9

함수

액션스크립트에서 가장 강력한 부분 중 하나인 함수에 대해 설명하려고 하니 필자의 가슴이 막 떨려온다. 사실 함수는 프로그램 전반에 걸쳐 언제든지 사용할 수 있는 코드 조각에 지나지 않는다. 함수를 이용하면 스크립트를 만들 때 융통성과 편리함을 증대시킬 수 있을 뿐만 아니라 플래시 무비의 요소를 제어할 수도 있다. 사실 함수 없는 프로그래밍은 상상하기조차 힘들다. 함수를 이용하면 단어를 정렬하는 것부터 두 무비 클립 사이의 거리를 계산하는 것에 이르기까지 다양한 작업을 편리하게 처리할 수 있기 때문이다. '12장. 객체와 클래스'에서 함수와 객체를 이용하여 복잡하고 강력한 프로그램을 만드는 법에 대해 배우기 전에 우선 이 장에서 함수에 대한 소개를 하고 넘어갈까 한다.

먼저 스크립트에서 사용자가 직접 만드는 '프로그램 함수(program function)'에 초점을 맞춰보자. 직접 함수를 만드는 법을 배우면 다음과 같은 기본적인 내용에 익숙해질 것이다.

함수 선언

스크립트에서 사용할 함수를 만드는 것

함수 호출

함수를 실행하는 것. 바꾸어 말하면 함수의 코드를 실행하는 것

함수 인자

함수를 호출했을 때 함수에서 사용하기 위해 전달하는 데이터

함수 종료

함수 실행을 멈추고 상황에 따라 결과를 리턴할 수도 있다.

함수 영역

함수를 사용할 수 있는 범위와 기간을 결정하는 것. 함수 본체에서 참조한 변수의 접근성

함수에 대한 위와 같은 내용을 이해하고 나서 액션스크립트에 내장된 '내부 함수(internal function)'에 이런 용어들이 어떤 식으로 적용되는지 생각해 보자.

함수 생성

기본적으로 함수를 만들려면 다음과 같이 함수의 이름과 실행할 선언문 블록만 있으면 된다.

```
function funcName () {
  statements
}
```

function 키워드는 새로운 함수의 '선언(declaration)'을 시작한다는 것을 의미한다. 그 뒤의 funcName 자리에는 함수 이름이 들어가며 이 이름은 함수를 호출할 때 쓰인다. funcName은 반드시 제대로 된 인식자여야 한다.[1] 그 뒤에는 괄호가 오는데, 여기에는 잠시 후 배우게 될 인자가 들어간다. 함수에 인자가 없다면 괄호 안에 아무 내용도 넣지 않으면 된다. 마지막으로 함수 본체(선언문 블록)가 들어가는데, 여기에는 함수를 호출했을 때 실행할 코드가 들어간다.

1) '제대로 된 인식자'의 조건은 '14장. 렉시컬 구조'에서 찾아볼 수 있다.

아주 간단한 함수를 직접 만들어 보자.

1. 새로운 플래시 무비를 시작한다.

2. 메인 무비 타임라인의 1번 프레임에 다음과 같은 코드를 추가한다.

```
function sayHi () {
  trace("Hi there!");
}
```

이렇게 sayHi()라는 함수를 만들었다. 정말 간단하지 않은가? 이 함수를 실행하면 본체에 있는 trace() 선언문이 실행된다. 잠시 후에 함수를 실행하는 법을 배울 것이므로 방금 만든 무비를 아직 닫지 말고 기다리기 바란다.

함수 실행

함수를 실행할 때는 마법사가 주문을 외우듯이 함수의 이름 뒤에 '5장. 연산자'에서 배운 함수 호출 연산자인 ()를 덧붙이면 된다.

```
funcName()
```

괄호 안에는 함수에서 정의된 인자가 들어간다. 만약 함수에서 인자를 정의하지 않으면 괄호 안에 아무것도 입력하지 않으면 된다. 조금 전에 만든 sayHi() 함수(이 함수에는 인자가 없다)를 호출하여 기본적인 함수 호출 방법을 알아보자.

sayHi() 함수는 다음과 같이 선언되었다.

```
function sayHi () {
  trace("Hi there!");
}
```

sayHi() 함수를 호출할 때는 다음과 같이 하면 된다.

```
sayHi();
```

위 코드를 실행시키면 sayHi() 함수가 호출되어 그 함수 본체가 실행되기 때문에, Output 창에 "Hi there!"가 출력된다.

sayHi();라고 입력하는 것이 trace() 선언문을 전부 입력하는 것보다 훨씬 편하다. 또한 sayHi라는 함수 이름은 그 코드의 의미를 더 명확하게 설명해준다. 의미 있는 함수 이름을 사용하면 코드의 가독성을 향상시킬 수 있다.

