대규모 개발에서도 통용되는 작성법 익히기 ~ 객체지향 구문

C • H • A • P • T • E • R

5

5-1 JavaScript의 객체지향 특징

「클래스」는 없고 「프로토타입」만 있다

지금까지 살펴본 대로 JavaScript는 훌륭한 객체지향 언어다. 그러나 Java나 C++, C# 등의 객체지향 언어와는 근본적으로 다른 점이 있다.

그것은 인스턴스화 및 인스턴스라는 개념은 존재하나, 이른바 클래스가 없고 「프로토타입(모형)」이라고 하는 개념만이 존재한다는 점이다.

프로토타입이란 「어떤 객체의 원본이 되는 객체」로서 JavaScript에서는 이것을 이용하여 새로운 객체를 생성해 나간다. 이러한 성질 때문에 JavaScript의 객체지향은 프로토타입 베이스의 객체지향이라고도 불린다. 클래스 베이스의 객체지향에 흠뻑 빠져 버린 사람에게 있어서는 약간 감을 잡기가 어려울지도 모르겠으나, 프로토타입이란 「보다 속박이 약한 클래스와 같은 것」이라고 생각해두기 바란다. 또한 이 책에서도 프로토타입을 편의적으로 「클래스」라고 칭하는 경우가 있기 때문에 이 점 양해 바란다.

가장 간단한 클래스 정의하기

구체적인 예를 직접 보는 편이 알기 쉬울 거라 생각되므로, 먼저 아무런 내용이 없는(가장 간단한) 「클래스」를 정의한 예를 보도록 하자.

리스트 5-01 simple.html(전반)

```
var Member = function() {};
```

「변수 Member에 대해 공란의 함수 리터럴을 대입하고 있을 뿐이지 않나」라고 생각될지도 모르겠으나, 이것이 JavaScript의 클래스다.

실제로 이 Member 클래스는 다음과 같이 new 연산자로 인스턴스화가 가능하다.

리스트 5-02 simple.html(후반)

```
var mem = new Member();
```

반복해서 말하지만, JavaScript의 세계에서는 이른바 엄밀한 의미의 클래스라는 개념이 존재하지 않는다. 여기에서는

JavaScript에서는 함수(Function 객체)에 클래스로서의 역할을 부여하고 있다

라고 기억해두자.

생성자로 초기화하기

이와 같이 new 연산자에 의해서 객체를 생성하는 것을 규정한 함수 객체를 생성자라고 말한다. 이미 Java나 C#의 지식이 있는 사람이라면 이해하고 있을 거라 생각하지만, 생성자라는 것은 「인스턴스(객체)를 생성할 때 객체의 초기화 처리를 기술하기 위한 특수한 메소드(함수)」를 말한다. 리스트 5-01에서 정의한 Member 함수 또한 new 연산자에 의해서 호출되어 객체를 생성한다는 의미로, 엄밀하게는 클래스 그 자체라고 하기보다는 생성자라고 부르는 것이 보다 올바른 표현일 것이다.

생성자의 이름은 보통(생성자가 아닌) 함수와 구별하기 위해서 대문자로 시작하는 것이 일반적이다.

■ 프로퍼티와 메소드

그러면 방금 전에 리스트 5-01로 작성한 생성자에 초기화 처리를 기술해보자.

리스트 5-03 simple2.html

```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
  this.getName = function(){
    return this.lastName + ' ' + this.firstName;
  }
};

var mem = new Member('요시히로', '야마다');
document.writeln(mem.getName());  // 야마다 요시히로
```

여기서 주목해야 할 것은 this 키워드다. this 키워드는 생성자에 의해서 생성되는 인스턴스(즉, 자기 자신)를 나타내는 것이다. this 키워드에 대해서 변수를 지정함으로써 인스턴스의 프로퍼티를 설정할 수 있다.

구문 프로퍼티의 정의

```
this.프로퍼티명 = 값;
```

또한 프로퍼티에는 문자열이나 정수, 날짜 등만이 아닌, 함수 객체(함수 리터럴)를 지정할 수 있다는 점에도 주목해야 한다. JavaScript에 있어서 엄밀하게는 메소드라고 하는 독립된 개념은 없어,

함수 객체로서의 값이 프로퍼티의 메소드로 인지된다

라는 의미가 된다. 여기에서는 getName 프로퍼티에 익명 함수를 인도하고 있기 때문에 이른바 「getName 메소드」를 선언한 셈이 되는 것이다.

실제로 Member 객체를 인스턴스화하고 getName 메소드를 호출해보면, 분명히 「야마다 요시히로」라고 하는 문자열이 표시되는 것을 확인할 수 있을 것이다.

Note **생성자에는 반환값이 불필요**

생성자 함수는 반환값을 돌려주어서는 안 된다. 객체를 생성한다고 하면 반환값으로서 객체를 돌려주고 싶을지도 모르겠지만, 어디까지나 생성자의 역할은 「지금부터 생성하는 객체를 초기화한다」라는 것이 목적이므로 반환값 자체는 불필요하다.

이것은 클래스 베이스의 객체지향 구문에 친숙해진 사람이라면 매우 당연한 포인트이지만, 객체지향 구문이 처음인 사람은 주의해야 한다.

글로벌 변수/함수는 가능한 한 줄일 것, 그러기 위해 관련된 기능이나 정보는 정적 멤버로 정리하도록 하여 조심해서 대체하도록 하자.

동적으로 메소드 추가하기

메소드는 생성자에서만 정의할 수 있는 것이 아니다. new 연산자로 일단 인스턴스화해 버린 객체에 대해서 나중에 메소드를 추가할 수도 있다. 예를 들어, 다음의 예는 방금 전의 리스트 5-03에서 생성자 안에 정의된 getName 메소드를 나중에 연관시켜 정의하도록 고쳐 쓴 예다.

리스트 5-04 dynamic.html

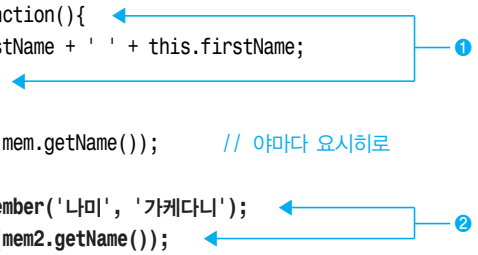
```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

var mem = new Member('요시히로', '야마다');
mem.getName = function(){
  return this.lastName + ' ' + this.firstName;
}

document.writeln(mem.getName());  // 야마다 요시히로
```

역시 이 경우에도 올바르게 getName 메소드가 동작하고 있는 것을 확인할 수 있다.

다만, 인스턴스에 대해서 직접 멤버(프로퍼티나 메소드)를 추가한 경우에는 주의해야 할 점도 있다. 다음과 같은 예를 보자.

리스트 5-05 dynamic2.html

```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

var mem = new Member('요시히로', '야마다');
mem.getName = function(){
  return this.lastName + ' ' + this.firstName;
}

document.writeln(mem.getName());       // 야마다 요시히로

var mem2 = new Member('나미', '가케다니');
document.writeln(mem2.getName());
```



추가로 기술한 것은 굵은 글씨 부분이다. ❷에서 새롭게 생성한 인스턴스 mem2로부터 동적으로 추가한 getName 메소드를 호출하려고 하면, 「개체가 이 속성 또는 메서드를 지원하지 않습니다.」라고 하는 메시지가 반환되는 것을 확인할 수 있을 것이다(Internet Explorer의 경우. 에러 메시지는 브라우저에 따라 다르다).

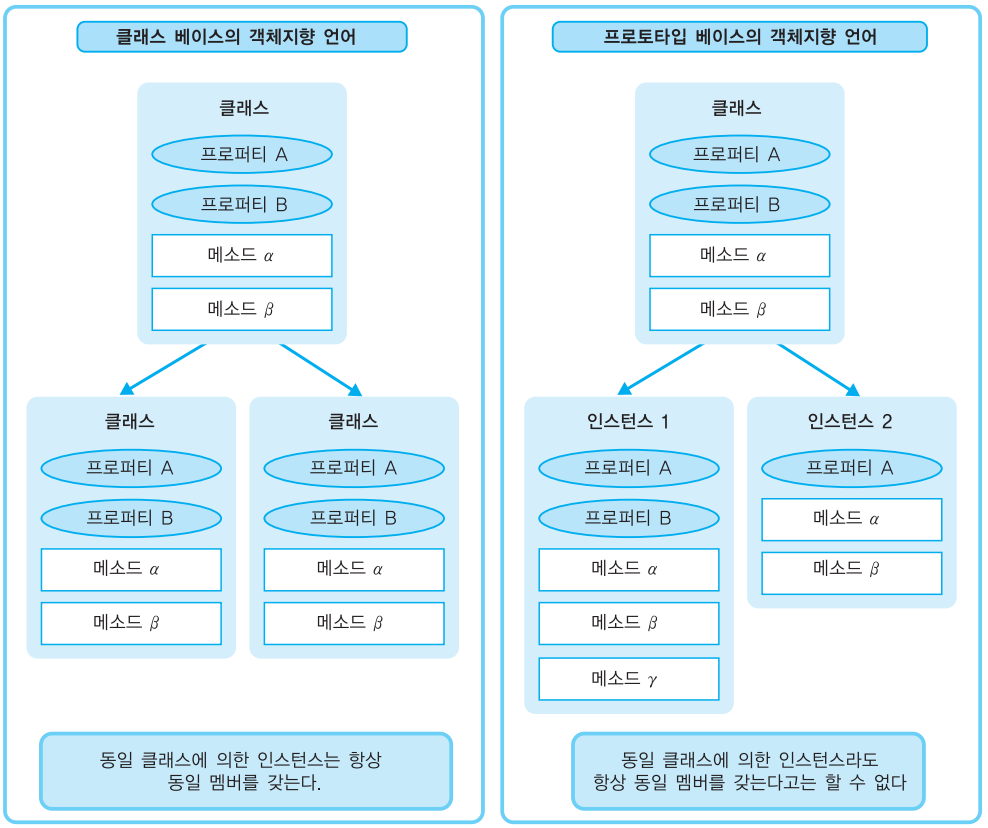
중요한 것은 ❶의 경우에는 Member 클래스 그 자체가 아니고, 「생성된 인스턴스에 대해서 메소드가 추가되고 있다」라는 점이다.

