12

객체와 클래스

이 장에서는 객체지향 프로그래밍(OOP, object-oriented programming)이라는 것을 다룬다. 객체지향 프로그래밍이라는 단어가 생소하게 느껴지는 독자들이 많으리라 생각한다. 우선 몇 가지 용어에 대해 알아보고 그 용어를 확실히 이해할 수 있도록 예제를 살펴보기로 하자. OOP는 뭔가 신비롭고 이해하기 어려운 것이라는 얘기를 들어본 독자들도 많을 것이다. 하지만 그와는 반대로 OOP의 개념은 매우 직관적이고 OOP를 이용한 작업은 많은 독자들이 생각하는 것보다 훨씬 쉽다. OOP의 핵심은 프로그램의 일부를 독립적인 객체로 다룬다는 점이다. 실생활에서 다루는 거의 모든 것이 사실 독립적인 객체에 해당한다는 점을 생각해 보면 이해하기 쉽다. 애완견, 부모님, 자동차, 컴퓨터 등은 모두 독립적인 객체라고 볼 수 있다. 독자적으로 어떤 일을 하기도 하고 그 안에서 어떤 일이 일어나는지 잘 모르더라도 뭔가를 부탁하면 그 일을 처리해줄 수 있다는 면에서 독립적이라고 부를 수 있다.

막대를 던져 놓고 애완견에게 물어오라고 한다고 해서 꼭 애완견의 내부 구조를 잘 알아야 할 필요는 없으며, 자동차를 몰기 위해서 자동차 엔지니어일 필요도 없다. 정신분석가가 아니더라도 부모님과 대화를 할 수 있고 컴퓨터를 전공하지 않았더라도 이메일을 확인할 수 있다. 사용자는 객체에서 사용할 수 있는 명령어(메소드라고 부른다)와 그러한 명령어에 의한 결과만 알면 된다. 예를 들어 자동차에서 엑셀레이터를 밟으면 자동차가 가속된다는 것을 알고 있다. 애완견에게 앉으라고 말하면 애

완견이 앉을 것이라는 사실을 알고 있다. 이제 상식적인 수준에서 실생활에서의 객체가 무엇인지를 이해했다면 그러한 개념을 액션스크립트에도 적용시켜보자.

프로그래밍 객체의 좋은 예로 튀어 다니는 공을 생각할 수 있다. 실제 공과 마찬가지로 공 객체에는 공의 반지름, 색, 질량, 위치, 튀는 정도(탄성)와 같이 그 공의 특징을 나타내는 속성이 있다. 프로그램에서 공을 나타낼 때는 radius(반지름)와 같은 속성을 지정하여 ball 객체를 만들면 된다. 객체의 속성은 그 객체의 상태를 나타내며, 속성 중 어떤 것은 시간에 따라 변하기도 한다. 예를 들어 공을 움직이게 하려면 뉴턴의 운동법칙을 적용해야 한다. 즉 공이 시간에 따라 움직이는 방법을 컴퓨터에서 쓸 수 있는 용어로 기술해야 한다. 가장 간단한 경우를 생각하면 속력에 시간을 곱하면 거리가 된다는 것을 이용하여 다음과 같은 간단한 공식을 이용해 공의 수평 위치를 알아낼 수 있다.

```
ball.xPosition += ball.xVelocity * (elapsedTime)
```

위 식에서는 공의 위치에 그 공이 움직인 거리(속도와 경과된 시간을 곱한 값)를 더하여 새로운 위치를 구한다. 객체의 행동 양식은 위에서 구한 공이 시간에 따라 움직이는 위치를 계산하는 식과 같이 어떤 객체를 지배하는 규칙을 의미한다. 일반적으로 이러한 행동 양식을 '메소드(method)'라는 것으로 포장하는데, 메소드란 어떤 객체의 행동양식을 구현하는 함수에 지나지 않는다. 예를 들어 ball 객체에 위에 있는 식을 이용하는 move()라는 메소드를 만들 수 있다.

따라서 메소드는 어떤 객체에 적용시킬 수 있는 명령이라고 볼 수 있다. 물론 객체에는 여러 개의 메소드가 들어갈 수 있다. 공이 튀도록 만들어보자. 공의 운동방향을 반대로 뒤집고 속도를 줄이는(공이 완전 탄성체가 아닌 경우) bounce()라는 메소드를 만들 수 있다. bounce() 메소드는 다음과 같은 식으로 구현할 수 있다.

```
ball.xVelocity =  -(ball.xVelocity) * 0.95        // 탄성 계수가 95%인
                                                  // 공의 경우
```

객체를 만들고 속성을 추가하고 메소드를 구현하는 복잡한 내용을 본격적으로 다루기 전에 몇 가지 정의를 확실히 짚고 넘어가도록 하자. '객체(object)'는 기술적으로 본다면 그 객체와 관련된 '속성(property)'과 메소드(함수)를 모아놓은 자료구조이다. 객체에서는 보통 그 행동 양식을 '캡슐화(encapsulation)'한다. 캡슐화라는 용어는 그 객체의 기능을 수행하는 자세한 내용을 객체 밖에서는 알지 못하게 하

는 것을 의미한다. 프로그램에서는 이와 같이 캡슐화된 객체를 인터페이스(객체 밖으로 공개된 메소드)라는 것을 통해 다루게 된다. 객체 이외의 부분에서는 객체 안에서 어떤 일을 어떻게 처리하는지 걱정할 필요가 없다. 대신 프로그램에서 객체에 어떤 것을 입력하고 가능한 경우에는 그 입력에 따른 출력(결과)을 확인하기만 하면 된다. '클래스(class)'와 '인스턴스(instance)'라는 말도 많이 사용할 것이다. 클래스는 일반적인 객체의 범주를 뜻하며, 인스턴스는 특정한 객체의 복사본을 의미한다. 예를 들어 애완견이 한 마리 있다면 그 애완견은 Dog라는 클래스의 인스턴스가 된다. Dog 클래스에 속하는 모든 개는 짖을 줄 알고 네 개의 다리를 가지고 있지만, 특정 인스턴스별로 키, 몸무게, 색깔과 같이 그 개의 특징을 나타내는 서로 다른 속성을 가지게 된다.

객체지향 프로그래밍(OOP)은 객체를 사용하는 프로그래밍을 일컫는 용어일 뿐이다. 객체와 OOP는 액션스크립트에서 워낙 많이 쓰이기 때문에 독자 자신도 모르는 사이에 많이 사용했을 것이다. 무비 클립은 플래시에서 많이 쓰이는 객체이며 다른 객체와 마찬가지로 속성과 메소드의 모음으로 구현된다. someClip._height를 이용하여 무비 클립의 높이를 알아낼 때는 클립 객체의 _height 속성을 사용한다. someClip.play()를 이용하여 어떤 무비 클립을 재생할 때는 클립 객체의 play() 메소드를 호출한다.

보통 어떤 객체 클래스의 모든 인스턴스는 같은 메소드와 속성 이름을 공유한다. 대신 각 인스턴스는 같은 클래스에 있는 다른 인스턴스와 다른 속성 값을 가진다.

객체를 직접 만들거나 액션스크립트에 내장된 객체를 사용하거나 상관없이 OOP를 이용하면, 프로그램의 구성요소를 다른 구성요소와 깨끗하게 분리(캡슐화)할 수 있으며, 객체끼리 상대방의 내부 동작을 전혀 모르더라도 서로 상호작용할 수 있다. 이렇게 하면 어떤 메소드 자체가 바뀌지만 않으면(즉, 외부와의 인터페이스가 바뀌지만 않는다면) 객체를 사용하는 프로그램의 다른 부분에는 영향을 미치지 않으면서 객체의 내부 기능을 변경할 수 있다. 예를 들어 ball 객체를 다시 생각해 보면 물리 법칙에 의해 공이 움직이는 양식이 바뀐다고 하더라도 프로그램의 다른 부분은 고치지 않아도 된다. 단지 공의 move() 메소드를 호출하기만 하면 객체 내부에서 어떤 일이 일어나더라도 상관없다.

OOP의 또 다른 장점으로 서로 다른 행동 양식을 가지는 서로 다른 객체도 같은 이름을 가진 메소드를 구현한다면 똑같은 방법으로 다룰 수 있다는 점을 들 수 있다. 예를 들어 circle 객체와 square 객체가 있다고 가정하자. 두 객체에서 모두 도형의 넓이를 리턴하는 area()라는 메소드를 구현한다면, 각 객체에서 넓이를 계산할 때는 다른 것은 신경쓸 필요 없이 area() 메소드를 호출하기만 하면 된다.