다른 내용을 배우기 전에 함수를 호출한 줄의 끝에 있는 세미콜론을 살펴보자.

```
sayHi();
```

함수 호출 하나만으로도 하나의 선언문이 될 수 있으므로 세미콜론을 추가한다. 모든 선언문은 세미콜론으로 끝나야 하기 때문이다.

너무 간단하다는 생각이 들지도 모르지만, 방금 배운 것이 바로 함수 생성과 함수 호출의 기본이 되는 내용이다. 조금 썰렁해 보이긴 하지만 그래도 배울 것은 모두 배운 셈이다. 함수를 이용하면 코드를 한 군데로 집중시킬 수 있고 프로그램 여기저기서 같은 작업을 반복적으로 처리해야 하는 경우에 코드를 관리하기가 매우 편해진다. 또한 다음에 배울 인자와 함께 사용하면 훨씬 강력한 기능을 발휘할 수 있다.

함수에 정보 전달하기

앞 절에서 간단한 trace() 선언문을 실행하는 함수(그다지 대단하다고 할 수는 없는 함수이다)를 만들어 보았다. 아래 예는 ball이라는 무비 클립 인스턴스를 약간 이동시키는 코드이다.

```
function moveBall () {
  ball._x += 10;
  ball._y += 10;
}
```

위와 같이 정의된 moveBall() 함수를 이용하면 언제든지 다음과 같이 함수를 호출하기만 하면 ball 인스턴스를 대각선 방향으로 움직일 수 있다.

```
moveBall();
```

이렇게 하면 공이 오른쪽 아래 대각선 방향으로 움직인다(원점인 (0, 0)은 메인 스테이지의 왼쪽 위 모서리 부분에 있다. 따라서 _x를 증가시키면 공이 오른쪽으로 움

직이지만 일반적인 직교좌표계와는 달리 _y를 움직일 때는 공이 위가 아닌 아래쪽으로 움직이게 된다).

moveBall() 함수도 편리하긴 하지만 아직은 융통성이 좀 부족하다. 단 하나의 무비 클립(ball)에만 적용할 수 있으며, 한 방향으로만 움직일 수 있는 데다 움직이는 거리도 고정되어 있기 때문이다.

잘 정의된 함수라면 하나의 코드만으로 여러 상황에서 사용할 수 있어야 한다. moveBall() 함수를 일반화시켜 임의의 클립을 원하는 거리만큼, 원하는 방향으로 움직이게 할 수도 있다. 어떤 함수든지 일반화하려면 어떤 요인이 어떤 기능을 결정하는지 알아야 한다. moveBall() 함수에서는 이동시킬 무비 클립의 이름, 수평 방향으로 이동시킬 거리, 수직 방향으로 이동시킬 거리가 그러한 요인에 해당한다. 이와 같은 요인을 함수의 '인자(parameter)'라고 부르며, 함수를 호출할 때마다 해당 정보를 조절할 수 있다.

인자가 있는 함수 만들기

앞에서도 배웠듯이 간단한 함수 선언 문법은 다음과 같다.

```
function funcName () {
  statements
}
```

함수에서 사용할 변수인 인자를 추가하려면 함수 선언부의 괄호 안에 인식자 목록을 집어넣어야 한다. 각 인자는 다음과 같이 쉼표로 구분한다.

```
function funcName (param1, param2, param3, ...paramn) {
  statements
}
```

인자를 정의하고 나면 함수 안에서 다른 변수를 다룰 때와 같은 방법으로 그 인자 값을 이용할 수 있다.

```
function say(msg) {                // msg 인자를 정의한다.
  trace("The message is" + msg); // trace() 선언문에서 msg 인자를 사용한다.
}
```

위 함수 선언에서는 msg라는 인자를 정의한다. trace() 선언문에서는 다른 변수를 사용하는 것과 똑같은 방법으로 인자를 사용하여 Output 창에 그 값을 출력한다.

이제 [예제 9-1]에 나온 것처럼 moveBall() 함수에도 인자를 추가하여 매번 실행할 때마다 클립의 이름, 수평 이동거리, 수직 이동거리를 다르게 설정할 수 있도록 만들어 보자.