Java와 같은 클래스 베이스의 객체지향에 익숙한 사람이라면 「동일한 클래스를 기초로 생성된 인스턴스는 동일한 멤버를 가진다」라는 것이 상식이지만, 프로토타입 베이스의 객체지향(JavaScript)의 세계에서는

> 동일한 클래스를 기초로 생성된 인스턴스라 할지라도 각각이 가지는 멤버가 동일하다라고 한정할 수 없다

라는 점을 알아두어야 한다. 여기에서는 새롭게 멤버를 추가하고 있을 뿐이지만, delete 연산자(61쪽)로 인스턴스로부터 기존의 멤버를 삭제할 수도 있다.

이러한 유연함이 앞서 말한 프로토타입이 「보다 속박이 약한 클래스와 같은 것」이라고 한 이유다.



◉ **프로토타입은 「보다 속박이 약한 클래스」**

5-2 생성자의 문제점과 프로토타입

메소드는 프로토타입으로 선언한다 – prototype 프로퍼티 –

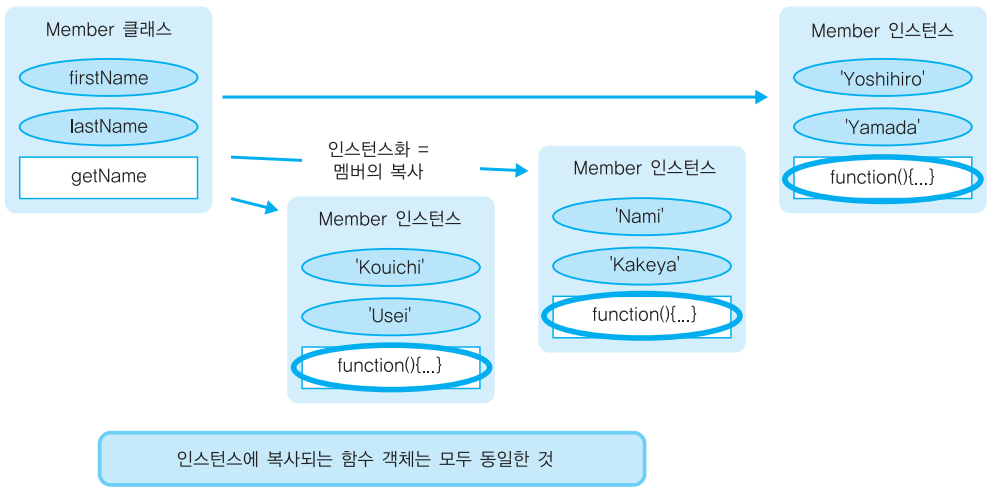
앞 절을 통해 인스턴스 공통의 메소드를 정의하려면 적어도 생성자로 메소드를 정의할 필요가 있음을 알 수 있었다. 그러나 실은 생성자에 의한 메소드 추가에는

메소드의 수에 비례하여 「쓸데없이」 메모리를 소비한다

라는 큰 문제점이 있다.

생성자는 인스턴스를 생성할 때마다 각각의 인스턴스를 위해 메모리를 확보한다. 예를 들어, 리스트 5-03의 예라면 인스턴스를 위해 Member 클래스에 속하는 firstName/lastName 프로퍼티, getName 메소드를 복사한다는 뜻이다.

그러나 getName과 같은 메소드(함수 리터럴)는 모든 인스턴스에서 내용이 동일할 것이기 때문에 인스턴스 단위로 메모리를 확보하는 것은 쓸데없는 일이다.



◉ **생성자에서 메소드를 정의하는 것에 대한 문제**

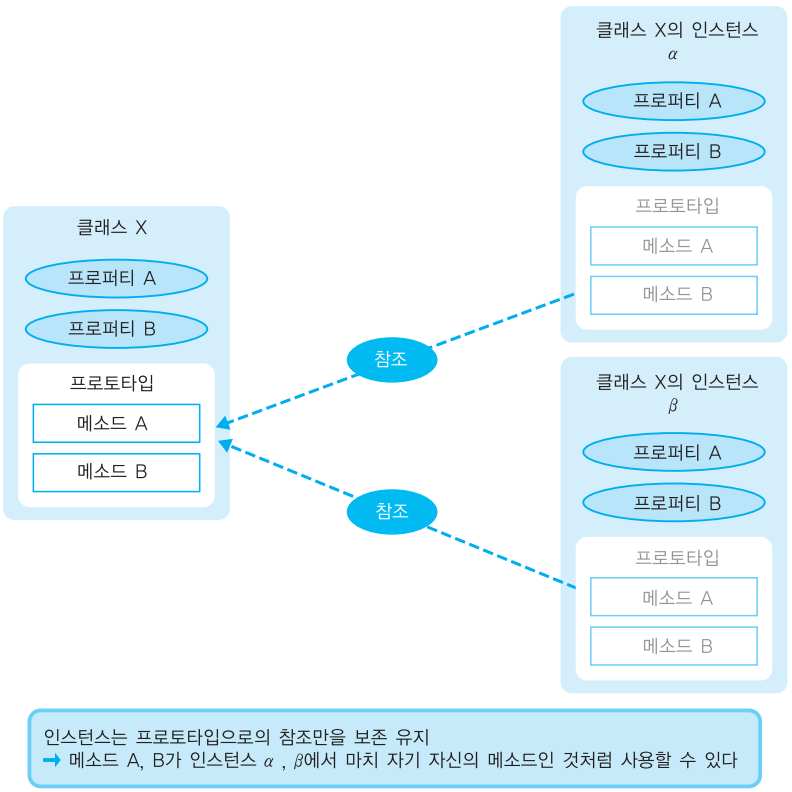
여기에서는 getName 메소드 하나이기 때문에 그다지 문제는 되지 않을지도 모르겠다. 그러나 이것이 더 복잡한 메소드를 10개, 20개씩 갖는 클래스라고 하면 어떨까? 그러한 클래스를 몇 개씩 인스턴스화하는 코드에서 그 모든 것을 하나하나 인스턴스화할 때마다 복사한다는 것은 쓸데없는 일이라 할 수 있다.

그래서 JavaScript에서는 객체에 멤버를 추가하기 위한 prototype이라고 하는 프로퍼티를 준비하고 있다. prototype 프로퍼티는 디폴트로 빈 객체를 참조하고 있지만, 이것에 프로퍼티나 메소드를 추가할 수 있다.

그리고 이 prototype 프로퍼티에 대입된 멤버는 인스턴스화된 앞의 객체에 인계된다. 좀 더 깊이 설명하자면, prototype 프로퍼티에 대해 추가된 멤버는 그 클래스(생성자)를 기초로 생성된 모든 인스턴스로부터 이용할 수 있다는 것이다. 또한 이를 약간 어려운 말투로 표현하자면,

> 객체를 인스턴스화했을 경우 인스턴스는 베이스가 되는 객체에 속하는 prototype 객체에 대해서 암묵적인 참조를 갖게 된다

라고 말할 수 있겠다.



◉ **프로토타입 객체**

이야기가 약간 어려워졌으므로 구체적인 코드를 보기로 하자. 리스트 5-06은 방금 전의 Member 객체에 포함되는 getName 메소드를 프로토타입으로 정의한 것이다.

리스트 5-06 prototype.html

```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

Member.prototype.getName = function(){
  return this.lastName + ' ' + this.firstName;
};

var mem = new Member('요시히로', '야마다');
document.writeln(mem.getName());     // 야마다 요시히로
```

프로토타입 객체(prototype 프로퍼티가 참조하는 객체)에 추가된 getName 메소드가 Member 클래스의 인스턴스(변수 mem)로부터도 올바르게 참조되고 있는 것을 확인할 수 있을 것이다.

「프로토타입(모형) 기반의 객체지향」이라는 식으로 말해 버리면, 특히 「클래스 기반의 객체지향」에 익숙한 사람들이 다가서기 힘들어 할 수도 있을 것 같지만, 요점은 JavaScript에서의 관점에서 「클래스라고 하는 추상적인 설계도가 존재하지 않는다」라고 생각하면 될 것 같다.

JavaScript의 세계에서 존재하는 것은 어디까지나 실체화된 객체만이며, 새로운 객체를 생성하는 경우에도 (클래스가 아닌) 객체가 토대가 된다. 그리고 새로운 객체를 만들기 위한 원형(모형)이 바로, 각각의 객체에 속하는 「프로토타입」이라고 하는 특별 객체인 것이다.

어떤가? 「프로토타입 기반의 객체지향」이 조금은 가깝게 느껴지는가?

프로토타입 객체를 이용하는 두 가지 이점

이와 같이 프로토타입 객체를 개입시켜 메소드를 정의하는 것에는 다음의 두 가지 이점이 있다.

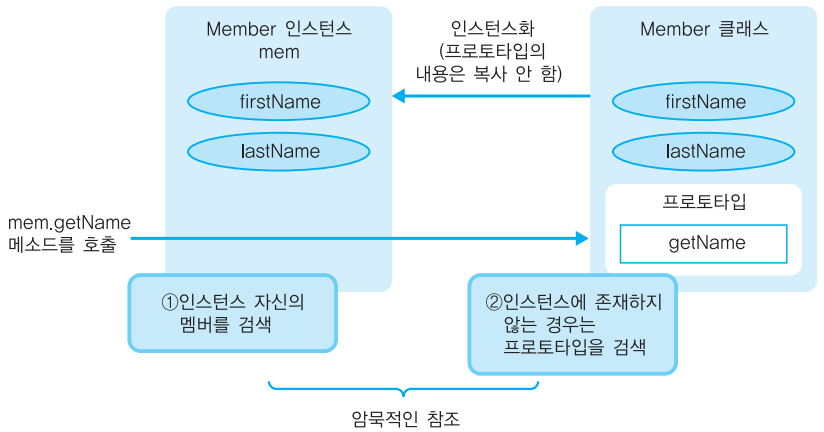
(1) 메모리의 사용량을 절감할 수 있다

반복해서 말하지만, 프로토타입 객체의 내용은 어디까지나 「베이스가 되는 객체로부터 암묵적으로 참조만」 될 뿐 인스턴스에 복사되는 것은 아니다. 194쪽의 그림과 같이 인스턴스는 원래 객체의 프로토타입 객체에 대해서 암묵적인 참조만을 갖는다.