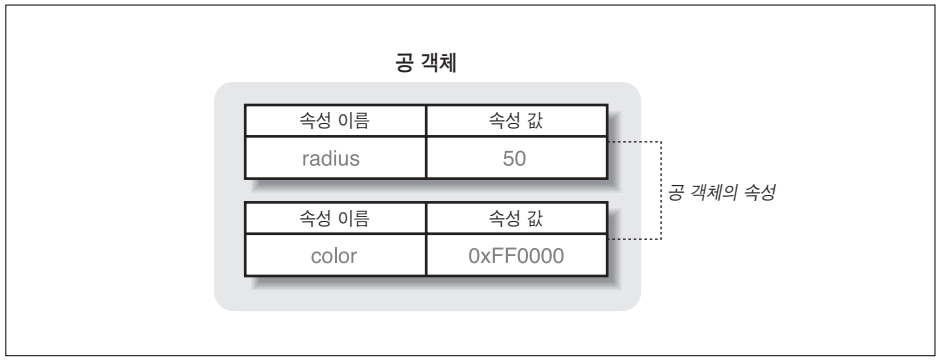
이 장에서는 기본적인 객체를 만드는 법과 객체(즉 클래스)의 범주를 정의하는 법에 대해 살펴본다. 일단 기초적인 내용을 끝내고 나면 '상속(inheritance)'을 이용하여 클래스와 객체 사이에서 같은 특성을 공유하는 방법(즉 가계도(family tree)를 만드는 방법)을 알아보자. 예를 들어 하나의 Mammal(포유류) 클래스에서 Horse(말)와 Dog(개) 클래스를 만들 수 있다. Mammal 클래스에서는 모든 포유류에 공통적인 메소드와 속성(털이 있다는 점, 젖을 먹인다는 점, 항온동물이라는 점 등)을 구현할 수 있다. 마지막으로 액션스크립트의 내장 객체와 클래스를 통해 플래시 환경을 제어하는 데 OOP가 어떻게 쓰이는지 살펴보기로 하자.

지금까지 설명한 내용을 통해 객체와 OOP에 대한 기초적인 개념을 이해할 수 있기 바란다. 이제 자세한 내용을 본격적으로 알아보자.

객체 구조

배열과 마찬가지로 객체는 컨테이너의 컨테이너이다. 배열에서는 서로 다른 원소에 여러 개의 데이터 값을 저장한다. 객체에서는 서로 다른 속성에 여러 개의 데이터 값을 저장한다. 하지만 객체의 속성은 번호 인덱스를 사용하지 않고 각 속성마다 서로 다른 이름을 가진다. 배열에서는 원소들을 번호가 있는 리스트를 이용하여 저장하지만, 객체에서는 속성을 따로 순서가 정해지지 않은 고유의 인식자를 이용하여 저장한다. 배열 원소를 액세스할 때는 인덱스 번호를 알아야 하지만, 객체 속성을 액세스하려면 이름(인식자)을 알아야 한다.

〔그림 12-1〕에 ball이라는 가상적인 객체(공 객체)의 속성을 나타내 보았다. ball 객체에서는 radius(반지름)와 color(색)라는 두 가지 속성이 저장된다. 이 두 속성 값은 각각 50과 0xFF0000(빨간색의 16진수값)이다. 각 속성에는 변수와 비슷한 식으로 각기 고유의 이름이 붙어있다. 각 속성의 이름은 서로 다르지만 그 속성들은 모두 ball이라는 하나의 객체에 포함된다.



〔그림 11-1〕 객체 구조의 예

물론 객체에서는 그 객체와 연관된 의미를 가지는 속성을 정의한다. 또한 객체의 속성은 어떤 한 객체와 다른 객체를 서로 구분하는 데 도움이 될 수 있도록 골라야 한다. 예를 들어 무비 클립 객체에는 프레임 수(_totalframe)나 위치(_x와 _y)와 같이 무비 클립에만 있는 속성들이 포함되어 있다.

객체 속성에는 번호가 아닌 이름이 붙어있으므로 배열에서 사용하는 원소관리 도구(shift(), unshift(), push(), splice() 등)는 객체에는 포함되지 않는다. 객체 속성은 객체의 캡슐화를 위해 객체의 메소드를 이용하여 설정하는 것이 일반적이다. 즉 엄격한 OOP 개념에서 보면 객체 자체에서 직접 속성을 설정해야 한다. 객체 바깥에서 그 객체의 속성을 설정하고 싶다면 객체에 있는 적절한 메소드를 호출해서 객체 속성을 설정해야 한다. 예를 들어 원칙을 중시하는 프로그래머라면 배열의 length 속성을 직접 설정하는 것을 싫어할 것이다. 대신 Array 객체의 메소드를 호출하는 방식으로 Array 객체에서 length 속성을 직접 설정하여 객체 밖에서 객체 안의 속성을 변경할 때는 간접적인 방법만을 사용하는 쪽을 선호할 것이다. 그렇게 하면 다른 사용자들에게는 영향을 미치지 않으면서 Array 클래스를 관리하는 사람이 length 속성의 이름을 len으로 바꿀 수 있다.

객체 인스턴스 만들기

아직 배우진 않았지만 일단 어떤 객체 클래스를 만들었다고 가정해 보자. 객체를 배우는 법을 아직 배운 것은 아니지만, 액션스크립트에서는 여러 내장 클래스를 제공하며 우리가 만들게 될 사용자 정의 클래스도 결국 비슷한 식으로 작동하므로 인스턴스 만드는 법을 먼저 배우는 것도 괜찮다. 어떤 객체 클래스가 있다면, 그 클래스를 기반으로 하는 특정 객체 인스턴스(즉 복사본)를 만들어야 한다. 예를 들어 액션스크립트에서는 Array 객체 클래스를 지원하지만 각 배열을 만드는 것은 사용자가 직접 해야 한다.

객체의 인스턴스를 만들려면 new 연산자와 생성자 함수(객체를 초기화시키는 함수)를 사용하면 된다. 일반적인 문법은 다음과 같다.

```
new ConstructorFunction()
```

액션스크립트에 내장된 생성자인 Object()를 이용하여 객체의 인스턴스를 만들어 보자. Object() 생성자에서는 그 이름에서 알 수 있듯이 다른 모든 객체의 기반이 되는 기본적인 객체 유형인 범용 객체를 만든다. 다음과 같은 코드를 사용하면 새로운 범용 객체가 만들어진다.

```
new Object()
```

객체의 인스턴스를 만들면 보통 그 결과로 만들어진 인스턴스를 나중에 사용할 수 있도록 변수, 배열 원소, 또는 객체 속성 등으로 저장한다. 예를 들면 다음과 같다.

```
var myObject = new Object();
```

이처럼 객체의 인스턴스를 만들면 속성이 하나도 없고 두 개의 일반적인 메소드만 들어 있는 비어있는 객체가 만들어진다. 따라서 범용 객체는 그다지 쓸모가 없고 객체의 진가는 특화된 객체 클래스에서 발휘된다. 클래스를 직접 만드는 법을 배우기 전에 이미 만들어진 클래스의 속성과 메소드를 사용하는 법을 먼저 알아보기로 하자.

객체 속성

속성은 객체와 연관된 이름이 있는 데이터 컨테이너이다. 속성은 객체의 클래스에 의해 정의되며 각 객체 인스턴스에서 별도로 그 값이 설정된다. 변수와 마찬가지로 객체 속성에는 어떤 데이터라도 저장할 수 있다(문자열, 숫자, 부울, null, undefined, 함수, 배열, 무비 클립, 심지어 다른 객체를 저장할 수도 있다).

속성 참조

이미 익숙해진 점 연산자를 이용하면 객체 속성에 액세스할 수 있다. 다음과 같이 속성 이름과 객체 이름 사이에 점을 찍으면 된다.

```
objectName.propertyName
```

objectName은 객체의 이름이고 propertyName은 objectName에 들어있는 속성의 이름과 같은 인식자이다.

예를 들어 radius라는 속성을 가지는 ball 객체가 있다고 가정하면, 다음과 같은 표현식을 이용하여 radius를 액세스할 수 있다.

```
ball.radius
```

또는 다음과 같이 [] 연산자를 이용하여 속성을 참조하는 것도 가능하다.

```
objectName[propertyName]
```

[] 연산자를 이용하면 propertyName 자리에 문자열 값을 가지는 어떤 표현식이라도 사용할 수 있다. 예를 들면 다음과 같은 표현식도 가능하다.

```
trace(ball["radius"]);

var prop = "radius";
trace(ball[prop]);  // prop은 "radius"가 된다.
```

액션스크립트의 내장 속성도 똑같은 식으로 액세스할 수 있다. 파이(π) 값을 구하는 다음과 같은 표현식을 다시 생각해 보자.

```
Math.PI
```

이 표현식에서는 Math 객체에 내장된 PI 속성을 액세스한다. 하지만 순수한 OOP에서는 객체 속성을 직접 액세스하는 일은 거의 없다. 대신 메소드를 이용하여 속성 값을 액세스하게 된다. 예를 들어 Sound라는 내장 클래스의 volume 속성을 확인할 때는

```
trace(mySound.volume);
```

이 아니라

```
trace(mySound.getVolume());
```

을 사용한다.

for-in 루프를 이용한 객체 속성 액세스

'8장. 순환문'에서 for-in 루프를 사용하면 모든 객체 속성에 대해 루프를 돌릴 수 있다는 것을 배웠다. 이제 객체에 대해 조금 더 배웠으므로 객체 속성을 조작하는 데 for-in 선언문을 사용하는 법에 대해 다시 알아보는 것이 좋겠다.

다른 모든 루프와 마찬가지로 for-in 선언문에도 헤더와 본체가 있다. for-in 선언문의 본체는 주어진 객체의 각 속성에 대해 한 번씩 자동으로 실행된다. 속성 이름이나 개수를 알 필요는 없다. 매번 루프가 실행될 때마다 반복자 변수가 자동으로 그 속성의 이름이 되기 때문이다. 따라서 다음과 같이 객체의 모든 속성을 액세스할 수 있다.

```
// ball 객체의 모든 속성을 출력한다.
for (var prop in ball) {
  trace("Property " + prop + " has the value " + ball[prop]);
}
```

위 예에서 반복자 변수로 쓰인 prop은 for 루프에서처럼 정수가 아니다. 일반적으로 숫자 인덱스를 사용하는 배열 원소를 액세스할 때 쓰이는 표준적인 for 루프와 객체 속성을 액세스하기 위한 for-in 루프를 혼동하지 않도록 주의하자. for-in 루프에 대한 자세한 내용은 8장을 참조하기 바란다.