[예제 9-1] 일반화된 moveClip 함수

```
function moveClip (theClip, xDist, yDist) {
  theClip._x += xDist;
  theClip._y += yDist;
}
```

위 예제에서는 함수 이름을 moveBall()보다 더 일반적인 이름인 moveClip()으로 바꿨다. 또한 theClip(이동시키고자 하는 무비 클립), xDist(수평 방향 이동거리), yDist(수직 방향 이동거리)와 같이 세 개의 인자를 정의했다. 함수 본체에서는 이동거리가 고정되어 있던 원래 예제와는 달리 주어진 인자를 활용하여 어떤 클립이든 원하는 만큼 수평, 수직 방향으로 이동시킬 수 있다.

인자를 이용한 함수 호출

함수를 만들 때는 인자 이름도 함께 정의하며, 함수를 호출할 때는 각 인자 자리에 들어갈 값들을 함께 전달한다.

매개변수와 인자라는 단어를 보통 많이 사용하는데, 이 책에서는 인자라는 표현을 주로 사용할 것이다. 기술적으로 볼 때 인자는 함수를 호출할 때 쓰이는 값을, 매개변수는 그러한 인자를 받아들이기 위한 저장고와 같은 역할을 의미하지만, 실제 두 단어의 의미에는 거의 차이가 없다.

앞에서도 배웠듯이 인자 없이 함수를 호출할 때는 다음과 같은 문법을 사용한다.

```
funcName()
```

함수에 인자를 제공(전달)할 때는 함수를 호출할 때 괄호 안에 값 목록을 집어넣으면 된다.

```
funcName(arg1, arg2, arg3,...argn)
```

복합 표현식을 포함한 액션스크립트에서 사용할 수 있는 모든 표현식은 인자로 사용할 수 있다. 예를 들어 앞에서 msg라는 하나의 인자를 받아들이는 함수인 say()를 정의한 적이 있다.

```
function say(msg) {
  trace("The message is" + msg);
}
```

say() 함수를 호출할 때는 다음과 같은 선언문을 사용한다.

```
say("This is my first argument...how touching");
```

또는 다음과 같은 선언문을 써도 된다.

```
say(99);
```

여기서 "This is my first argument...how touching"과 99라는 값의 데이터형이 서로 다르다는 것을 알 수 있다. 액션스크립트에서는 함수에서 주어진 값을 처리할 수만 있다면 어떤 데이터형의 어떤 데이터라도 함수에 전달할 수 있다(C와 같은 언어에서는 각 인자마다 데이터형이 미리 정의되어 있어서 잘못된 유형의 데이터를 전달하면 오류가 발생한다).

각 인자는 전달되기 전에 그 값이 완전히 결정된다. 따라서 다음과 같이 복합 표현식을 인자로 사용하여 함수를 호출할 수도 있다.

```
var name = "Gula";
say("Welcome to my web site" + name);
```

"Welcome to my web site " + name이라는 표현식은 say() 함수로 전달되기 전에 그 값이 계산되므로 함수에서는 결국 최종적으로 결정된 값인 "Welcome to my web site Gula"라는 값을 받게 된다. 즉 프로그램을 통해 함수의 인자를 만들 수도 있다. 꽤 강력한 기능이다.

한 개 이상의 인자를 함수에 전달할 때는 쉼표로 각 인자를 구분한다. [예제 9-1]에 나온 moveClip() 함수에서는 세 개의 인자(theClip, xDist, yDist)를 받아들인다. 따라서 moveClip()을 호출할 때는 다음과 같이 해야 한다.

```
moveClip(ball, 5, -15);
```

세 개의 인자는 각각 함수를 선언할 때 그 위치에 있었던 매개변수 값으로 저장된다. 따라서 theClip에는 ball이, xDist에는 5가, yDist에는 -15가 저장된다. moveClip()을 호출할 때 인자만 바꿔주면 어떤 클립이라도 원하는 거리만큼 이동시킬 수 있다. 아래 예에서는 square라는 클립 인스턴스를 오른쪽으로 2픽셀, 아래로 100픽셀 이동시킨다.

```
moveClip(square, 2, 100);
```

함수 종료 및 함수 값 리턴

따로 정해주지 않으면 인터프리터에서는 함수 본체에 있는 마지막 선언문을 실행하고 나면 그 함수를 종료하게 된다. 하지만 마지막 선언문을 실행하지 않고도 함수를 종료시키는 방법이 있다. 또한 함수에서 그 함수를 호출한 코드에 어떤 결과를 리턴(계산된 값을 보내는 것)할 수도 있다. 이러한 작업이 처리되는 과정을 자세히 알아보자.