즉, JavaScript에서는 객체의 멤버가 호출되었을 때에

- 인스턴스 측에 요구된 멤버가 존재하지 않는지를 확인
- 존재하지 않는 경우는 암묵적인 참조를 통해 프로토타입 객체를 검색

하는 처리를 실시해 멤버를 취득하게 된다.

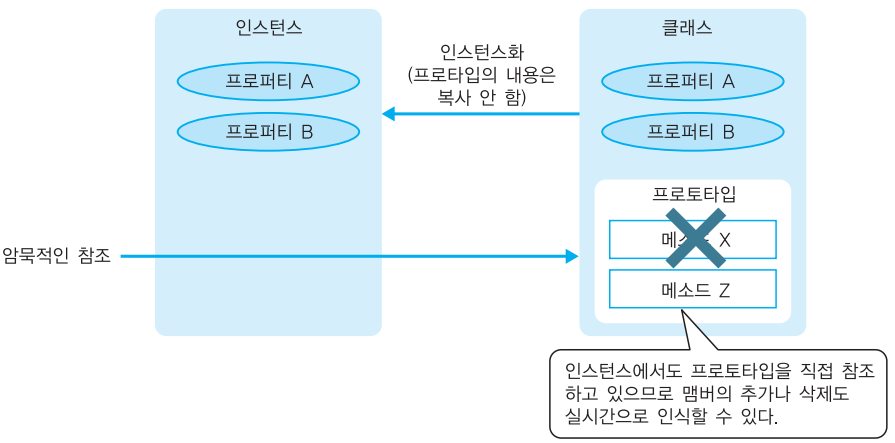


◉ **암묵적인 참조**

이로 인해 생성자 경유로 메소드를 정의하는 경우에 발생하는 「메모리를 쓸데없이 소비」하는 문제를 회피할 수 있게 된다.

(2) 멤버의 추가나 변경을 인스턴스가 실시간으로 인식할 수 있다

「인스턴스에 멤버를 복사하지 않는다」라고 하는 것은 「프로토타입 객체에의 변경(추가나 삭제)을 인스턴스 측에서 동적으로 인식할 수 있다」라는 뜻도 된다.



◉ **프로토타입에의 변경도 실시간으로 인식**

시험삼아 아래와 같은 코드로 실제의 동작을 확인해보자.

리스트 5-07 dynamic3.html

```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

var mem = new Member('요시히로', '야마다');

Member.prototype.getName = function(){
  return this.lastName + ' ' + this.firstName;
};

document.writeln(mem.getName());  // 야마다 요시히로
```

어떠한가? 리스트 5-06과 다른 점은 「getName 메소드를 new 연산자에 의해서 인스턴스를 생성한 후에 추가하고 있다」라고 하는 점이다. 이 경우에도 아무런 문제 없이 메소드를 인식함을 확인할 수 있다.

인스턴스를 「객체(클래스)의 복사본」이라고 생각하고 있으면 이것은 이상하게 생각될지도 모르겠으나, 방금 전의 「암묵의 참조」라는 개념을 이해하고 있으면 오히려 당연하다고 말할 수 있는 동작이 된다.

프로토타입 객체의 불가사의(1) – 프로퍼티의 설정 –

그럼 프로토타입 객체로 프로퍼티를 선언하면 어떻게 될까? 「암묵의 참조」라는 개념으로 보면 프로퍼티의 값은 모든 인스턴스에 공유되는 것처럼 생각된다. 어떤 인스턴스에서 프로퍼티의 값을 변경하면 모든 인스턴스에 그 변경이 반영되어 버리는 건 아닐까?

자, 그럼 실제의 동작을 확인해보자.

리스트 5-08 prototype2.html

```
var Member = function(){ };
Member.prototype.sex = '남자';

var mem1 = new Member();
var mem2 = new Member();
document.writeln(mem1.sex + '|' + mem2.sex);  // 남자 | 남자  ←❶
```

```
mem2.sex = '여자';    ←❷
document.writeln(mem1.sex + '|' + mem2.sex);  // 남자 | 여자   ←❸
```

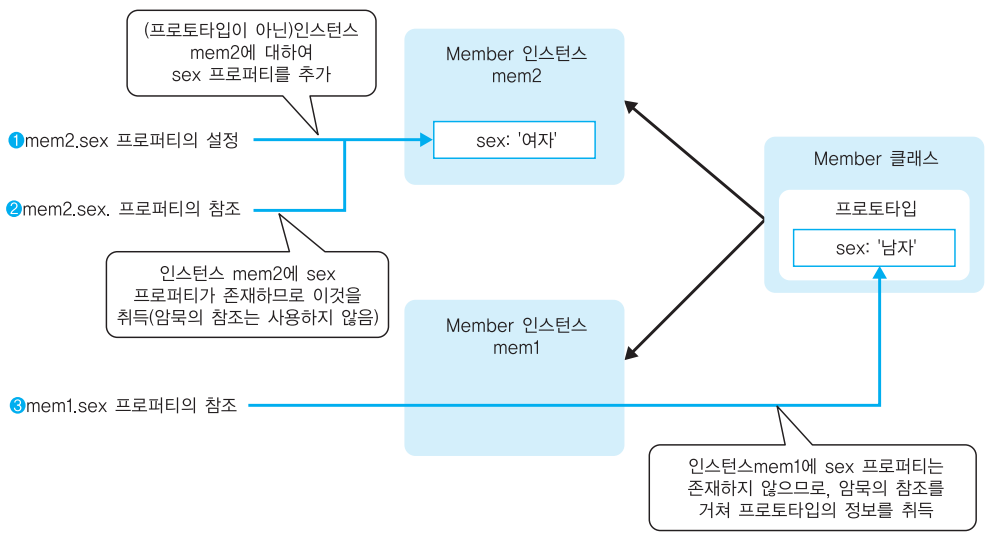
주목해주길 바라는 부분은 ❸의 부분이다.

❷에서 인스턴스 mem2의 sex 프로퍼티를 변경하고 있기 때문에 원래의 프로토타입 객체도 바뀌게 되어, ❸은 양쪽 모두 「여자」가 될 것이라고 생각할 수 있다. 그러나 결과는 인스턴스 mem1의 내용은 그대로이며, 인스턴스 mem2의 내용만이 변경되었다.

이것 참, 도대체 어떻게 된 일인가?

결론부터 말하자면, 프로토타입 객체가 사용되는 것은 「값을 참조할 때뿐」이다. 값의 설정은 항상 인스턴스에 대해서 행해진다.

프로퍼티 설정/참조의 내부적인 동작을 좀 더 자세하게 보도록 하자.



◉ **암묵의 참조(프로퍼티의 설정)**

우선 ❶의 시점에서는 인스턴스 mem1, mem2는 둘 다 sex 프로퍼티를 가지지 않으므로 프로토타입 객체가 가지는 sex 프로퍼티를 「암묵적으로 참조」한다. 그런데 ❷의 시점에서 인스턴스 mem2의 sex 프로퍼티가 변경된다면, 「인스턴스 mem2 자신이 sex 프로퍼티를 가지게 됨」으로써 인스턴스 mem2는 프로토타입을 참조할 필요가 없어진다. 그 결과, 인스턴스 mem2에 설정되어 있는 sex 프로퍼티가 참조되는 것이다(이것을 「인스턴스의 sex 프로퍼티가 프로토타입의 name 프로퍼티를 은폐한다」라고 말한다).

물론, 이 시점에서도 인스턴스 mem1은 여전히 sex 프로퍼티를 가지고 있지 않기 때문에

그대로 암묵의 참조를 하여 프로토타입 객체에서 보유하는 sex 프로퍼티를 참조하게 된다.

이와 같이 프로퍼티를 프로토타입으로 정의해도 동작상으로는 「인스턴스가 개별적으로 프로퍼티를 보유하고 있는 것처럼 보이기」 때문에 문제는 없다. 다만, 본래 인스턴스 단위로 값이 달라야 할 프로퍼티를 프로토타입 객체로 선언하는 것은 의미가 없다(독해 전용의 프로퍼티는 별도다). 보통 다음과 같이 구분하여 사용하기 바란다.

- 프로퍼티의 선언 ➡ 생성자로
- 메소드의 선언 ➡ 프로토타입으로

■ 정적 프로퍼티/정적 메소드를 정의하려면

96쪽에서도 말했듯이, 정적 프로퍼티/정적 메소드란 인스턴스를 생성하지 않아도 객체로부터 직접 호출할 수 있는 프로퍼티/메소드다(참고로, 인스턴스 경유로 호출하는 프로퍼티/메소드는 인스턴스 프로퍼티/인스턴스 메소드라고 했다).

이러한 정적 프로퍼티/정적 메소드를 정의하는 경우 프로토타입 객체에는 등록할 수 없기 때문에 주의가 필요하다(프로토타입 객체는 어디까지나 「인스턴스」로부터 암묵적으로 참조되는 것을 목적으로 한 객체이므로 이는 당연하다). 정적 프로퍼티/정적 메소드는 다음과 같이 생성자(객체)에 직접 추가한다.

구문 정적 프로퍼티/정적 메소드

```
객체명.프로퍼티명 = 값
객체명.메소드명 = function() { /* 메소드의 정의 */ }
```

구체적인 예를 들어 보자. 다음은 기본 도형의 면적을 구하는 기능을 정리한 유틸리티 클래스인 Area 클래스를 정의한 예다. Area 클래스에는 클래스의 버전 번호를 나타내는 version 프로퍼티를 시작해 삼각형/마름모의 면적을 요구하는 triangle/diamond 메소드가 포함되도록 하였다.