메소드

메소드는 어떤 객체와 연관된 함수이며 일반적으로 객체의 데이터를 액세스하거나 어떤 작업을 수행하는 데 쓰인다. 메소드를 호출할 때는 점 연산자를 이용하여 객체의 이름과 메소드의 이름을 구분하고 메소드 이름 뒤에 함수 호출 연산자를 사용한다.

```
objectName.methodName()
```

예를 들면 다음과 같이 쓸 수 있다.

```
ball.getArea();  // ball 객체의 getArea() 메소드를 호출한다.
```

속성과 마찬가지로 메소드는 클래스에서 정의되며 각 객체 인스턴스에서 그 메소드를 호출한다. 하지만 클래스에서 메소드를 만드는 법을 배우기 전에 좋은 객체 지향 프로그래밍 기법은 잠시 접어두고 내장 Object 클래스의 객체 인스턴스에 메소드를 하나 추가해 보자. 개별 객체에서 한 메소드가 동작하는 법을 이해하고 나면 그 개념을 새로 만드는 클래스에 대해서도 적용할 수 있다.

메소드는 사실 어떤 객체 속성에 저장된 함수에 지나지 않는다. 객체 속성에 함수를 저장하면 그 함수가 객체의 메소드가 된다.

```
// 객체를 만든다.
myObject = new Object();

// 함수를 만든다.
function greet () {
  trace("hello world");
}

// sayHello라는 속성에 greet 함수를 대입한다.
myObject.sayHello = greet;
```

어떤 함수를 객체 속성에 대입하고 나면 그 함수를 다음과 같이 메소드로 호출할 수 있다.

```
myObject.sayHello();  // "hello world"가 출력된다.
```

배열에서도 이와 마찬가지로 원소에 함수를 대입하면 다음과 같이 함수 호출 연산자를 이용하여 그 함수를 호출할 수 있다.

```
var myList = new Array();
myList[0] = greet;
myList[0]();          // "hello world"가 출력된다.
```

하지만 어떤 함수를 객체의 메소드로 호출하면 그 함수가 들어있는 객체의 속성 값을 구하거나 그 속성에 어떤 값을 대입할 수 있는 특별한 기능이 생긴다. sayHello 속성을 이용하여 이러한 기능이 어떤 식으로 돌아가는지 알아보자. 우선 범용 객체에 msg 속성을 추가해 보자(지금은 독자들에게 예를 보여주기 위해 OOP의 규칙을 어기고 있는 셈이다. 개별 객체 인스턴스에 사용자 정의 속성을 추가하는 것은 그다지 권장할만한 방법이 아니다).

```
myObject.msg = "Nice day, isn't it?";
```

이제 greet() 함수를 고쳐서 msg 값을 출력하도록 만들어 보자.

```
function greet () {
  trace(this.msg);
}

// sayHello를 다시 호출한다.
myObject.sayHello();  // "Nice day, isn't it?"이 출력된다.
```

둘째 줄의 뒷부분을 보면 this라는 키워드가 보일 것이다. 바로 이 키워드가 myObject를 가리키는 키워드이다. sayHello()를 myObject의 메소드로 실행하면 그 메소드에는 myObject를 가리키고 있는 보이지 않는 인자인 this가 전달된다고 볼 수 있다. 함수의 본체에서는 this.msg와 같이 this를 이용하여 myObject의 msg 속성을 액세스할 수 있다. myObject.sayHello()라고 하면 this.msg는 myObject.msg로 해석되어 결국 "Nice day, isn't it?"이라는 문자열이 된다.

메소드에서는 그 메소드의 호스트 객체의 속성 값을 구하거나 그 속성 값을 설정할 수 있다. [예제 12-1]에서는 객체의 두 속성 값을 알아보고 그 두 값을 기반으로 계산을 처리한 다음 세 번째 새로운 속성 값을 설정한다(이번에도 일단 메소드에 대해 알아보기 위해 객체 인스턴스를 사용한다. 클래스에 객체 메소드를 추가하는 방법은 다음 절에서 자세히 배워보도록 하자).

〔예제 12-1〕 속성 값을 설정하는 메소드 구현

```
// 새로운 객체를 만들고 rectangle에 그 객체를 저장한다.
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;

// rectangle(직사각형)의 면적(area)을 구하는 함수를 정의하고
// 그 속성(area)을 설정한다. this 키워드의 사용법을 자세히 보자.
function rectArea() {
  this.area = this.width * this.height;
}

// 함수를 rectangle의 메소드로 대입한다.
rectangle.setArea = rectArea;

// rectangle의 setArea() 메소드를 호출한다.
rectangle.setArea();    // rectangle.area가 50이 된다.

// 마지막으로 setArea()에 의해 만들어진 새로운 속성을 확인한다.
trace(rectangle.area);  // 50이 출력된다.
```

일반적으로 메소드에서는 어떤 값을 설정하기보다는 리턴하는 경우가 더 많다. 메소드에서 값을 리턴하도록 setArea() 메소드를 수정한 예가 〔예제 12-2〕에 나와 있다.

```
// rectangle을 만들고 설정한다. 코드에 rectArea 함수가 나오기 전에
// area 메소드에 rectArea 함수를 대입해도 괜찮다.
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;
rectangle.area = rectArea;

// 이번에는 어떤 속성에 대입하지 않고 넓이를 그냥 리턴한다.
function rectArea() {
  return this.width * this.height;
}

// 이제 사용하기도 더 쉽다.
trace("The area of rectangle is: " + rectangle.area());

// "The area of rectangle is: 50"이 출력된다.
```

this 키워드를 인터프리터에서 해석할 때는 약간의 트릭이 필요하기 때문에 생각보다 그 개념이 까다로울 수도 있다. 위 코드가 작동하는 과정을 이해하기 위해 this와 객체, 메소드와의 관계를 보여주는 〔예제 12-3〕을 살펴보자(사실 〔예제 12-3〕을 그대로 실행시키면 오류가 발생한다. this는 예약된 키워드이므로 이 예제에 나온 것처럼 사용할 수 없다).

〔예제 12-3〕 this를 직접 사용하는 가상 예제

```
// rectangle 객체를 만들고 설정한다.
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;
rectangle.area = rectArea;

// rectArea() 함수에서 this를 직접 인자로 사용한다.
function rectArea(this) {
  return this.width * this.height;
}

// 메소드를 호출할 때 rectangle 객체의 레퍼런스를 직접 전달한다.
trace("The area of rectangle is: " + rectangle.area(rectangle));
```

메소드는 함수 리터럴로 객체 속성에 대입할 수도 있다. 함수 리터럴을 이용하여 메소드를 만들면 함수를 먼저 만들고 나서 속성에 대입하지 않아도 된다. 〔예제 12-4〕에는 〔예제 12-2〕를 함수 리터럴을 사용하도록 고친 예가 나와 있다.

〔예제 12-4〕 함수 리터럴을 이용한 메소드 구현

```
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;

// 함수 리터럴을 대입하는 부분
rectangle.area = function () {
  return this.width * this.height;
};  // 리터럴의 끝에는 세미콜론을 사용해야 한다.

// 함수를 호출할 때는 이전과 같은 코드를 사용하면 된다.
trace("The area of rectangle is: " + rectangle.area());
```

메소드를 이용하면 객체를 자유자재로 제어할 수 있다. 내장 객체의 메소드는 플래시 무비 환경을 제어하는 데 가장 많이 쓰이는 도구이다.

이 절에서는 속성에 함수를 저장하는 법(메소드를 만드는 법)과 속성을 통해 함수를 호출하는 법에 대해 알아보았다. 이 두 가지 문법은 그다지 차이가 나지 않지만, 그 결과는 전혀 다르다. 그 차이점을 꼭 이해하고 넘어가기 바란다.

속성에 메소드를 대입할 때는 괄호를 사용하지 않는다.

```
myObj.myMethod = someFunction;
```

메소드를 호출할 때는 괄호를 사용한다.

```
myObj.myMethod();
```

문법이 조금 다르다. 이 둘 사이의 차이점을 확실히 이해하고 넘어가도록 하자.

클래스와 객체지향 프로그래밍

프로그래밍을 하다보면 쇼핑몰 시스템에 들어갈 제품에서 비디오 게임에 들어가는 인공지능을 갖춘 악당에 이르기까지 다양한 정보를 저장하기 위한 복잡한 객체를 만들어야 하는 경우도 종종 있다. 객체를 빠르게 만들고 객체 계층(객체간의 관계)을 정의하기 위해 객체 클래스를 사용한다. 클래스는 객체의 모든 범주를 템플릿 스타일로 정의한 것이다. 도입부에서 배웠듯이 클래스는 '모든 개의 다리는 네 개이다'와 같이 어떤 객체 종류의 일반적인 특징을 기술한다.