함수 종료

'6장. 선언문' 에서 소개한 return 선언문은 함수를 종료하고 상황에 따라 결과를 리턴하는 데 쓰인다. 인터프리터에서 함수를 실행하는 도중에 return 선언문을 발견하면 함수에 있는 나머지 선언문을 건너뛴다. 아래 예를 생각해 보자.

```
function say(msg) {
  return;
  trace(msg);          // 이 줄은 절대 실행되지 않는다.
}
```

위와 같은 예는 사실 실전에서 전혀 쓰이지 않는 코드이다. return 선언문 때문에 trace() 선언문을 실행해보지도 못하고 함수가 종료되기 때문이다. 따라서 return 선언문은 조건 선언문 안에 있는 경우를 제외하면 보통 함수 본체에서 마지막 선언문으로 쓰인다. 아래 예에서는 비밀번호가 틀린 경우에 함수를 그냥 종료시키기 위해 return 선언문을 사용한다.

```
var correctPass = "cactus";
function enterSite(pass) {
  if (pass != correctPass) {
    // 비밀번호가 틀리면 그대로 종료한다.
    return;
  }
  // 비밀번호가 맞는 경우에만 이 부분이 실행된다.
  gotoAndPlay("intro");
}

enterSite("hackAttack");  // 함수가 제대로 실행되지 않고 종료된다.
enterSite("cactus");      // 함수가 제대로 실행된다.
```

그 이름에서 알 수 있듯이 return 선언문은 인터프리터에서 함수를 호출한 곳으로 돌아가도록 하는 역할을 한다. return 선언문이 없으면 액션스크립트에서는 함수 본체의 마지막 줄에 return 선언문이 있는 것처럼 처리한다.

```
function say(msg) {
  trace(msg);
  return;         // 이런 경우에는 굳이 return 선언문을 쓰지 않아도 된다.
}
```

return 선언문의 사용 여부와 상관없이 어떤 함수가 종료되고 나면 그 함수를 호출한 선언문 바로 뒤부터 나머지 부분을 계속해서 실행하게 된다. 예를 들면 다음과 같다.

```
say ("Something");  // say() 함수에 있는 코드를 실행시킨다.
// say() 함수가 종료되고 나면 이 지점부터 다시 프로그램이 실행된다.
trace ("Something else");
```

함수 값 리턴

방금 전에 배웠듯이 return 선언문을 사용하면 모든 함수가 종료된다. 하지만 다음과 같은 문법을 이용하면 그 함수를 호출한 스크립트에 어떤 값을 돌려보낼 수도 있다.

```
return expression;
```

expression 값이 함수 호출의 결과가 된다. 예를 들면 다음과 같다.

```
// 두 개의 숫자를 더하는 함수를 정의한다.
function combine(a, b) {
  return a + b;  // 두 인자의 합을 리턴한다.
}

// 함수 호출
var total = combine(2, 1);  // total 값에 3이 대입된다.
```

return 선언문에 의해 리턴된 표현식이나 결과를 그 함수의 '리턴 값(return value)'이라고 부른다.

combine( ) 함수에서는 단지 두 숫자의 합을 계산하여(또는 두 문자열을 합쳐서) 그 결과를 리턴할 뿐이다. sayHi( ) 함수(메시지를 화면에 표시함)나 moveClip( ) 함수(무비 클립의 위치를 바꿈)처럼 어떤 가시적인 결과를 보여주는 것은 아니다. 다음과 같이 함수의 리턴 값을 이용하여 어떤 변수에 새로운 값을 대입할 수도 있다.

```
var total = combine (5, 6);                // total 값이 1이 된다.
var greet = combine ("Hello", "Cheryl")    // greet에 "Hello Cheryl"이
                                           // 저장된다.
```

함수를 호출한 결과는 그냥 일반적인 표현식이 된다. 따라서 그 값을 다른 표현식에서 사용해도 된다. 다음 예에서는 phrase에 "11 people were at the party"라는 값을 대입한다.

```
var phrase = combine(5, 6) + "people were at the party";
```

함수의 리턴 값은 종종 복합 표현식의 일부로 쓰이며, 때에 따라 다른 함수를 호출하기 위한 인자로 쓰이는 수도 있다.

```
var a = 3;
var b = 4;
function sqr(x) {  // 숫자의 제곱을 구한다.
  return x * x;
}
var hypotenuse = Math.sqrt(sqr(a) + sqr(b));
```

