TMyComponent 형의 모든 컴포넌트와 사용할 TMyEditor 라는 컴포넌트 에디터를 등록한다.

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

RegisterComponentEditor 는 Register 프로시저에서 호출하도록 한다. 예를 들어 TMyComponent 라는 컴포넌트와 이 컴포넌트를 위한 컴포넌트 에디터인 TMyEditor 가 하나의 유닛에 구현되어 있다면 다음의 예제 코드는 컴포넌트와 컴포넌트 에디터를 등록할 것이다.

```
Procedure Register;
Begin
   RegisterComponents(' Miscellaneous' , [TMyComponent]);
  RegisterComponentEditor(classes[0], TMyEditor);
End;
```

6) 패키지의 컴포넌트 컴파일

컴포넌트가 등록되면 컴포넌트가 IDE 에 인스톨 되도록 패키지로 컴파일 되어야 한다. 이 과정은 델파이의 도움말을 참조하기 바란다.

## 5. 데이터베이스 관련 컨트롤 생성

데이터베이스 연결을 이용하여 작업하는 경우에는 관련 컨트롤을 가지고 작업하는 것이 더 편리하다. 애플리케이션은 컨트롤과 데이터베이스의 한 부분 사이에 연결을 갖는 것이다. 델파이는 여러 종류의 데이터베이스 관련 컴포넌트를 가지고 있는데, 여기에 사용자가 새로운 컴포넌트를 생성하여 사용할 수 있다.

이러한 데이터베이스 관련 컨트롤을 모두 몇 가지 종류로 나누어 생각할 수 있는데, 먼저 데이터를 읽기만 할 수 있는 컨트롤을 생각할 수 있다. 다음은 반대로 편집이 가능한 컨트롤이다. 이러한 분류는 가장 간단한 분류일 것이고, 이 외에도 다양하게 분류하는 것이 가능하다.

여기서는 먼저 가장 간단한 컨트롤인 읽기 전용 컨트롤, 즉 데이터를 보기 위한 컴포넌트부터 생성해 보도록 한다. 이용하게 되는 컨트롤은 TCalendar 로써 이 컨트롤을 데이터베이스 관련 컨트롤로 재작성하게 된다.

### 5.1 데이터 읽기 전용 컴포넌트

작성하게 될 컴포넌트가 읽기 전용이건, 아니면 데이터를 변경할 수 있던 간에 다음과 같은 순서를 따르게 된다.

- 컴포넌트 생성 및 등록
- 데이터 연결 추가
- 데이터 변경에 대한 응답

1) 컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 같은 방법에서 출발한다. 그 방법은 앞에서 이미 설명했기 때문에 여기서는 우리가 만들 컴포넌트에 직접 적용하여 알아보도록 하자. 먼저 컴포넌트를 생성하기 위한 기본 골격이 있다고 설명했었다. 그 기본 골격은 델파이의 Component|New Component 를 이용하여 생성해 보자. 우리가 상속 받을 컨트롤은 TCalendar 이다. 이 작업이 끝나면 나타나는 코드는 다음과 같을 것이다.

```
unit DBCalendar;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Grids, Calendar;

type
  TDBCalendar = class(TCalendar)
   private
    { Private declarations }
   protected
    { Protected declarations }
   public
    { Public declarations }
   published
    { Published declarations }
   end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TDBCalendar]);
end;

end.
```

[코드 3] TDBCalendar의 골격 코드

2) 읽기 전용 컨트롤로 만들기

이 컨트롤이 데이터베이스를 읽기 전용으로 사용하도록 만들려면 가장 간단하고 확실한 방법은 컨트롤 자체를 읽기 전용으로 만드는 것이다. 컨트롤을 읽기 전용으로 만들려면 ReadOnly 속성을 추가하고 필요한 갱신은 가능하도록 해야 한다.

* ReadOnly 속성 추가
TCalendar 는 이미 ReadOnly 속성을 가지고 있기 때문에 새롭게 생성할 필요는 없다. 여기서는 간단하게 ReadOnly 속성을 추가하는 예를 보도록 하자.

```
Type
  TDBCalendar = class(TCalendar)
   Private
     FReadOnly: Boolean;
   Public
    Constructor Create(AOwner: TComponent); override;
   Published
    Property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
   End;
:
constructor TDBCalendar.Create(AOwner: TComponent);
begin
   inherited Create(AOwner);
  FReadOnly := True;
End;
```