Object 클래스

클래스를 사용하는 법을 배우기 전에 클래스를 사용하지 않으면 어떻게 될지 생각해 보자. ball이라는 객체를 만드는 데 클래스를 이용하여 만들지 않고 내장된 Object 클래스라는 범용 객체를 이용하는 경우를 생각해 보자. ball 객체에 radius, color, xPosition, yPosition이라는 속성을 추가한다. 그리고 객체의 위치를 변경하

고 그 객체가 차지하는 영역의 넓이를 구하기 위한 moveTo() 메소드와 area() 메소드를 추가하자.

코드는 다음과 같다.

```
var ball = new Object();
ball.radius = 10;
ball.color = 0xFF0000;
ball.xPosition = 59;
ball.yPosition = 15;
ball.moveTo = function (x, y) { this.xPosition = x; this.yPosition = y; };
ball.area = function () { return Math.PI * (this.radius *
this.radius); };
```

이렇게 해도 필요한 작업을 처리할 수는 있지만 한계가 있다. ball과 같은 객체를 만들 때마다 일일이 ball을 초기화하는 코드를 반복해야 하는데, 이러한 작업은 귀찮을 뿐만 아니라 오류가 생기기도 쉽다. 게다가 이러한 식으로 여러 개의 ball 객체를 만들면 불필요하게 똑같은 moveTo()와 area() 코드를 반복해야 하므로 쓸데없이 메모리를 낭비하게 된다.

같은 속성을 가지는 일련의 객체를 효율적으로 만들기 위해서는 클래스를 사용해야 한다. 클래스를 이용하면 모든 ball 객체에 필요한 속성을 정의할 수 있다. 게다가 ball 클래스의 인스턴스에서 모두 같은 고정된 값을 사용하는 속성을 공유할 수도 있다. 사람들이 하는 말로 쓴다면 ball(공) 클래스는 다음과 같이 설명할 수 있다.

> ball은 객체의 일종이다. 모든 ball 객체의 인스턴스에는 radius, color, xPosition, yPosition이라는 속성이 들어가며 이 속성들은 각 ball 인스턴스에서 개별적으로 설정된다. 모든 ball 객체에서는 moveTo()와 area() 메소드를 공유하는데 이 메소드는 Ball 클래스의 모든 멤버에서 똑같은 코드로 쓰인다.

이제 이러한 이론적인 내용이 실전에서 어떻게 쓰이는지 알아보자.

클래스 만들기

액션스크립트에는 특별히 정해진 '클래스를 선언하기 위한 도구'가 없다. 새로운 변수를 선언하기 위한 var 선언문과 같이 클래스를 선언하기 위한 class 선언문

같은 것이 있는 것도 아니다. 대신 클래스의 새로운 인스턴스를 만들어주는 '생성자(constructor)' 함수라는 특별한 종류의 함수를 정의한다. 생성자 함수를 정의하면 클래스 템플릿 또는 클래스 정의를 만드는 셈이 된다.

생성자 함수(또는 줄여서 생성자라고도 함)는 다음과 같이 일반적인 함수를 만드는 것처럼 만들면 된다.

```
function Constructor () {
  statements
}
```

클래스 생성자 함수의 이름은 함수 이름으로 사용할 수 있는 것이면 다 쓸 수 있지만 그 함수가 클래스 생성자임을 나타내기 위해 첫째 글자를 대문자로 사용하는 것이 일반적이다. 생성자의 이름은 Ball, Product, Vector2d와 같이 그 생성자에서 만드는 객체의 클래스를 잘 설명할 수 있는 것이어야 한다. 생성자 함수의 statements 자리에서는 객체를 초기화한다.

우선 Ball 생성자를 최대한 간단하게 만드는 것부터 시작해 보자. 우선 다음과 같이 비어있는 객체를 만들어 보자.

```
// Ball 생성자를 만든다.
function Ball () {
  // 여기에서 뭔가 작업을 처리한다.
}
```

그다지 어렵지 않다. 이제 Ball 생성자 함수를 이용하여 ball 객체를 만드는 법을 알아보자.

클래스 멤버 만들기

생성자 함수는 클래스를 정의하면서 동시에 클래스의 인스턴스를 새로 만드는 데도 쓰인다. 앞에서 배운 것처럼 new 연산자와 생성자를 호출하면 생성자 함수에서는 객체 인스턴스를 만들어서 리턴한다. 생성자 함수를 이용하여 새로운 객체를 만드는 일반적인 문법을 다시 떠올려보자.

```
new Constructor();        // Constructor 클래스의 인스턴스를 리턴한다.
```

Ball 클래스를 이용하여 ball 객체(인스턴스)를 만들려면 다음과 같이 하면 된다.

```
myBall = new Ball();      // myBall에 Ball 클래스의 인스턴스를 저장한다.
```

하지만 위에서 만든 Ball 클래스에서는 그 클래스에서 만드는 객체에 속성이나 메소드를 추가하지 않는다. 이제 속성과 메소드를 추가하는 법을 알아보자.

클래스 객체에 사용자 정의 속성 대입하기

객체 생성 단계에서 객체 속성을 만들 때도 this 키워드를 사용한다. 생성자 함수 내에서 this 키워드에는 현재 만들고 있는 객체의 레퍼런스가 저장된다. 이 레퍼런스를 이용하여 객체에 원하는 속성을 추가할 수 있다. 일반적인 방법은 다음과 같다.

```
function Constructor () {
  this.propertyName = value;
}
```

this는 지금 만들고 있는 객체이고 propertyName은 객체에 추가하고자 하는 속성이며 value는 그 속성에 대입할 값이다.

이러한 기술을 Ball 클래스 예제에 적용해 보자. Ball 클래스에는 radius, color, xPosition, 그리고 yPosition 속성을 집어넣는다.

Ball 클래스 생성자에서 인스턴스에 속성을 대입하는 방법은 다음과 같다(this 키워드를 사용하는 법을 자세히 살펴보자).

```
function Ball () {
  this.radius = 10;
  this.color = 0xFF0000;
  this.xPosition = 59;
  this.yPosition = 15;
}
```

이렇게 만든 Ball 생성자를 이용하면 앞에서 한 것처럼 new 연산자와 함께 Ball() 생성자를 호출하기만 하면 미리 정의된 속성(radius, color, xPosition, yPosition)이 들어있는 객체의 인스턴스(클래스 멤버)를 만들 수 있다. 예를 들면 다음과 같다.

```
// 새로운 Ball 인스턴스를 만든다.
bouncyBall = new Ball();

// Ball() 생성자를 이용해서 인스턴스를 만들 때 설정된
// 속성 값을 알아본다.
trace(bouncyBall.radius);     // 10이 출력된다.
trace(bouncyBall.color);      // 16711680이 출력된다.
trace(bouncyBall.xPosition);  // 59가 출력된다.
trace(bouncyBall.yPosition);  // 15가 출력된다.
```

하지만 이렇게 만든 Ball() 생성자에서는 새로 만든 객체에 속성을 대입할 때 고정된 값(예를 들면 this.radius = 10)을 사용한다. 따라서 Ball 클래스의 모든 객체가 같은 속성 값을 가지게 되어 객체 지향 프로그래밍의 취지와는 정반대의 결과가 나오고 만다(똑같은 객체를 만들어내는 클래스는 필요가 없다. 하나의 클래스에서 여러 객체를 만드는 이유는 서로 다른 내용을 저장할 수 있는 객체가 필요하기 때문이다).

클래스의 인스턴스에 속성 값을 동적으로 대입하려면 생성자를 고쳐서 인자를 받아들이도록 하면 된다. 우선 일반적인 문법을 알아보고 나서 Ball 예제를 고쳐보자.

```
function Constructor (value1, value2, value3) {
  this.property1 = value1;
  this.property2 = value2;
  this.property3 = value3;
}
```

Constructor 함수 내부에서 새로 만들고 있는 객체를 this 키워드를 이용하여 참조하는 것은 똑같다. 하지만 이번에는 객체 속성 값을 코드에 직접 넣지 않는다. 대신 value1, value2, value3 인자의 값을 객체 속성에 대입한다. 새로운 클래스 멤버를 만들 때는 클래스 생성자 함수에 새로운 인스턴스의 초기 속성 값을 전달하면 된다.

```
myObject = new Constructor (value1, value2, value3);
```

〔예제 12-5〕에 이러한 방법을 Ball 클래스에 적용한 예가 나와 있다.

〔예제 12-5〕 일반화된 Ball 클래스

```
// Ball() 생성자에서 속성 값을 인자로 받아들일 수 있도록 만든다.
function Ball (radius, color, xPosition, yPosition) {
  this.radius = radius;
  this.color = color;
```

```
    this.xPosition = xPosition;
    this.yPosition = yPosition;
}

// 생성자를 호출할 때 객체의 속성 값으로
// 사용할 인자를 전달한다.
myBall = new Ball(10, 0x00FF00, 59, 15);

// 제대로 되었는지 확인하자.
trace(myBall.radius);  // 10이 출력된다.
```

이제 Ball 클래스를 만드는 과정이 거의 끝났다. 하지만 앞에서 얘기했던 moveTo()와 area() 메소드는 아직 만들지 않았다. 클래스 객체에 메소드를 추가하는 방법에는 두 가지가 있다. 우선 간단하지만 약간 비효율적인 방법을 알아보고 상속에 대한 내용을 배운 뒤에 메소드 생성 방법을 다시 다루기로 하자.