위 예제에서 sqr() 함수의 리턴 값을 Math.sqrt() 함수의 인자로 전달하는 방법을 주의깊게 살펴보자. 이와 같은 방법으로 앞에서 만들었던 예제를 다음과 같이 고칠 수도 있다.

```
var phrase = combine (combine(5,6), "people were at the party");
```

이 예에서 안쪽에 있는 combine(5,6)이라는 표현식은 11을 리턴하고 이 값이 다시 바깥쪽에 있는 combine() 함수의 인자가 되어 결국 "people were at the party"라는 문자열과 합쳐지게 된다.

return 선언문에 리턴할 표현식이 없거나 아예 return 선언문을 사용하지 않으면 함수에서는 undefined를 리턴한다. 사실 이로 인해 생기는 오류가 상당히 많다. 예를 들면 아래 예에는 return 선언문이 없기 때문에 아무런 일도 하지 않게 된다.

```
function combine(a, b) {
  var result = a + b;  // 이 결과를 계산하지만, 아무것도 리턴하지 않는다.
}
```

마찬가지로 다음과 같은 코드도 잘못된 코드이다.

```
function combine(a, b) {
  var result = a + b;
  return;  // 리턴 값을 적어주지 않았다.
}
```

계산 결과를 리턴하기 위한 함수를 만들 때는 원하는 값을 리턴하기 위한 return 선언문을 포함시키는 것을 잊지 말도록 주의해야 한다. return 선언문이 없으면 리턴 값이 undefined가 되기 때문에 그 결과를 바탕으로 한 나머지 계산 결과가 틀리게 될 것이 거의 확실하기 때문이다.

함수 리터럴

액션스크립트에서는 임시 함수가 필요하거나 표현식이 들어갈 자리에 함수를 사용하고 싶을 때 편리하게 활용할 수 있는 '함수 리터럴(function literal)'을 만들 수 있다.

함수 리터럴의 문법은 함수 이름이 없고 선언문 블록 맨 뒤에 세미콜론이 들어간다는 점을 제외하면 표준적인 함수를 선언하는 데 쓰이는 문법과 똑같다.

```
function (param1, param2, ... paramn) { statements };
```

여기서 param1, param2, ... paramn은 매개변수 목록이며, 반드시 필요한 것은 아니다. statements는 함수 본체를 이루는 한 개 이상의 선언문이다. 함수 이름을 정해주지 않기 때문에 어떤 변수(또는 배열 원소나 객체 속성)에 저장하지 않으면 함수 리터럴은 다시 사용할 수 없다. 다음과 같이 함수 리터럴을 변수에 저장하면 나중에 사용할 수 있다.

```
// 함수 리터럴을 변수에 저장한다.
var mouseCoords = function () { return [ _xmouse, _  ymouse ]; };

// 함수를 호출한다.
mouseCoords();
```

액션스크립트에서는 자바스크립트에 있는 Function() 생성자를 지원하지 않으므로 자바스크립트에서처럼 실행중에 동적으로 함수를 만들 수 없다.

함수 사용 범위와 유효 기간

변수와 마찬가지로 프로그램 함수는 언젠가는 사라지게 마련이고 무비의 어느 부분에서나 직접 사용할 수 있는 것도 아니다. 함수를 확실히 호출하려면 무비의 서로 다른 위치에서 함수에 접근하는 법을 알아야 하며 함수를 호출하기 전에 그 함수가 실제로 존재하는지 알아야 한다.

함수 사용 범위

프로그램 함수(코드에서 직접 만든 함수)는 다음과 같은 위치에서만 직접 호출할 수 있다.

- 함수 선언이 포함된 무비 클립의 타임라인에 있는 코드
- 함수 선언이 포함된 무비 클립의 타임라인에 있는 버튼

'직접 호출' 한다는 것은 다음과 같이 무비 클립이나 객체에 대한 레퍼런스를 사용하지 않고 이름만 이용하여 함수를 호출하는 것을 의미한다.

```
myFunction();
```

점 구문을 사용하면 무비의 어느 위치에서나 간접적으로 호출할 수 있다. 다른 위치에서 변수를 사용하는 경우와 마찬가지로 다음과 같이 그 함수를 포함하고 있는 무비 클립에 대한 경로를 제공해야 한다.

```
myClip.myOtherClip.myFunction();
_parent.myFunction();
_root.myFunction();
```

메인 타임라인에 있는 rectangle이라는 클립 인스턴스에 area()라는 함수가 들어있다고 가정해 보자. 다음과 같이 rectangle에 대한 절대 경로를 이용하면 무비의 어느 위치에서든지 area() 함수를 호출할 수 있다.

```
_root.rectangle.area();
```

떨어져 있는 무비 클립 타임라인에서 어떤 함수를 참조하려면 '2장. 변수' 에서 다른 위치에 있는 변수를 참조하기 위해 사용한 것과 같은 규칙을 이용해야 한다 ('13장. 무비 클립' 에서 원격 함수를 호출하는 것에 대해 더 배우게 될 것이다).