이제 이렇게 생성한 ReadOnly 속성을 이용하여 데이터를 편집하거나 추가하는 작업을 못하도록 설정하기만 하면 된다.
* 필요한 갱신은 가능하도록 한다
여기서 생성하게 될 TDBCalendar 는 컴포넌트와 관련된 모든 변경 내용에 대해 Row 와 Col, 그리고 SelectCell 메소드를 사용한다. UpdateCalendar 메소드가 Row 와 Col 속성을 날짜가 변하게 되면 다시 설정하게 되지만, SelectCell 메소드가 변경을 막으면 실제로 달력에서는 날짜가 바뀌지 않게 된다. 이러한 문제점을 예방하기 위해서는 특정 부울형 플래그를 선언하여 그 플래그 값이 True 이면 변경이 가능하도록 해야 한다.

```
Type
  TDBCalendar = class(TCalendar)
   Private
     FUpdating: Boolean;
   Protected
    Function SelectCell(ACol, ARow: Longint): Boolean; override;
   Public
```

```
      Procedure UpdateCalendar; override;
    End;
:
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if (not FUpdating) and FReadOnly then Result := False
  else Result := inherited SelectCell(ACol, ARow);
end;

procedure TDBCalendar.UpdateCalendar;
begin
  FUpdating := True;
   Try
      Inherited UpdateCalendar;
   Finally
     FUpdating := False;
   End;
End;
```

이렇게 되면 사용자에 의한 변경은 계속 허용되지 않지만, 날짜 속성을 변경하면 그 값이 컴포넌트에 적용된다. 이렇게 되면 진정으로 읽기 전용 달력이 만들어진 것이고, 이제 데이터베이스와 연결만 하면 컨트롤이 완성된다.

3) 데이터 연결 추가

컨트롤과 데이터베이스와의 연결은 datalink 라는 클래스로 이루어진다. 하나의 필드와 컨트롤을 연결하는 데이터 연결을 위한 클래스는 TFieldDataLink 이다. 또한 테이블 전체와 연결하는 클래스도 있다. 데이터베이스 관련 컨트롤은 그 자체가 데이터 연결 클래스를 가지고 있다. 즉, 컨트롤은 직접 데이터 연결을 생성 및 소멸시키는 것이다.

* 클래스 필드 선언
달력에 데이터 연결을 위한 필드를 선언해 보자.

```
Type
  TDBCalendar = class(TCalendar)
   Private
```

```
    FDataLink: TFieldDataLink;
  :
  end;
```

한 가지 주의할 점은 애플리케이션을 컴파일하기 전에 uses 절에 DB 와 DBCtrls 를 추가해야 한다는 것이다.

* 접근 속성 선언
여기서는 우리가 만들고 있는 TDBCalendar 의 예를 통해 알아보도록 하자.

```
Type
  TDBCalendar = class(TCalendar)
  Private
     …
    function GetDataField: string;
    function GetDataSource: TDataSource;
    procedure SetDataField(const Value: string);
    procedure SetDataSource(Value: TDataSource);
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
:
function TDBCalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
End;

Function TDBCalendar.GetDataSource: TDataSource;
Begin
  Result := FDataLink.DataSource;
End;

Procedure TDBCalendar.SetDataField(const Value: string);
Begin
  FDataLink.FieldName := Value;
```

```
End;

Procedure TDBCalendar.SetDataSource(Value: TDataSource);
Begin
  FDataLink.DataSource := Value;
End;
```

앞의 예에서는 DataField 와 DataSource 속성을 선언하고 메소드를 선언하고 있다. 예에서도 볼 수 있겠지만, 이 두 속성은 FDataLink 의 FieldName 과 DataSource 의 값으로 할당된다. 이렇게 하여 여러분은 이 컨트롤과 데이터 연결간의 설정을 마쳤다. 그렇다면 이 데이터 연결은 어디서 생성해야 할까? 물론 컨트롤이 생성될 때 같이 생성되도록 해야 할 것이다. 그렇다면 이제 데이터 연결을 생성, 그리고 소멸 시키는 코드를 추가해 보자.