클래스 객체에 메소드 대입하기

메소드를 클래스에 추가하는 가장 간단한 방법은 함수가 들어있는 속성을 생성자에 대입하는 방법이다. 일반적인 문법은 다음과 같다.

```
function Constructor() {
    this.methodName = function;
}
```

둘째 줄의 function 자리에 들어갈 함수는 여러 가지 방법으로 만들 수 있다. 여기서는 area() 메소드를 Ball 클래스에 추가하는 예를 통해 그 방법들을 알아보자. 편의상 지금은 color, xPosition, yPosition 속성을 정의하는 부분을 제외하겠다.

다음과 같이 함수 리터럴을 function 자리에 넣을 수 있다.

```
function Ball (radius) {
    this.radius = radius;
    // area() 메소드를 추가한다.
    this.area = function () { return Math.PI * this.radius *
this.radius; };
}
```

또는 생성자 내부에서 함수를 정의해도 된다.

```
function Ball (radius) {
  this.radius = radius;
  // area 메소드를 추가한다.
  this.area = getArea;
  function getArea () {
    return Math.PI * this.radius * this.radius;
  }
}
```

마지막으로 생성자 외부에서 함수를 선언하여 생성자 내부에서 대입할 수도 있다.

```
// getArea() 함수를 선언한다.
function getArea () {
  return Math.PI * this.radius * this.radius;
}

function Ball (radius) {
  this.radius = radius;
  // area() 메소드를 추가한다.
  this.area = getArea;
}
```

세 가지 방법은 서로 크게 다르지 않으며 어느 것을 사용해도 무방하다. 대부분 함수 리터럴이 가장 편리하지만 생성자 밖에서 함수를 정의하는 것에 비해 코드 재사용이 힘들다. 생성자 밖에서 함수를 정의하면 다른 생성자에서도 그 함수를 이용할 수 있다. 하지만 방금 알아본 세 가지 방법은 모두 효율이 그다지 좋지 않다.

지금까지는 Ball 클래스에 있는 각 객체에 서로 다른 속성 값을 추가하였다. 모든 ball 객체에는 서로 다른 radius와 color 속성이 있어야 다른 객체와 차별될 수 있다. 하지만 방금 배운 기법을 이용하여 클래스 객체에 고정된 메소드를 추가한다면 불필요하게 클래스의 모든 객체에 똑같은 메소드를 추가하게 된다. 모든 ball 객체에서 넓이를 구하는 공식은 똑같기 때문에, 이러한 작업을 처리하는 코드는 모두 한 군데로 모아 일반화해야 한다.

고정된 값을 가지는 속성 또는 고정된 메소드를 효율적으로 클래스에 추가하고 싶다면 다음에 배울 '상속(inheritance)'을 이용해야 한다.

객체 속성 상속

상속된 속성은 클래스의 각 객체에 모두 들어가지는 않는다. 이러한 속성은 일단 클래스 생성자에 추가되며 필요에 따라 객체에서도 그 속성을 차용한다. 이러한 속성을 상속된 속성이라고 한다. 모든 객체에서 정의되는 것이 아니라 객체로 전달되기 때문이다. 상속된 속성은 객체를 통해 참조되므로 그 객체에 속한 것처럼 보이지만 사실은 객체의 클래스 생성자의 일부일 뿐이다.

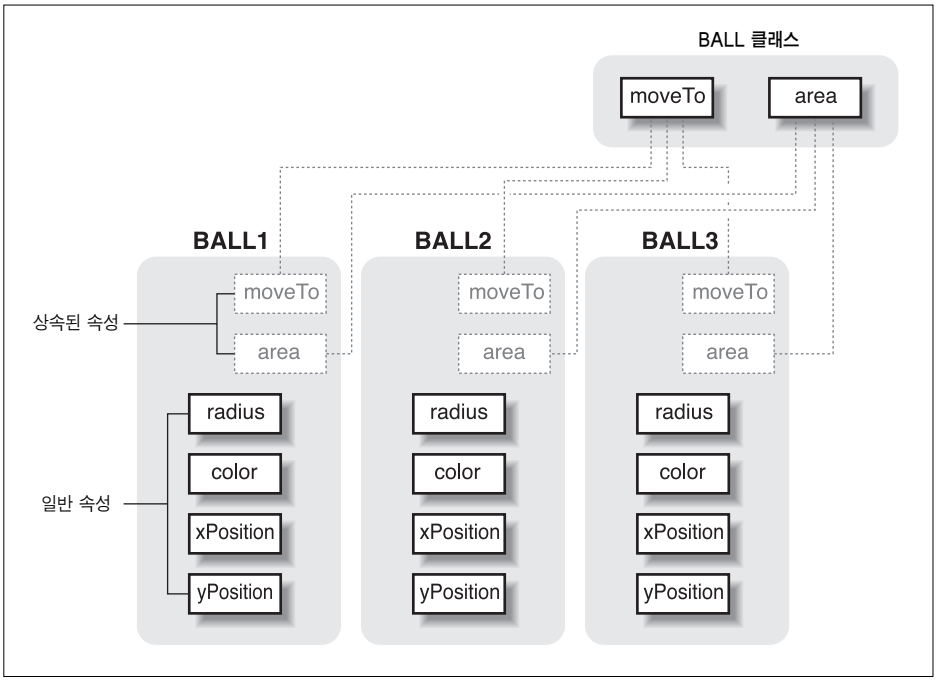
〔그림 12-2〕에 Ball 클래스를 예로 든 일반적인 상속된 속성의 모델이 나와 있다. Ball 클래스의 moveTo()와 area() 메소드는 인스턴스에 따라 달라지는 것이 아니므로 그러한 메소드는 상속된 메소드로 구현하기 좋다. 그러한 메소드는 클래스에 속하며 각 ball 객체는 레퍼런스를 통해서만 그 메소드를 액세스하게 된다. 반면 Ball 클래스의 radius, color, xPosition, yPosition 속성은 모든 ball 객체에서 각각 그 객체만의 속성을 가지고 있어야 하기 때문에, 일반적인 속성과 마찬가지로 각 객체에 대입된다.

상속은 가계도처럼 계층 사슬을 따라 이루어진다. 어떤 객체의 메소드를 호출하면 인터프리터에서는 그 객체에서 주어진 메소드를 구현하는지 알아본다. 만약 그 메소드가 없으면 인터프리터에서는 그 객체가 속한 클래스에 메소드가 있는지 확인한다.

예를 들어 ball1.area()를 실행하면 인터프리터에서는 ball1에서 area() 메소드를 정의하는지 확인한다. 만약 ball1 객체에 area() 메소드가 없다면 인터프리터에서는 Ball 클래스에 area() 메소드가 정의되어 있는지 확인한다. 만약 그렇다면 인터프리터에서는 그 메소드가 Ball 클래스가 아닌 ball1 객체에 있는 것처럼 area() 메소드를 호출한다. 이렇게 하면 메소드가 클래스 자체가 아닌 ball1 객체에 작용하여 필요에 따라 ball1의 속성 값을 확인하거나 그 값을 설정할 수 있다. 바로 이러한 점은 OOP의 핵심적인 장점 중 하나이다. 한 군데(Ball 클래스)에서 함수를 정의하고 여러 곳(ball 객체)에서 그 메소드를 사용할 수 있기 때문이다.

일반적인 속성과는 달리 상속된 속성은 객체에서 그 값을 읽을 수만 있고 그 속성에 어떤 값을 대입할 수 없을 수도 있다.

지금까지 이론적으로 배운 내용들을 직접 코드를 보면서 확인해 보자.



〔그림 12-2〕 상속된 속성과 일반 속성

prototype 속성을 이용하여 상속된 속성 만들기

상속된 속성을 만드는 과정은 〔예제 12-5〕에 나온 것과 같이 클래스 생성자 함수를 만드는 것부터 시작된다.

```
function Ball (radius, color, xPosition, yPosition) {
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}
```

'9장. 함수'에서 배웠듯이 함수도 객체와 마찬가지로 중복될 수 있으므로 속성을 가질 수 있다.

생성자 함수가 만들어지면 인터프리터에서는 자동으로 그 생성자 함수에 prototype이라는 속성을 추가한다. 인터프리터는 prototype 속성에 범용 객체를 집어넣는다. prototype 객체에 들어가는 속성은 그 생성자 함수로 정의되는 클래스의 모든 인스턴스에 상속된다.

어떤 클래스의 모든 객체에 상속되는 속성을 만들려면 그 생성자 함수로 정의되는 클래스에 속하는 미리 만들어진 prototype 객체에 속성을 추가하면 된다. 일반적인 문법은 다음과 같다.

```
Constructor.prototype.propName = value;
```

여기서 Constructor는 클래스의 생성자 함수(우리가 사용하는 예에서는 Ball)의 이름이고, prototype은 상속되는 속성을 집어넣기 위한 자동으로 생성되는 속성이며, propName은 상속되는 속성의 이름, value는 그 상속되는 속성의 값이다. 예를 들어 Ball 클래스에 속하는 모든 객체에 적용되는 gravity라는 전역 속성을 추가해 보자.

```
Ball.prototype.gravity = 9.8;
```

이렇게 gravity 속성을 정해 놓으면 Ball 클래스의 모든 멤버에서 gravity를 사용할 수 있다.

```
// Ball 클래스의 인스턴스를 새로 만든다.
myBall = new Ball(5, 0x003300, 34, 220);

// 상속된 gravity 속성을 출력한다.
trace(myBall.gravity);  // 9.8이 출력된다.
```

myBall에서는 Ball의 prototype 객체에 속한 속성을 모두 상속하므로 myBall을 통해 gravity 속성을 사용할 수 있다.