리스트 5-09 static.html

```
var Area = function() {};  // 생성자

Area.version = '1.0';  // 정적 프로퍼티의 정의

  // 정적 메소드 triangle의 정의
Area.triangle = function(base, height) {
  return base * height / 2;
}
```

```
  // 정적 메소드 diamond의 정의
Area.diamond = function(width, height) {
  return width * height / 2
}

document.writeln('Area 클래스의 버전 : ' + Area.version);          // 1.0
document.writeln('삼각형의 면적 : ' + Area.triangle(5, 3));        // 7.5
var a = new Area();
document.writeln('마름모의 면적 : ' + a.diamond(10, 2));           // 에러
```

확실히 객체로부터 정적 프로퍼티/정적 메소드가 직접 호출될 수 있는 것을 확인할 수 있다.

또 인스턴스 경유로 정적 메소드를 호출하려고 한 경우에 「객체가 이 속성 또는 메서드를 지원하지 않습니다.」라는 에러(Internet Explorer의 경우)가 발생한다. 정적 멤버는 어디까지나 Area라고 하는 함수 객체에 동적으로 추가된 멤버이지 Area가 생성하는 인스턴스에 추가된 것이 아니기 때문이다.

■ 정적 프로퍼티/정적 메소드를 정의할 때의 두 가지 주의점

정적 프로퍼티/정적 메소드를 정의하는 경우에는 다음의 사항에 주의하기 바란다.

(1) 정적 프로퍼티는 기본적으로 읽기 전용

정적 프로퍼티는 인스턴스 프로퍼티와는 달리 「클래스 단위로 보유되는 정보」다. 즉, 그 내용을 변경했을 경우는 스크립트 내의 모든 내용에 변경이 반영되어 버리게 된다. 원칙적으로 정적 프로퍼티로 정의한 값은 읽기 전용의 용도로 한정해야 한다.

(2) 정적 메소드 안에서는 this 키워드는 사용할 수 없다

인스턴스 메소드 안에서의 this 키워드는 (방금 전에도 말한 것처럼) 인스턴스 자신을 나타낸다. 한편, 정적 메소드 안에서의 this 키워드는 생성자(함수 객체)를 나타낸다. 당연한 이야기이지만, 인스턴스가 없기 때문에 정적 메소드로부터 인스턴스 프로퍼티의 값에 액세스할 수 없다는 점을 주의해야 한다.

정확하게는 「정적 메소드 안에서는 this 키워드를 사용해도 의미가 없다」라는 것이 보다 올바른 표현이겠지만, 인스턴스 메소드와 같은 의미로서의 「this 키워드는 사용할 수 없다」라고 기억해두는 것이 편할 것이다.

Note **왜 정적 멤버를 정의하는 것인가**

정적 멤버는 기능적으로는 글로벌 변수/함수와 서로 아무런 차이가 없다. 「그러면 정적 프로퍼티/정적 메소드 등을 사용하지 않고 그냥 글로벌 변수/함수를 사용하면 좋지 않을까」라고 생각할지도 모르겠다.

그러나 이것은 불가하다.

왜냐하면 글로벌 변수/함수는 이름이 충돌하는 원인이 되기 때문이다.

예를 들어, 100개의 글로벌 변수/함수를 포함한 라이브러리를 생각해보자. 애플리케이션으로부터 이 라이브러리를 사용하려고 했을 경우, 여기서 정의된 100개의 이름은 이른바 예약어(reserved word)이기 때문에 애플리케이션에서는 사용할 수가 없다(만약 잘못해 덮어 쓰기해 버렸을 경우에는 원래의 기능이나 정보를 잃게 된다). 글로벌 변수/함수가 많아지면 많아질수록 애플리케이션 측에서는 이름이 충돌할지 모른다는 것을 의식하여 코딩하지 않으면 안 된다.

이것은 당연히 바람직한 상태가 아니기 때문에 글로벌 변수/함수는 가능한 한 적게 사용하도록 해야 한다. 정적 멤버를 이용하면 변수/함수는 클래스 밑에 속하게 되기 때문에 이러한 경합(충돌)의 가능성을 줄일 수 있다(예를 들어, 글로벌 변수 version과 정적 프로퍼티 Area.version은 별개다).

글로벌 변수/함수는 가능한 한 줄일 것. 그러기 위해서는 관련된 기능이나 정보는 정적 멤버로 정리할 것. 이 점에 대해서 유의하도록 하자.

프로토타입 객체의 불가사의(2) – 프로퍼티의 삭제 –

앞의 절에서는 프로토타입으로 멤버를 추가하는 경우에 대해서 이야기했으나, 삭제하는 경우에도 「인스턴스의 단위로 행해진다」라는 사실에 주목해야 한다. 다음의 구체적인 예를 보자.

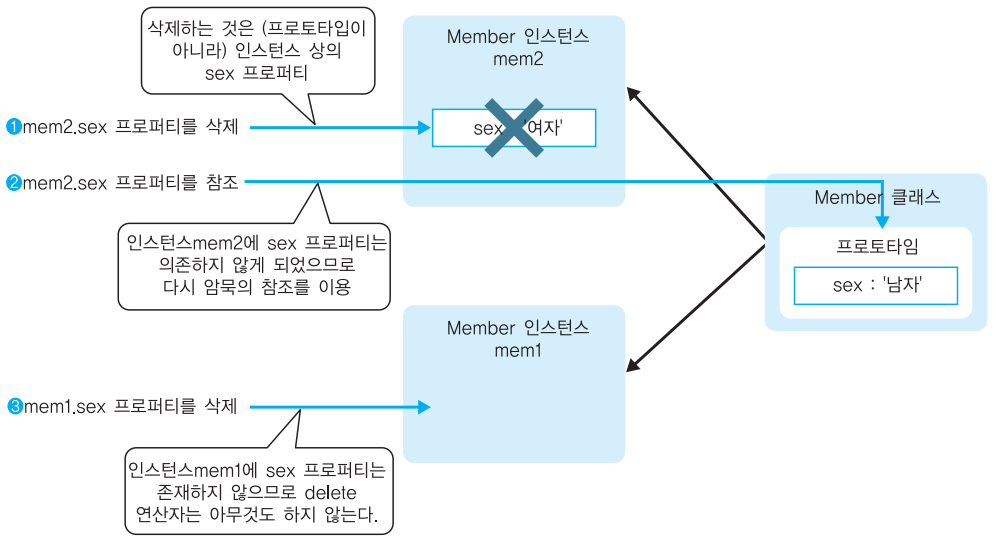
리스트 5-10 prototype3.html

```
var Member = function(){ };

Member.prototype.sex = '남자';

var mem1 = new Member();
var mem2 = new Member();
document.writeln(mem1.sex + '|' + mem2.sex);     // 남자 | 남자
mem2.sex = '여자';
document.writeln(mem1.sex + '|' + mem2.sex);     // 남자 | 여자
delete mem1.sex  ←❶
delete mem2.sex  ←❷
document.writeln(mem1.sex + '|' + mem2.sex);     // 남자 | 남자  ←❸
```

리스트 5-10은 방금 전의 리스트 5-08의 말미에 delete 연산자(61쪽)를 사용한 코드가 추가된 것이다. 이 경우의 결과(❸)에 주목해보자.

우선 ❶에서 delete 연산자는 「인스턴스 mem1의」 sex 프로퍼티를 삭제하려고 하는데, 인스턴스 mem1은 sex 프로퍼티를 가지지 않으므로 delete 연산자는 아무것도 실시하지 않는다(프로토타입까지 거슬러 올라가 삭제할 것은 없다). 그 결과, ❸에서 인스턴스 mem1은 암묵의 참조를 더듬어 프로토타입 객체의 sex 프로퍼티를 반환해준다. 즉, 결과는 「남자」가 된다.

한편, ❷는 어떤가? 이번에는 인스턴스 mem2가 자기 자신의 sex 프로퍼티를 가지기 때문에 delete 연산자는 이것을 삭제한다. 그 결과, ❸에서 인스턴스 mem2는(독자적인 프로퍼티 값을 가지지 않게 되었으므로) 다시 암묵의 참조를 더듬어 프로토타입 객체의 sex 프로퍼티(즉, 「남자」)를 돌려주게 되는 것이다.



◉ **암묵의 참조(프로퍼티의 삭제)**

인스턴스 측에서의 멤버 추가나 삭제의 조작이 프로토타입 객체에까지 영향을 미칠 일은 없다

라는 것을 재차 유념해두자.

■ 부연설명: 인스턴스 단위로 프로토타입의 멤버를 삭제하려면

무엇보다 엄밀하게는 다음과 같이 기술함으로써 프로토타입 객체의 멤버를 삭제할 수도 있다.

```
delete Member.prototype.sex
```

다만, 이러한 코드는 이 프로토타입을 참조하고 있는 모든 인스턴스에 영향을 미친다는(모든 인스턴스의 sex 프로퍼티가 삭제되어 버린다) 점에 주의해야 한다.

만약 프로토타입으로 정의된 멤버를 「인스턴스 단위로」삭제하고 싶은 경우, 약간의 트릭이지만 정수 undefined를 이용하는 방법이 있다.

리스트 5-11 undefined.html

```
var Member = function(){ };

Member.prototype.sex = '남자';

var mem1 = new Member();
var mem2 = new Member();
document.writeln(mem1.sex + '|' + mem2.sex);  // 남자 | 남자
mem2.sex = undefined;
document.writeln(mem1.sex + '|' + mem2.sex);  // 남자 | undefined
```

여기에서는 sex 프로퍼티의 값을 정수 undefined로 덮어씀으로써 「유사적으로」 멤버를 무효화하고 있다.

다만, 이 방법은 어디까지나 「멤버의 존재 자체는 그대로 두고 값을 강제적으로 미정의값으로 대입하고 있는 것」에 지나지 않는다. 엄밀하게는 멤버를 삭제하고 있는 것이 아니기 때문에 for...in 루프로 객체 내의 멤버를 열거했을 경우에는 sex 프로퍼티는 여전히 존재하는 것으로 표시된다.