* 데이터 연결 초기화

여기서도 직접 예제를 보면서 이해해 보도록 하자. 데이터 연결을 초기화하기 위해서는 Create 와 Destroy 메소드를 오버라이드하여 구현한다.

```
Type
  TDBCalendar = class(TCalendar)
   Public
      Constructor Create(AOwner: TComponent); override;
      Destructor Destroy; override;
       :
   end;
:
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  FDataLink := TFieldDataLink.Create; //FDataLink 생성
   Inherited Create(AOwner);
  FReadOnly := True;
End;

Destructor TDBCalendar.Destroy;
Begin
  FDataLink.Free; //FDataLink 소멸
   Inherited Destroy;
```

```
End;
```

이제 컨트롤은 완전한 데이터 연결을 갖게 되었다. 그러나 이 컴포넌트에서는 아직 구현되지 않은 부분이 있다. 바로 위와 같이 데이터 연결을 갖게 되어 그 해당 데이터를 보여 줄 수 있어야 하는데 이 컴포넌트는 아직 구현되어 있지 않다. 그럼 그 데이터가 어떤 값인지를 컴포넌트가 나타낼 수 있도록 구현해 보자.

4) 데이터 변경에 대한 응답

컨트롤이 데이터 소스나 데이터 필드에 대한 값을 가지게 되어 데이터 연결이 이루어진 후 필드의 데이터가 변경되면 함께 변경되어야 한다. 데이터 연결 클래스는 OnDataChange 라는 이벤트를 가지고 있다. 데이터 소스가 데이터의 변경을 인지하면, 데이터 연결 객체는 OnDataChange 이벤트에 연결된 이벤트 핸들러를 호출한다. 즉, 컨트롤이 데이터의 변경에 대해 적당한 응답을 하게 하려면 OnDataChange 이벤트에 핸들러를 작성해 주면 된다. 우리가 작성하는 컨트롤도 이러한 핸들러를 작성해서 OnDataChange 이벤트에 연결해 주게 된다. 여기서 우리가 작성하는 핸들러를 이벤트에 할당하는 작업은 생성자에서 이루어진다. 역시 소멸자에서는 객체를 소멸하기 전에 이벤트로부터 핸들러의 연결을 해제하게 된다.

```
Type
  TDBCalendar = class(TCalendar)
   Private
     Procedure DataChange(Sender: TObject);
   End;
:
constructor TDBCalendar.Create(AOwner: TComponent);
begin
   inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange; //핸들러 할당
End;

Destructor TDBCalendar.Destroy;
Begin
   FDataLink.OnDataChange := nil; //핸들러 할당 삭제
   FDataLink.Free;
  Inherited Destroy;
End;

Procedure TDBCalendar.DataChange(Sender: TObject);
Begin
  If FDataLink.Field = nil then
     CalendarDate := 0
  Else CalendarDate := FDataLink.Field.AsDateTime;
End;
```

이제 여러분은 데이터베이스에서 사용할 수 있는 읽기 전용 달력을 완성하였다.

## 5.2 데이터 변경 가능 컨트롤 생성

데이터 변경이 가능한 컨트롤을 생성할 때 컴포넌트를 생성, 등록 그리고 데이터 연결을 추가하는 작업들은 앞에서 설명한 읽기 전용 컨트롤을 생성하는 것과 동일하다. 그러나 데이터 변경이 가능한 컨트롤을 생성하려면 필드에 변경된 데이터를 적용하는 부분을 추가해야 하고, 이 외에도 몇 가지 더 신경을 써야 할 부분이 있다. 이제 앞에서 만든 달력을 읽기 전용이 아니라, 데이터를 직접 변경할 수 있는 컨트롤로 수정해 보자.

### 1) FReadOnly 의 기본 값 변경

이제부터 만들 컨트롤은 데이터를 변경할 수 있는 컨트롤이므로 ReadOnly 속성의 기본 값은 False 로 선언 되야 한다. ReadOnly 속성의 값을 False 로 선언하려면 생성자에서 그 값을 변경하면 된다.

```
Constructor TDBCalendar.Create(AOwner: TComponent);
Begin
   :
   FReadOnly := False;
   :
end;
```

### 2) Mouse-Down 과 Key-Down 메시지 핸들링

데이터의 변경이 가능하다는 것은 데이터가 변경되려면 사용자로부터 입력을 받게 되므로

Mouse-Down(WM_LBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONDOWN)과 Key-Down(WM_KEYDOWN)메시지를 처리해야 한다는 것을 의미한다. 그러므로 여러분은 이 두 가지의 메시지를 처리하는 핸들러를 정의해야 한다.