대부분의 경우에 한 클래스에 속하는 모든 인스턴스에서 같은 메소드를 사용하므로 상속된 속성으로 저장하는 것이 좋다. Ball 클래스에 상속되는 area() 메소드를 추가해 보자.

```
Ball.prototype.area = function () {
  return Math.PI * this.radius * this.radius;
};  // 함수 리터럴이므로 세미콜론을 사용해야 한다.
```

이번에는 함수 리터럴 대신 따로 정의한 함수를 이용하여 상속 가능한 moveTo() 메소드를 추가해 보자.

```
function moveTo (x, y) {
  this.xPosition = x;
  this.yPosition = y;
}

Ball.prototype.moveTo = moveTo;
```

어떤 함수를 상속되는 속성으로 정의하고 나면 다른 메소드와 마찬가지로 호출할 수 있다.

```
// 새로운 공을 만든다.
myBall = new Ball(15, 0x33FFCC, 100, 50);

// myBall의 상속 가능한 메소드인 area() 메소드를 호출한다.
trace(myBall.area());  // 706.858347057703이 출력된다.
```

또한 생성자의 prototype 객체를 전부 다른 객체로 바꾸는 것도 가능하다. 이렇게 하면 한 번에 여러 개의 상속 가능한 속성을 추가할 수 있다. 하지만 이러한 방법을 사용하면 상속 사슬이 변경된다. 상속 사슬에 대한 내용은 '수퍼클래스와 서브클래스'에서 자세히 배울 것이다.

상속된 속성 무효화

하나의 객체에 대해서만 상속된 속성을 변경하려면 상속된 속성과 같은 이름을 가지는 속성을 새로 설정하면 된다. 예를 들어 다음과 같이 공 하나만 다른 공에 비해 적은 중력을 받도록 만들 수도 있다.

```
// Ball 클래스 생성자를 만든다.
function Ball ( radius, color, xPosition, yPosition ) { ... } // 생략

// 상속 가능한 gravity 속성을 만든다.
Ball.prototype.gravity = 9.8;

// Ball 객체를 만든다.
lowGravBall = new Ball ( 200, 0x22DD99, 35, 100 );

// 상속된 gravity 속성을 무효화한다.
lowGravBall.gravity = 4.5;
```

한 객체에서 어떤 속성을 설정하면 같은 이름을 가진 상속된 객체를 '무효화(overriding)' 할 수 있다. 상속 사슬은 원래 이러한 방식으로 작동한다. 어떤 객체에서 메소드를 새로 정의하면 클래스에서 상속된 메소드도 무효화할 수 있다. ball1에서 area() 메소드를 정의하고 있다면, ball1.area()를 호출하면 ball1에서 정의한 area() 메소드가 호출된다. 인터프리터에서는 Ball 클래스의 prototype을 확인하지도 않는다.

constructor 속성

생성자의 prototype 객체가 만들어지면 인터프리터에서는 그 객체에 자동으로 constructor라는 특별한 속성을 추가한다. constructor 속성은 prototype의 클래스 생성자 함수에 대한 레퍼런스이다. 예를 들어 다음과 같은 두 표현식은 모두 Ball 생성자 함수에 대한 레퍼런스를 출력한다.

```
trace(Ball);                          // [Function]이 출력된다.
trace(Ball.prototype.constructor);    // [Function]이 출력된다.
                                      // 위에서 사용한 것과 같은 레퍼런스
```

constructor 속성에는 함수 이름을 문자열로 나타낸 값이 아니라 생성자 함수에 대한 레퍼런스가 들어간다는 점에 주의하자.

__proto__ 속성

어떤 객체가 만들어지면 인터프리터에서는 자동으로 __proto__(이름 양쪽에 각각 밑줄이 두 개씩이라는 점에 주의)라는 속성을 할당한다. 객체의 __proto__ 속성은 그 객체의 생성자 함수의 prototype 속성에 대한 레퍼런스이다. 예를 들어 myBall이라는 Ball 클래스의 인스턴스를 만들면 myBall.__proto__는 Ball.prototype이 된다.

```
myBall = new Ball(6, 0x00FF00, 145, 200);
trace(myBall.__proto__ == Ball.prototype);  // true가 출력된다.
```

__proto__ 속성은 주로 액션스크립트 인터프리터에서 어떤 객체의 상속된 속성을 확인하는 데 쓰인다. 예를 들어 myBall.area()와 같은 식으로 myBall 객체를 통해 상속된 메소드인 area()를 호출한다면 액션스크립트에서는 myBall.__proto__를 통해 Ball.prototype.area 메소드를 호출한다.

〔예제 12-6〕에 나온 것처럼 __proto__를 직접 사용하여 어떤 객체가 특정 클래스에 속하는지 확인할 수도 있다.

〔예제 12-6〕 객체의 클래스를 알아내는 법

```
function MyClass (prop) {
  this.prop = prop;
}
myObj = new MyClass();
```

```
if (myObj.__proto__ == MyClass.prototype) {
  trace("myObj is an instance of MyClass");
}
```

수퍼클래스와 서브클래스

고급 객체 지향 프로그래밍에서 매우 중요한 특징 중 하나로 속성을 공유할 수 있다는 점이 있다. 즉 클래스 전체에서 속성을 상속할 수 있고 다른 클래스에 그 속성을 전달할 수도 있다. 복잡한 상황에서는 '다중 클래스 상속(multiclass inheritance)'이 필요한 경우도 있다(다중 클래스 상속을 직접 사용하지는 않더라도 이 내용을 제대로 이해한다면 액션스크립트의 내장 클래스를 사용하는 데 도움이 될 것이다).

조금 전에 클래스 생성자의 prototype 객체를 이용하여 객체에서 속성을 상속하는 법에 대해 배웠다. 상속은 하나의 객체/클래스 관계에만 한정된 것은 아니다. 클래스 자체도 다른 클래스로부터 속성을 상속할 수 있다. 예를 들어 모든 원형 물체의 넓이를 계산할 수 있는 일반적인 메소드인 area()를 정의하고 있는 Circle이라는 클래스가 있다고 가정하자. Ball과 같은 클래스에서 area() 함수를 따로 정의하는 대신 Circle에 있는 area() 메소드를 상속할 수도 있다. 계층 구조를 확실히 이해하도록 하자. 가장 간단한 클래스인 Circle에서는 가장 일반적인 메소드와 속성을 정의한다. 다른 클래스인 Ball은 단순한 Circle 클래스를 기반으로 모든 원형 물체에 대해 Circle을 통해 구현하고 Ball 클래스의 인스턴스에만 필요한 특징을 추가하여 만들 수 있다. 전통적인 객체지향 프로그래밍에서는 Ball 클래스가 Circle 클래스를 '확장(extend)'한다고 말한다. 즉 Ball은 Circle의 서브클래스이고 Circle은 Ball의 수퍼클래스이다.

수퍼클래스 만들기

앞에서 클래스의 생성자 함수에 있는 prototype 객체의 상속 가능한 속성을 정의하는 방법을 배웠다. 주어진 클래스의 '수퍼클래스(superclass)'를 만들려면 클래스의 prototype 객체를 원하는 수퍼클래스의 새로운 인스턴스로 완전히 바꿔야 한다. 일반적인 문법은 다음과 같다.

```
Constructor.prototype = new SuperClass();
```

Constructor의 prototype 객체를 SuperClass의 인스턴스로 바꾸면 Constructor의 모든 인스턴스에서 SuperClass의 인스턴스에서 정의된 속성을 상속받게 된다. [예제 12-7]에서는 우선 모든 인스턴스에 area() 메소드를 집어넣는 Circle이라는 클래스를 만든다. 그리고 나서 Circle의 인스턴스를 Ball.prototype에 대입하여 모든 ball 객체에서 Circle의 area() 메소드를 상속하도록 만든다.

[예제 12-7] 수퍼클래스 만들기

```
// Circle (수퍼클래스) 생성자를 만든다.
function Circle() {
  this.area = function () { return Math.PI * this.radius * this.radius; };
}

// Ball 클래스 생성자를 만든다.
function Ball ( radius, color, xPosition, yPosition ) {
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Circle의 인스턴스를 Ball 클래스 생성자의 prototype에 대입하여
// 수퍼클래스를 만든다.
Ball.prototype = new Circle();

// 이제 Ball의 인스턴스를 만들고 속성을 확인해보자.
myBall = new Ball ( 16, 0x445599, 34, 5);
trace(myBall.xPosition);     // Ball의 일반적인 속성인 34가 출력된다.
trace(myBall.area());        // 804.24...가 출력된다.
                             // area()는 Circle에서 상속됨
```

하지만 이렇게 하면 클래스 계층이 이상해진다. Ball에서는 radius를 정의하지만 사실 radius는 모든 원(Circle 클래스)에 대해 공통적인 속성이므로 Circle 클래스에 속하게 된다. xPosition이나 yPosition의 경우도 마찬가지이다. 이러한 구조를 고치기 위해 radius, xPosition, yPosition을 모두 Circle로 옮기고 color만 Ball에 남겨야 한다(이 예제에서는 편의상 color라는 속성을 공에만 적용되는 것으로 간주한다).