모든 함수가 무비 클립 타임라인에 붙어있는 것은 아니다. 프로그램 함수 중에는 사용자 정의 객체 또는 내장 객체에 포함되는 것도 있다. 함수를 객체에서 정의하면 호스트 객체를 통해서만 그 함수를 사용할 수 있다. 메소드(객체에 포함된 함수)의 사용 범위에 대한 내용은 12장에서 알아보기로 하자.

함수의 유효 기간

무비 클립 타임라인에서 정의한 함수는 그 클립이 스테이지에서 없어지면 함께 없어진다. 더 이상 존재하지 않는 클립에서 정의한 함수를 호출하려고 하면 아무런 경고 메시지 없이 함수 호출에 실패하게 된다. 메인 타임라인은 무비가 실행되는 동안 절대 사라지지 않으므로 함수가 없어지는 것을 막으려면 메인 무비 타임라인에서 함수를 정의하면 된다.

프레임의 스크립트에 있는 함수는 스크립트가 들어있는 프레임에 플레이헤드가 처음 들어갈 때 초기화되면서 사용할 수 있게 된다. 따라서 같은 스크립트 안에 포함되어 있다면 다음과 같이 함수를 선언하기 전에 함수를 호출하는 것도 가능하다.

```
// 함수를 선언하기 전에 tellTime() 함수를 호출한다.
tellTime();

// tellTime() 함수를 선언한다.
function tellTime() {
  var now = new Date();
  trace(now);
}
```

함수 영역

액션스크립트 선언문에는 영역 또는 그 선언문이 유효한 범위가 정해져 있다. 무비 클립에 선언문을 추가하면 그 선언문의 영역은 선언문을 포함하고 있는 클립으로 제한된다. 예를 들어 다음 코드에서는 대입 선언문에서 score라는 변수를 참조한다.

```
score = 10;
```

만약 위 선언문이 clipA에 들어 있다면 인터프리터에서는 clipA에 있는 score 변수의 값을 설정한다. 이 선언문의 영역은 clipA로 제한되어 있기 때문이다. 만약 이 선언문이 clipB에서 쓰인다면 이 선언문의 영역이 clipB로 제한되므로 clipB에 있는 score 변수의 값이 새로 설정된다. 즉 선언문의 위치로부터 영역, 즉 그 선언문의 효과가 결정된다.

함수 본체에 있는 선언문은 '지역 영역(local scope)'이라는 별도의 영역 내에서 동작한다. 함수의 지역 영역은 그 함수만을 위한 공중전화 부스와 비슷하다고 보면 된다. 이 영역은 그 함수를 포함하고 있는 클립이나 객체의 영역과는 다르다. 함수의 지역 영역은 함수가 호출될 때 만들어지고 함수가 종료될 때 없어진다. 함수 본체에서 쓰인 변수 값을 알아낼 때 인터프리터에서는 우선 함수 자체의 영역을 검색한다.

예를 들어 함수의 매개변수는 함수의 지역 영역 안에서만 정의된다. 그 함수를 포함하고 있는 타임라인의 영역에서 정의되는 것이 아니다. 따라서 매개변수는 함

수가 실행되는 도중에 함수 본체에 있는 선언문에서만 사용할 수 있다. 함수 밖에 있는 선언문에서는 함수의 매개변수를 사용할 수 없다.

함수의 지역 영역에서는 함수에서만 사용하기 위한 임시 변수를 위한 공간을 제공한다. 이렇게 하면 함수 변수와 타임라인 변수 중 이름이 같은 변수 때문에 생기는 혼동을 방지할 수 있고 전체적인 메모리 사용량을 줄일 수 있다.

영역 사슬

함수는 그 함수만의 지역 영역 안에서 작동하지만 일반적인 타임라인 변수도 함수 본체에서 사용할 수 있다. 인터프리터에서 변수를 찾을 때는 우선 함수의 지역 영역을 찾아보고 그 변수가 없으면 그 함수를 포함하고 있는 객체나 무비 클립을 검색한다.