리스트 5-12 undefined2.html

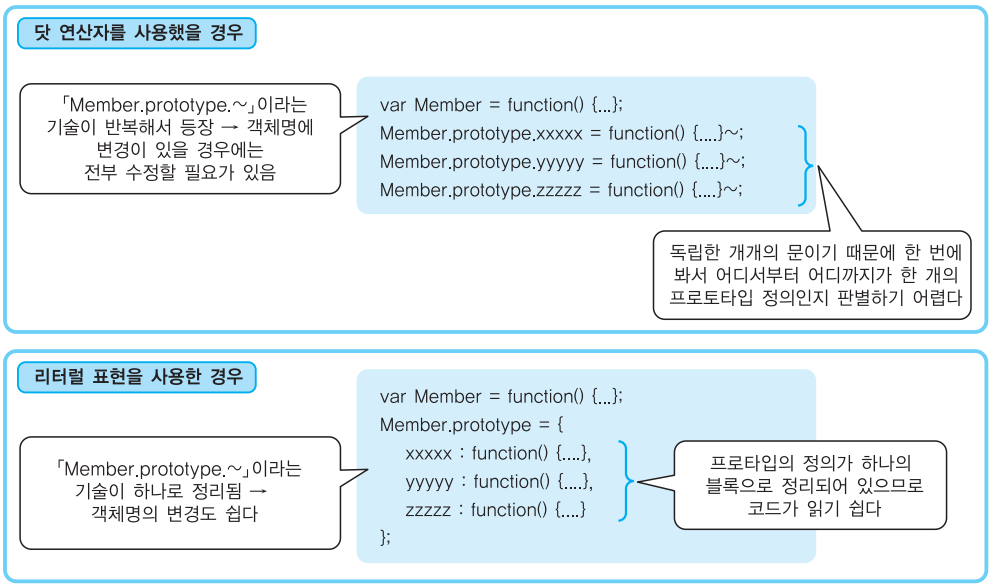
```
var Member = function(){ };
Member.prototype.sex = '남자';

var mem = new Member();
mem.sex = undefined;
for (var key in mem) {
  document.writeln(key + ":" + mem[key]);
}       // sex:undefined
```

객체 리터럴로 프로토타입 정의하기

자, 지금까지의 예에서는 닷 연산자(.)를 사용하여 프로토타입에 멤버를 추가해 왔다. 구문적으로 그것은 그것대로 맞지만, 멤버 수가 많아졌을 경우 코드가 아무래도 장황하게 될 수밖에 없기 때문에 바람직한 작성 방법이라고는 할 수 없다.

사소한 일이긴 하지만 매회 「Member.prototype.~」이라는 기술을 반복해야 하는 것도 귀찮고, 원래의 객체명(여기에서는 Member)이 변경이 되었을 경우에 정의된 모든 곳들을 고쳐 써야 하는 것 또한 좋다고는 말할 수 없다. 또 개개의 멤버 정의가 독립된 블록에 기술되어 있다는 점에서 「어디서부터 어디까지가 같은 객체의 멤버 정의인지를 언뜻 봤을 때 알아내기 어렵다」라는 가독성의 문제도 있다.

그래서 등장하는 것이 44쪽에서도 설명한 객체의 리터럴 표현이다. 리터럴 표현을 이용하면 리스트 5-13은 리스트 5-14와 같이 고쳐 쓸 수 있다.



◉ 닷 연산자와 리터럴 표현

리스트 5-13 literal.html

```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

Member.prototype.getName = function(){
  return this.lastName + ' ' + this.firstName;
};

Member.prototype.toString = function(){
  return this.lastName + this.firstName;
};
```

리스트 5-14 literal2.html

```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

Member.prototype = {
  getName : function(){return this.lastName + ' ' + this.firstName;},
  toString : function(){return this.lastName + this.firstName;}
};
```

이와 같이 객체 리터럴을 이용함으로써

- 「Member.prototype.~」과 같은 기술이 최소한으로 억제된다
- 그 결과, 객체명의 변경이 있을 경우에도 영향받는 범위는 한정 가능하다
- 동일 객체의 멤버 정의가 하나의 블록에 담겨질 수 있기 때문에 코드의 가독성이 향상된다

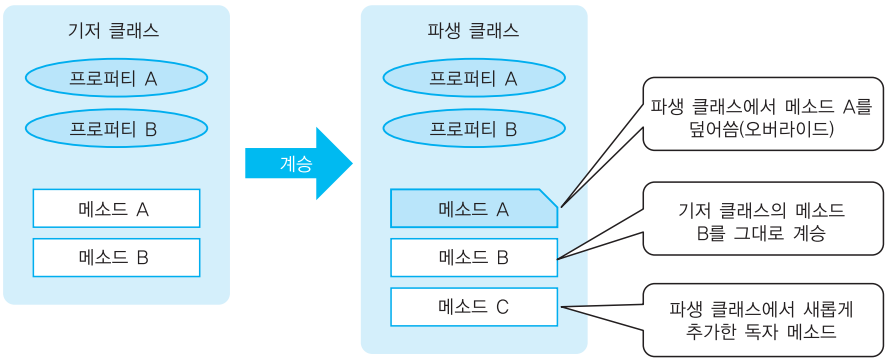
라는 효과가 있다. 리스트 5-13과 같이 뿔뿔이 흩어져 기술하던 것에 비하면 대단히 코드가 깨끗하게 정리되었다는 생각이 들지 않는가?

보통 프로토타입을 정의하는 경우에는 이와 같이 리터럴 표현을 이용하길 바란다.

5-3 객체의 계승 – 프로토타입 체인 –

계승이란?

객체지향 언어를 이해하는 데 있어서 중요한 개념 중의 하나가 계승(繼承)이다. 계승이란 베이스가 되는 객체(클래스)의 기능을 계승하여 새로운 클래스를 정의하는 기능을 말한다.



◉ 계승

이것을 이용하면 공통된 기능을 복수의 클래스에서 중복해 정의할 필요가 없어진다. 따라서 근원이 되는 클래스로부터의 차분 기능만을 기술하는 것만으로 끝나게 된다(차분 프로그래밍이라고도 말한다).

계승에 있어서 계승원이 되는 클래스를 슈퍼 클래스(기저 클래스: base class), 계승된 클래스를 서브 클래스(상속 클래스)라고 말한다.

프로토타입 체인의 기초

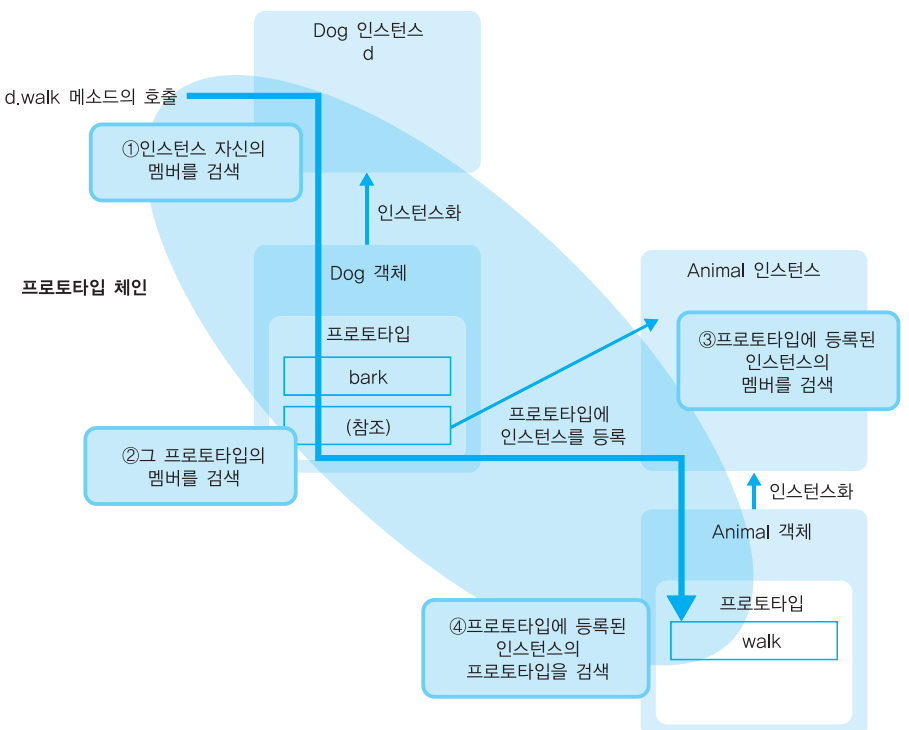
자, 이러한 계승의 구조를 JavaScript의 세계에서 실현하고 있는 것이 프로토타입 체인이라고 하는 것이다. 프로토타입 체인에 대해서는 우선 예제 코드를 보는 것이 빠를 것이다.

리스트 5-15 chain.html

```
var Animal = function() {}

Animal.prototype = {
  walk : function() { document.writeln('종종...'); }
};
var Dog = function() {};
Dog.prototype = new Animal();

Dog.prototype.bark = function() { document.writeln('멍멍!'); }

var d = new Dog();
d.walk();       // 종종…
d.bark();       // 멍멍!
```

주목해야 할 것은 리스트 내의 굵은 글씨 부분이다. Dog 객체의 프로토타입(Dog.prototype)이 Animal 객체의 인스턴스로 세트되고 있는 것을 확인할 수 있을 것이다. 이로 인해 Dog 객체의 인스턴스로부터 Animal 객체로 정의된 walk 메소드를 호출할 수 있다.

이 동작은 방금 전의 「암묵의 참조」를 생각해 내면 쉽게 이해할 수 있을 것이다. 구체적인 메소드 호출의 흐름은 아래와 같다. 그림으로 확인해보자.



◉ 프로토타입 체인

1. Dog 객체의 인스턴스 d로부터 멤버의 유무를 검색한다
2. 해당하는 멤버가 존재하지 않는 경우에는 Dog 객체의 프로토타입 – 즉, Animal 객체의 인스턴스를 검색한다
3. 거기서도 목적의 멤버가 발견되지 않는 경우에는 한 층 위의 Animal 객체의 프로토타입을 검색한다

이 예에서는 Animal 객체에서 walk 메소드가 발견되므로 검색은 거기서 종료하지만, 발견되지 않는 경우에는 한 층 더 위의 프로토타입(구체적으로는 최상위의 Object.prototype까지)으로 거슬러 올라가게 된다.

이와 같이 JavaScript에서는 프로토타입에 인스턴스를 설정함으로써 인스턴스끼리 「암묵의 참조」 하에 서로 연결되어 계승 관계를 갖게 할 수 있다. 그리고 이러한 프로토타입의 연결을 프로토타입 체인이라고 부른다.