* Mouse-Down 메시지에 응답

다음의 예로써 Mouse-Down 메시지에 어떻게 핸들러를 작성하는지 알아보도록 하자.

```
Type
  TDBCalendar = class(TCalendar)
   :
   protected
     procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X: Integer;
                                       Y: Integer); override;
   :
   end;

procedure TDBCalendar.MouseDown(Button: TMouseButton; Shift: TShiftState; X,
                                              Y: Integer);
var
  MyMouseDown: TMouseEvent;
Begin
  // ReadOnly 의 값이 False 이고 데이터가 수정 가능한지 확인
  If not ReadOnly and FDataLink.Edit then
      Inherited MouseDown(Button, Shift, X, Y)
   Else
  // 데이터 수정이 불가능하다면 사용자가 OnMouseDown 이벤트에 정의한 핸들러를 수행
   Begin
     MyMouseDown := OnMouseDown;
    If Assigned(MyMouseDown) then MyMouseDown(Self, Button, Shift, X, Y);
   End;
End;
```

* Key-Down 메시지에 응답

Key-Down 에 대한 핸들러 역시 앞에서 설명한 Mouse-Down 에 대한 핸들러와 유사하게 정의한다. 다음의 예로써 정의하는 방법을 알아보자.

```
Type
  TDBCalendar = class(TCalendar)
   :
   protected
     procedure KeyDown(var Key: Word; Shift: TShiftState); override;
   :
   end;

procedure TDBCalendar.KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  // ReadOnly의 값이 False이고 Key의 값이 커서를 제어하는 값이며 데이터가 수정 가능
    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
  // 데이터 수정이 불가능하다면 사용자가 OnKeyDown이벤트에 정의한 핸들러를 수행
   begin
     MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
   end;
end;
```

3) 필드 데이터 연결 클래스 갱신

데이터의 변경은 두 가지 형식으로 나타날 것이다. 그 중의 하나는 필드의 값이 변경되어 데이터베이스 관련 컨트롤이 함께 변경되어야 하는 경우일 것이고, 다른 하나는 데이터베이스 관련 컨트롤의 값이 변경되어 필드의 값이 변경되어야 하는 경우일 것이다. 우리가 만들고 있는 TDBCalendar 에는 이미 필드의 값이 변경되었을 때 컨트롤의 값을 변경하기 위한 DataChange 메소드가 정의되어 있다. 그러므로 이제는 데이터베이스 관련 컨트롤이 변경된 경우 필드의 값을 변경하기 위한 경우만 정의해 주면 된다. 이것은 OnUpdateData 이벤트와 관련이 있다. OnUpdateData이벤트는 사용자가 컨트롤의 값을 변경하면 발생하므로 이 이벤트의 핸들러인 UpdateData를 정의함으로써 해결된다.

```
Type
```

```
  TDBCalendar = class(TCalendar)
   Private
     Procedure UpdateData(Sender: TObject);
      :
   end;

procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;
End;

Constructor TDBCalendar.Create(AOwner: TComponent);
Begin
   Inherited Create(AOwner);
  FReadOnly := False;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
End;
```

4) Change 메소드 수정

TDBCalendar 의 Change 메소드는 새로운 날짜가 설정되면 호출된다. Change 는 OnChange 이벤트 핸들러를 호출한다. 달력의 날짜가 변경되면 데이터 셋은 변경이 일어났음을 알아야 한다. 즉, 컨트롤은 그 값을 변경하면 데이터 셋에 값을 변경했음을 알려야 하는 것이다. 이러한 작업은 Change 메소드를 오버라이드하여 구현할 수 있다.

```
Type
  TDBCalendar = class(TCalendar)
   Private
      Procedure Change; override;
      :
   end;
```

```
procedure TDBCalendar.Change;
begin
   FDataLink.Modified;
   Inherited  Change;
End;
```

5) **데이터 셋 갱신**

데이터베이스 관련 컨트롤의 생성에 있어서 최종 단계는 데이터를 변경했을 경우 그 변경된 데이터를 데이터 셋에 적용시키는 것이다. 이러한 적용은 사용자가 새로운 값으로 변경하고 그 컨트롤을 빠져 나가면 수행된다. 여기서 유용하게 사용할 수 있는 메시지가 있다. 바로 CM_EXIT 인데 이 메시지는 사용자가 컨트롤을 빠져 나가면 그 컨트롤로 보내진다. 이 메시지를 이용하여 사용자가 컨트롤을 빠져 나가는 순간에 작업을 수행하도록 할 수 있다. 여기서는 CM_EXIT 의 메시지 핸들러인 CMExit 메소드를 이용하여 변경된 값을 데이터 셋에 적용시켜 보자.

```
Type
  TDBCalendar = class(TCalendar)
   Private
      Procedure  CMExit(var Message: TWMNoParams); message CM_EXIT;
      :
   end;

procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
   try
      FDataLink.UpdateRecord;
   Except
     On  Exception  do  SetFocus;
   End;
   Inherited;
End;
```