예를 들어 clipA라는 무비 클립의 타임라인에 firstName이라는 변수가 정의되어 있다고 가정해 보자. 또한 getName()이라는 함수도 clipA에서 선언한다고 가정하자. getName()에 있는 trace() 선언문에서는 firstName이라는 변수를 참조한다.

```
firstName = "Christine";

function getName () {
  trace(firstName);
}

getName();
```

getName() 함수를 호출하면 인터프리터에서는 firstName 값을 알아내야 한다. getName()의 지역 영역에는 firstName이라는 변수가 없으므로 인터프리터에서는 clipA의 타임라인에서 firstName을 검색한다. 여기에 firstName이 있으므로 그 변수 값인 "Christine"을 출력한다.

위 예제에서는 getName() 자체에 firstName이라는 매개변수나 변수가 정의되어 있지 않기 때문에, getName() 함수의 본체에 있는 trace() 선언문에서 타임라인 변수를 참조할 수 있었다. 이제 getName() 함수에 firstName이라는 매개변수를 추가하면 어떻게 되는지 알아보자.

```
firstName = "Christine";

function getName (firstName) {
  trace(firstName);
}

getName("Kathy");
```

이번에는 getName()을 호출할 때 firstName 매개변수에 "Kathy"라는 값을 대입한다. 인터프리터에서 trace() 선언문을 실행하면 지역 영역에서 firstName 매개변수를 찾아내고 그 값이 "Kathy"라는 것을 알게 된다. 따라서 이번에는 "Chrisitine"이 아닌 "Kathy"라는 결과가 출력된다. 타임라인 변수 중에도 firstName이 있긴 하지만 함수의 지역 변수인 firstName이 우선 순위가 더 높다.

이 예제에서는 '영역 사슬(scope chain, 변수나 객체 속성에 대한 레퍼런스를 알아낼 때 인터프리터에서 사용하는 계층구조)'이 작동하는 방법을 볼 수 있다. 타임라인에 들어 있는 함수에서는 영역 사슬이 두 단계(지역 영역과 그 함수가 들어 있는 무비 클립의 영역)로 구성된다. 하지만 사용자 정의 객체나 클래스에 함수를 집어넣을 때는 영역 사슬이 더 여러 단계로 나뉠 수 있다. 자세한 내용은 12장의 '객체 속성 상속' 절에서 살펴보도록 하자.

우리가 함수 본체에서 사용하고자 하는 변수와 속성의 영역이 선언문의 영역과 다른 경우에는 점 구문을 이용하여 직접 어느 쪽을 사용할 것인지 지정해야 한다. 예를 들면 다음과 같다.

```
function dynamicGoto() {
  // 함수의 지역 영역 밖에 있는 속성을 사용한다.
  _root.myClip.gotoAndPlay(_root.myClip.targetFrame);
}
```

함수의 영역 사슬은 함수를 호출한 부분이 아니라 선언한 부분을 기준으로 상대적으로 결정된다는 점에 주의하자. 즉 함수 본체에 있는 코드의 영역은 그 함수를 호출한 선언문이 들어 있는 무비 클립이 아니라 그 함수를 선언한 부분의 무비 클립의 영역에 포함된다.

아래 예는 영역 사슬을 잘못 적용한 경우를 보여준다.

```
// 메인 무비 타임라인의 코드
// 이 함수의 영역 사슬은 메인 무비를 포함한다.
function rotate(degrees) {
  _rotation += degrees;
}
```

만약 _root.rotate를 이용하여 clipA를 회전시키려고 하면 clipA뿐만 아니라 메인 무비 전체를 회전시키는 결과를 낳게 된다.

```
// clipA의 타임라인에 들어가는 코드
_root.rotate(30);  // 이렇게 하면 무비 전체가 회전된다.
```

이러한 상황에서는 회전시킬 클립을 rotate() 함수의 인자로 전달하여 문제를 해결할 수 있다.

```
function rotate (theClip, degrees) {
   theClip._rotation += degrees;
}

// rotate()를 호출할 때 clipA에 대한 레퍼런스를 전달한다.
_root.rotate(clipA, 30);
```