Note **프로토타입 체인의 종단은 Object.prototype**

131쪽에서도 말했듯이, 모든 객체의 루트가 되는 것은 Object 객체다. 즉, 모든 객체는 암묵적으로(명시적으로 프로토타입을 설정하지 않아도) Object 객체를 계승해, Object.prototype을 참조하고 있다. 바꾸어 말하면, 프로토타입 체인의 종단에는 반드시 Object.prototype이 있다.

계승 관계는 동적으로 변경 가능

이와 같이 프로토타입 체인은 JavaScript로 「계승」 기능을 실현한 구조다. 그러나 Java나 C++와 같은 언어을 통해 객체지향(계승)을 배운 사람은 주의해야 한다. 왜냐하면 Java나 C++에 있어서의 계승은 어디까지나 정적인 계승(일단 결정한 계승 관계를 도중에 변경할 수 없다)인데 반해, JavaScript에 의한 계승은 동적이기 때문이다. 즉, 이 말은 동일한 객체가 어느 타이밍에서는 객체 X를 계승하다가도 이후의 다른 타이밍에서 객체 Y를 계승할 수도 있다는 뜻이다. 다음과 같은 예를 보면 이해하기 쉬울 것이다.

리스트 5-16 chainDynamic.html

```
var Animal = function() {}

Animal.prototype = {
  walk : function() { document.writeln('종종...'); }
};
```

```
var SuperAnimal = function() {}
SuperAnimal.prototype = {
  walk : function() { document.writeln('다다다닷!'); }
};

var Dog = function() {};
Dog.prototype = new Animal();  // Animal 객체를 계승
var d1 = new Dog();   ←❶
d1.walk();  // 종종...   ←❸

Dog.prototype = new SuperAnimal(); // SuperAnimal 객체를 계승
var d2 = new Dog();   ←❷
d2.walk();  // 다다다닷!   ←❹
d1.walk();  // ????   ←❺
```

❶에서는 Dog.prototype에 Animal 인스턴스를 세트한 상태로 인스턴스 d1을, ❷에서는 SuperAnimal 인스턴스로 전환한 다음 인스턴스 d2를 각각 생성하고 있다. 이 결과, ❸, ❹에서는 Animal/SuperAnimal 객체(정확하게는 그 프로토타입)로 정의된 walk 메소드가 실행되는 것을 확인할 수 있다.

여기까지는 극히 직관적으로 이해할 수 있는 동작이라고 생각한다.

그러나 마지막 ❺는 어떤가? 인스턴스 d1을 생성한 시점에서 Dog 객체의 프로토타입은 Animal 객체다. 그러나 ❺의 시점에서는 이미 SuperAnimal 객체로 변경되었다.

암묵의 참조가 실시간으로 변경을 인식하는 것을 생각하면, ❺에서는 SuperAnimal 객체의 walk 메소드가 호출되어 「다다다닷!」이 출력될 것처럼 예상된다. 그러나 결과는 「종종...」이다. — Animal 객체의 walk 메소드가 호출된다.

여기에서 말할 수 있는 것은 JavaScript의 프로토타입 체인은

인스턴스가 생성된 시점에서 고정되어 그 후의 변경에는 관여치 않고 보존된다

라는 것이다. 이것은 「JavaScript가 동적인 성질을 가지고 있다」라고 이해하고 있는 사람에게 있어 오해하기 쉬운 포인트이기도 하기 때문에 주의가 매우 필요한 부분이다.

5-4 본격적인 개발에 대비하기 위해서

지금까지 객체지향의 기본적인 구문이나 이용상의 주의점에 대해서 배워 보았다. 초보적인 코딩을 실시하는 데 있어서는 지금까지의 내용으로도 충분하지만, JavaScript로 보다 본격적인(대규모) 애플리케이션이나 라이브러리로 구축해 나가려는 경우를 대비하여 여기에서는 보다 높은 수준의 주제에 대해 소개할 것이다.

클래스와 비슷한 계승의 구조

앞 절에서는 프로토타입 체인을 이용한 계승에 대해 설명하였다.

프로토타입 체인에 의한 계승은 일단 한 번 이해하기만 하면 그다지 복잡하지는 않으나, 독특한 개념임에는 틀림없다. 특히 클래스 기반의 객체지향에 익숙한 사람에게 있어서는 위화감을 느낄 만한 부분이기도 하다.

그래서 여기에서는 JavaScript로 클래스 기반의 객체지향과 매우 비슷한 방식의 계승을 구현하는 방법에 대해 소개하고자 한다.

구체적인 예제를 보기로 하자. 여기에서는 미리 준비한 Member 클래스를 계승한 SpecialMember 클래스를 정의하고 있다.

리스트 5-17 inherit.html

```
function initializeBase(derive, base, baseArgs) {
  base.apply(derive, baseArgs);     ←❶
  for(prop in base.prototype) {
    var proto = derive.constructor.prototype;
    if(!proto[prop]) {
      proto[prop] = base.prototype[prop];
    }
  }                                 ←❸
}

      // Member 클래스를 정의
```

```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

Member.prototype.getName = function(){
  return this.lastName + ' ' + this.firstName;
};

      // Member 클래스를 계승한 SpecialMember 클래스를 정의
var SpecialMember = function(firstName, lastName, role) {
  initializeBase(this, Member, [firstName, lastName]);
  this.role = role;   ←❷
}

SpecialMember.prototype.isAdministrator = function(){
  return (this.role == 'Administrator');
};

var mem = new SpecialMember('요시히로', '야마다', 'Administrator');
document.writeln('이름 : ' + mem.getName());                    // 야마다 요시히로  ←❹
document.writeln('관리자 : ' + mem.isAdministrator());          // true
```

여기서 주목해야 하는 것은 initializeBase 함수다. initializeBase 함수는 상속 클래스의 생성자 안에서 호출될 것을 가정해서 만든 함수로서 아래와 같은 역할을 제공한다.

- 기저 클래스(base class)의 생성자를 호출한다
- 기저 클래스에서 정의된 멤버를 상속 클래스에 복사한다

initializeBase 함수를 호출하는 구문은 다음과 같다.

구문 initializeBase 함수의 호출

```
initializeBase(현재의 인스턴스, 부모 클래스, 인수 배열)
```

→ 기저 클래스의 생성자를 호출함과 동시에 현재의 클래스에 대해 부모 클래스에서 정의한 멤버를 복사한다. 「인수 배열」은 부모 클래스의 생성자에 건네주는 인수다(인수가 없는 경우는 빈 배열 「[]」을 건네준다)

그럼, 이러한 인수들의 의미를 염두에 두고 initializeBase 함수의 내용을 구체적으로 해석해보도록 하자.

우선 기저 클래스의 생성자를 호출하고 있는 것이 리스트 5-17의 ❶이다. 상속 클래스는 기저 클래스의 초기화 처리에 독자적인 초기화 처리를 추가하는 형태로 초기화하는 것이 일반적이다. 독자적인 처리는 상속 클래스의 생성자에 맡기면 좋기 때문에 initializeBase 함수

에서는 기저 클래스의 생성자를 호출하는 데까지만 행한다.

```
base.apply(derive, baseArgs);
```

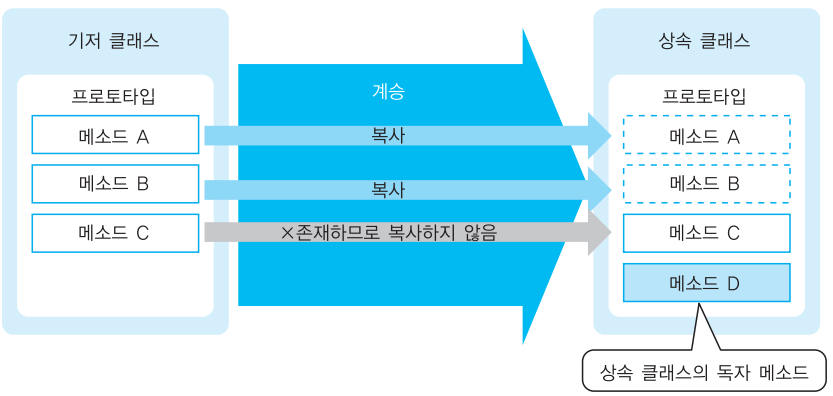
Member　this　[firstName, lastName]

apply 메소드는 Function 객체가 제공하는 메소드 중의 하나로, 함수를 마치 지정된 객체(여기에서는 인수 derive)의 메소드인 것처럼 호출할 수 있다.

여기에서는 인수 derive에 상속 클래스의 생성자로부터 this 키워드가 건네지고 있다. 즉, 지금 작성하려고 하고 있는 상속 클래스(인스턴스)의 메소드로서 부모 클래스의 생성자(Member 함수)를 호출하고 있는 것이다. 메소드(생성자) 호출에 필요한 인수는 apply 메소드의 제2인수로, 배열 리터럴로 지정한다.

참고로, 상속 클래스 독자적인 초기화 처리를 추가하고 싶은 경우에는 ❷와 같이 상속 클래스의 생성자 안에 처리를 기술한다.

그 다음, 기저 클래스에서 정의된 멤버를 상속 클래스에서도 사용할 수 있도록 복사하고 있는 것이 ❸이다. 앞서 이야기한 것처럼 기저 클래스에서 이용 가능한 멤버(메소드)는 프로토타입 객체(prototype 프로퍼티) 밑에 등록되어 있을 것이다. 그래서 여기에서는 이 프로토타입 객체의 내용을 for...in 루프를 이용하여 모두 꺼내 그대로 상속 클래스의 프로토타입 객체에 세트하고 있다. 다만, 상속 클래스에서 동일 이름의 메소드가 이미 정의되어 있는 경우를 고려해, 동일 이름의 메소드가 존재하지 않는 경우에만 복사 처리를 행하도록 한다.



◉ initializeBase 함수의 구조

또한 여기서 대입하려는 곳이 인스턴스의 프로토타입 객체가 아니라 생성자 함수(여기에서는 SpecialMember)의 프로토타입 객체인 점에도 주의해야 한다. 인수 derive에는 상속 클래스의 인스턴스가 세트되어 있을 것이므로 여기에서는 constructor 프로퍼티(133쪽)로 생성자

를 취득하고 있다.