간단하게 개념만 살펴본다면 새로 고친 Circle과 Ball의 생성자는 다음과 같다.

```
// Circle (수퍼클래스) 생성자를 만든다.
function Circle ( radius, xPosition, yPosition ) {
  this.area = function () { return Math.PI * this.radius *
this.radius; };
  this.radius = radius;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Ball 클래스 생성자를 만든다.
function Ball ( color ) {
  this.color = color;
}
```

속성을 모두 옮기긴 했는데 이번에는 새로운 문제에 직면하게 된다. Circle이 아닌 Ball을 이용하여 객체를 만들 때는 radius, xPosition, yPosition의 값을 어떻게 설정할까? 이러한 문제를 해결하기 위해 Ball 생성자 코드에서 한 가지를 더 수정해야 한다. 우선 Ball에서 필요한 속성을 모두 매개변수로 받아들이도록 설정하자.

```
function Ball ( color, radius, xPosition, yPosition ) {
```

다음에는 Ball 생성자 내부에서 Circle 생성자를 ball 객체의 인스턴스를 만드는 메소드로 선언한다.

```
this.superClass = Circle;
```

마지막으로 ball 객체에서 Circle 생성자를 호출한다. 이 때 radius, xPosition, yPosition의 값을 인자로 전달한다.

```
this.superClass(radius, xPosition, yPosition);
```

이렇게 완성된 클래스/수퍼클래스 코드는 〔예제 12-8〕에 나와 있다.

〔예제 12-8〕 클래스와 수퍼클래스

```
// Circle (수퍼클래스) 생성자를 만든다.
function Circle ( radius, xPosition, yPosition) {
  this.area = function () { return Math.PI * this.radius * this.radius; };
  this.radius = radius;
```

```
    this.xPosition = xPosition;
    this.yPosition = yPosition;
}

// Ball 클래스 생성자를 만든다.
function Ball ( color, radius, xPosition, yPosition ) {
    // Circle 수퍼클래스를 ball 객체의 인스턴스를 만들기 위한 메소드로 선언한다.
    this.superClass = Circle;
    // ball 객체에서 Circle 생성자를 호출한다. 이 때 Ball 생성자에
    // 전달된 인자의 값을 전달한다.
    this.superClass(radius, xPosition, yPosition);
    // ball 객체의 color 속성을 설정한다.
    this.color = color;
}

// Ball 클래스 생성자의 prototype에
// Circle 수퍼클래스의 인스턴스를 대입한다.
Ball.prototype = new Circle();

// 이제 Ball의 인스턴스를 만들고 속성을 확인해보자.
myBall = new Ball ( 0x445599, 16, 34, 5);
trace(myBall.xPosition);     // 34
trace(myBall.area());        // 804.24...
trace(myBall.color);         // 447836
```

Ball에 있는 superClass라는 단어는 예약된 특수 키워드가 아니다. 단순히 수퍼클래스 생성자 함수의 이름일 뿐이다. 또한 Circle의 area() 메소드는 Circle.prototype에서 정의된 것일 수도 있다. 실제로 클래스와 객체를 이용하여 프로그래밍을 시작하면 액션스크립트에서 클래스 계층을 만들고 상속성을 구현하기 위해 제공하는 도구에 상당한 유연성이 있다는 생각이 들 것이다. 독자가 사용하고자 하는 용도에 알맞게 이 장에서 설명한 접근법을 적당히 응용하는 것이 필요하다.

다형성

상속성을 이용하면 '다형성(polymorphism)'이라는 또 다른 핵심 OOP 개념을 사용할 수 있다. 다형성은 '다양한 형태'라는 뜻을 가진 단어이다. 다형성이란 객체에 어떤 일을 시키고 자세한 내용은 객체에서 알아서 처리하도록 한다는 것을 의미한다. 길게 말로 설명하는 것보다는 예제를 통해 알아보는 것이 낫다. 경찰과 도둑

게임을 만든다고 가정하자. 화면에 서로 다른 규칙에 따라 독립적으로 움직이는 경찰, 도둑, 무고한 시민이 각각 여러 명씩 동시에 나타난다. 경찰은 도둑을 쫓아가고 도둑은 경찰로부터 달아나고 시민들은 놀라서 여기저기로 특정한 방향이 없이 움직인다. 이 게임의 코드에서 각 종류의 사람들을 나타내기 위한 객체 클래스를 만든다고 가정하자.

```
function Cop() { ... }
function Robber() { ... }
function Bystander() { ... }
```

이 외에 Cop, Robber, Bystander에서 모두 상속할 수 있는 수퍼클래스인 Person을 만든다.

```
function Person() { ... }
Cop.prototype       = new Person();
Robber.prototype    = new Person();
Bystander.prototype = new Person();
```

플래시 무비의 각 프레임에서 화면에 있는 모든 사람은 그 클래스에서 정해진 규칙에 따라 움직여야 한다. 이렇게 하려면 각 객체에서 move()라는 메소드를 정의해야 한다(move() 메소드는 각 클래스마다 다르게 정의되어야 한다).

```
Person.prototype.move = function () { ... 기본적인 움직임 ... }
Cop.prototype.move = function () { ... 도둑을 쫓아간다 ... }
Robber.prototype.move = function () { ... 경찰로부터 도망간다 ... }
Bystander.prototype.move = function () { ... 아무 방향으로나 움직인다
... }
```

플래시 무비의 각 프레임에서 화면에 있는 모든 사람이 움직이도록 만들자. 모든 사람을 관리할 수 있도록 Person 객체의 배열을 만든다. persons 배열은 다음과 같은 방법으로 채울 수 있다.

```
// 경찰을 만든다.
cop1 = new Cop();
cop2 = new Cop();

// 도둑을 만든다.
robber1 = new Robber();
robber2 = new Robber();
robber3 = new Robber();
```

```
// 시민을 만든다.
bystander1 = new Bystander();
bystander2 = new Bystander();

// 경찰, 도둑, 시민이 모두 들어있는 배열을 만든다.
persons = [cop1, cop2, robber1, robber2, robber3, bystander1,
bystander2];
```

플래시 무비의 모든 프레임에서 다음과 같이 정의된 moveAllPersons()라는 함수를 호출한다.

```
function moveAllPersons() {
  for (var i=0; i < persons.length; i++) {
    persons[i].move();
  }
}
```

moveAllPersons()를 호출하면 모든 경찰, 도둑, 그리고 시민들이 각 클래스의 move() 메소드에서 정의한 규칙에 따라 움직이게 된다. 다형성은 바로 이러한 식으로 적용된다. 공통적인 특징을 가진 객체를 하나로 묶을 수 있긴 하지만 각각의 성질은 다르다. 경찰, 도둑, 그리고 시민은 비슷한 점이 많으며, 이러한 특징은 모두 Person이라는 수퍼클래스에 포함된다. 움직일 수 있다는 것과 같이 공통적으로 처리하게 되는 작업도 있다. 하지만 그러한 공통적인 작업을 서로 다른 방식으로 처리해야 하며, 그 외의 특정 클래스에서만 필요한 작업이나 데이터가 있을 수도 있다. 다형성을 이용하면 서로 다른 객체를 같은 방법으로 다룰 수 있다. 각 클래스의 move() 함수는 서로 다르더라도 모든 사람이 움직이게 하고 싶다면 move() 함수만 호출하면 된다.

어떤 객체가 수퍼클래스에 속하는지 알아내는 법

Shape라는 클래스와 Shape를 수퍼클래스로 가지는 Rectangle이라는 클래스가 있다고 가정하자. 어떤 객체가 Shape에서 만들어진 객체인지 확인하려면 〔예제 12-6〕에서 사용한 방법을 조금 고쳐서 prototype 객체 사슬(prototype 사슬이라고도 한다)을 따라갈 수 있는 코드를 만들어야 한다. 〔예제 12-9〕에 이러한 기법이 나와 있다.

〔예제 12-9〕 prototype 사슬 따라가기

```
// 이 함수에서는 theObj가 theClass에서 나온 클래스인지 확인한다.
function objectInClass(theObj, theClass) {
  while (theObj.__proto__ != null) {
    if (theObj.__proto__ == theClass.prototype) {
      return true;
    }
    theObj = theObj.__proto__;
  }
  return false;
}

// 새로운 Rectangle의 인스턴스를 만든다.
myObj = new Rectangle();

// myRect에서 Shape를 상속하는지 확인한다.
trace (objectInClass(myRect, Shape)); // true가 출력된다.
```

상속 사슬의 끝

모든 객체는 최상위 클래스인 Object 클래스에서 파생된다. 따라서 모든 객체는 Object 생성자에서 정의된 속성(toString()과 valueOf() 메소드)을 상속받는다. 따라서 Object 클래스에 새로운 속성을 추가하면 무비에 있는 모든 객체에 새로운 속성을 추가할 수 있다. Object.prototype에 들어있는 속성은 모든 내부 객체, 심지어는 movieclip 객체에 이르기까지 전체 클래스 계층에 포함된다. 이렇게 하면 진정한 전역 변수 또는 전역 메소드를 만들 수도 있다. 아래 코드에서는 Object 클래스의 prototype에 스테이지의 너비와 높이를 직접 집어넣는다. 이렇게 하면 그러한 정보를 모든 무비 클립에서 액세스할 수 있다.

```
Object.prototype.mainstageWidth = 550;
Object.prototype.mainstageHeight = 400;
trace(anyClip.mainstageWidth);  // 550이 출력된다.
```

Object.prototype 속성은 무비 클립뿐 아니라 모든 객체에서 상속받게 되므로, 사용자 정의 객체나 Date, Sound와 같은 내장 클래스도 Object.prototype에 추가된 속성을 상속받는다.