이상으로 계승의 절차가 완료되었다. 리스트 5-17의 ❹를 보면 알겠지만, 상속 클래스에서 확실히 기저 클래스의 멤버를 액세스하고 있음을 확인할 수 있다.

어떠한가? initializeBase 함수를 이용하는 것이 프로토타입 체인을 이용했을 경우에 비해 계승에 있어 꽤 직관적인 조작이라고 느껴지진 않는가?

다만, 이 방법을 이용했을 경우에는 한 가지 주의가 필요하다. 바로 「계승되는 멤버는 인스턴스화될 때 정적으로 정해져 버린다」라는 점이다. 프로토타입 체인 때와 같이 기저 클래스의 프로토타입 객체에 대한 변경을 실시간으로 인식할 수 없음을 유의하라.

■ 부연설명: 기저 클래스의 메소드를 상속 클래스로 이용하려면

상속 클래스로 메소드를 정의하는 경우에는 단지 새로운 메소드를 추가하는 것만이 아닌, 기저 클래스로 정의된 메소드를 오바라이드(덮어쓰기)하는 일도 있다. 또한 그런 경우, 처음부터 그 기능을 구현하는 것이 아니라 기저 클래스에서 정의된 기능을 유용하면서 상속 클래스의 독자적인 기능을 추가하고 싶은 경우도 있을 수 있다.

그러한 때에는 「상속 클래스의 메소드로부터 기저 클래스의 메소드를 호출」하는 행위가 필요하게 된다.

구체적인 예를 보자. 다음은 Member 클래스에서 정의된 getName 메소드를 덮어쓰고, 이름의 말미에 롤(role) 정보를 추가해본 것이다.

리스트 5-18 inherit2.html(발췌)

```
var SpecialMember = function(firstName, lastName, role) {
  initializeBase(this, Member, [firstName, lastName]);
  this.role = role;
}

SpecialMember.prototype.getName = function(){
  var result = Member.prototype['getName'].apply(this, []);  ←❶
  return result + '(' + this.role + ')';
};

var mem = new SpecialMember('요시히로', '야마다', 'Administrator');
document.writeln('이름 : ' + mem.getName());     // 이름:야마다 요시히로(Administrator)
```

포인트가 되는 부분은 ❶이다. 기저 클래스에서 정의된 메소드는 방금 전과 같이 apply 메소드로 호출한다.

구문 기저 클래스의 메소드 호출

```
기저 클래스명.prototype[메소드명].apply(현재의 인스턴스, 인수)
```

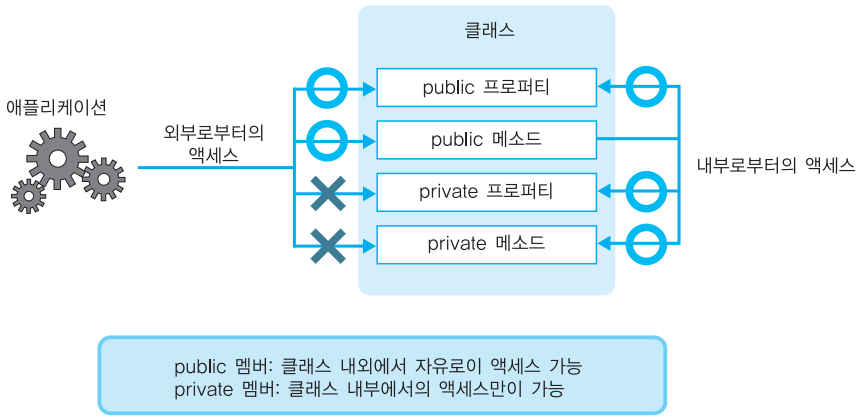
→ 기저 클래스에서 정의된 메소드를 호출한다.

여기에서는 Member.getName 메소드로 얻은 결과를 일단 변수 result에 세트한 다음, 이것을 가공한 것을 새로운 SpecialMember.getName 메소드의 반환값으로 되돌리고 있다.

private 멤버 정의하기

private 멤버란 클래스 내부의 메소드에서만 호출할 수 있는 프로퍼티/메소드를 말한다. 클래스 내부에서만 사용하는 정보나 처리를 정의하고 싶은 경우에는 private 멤버로 정의해 둠으로써 자칫 실수로 외부로부터 액세스될 염려를 하지 않아도 된다.

이와 더불어, 클래스 내외로부터 자유롭게 액세스할 수 있는 멤버(지금까지 정의해 온 멤버들이다)를 public 멤버라 한다. JavaScript에서는 아무런 고려 없이 멤버를 정의하면, public 멤버가 된다.



◉ public 멤버와 private 멤버

JavaScript에서는 엄밀하게는 private 멤버를 정의하기 위한 구문은 없으나, 클로저를 이용함으로써 유사적으로 private 멤버를 정의할 수 있다("유사적"이라고 말한 의미는 나중에 설명하겠다). 구체적인 예제를 보기로 하자.

리스트 5-19 private.html

```
function Triangle() {
    // private 프로퍼티의 정의(밑변/높이를 보존)
  var _base;
  var _height;
    // private 메소드의 정의(인수가 올바른 수치인가를 체크)
  var _checkArgs = function(val) {
    return (!isNaN(val) && val > 0);
  }
    // private 멤버에 액세스하기 위한 메소드의 정의
  this.setBase = function(base) {
    if(_checkArgs(base)){_base = base;}
  }
  this.setHeight = function(height) {
    if(_checkArgs(height)){_height = height;}
  }
  this.getBase = function() { return _base; }
  this.getHeight = function() { return _height; }
}

    // private 멤버에 액세스하지 않는 보통의 메소드를 정의
Triangle.prototype.getArea = function() {
  return this.getBase() * this.getHeight() / 2;
}

var t = new Triangle();
t._base = 10;
t._height = 2;
document.writeln('삼각형의 면적 : ' + t.getArea());  // NaN
t.setBase(10);
t.setHeight(2);
document.writeln('삼각형의 밑변 : ' + t.getBase());    // 10
document.writeln('삼각형의 높이 : ' + t.getHeight());  // 2
document.writeln('삼각형의 면적 : ' + t.getArea());    // 10
```

❶ ❷ ❸

여기서 주목해야 할 포인트는 두 가지다.

(1) private 멤버는 생성자 함수에서 정의한다

private 멤버는 다음의 구문으로 「생성자 함수 안에서」 정의할 필요가 있다.

구문 private 프로퍼티/메소드

```
var 프로퍼티명
var 메소드명 = function([인수]) {/*메소드의 처리*/}
```

이와 같이 private 멤버를 정의하는 경우에는 「this.프로퍼티명=~」, 「this.메소드명=~」과 같이 this 키워드가 아니고 var 키워드로 선언하는 점에 주목해야 한다.

리스트 5-19의 예에서는 _base /_height 프로퍼티, _checkArgs 메소드가 private 멤버가 된다. ❸에서는 private 멤버에 외부로부터 직접 액세스하려고 하고 있으나, 의도한 것처럼 값을 세트할 수 「없다」라는 것을 확인할 수 있다.

(2) 「privileged 메소드」를 정의해 private 멤버에 액세스하기

private 멤버에 액세스할 수 있는 메소드를 privileged 메소드라 한다. privileged 메소드는 「생성자 함수 안에서 정의한다」라고 하는 점을 빼고는 일반적인 메소드와 같은 방법으로 정의할 수 있다.

privileged 메소드는 생성자 안에서 내부 정의된 함수 – 즉, 180쪽에서도 등장한 클로저다. 그 때문에 생성자 함수 안에서 정의된 private 멤버(요점은 로컬 변수)에도 자유롭게 액세스할 수 있다.

앞에서 다루었던 클로저를 생각해보면 알 수 있듯이, 여기에서는 인스턴스에 속한 privileged 메소드(setBase/setHeight/getBase/getHeight)가 각각 private 멤버를 참조하고 있기 때문에 private 멤버는 인스턴스가 존재하는 동안 계속 유지된다. 즉, private 멤버는 「privileged 메소드에 의해서 계속 활용되고 있다」라고도 말할 수 있다.

덧붙여서, privileged 멤버에는 public 멤버나 클래스 외부로부터 자유롭게 액세스할 수 있다.

Note **privileged 메소드는 JavaScript 고유의 생각**

Java나 C#과 같은 객체지향 언어에 익숙한 사람에게 privileged 메소드와 같은 방식은 익숙하지 않을 것이다. Java나 C# 등에서는 같은 클래스 내의 멤버라면 private 멤버에 자유롭게 액세스할 수 있기 때문이다.

privileged 메소드는 클로저에 의해서 유사적으로 private 멤버를 실현하고 있는 JavaScript 고유의 생각이다. 이른바 보통(특권을 가지지 않는다)의 public 멤버로부터 JavaScript의 private 멤버에는 액세스할 수 없다라는 점에 주의해야 한다.

■ 부연설명: 액세서메소드 경유로 프로퍼티를 공개하기

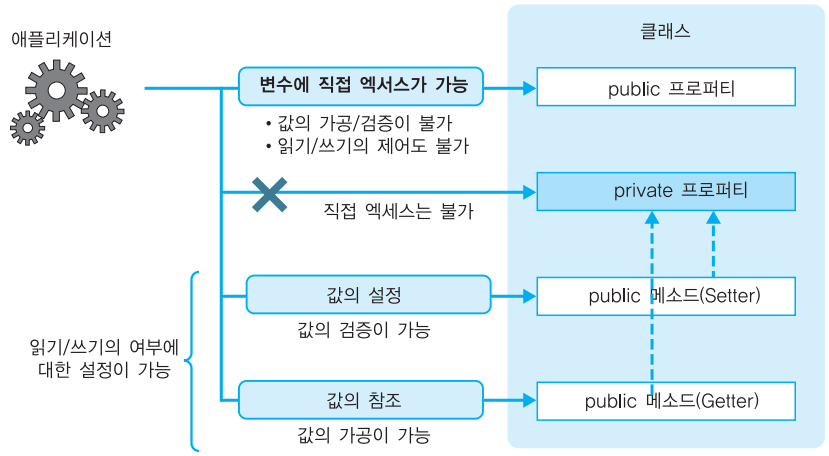
리스트 5-19의 예와 같이 「프로퍼티 그 자체는 클래스 외부로부터 액세스할 수 없게 해두고, 대신에 프로퍼티에 액세스하기 위한 메소드를 준비하는」 케이스는 실제 구현에 있어 자주 있는 코딩 방식이다. 이러한 메소드를 액세서메소드(AccessorMethod)라고 한다(보다 세분화해서 참조용의 메소드를 겟터메소드(GetterMethod), 설정용의 메소드를 셋터메소드(SetterMethod)라고 하는 경우도 있다).

엑세서메소드 경유로 프로퍼티를 공개하는 것은 직접 프로퍼티를 공개하는 것과 비교해 멀리 돌아가는 듯한 느낌이지만, 다음과 같은 경우에 섬세한 제어를 가할 수 있는 이점이 있다.

- 값을 읽기(쓰기) 전용으로 하고 싶다
- 값 참조 시에 데이터를 가공하고 싶다
- 값 설정 시에 타당성을 검증하고 싶다

예를 들어, 겟터메소드만을 준비하고 셋터메소드를 생략하면 프로퍼티는 읽기 전용으로 할 수 있고, 리스트 5-19와 같이 값 설정 시에 인수의 타당성을 체크할 수도 있다.

또 리스트 5-19에서는 단지 private 프로퍼티의 값을 그대로 출력하고 있을 뿐이지만, getBase/getHeight 메소드 안에서 어떠한 가공이나 변환을 행할 수도 있을 것이다.



◉ 엑세서메소드의 역할

엑세서메소드를 도입함으로써 프로퍼티를 보다 안전하게 (그리고 확실하게) 조작할 수 있게 된다.

엑세서메소드는 일반적으로 「get<프로퍼티명>」「set<프로퍼티명>」의 형식으로 명명한다. 그리고 프로퍼티명의 머리 글자는 대문자로 한다.

■ 부연설명: 본래 private 멤버를 사용해야 하는 것인가?

지금까지 봐서 알겠지만, JavaScript에 있어서의 private 멤버의 정의 방법은 약간 독특하다. 원래 다음의 구분은 코드를 이해하기 어렵게 하는 원인이 되기도 할 것이다.

- private 메소드 ➡ 생성자에서
- public 메소드 ➡ 프로토타입 객체에서
- public 메소드에서도 특권을 갖는 것 ➡ 생성자에서

또 「일반적인 퍼블릭 메소드를 privileged 메소드로 하고 싶은 경우에는 메소드 자체의 정의 장소를 이동시키지 않으면 안 된다」라는 문제도 있다(이것은 사소한 것이긴 하지만 의외로 실수의 원인이 된다).

물론, 「그래도 private 멤버를 이용하고 싶다」라는 상황이 있을 거라 생각하지만, 일반적으로는 「모든 것을 public 멤버로 해 버리는」 편이 좋다라고 필자는 생각한다.

대신, 단지 public 멤버로 하는 것으로 끝나는 것이 아니라 private 멤버인 것을 나타내기 위해 프로퍼티/메소드명의 머리에 언더스코어(_)를 붙인다(예를 들어, 「_base」, 「_height」, 「_checkArgs()」와 같이). 어디까지나 이것은 명명상의 관례에 불과하며, 객체 외부로부터의 액세스를 금지하는 것은 아니나, 적어도 이것이 private 멤버인 것을 식별할 수는 있다. 일반적인 용도에서는 이것으로 충분한 케이스도 많을 것이다.

때와 상황에 따라 다르다고 생각하나, 개발의 생산성/보수성, 클래스의 안전성 등을 저울질하여 private 멤버의 사용 여부를 판단하기 바란다.

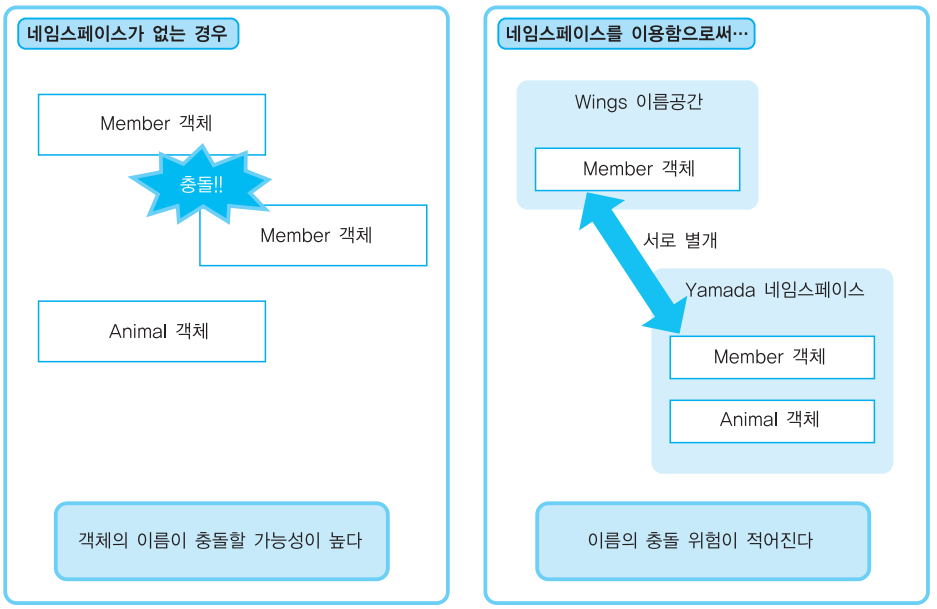
네임스페이스/패키지 작성하기

클래스의 수가 많아지면 「클래스명이 라이브러리끼리 혹은 라이브러리와 라이브러리를 이용하고 있는 애플리케이션에서 충돌하는」 케이스가 발생할 수도 있을 것이다(이름의 충돌에 대해서는 201쪽을 참조하기 바란다).

이것은 당연히 바람직한 일이 아니기 때문에 어느 정도 규모가 있는 클래스 라이브러리를 작성하는 경우에는 이것을 네임스페이스(패키지) 하에서 정리해 둘 것을 추천한다.

네임스페이스(패키지)란 클래스를 정리하기 위한 상자와 같은 것이다. 예를 들어, 「Wings 네임스페이스에 속하는 Member 객체」와 「Yamada 네임스페이스에 속하는 Member 객체」란 서로 별개로 구별 가능하게 된다.

Java나 C#과 같은 언어에서는 이러한 네임스페이스(패키지)의 구조를 표준으로 갖추고 있지만, 유감스럽게도 JavaScript에는 없다. 그래서 JavaScript에서는 빈 객체를 이용해 유사적으로 「네임스페이스와 같은 것」을 작성한다.



◉ 네임스페이스의 역할

리스트 5-20 namespace.html

```
var Wings = function() {};

Wings.Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

Wings.Member.prototype = {
  getName : function(){
    return this.lastName + ' ' + this.firstName;
  }
};

var mem = new Wings.Member('요시히로', '야마다');
document.write(mem.getName());       // 야마다 요시히로
```

네임스페이스를 정의하려 한다면 (앞서 이야기한 것처럼) 단순히 「빈 생성자 함수를 생성」하기만 하면 된다. 여기에서는 ❶에서 Wings 네임스페이스를 정의하고 있다.

그리고서는 Wings 네임스페이스(객체)에 대해 정적 프로퍼티(199쪽)을 추가하는 것과 같은 요령으로 밑에 두고 싶은 클래스(생성자)를 정의할 뿐이다(❷). 이에 의해서 Wings 네임스페

이스 하에 속하는 Member 클래스가 정의되었다.

네임스페이스 하의 클래스를 인스턴스화하는 경우, 네임스페이스도 포함한 이름(완전 수식명)으로서 클래스명을 지정할 필요가 있으므로 주의해야 한다(❸).

Column

기대가 높아지고 있는 차세대 표준사양 「HTML 5」

HTML 5란 그 이름처럼 HTML(HyperText Markup Language)의 최신 버전이다.

이 책의 집필 시점에서는 워킹 드래프트의 단계였지만, 이미 많은 브라우저가 사양 권고에 선행해 주요한 기능을 포함하고 있는 것만 봐도 기대치가 상당하다.

그럼 왜 HTML 5가 그렇게 주목받고 있는 것일까? 다양한 관점이 있지만, 가장 큰 이유는 「HTML 5를 이용하면 보다 고급스런 애플리케이션이 HTML과 JavaScript만으로도 구축할 수 있게 된다」라는 점에 있다.

구체적으로는 다음과 같은 HTML 5의 주된 기능이 있다(엄밀하게는 사양으로서 HTML 5 본체에 포함되지 않는 것도 있으나, 넓은 의미에서 HTML 5 관련 API라고 정리해 취급하였다).

- **문서 구조를 보다 명확하게 정의하는 〈article〉, 〈section〉, 〈navi〉 요소**
- **동영상, 음성을 재생하기 위한 〈video〉, 〈audio〉 요소, 조작하기 위한 Video, Audio 객체**
- **입력 지원이나 검증 기능을 갖춘 새로운 폼 요소**
- **JavaScript로부터 동적으로 이미지를 그릴 수 있는 〈canvas〉 요소**
- **크로스 도메인 통신에 대응한 XMLHttpRequest 객체**
- **드래그 & 드롭, 로컬 파일 로딩에 대응한 File API**
- **백그라운드에서의 JavaScript 실행을 가능하게 하는 Web Workers**
- **서버와의 쌍방향 통신을 가능하게 하는 Web Socket API**
- **위치 정보를 취득하기 위한 Geolocation API**
- **브라우저에 로컬 데이터를 보존하기 위한 Web Storage**

사양 권고 전이라는 점에서 이 책에서는 HTML 5에 관련하는 JavaScript의 기능에 대해 다루지 않고 있지만, HTML 5는 향후 확실히 보급될 기술이다. 서적, 인터넷을 불문하고 정보 또한 많아지고 있으므로 서둘러 정보를 수집해 두기를 권한다.