연습 문제: 어떤 내장 클래스에도 새로운 속성이나 메소드를 추가할 수 있다. [예제 11-6]에 있는 대소문자를 구분하지 않는 알파벳순 정렬 함수를 Array 클래스에 상속된 메소드 형태로 추가해 보자.

자바 용어와 비교

클래스와 클래스 계층에 대해 알아보면서 객체, 클래스 생성자, 클래스 prototype에서 정의된 속성에 대해 알아보았다. 자바와 C++에는 여러 종류의 클래스와 객체 속성에 특별한 이름이 정해져 있다. 자바 프로그래머들을 위해 [표 12-1]에 자바와 액션스크립트 용어를 비교해 보았다.

[표 12-1] 자바와 액션스크립트 속성

자바	설명	액션스크립트	액션스크립트 예제

인스턴스 변수	객체 인스턴스의 지역 변수	클래스 생성자 함수에서 정의되어 객체에 복사되는 속성	function Square (side) {
			this.side = side;
			}

인스턴스 메소드	객체 인스턴스에서 호출되는 메소드	클래스 생성자 함수의 prototype 객체에서 정의되며 어떠 인스턴스에서 호출하면 자동으로 prototype 객체를 통해 실행되는 메소드	Square.prototype.area = squareArea; mySquare.area( );
클래스 변수	한 클래스의 모든 객체 인스턴스에서 같은 값을 가지는 변수	클래스 생성자 함수에서 함수 속성으로 정의된 속성	Square.numSides = 4;
클래스 메소드	클래스를 통해 실행되는 메소드	클래스 생성자 함수에서 함수 속성으로 정의된 메소드	Square.findSmaller = function (square1, square2) { ... }

객체 지향 프로그래밍 요약

클래스와 상속성을 이용하면 여러 객체에서 공통적인 정보를 공유할 수 있다. 객체 지향 프로그래밍은 모든 액션스크립트 코드의 기반이 된다. 스크립트에서 클래스와 객체의 모든 면을 이용하는 것과는 상관없이 일반적인 개념을 이해하는 것

은 플래시 프로그래밍 환경을 이해하는 데 필수적이다. 이 장의 마지막 절에서 알 수 있겠지만 객체 지향 프로그래밍에 대한 자신감이 생기면 전반적인 액션스크립트에 대한 자신감이 생기게 될 것이다.

ECMA-262 기반 언어에서의 객체 지향 프로그래밍에 대한 내용은 넷스케이프의 자바스크립트에 대한 문서인 Details of Object Model에 자세히 나와 있다.

http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm

데이비드 플라나긴의 명저 『자바스크립트 핵심 가이드』(한빛미디어, 2001)에도 자바스크립트에서의 OOP에 대한 좋은 정보가 실려 있다. 자바 관점에서 보는 OOP에 대한 일반적인 내용은 썬의 Object Oriented Programming Concepts(Java™ 튜토리얼)을 참조하기 바란다.

http://java.sun.com/docs/books/tutorial/java/concepts

액션스크립트 내장 클래스와 객체

'1장. 프로그래밍을 모르는 독자들을 위한 기본 소개'에서 시작하여 지금까지 정말 먼 길을 달려왔다. 액션스크립트의 구성요소를 처음으로 살펴본 부분은 Actions 패널의 + 버튼 밑에 있는 아이템을 쭉 알아보는 것이었다. 그 다음부터 데이터와 표현식, 연산자, 선언문, 함수, 그리고 클래스와 객체에 대한 개념까지 자세히 살펴보았다. 드디어 액션스크립트 언어를 그려가는 과정에서 마지막 점을 찍을 때가 되었다.

액션스크립트에는 다양한 문법적인 도구가 들어있다. 표현식에는 데이터가 들어있고 연산자는 데이터를 조작하고 선언문은 명령을 내리며 함수는 여러 개의 명령어를 이식성이 좋은 하나의 명령어로 묶어준다. 이러한 것은 훌륭한 도구이긴 하지만 단지 도구에 지나지 않는다. 즉 스크립트에서 명령을 내리기 위한 문법에 불과하다. 우리는 지금까지 정작 주제에 관한 것은 다루지 않았다. 이제 액션스크립트로 말하는 방법을 거의 완벽하게 배우긴 했지만 아직 무엇에 대해 말할지를 배우지 않았다. 액션스크립트의 내장 클래스와 객체는 바로 그 빈 자리를 메꿔줄 것이다.

내장 클래스

우리가 원하는 규격에 맞추어 만들어진 객체를 설명하고 조작하기 위해 클래스를 정의하는 것과 마찬가지로 액션스크립트에서는 액션스크립트만의 데이터 클래스를 정의한다. Object 클래스를 포함한 다양한 클래스가 이미 만들어져 있으며, 액션스크립트 언어에 내장되어 있다. 내장 클래스를 이용하면 플래시 무비의 물리적인 환경을 제어할 수 있다.

예를 들어 내장 클래스 가운데 하나인 Color 클래스에서는 무비 클립의 색을 감지하거나 설정할 수 있는 메소드를 정의한다. 이러한 메소드를 사용하려면 우선 다음과 같이 Color() 클래스 생성자를 이용하여 Color 객체를 만들어야 한다.

```
clipColor = new Color(target);
```

clipColor에는 Color 객체를 저장하기 위한 임의의 변수, 배열 원소 또는 객체 속성이 들어간다. Color() 생성자에서는 target 자리에 들어가는 하나의 인자만을 받아들이는데, 이 인자는 우리가 알아보거나 설정하려는 무비 클립의 이름을 나타낸다.

square라는 무비 클립이 있는데 그 색을 바꾸고 싶은 경우를 생각해 보자. 우선 다음과 같이 새로운 Color 객체를 만든다.

```
squareColor = new Color(square);
```

그리고 나서 square 클립의 색을 설정하기 위해 squareColor 객체에서 Color 메소드 중 하나를 실행시킨다.

```
squareColor.setRGB(0x999999);
```

setRGB() 메소드에서는 Color 객체의 target, 이 경우에는 square의 RGB 값을 설정한다. 따라서 위와 같은 코드를 실행시키면 square의 색이 회색으로 설정된다.

Color 객체는 내장 클래스이므로 무비 클립의 색을 직접 설정할 수 있다. 또한 사운드, 날짜, XML 문서와 같은 것을 제어할 수 있는 클래스들도 있다. 내장 클래스에 대해서는 '3부. 레퍼런스'에서 알아보기로 하자.

내장 객체

내장 클래스와 마찬가지로 내장 객체를 이용하면 무비 환경을 제어할 수 있다. 예를 들어 Key 객체에는 컴퓨터의 키보드 상태를 알려주는 속성과 메소드가 정의되어 있다. 이러한 속성과 메소드를 사용할 때는 Key 객체의 인스턴스를 만드는 대신 Key 객체를 직접 사용하면 된다. 내장 객체는 플래시 플레이어가 시작할 때 인터프리터에 의해 자동으로 만들어지며 무비 전체에 걸쳐 언제든지 사용할 수 있다.

아래 코드에서는 Key 객체의 getCode() 메소드를 호출하여 현재 눌려 있는 키의 키코드를 출력한다.

```
trace(Key.getCode());
```

그리고 다음 코드에서는 isDown() 메소드를 호출하면서 스페이스바의 키코드를 인자로 전달하여 스페이스바가 눌려 있는지 확인한다.

```
trace(Key.isDown(Key.SPACE));
```

3부에서는 Math, Mouse, Selection과 같이 수학적인 정보, 마우스 포인터, 텍스트 필드 선택영역을 액세스할 수 있는 다른 내장 객체에 대해서도 다룬다.

프로그래밍 요령 익히기

제대로 된 액션스크립트 코드를 만드는 법을 배우는 것은 플래시 프로그래밍을 배우는 데 절반 정도의 비중을 차지한다. 나머지 절반은 내장 클래스와 내장 객체, 그리고 그 클래스와 객체의 속성 및 메소드를 배우는 데 있다. 이러한 내용은 3부에서 자세히 배울 것이다. 하지만 한 번에 모든 클래스와 객체를 배울 필요는 없다. 우선 무비 클립을 배우는 것부터 시작하여 상황에 따라 필요한 것을 공부하면 된다.

시간이 지남에 따라 자신이 하고자 하는 일을 제대로 하려면 어떤 것이 필요한지 알 수 있게 될 것이다. 중요한 것은 객체 지향 프로그래밍의 일반적인 구조를 이해하는 것이다. 일단 시스템 규칙을 알고 나면 새로운 객체나 클래스를 배우는 일은 그 객체나 클래스의 메소드 및 속성 이름을 찾아보는 것으로 충분하기 때문이다.

앞으로 배울 내용

지금까지 정말 많은 내용을 다루었다. 많은 독자들이 필자처럼 계속해서 열정을 가지고 공부하기 바란다. 다음 장에서는 액션스크립트에서 가장 중요한 객체 클래스인 무비 클립에 대해 알아보자.