지역 변수

함수의 지역 영역에서 정의된 변수는 '지역 변수(local variable)'라고 부른다. 매개변수를 비롯한 지역 변수는 그 변수가 정의된 함수 본체 안에 있는 선언문에서만 사용할 수 있으며 그 함수가 실행되는 동안만 존재한다. 지역 변수를 만들려면(매개변수는 자동으로 지역 변수가 되므로 따로 정의할 필요가 없다), 다음과 같이 함수 안에서 var 선언문을 이용하기만 하면 된다.

```
function funcName() {
  var temp = "just testing!";  // temp라는 지역 변수를 선언한다.
}
```

지역 변수는 어떤 정보를 임시로 저장하는 데 쓰인다. 아래 예제에서는 lastSpacePlusOne이라는 지역 변수를 만들어서 중간 결과를 저장하는 데 사용한다. 모든 지역 변수가 그렇듯이 이 변수도 함수가 끝날 때 제거된다.

```
function getLastWord(text) {
  var lastSpacePlusOne = text.lastIndexOf(" ") + 1;  // 지역 변수
  var lastWord = text.subString(lastSpacePlusOne, text.length);
                                                      // 지역 변수
  return lastWord;
}

// "word"가 출력된다.
trace(getLastWord("Tell me the last word"));

// undefined 가 출력된다. lastSpacePlusOne은 지역 변수이므로
// getLastWord() 함수 밖에서는 사용할 수 없다.
trace(lastSpacePlusOne);
```

함수가 종료되면서 지역 변수의 유효 기간이 끝나면 그 변수와 연결된 메모리가 비워진다. 지역 변수를 이용하여 임시 값을 저장하면 프로그램에서 메모리를 절약할 수 있다. 게다가 지역 변수를 선언하는 경우에는 같은 이름의 타임라인 변수와 혼동할까봐 걱정하지 않아도 된다.

물론 함수에서 사용하는 모든 변수가 지역 변수일 필요는 없다. 함수 안에서 타임라인 변수를 사용할 수 있다는 것은 이미 배웠다. 하지만 단지 읽을 수만 있는 것이 아니라 타임라인 변수를 새로 만들고 그 변수에 새로운 값을 대입할 수도 있다. 함수 안에 있는 변수 대입 선언문도 지역 변수로 선언하지 않는다면 함수 영역에만 국한되는 것이 아니라 타임라인 영역에 속하게 된다. 아래 예제에서 x는 지역 변수이지만 y와 z는 타임라인 변수이다.

```
var z = 1;

function createVars() {
  var x = 10;    // 지역 변수 x를 만든다.
  y = 13;        // 타임라인 변수 y를 만든다.
  z = 2;         // 타임라인 변수 z를 변경한다.
}
createVars();// 함수를 호출한다.
trace(x);    // x는 정의되지 않은 값이다(함수가 종료되면 x도 함께 없어진다).
trace(y);    // y는 13이다(함수가 종료된 후에도 y는 남는다).
trace(z);    // z는 2이다(함수가 종료된 후에도 z는 변경된 채로 남는다).
```

타임라인 변수를 만드는 규칙은 함수가 어떤 객체의 메소드인 경우에도 그대로 적용된다. 12장에 들어가기 전에는 객체에 대해 자세히 다루진 않겠지만 객체지향 프로그래밍에 익숙한 독자들을 위해 〔예제 9-2〕를 통해 그러한 사실을 확인해 보도록 하자. x는 newFunc()의 지역 영역이나 newObj의 속성에 속하지 않으므로 타임라인 상에서 정의되는 변수가 된다.

〔예제 9-2〕 객체 메소드 안의 변수 영역

```
newObj = new Object();                          // 변수를 만든다.
newObj.newFunc = function() { x = 12; };    // 새로운 메소드를 추가한다.
newObj.newFunc ();    // 메소드를 호출한다.

// x를 체크해 보자.
trace("x is" + x);                  // x의 값은 12이다.
trace("newObj.x is" + newObj.x);    // newObj.x는 정의되지 않은 속성이다.
```

함수 매개변수 다시 보기

이제 함수가 작동하는 방법에 어느 정도 익숙해졌으므로 함수 매개변수에 대해 다시 생각해 보도록 하자. 지금 알아보려는 주제를 이해하려면 객체에 대한 지식이 약간 필요하기 때문에, 프로그래밍을 처음 배우는 독자라면 이 절을 읽기 전에 12장을 미리 읽어두는 편이 좋을 것이다.

매개변수의 개수

이미 앞에서 함수의 매개변수는 함수를 만들 때 선언한다는 것을 배웠다. 다음과 같은 문법을 다시 떠올려 보자.

```
function funcName (param1, param2, param3,...paramn) {
  statements
}
```

약간 이상하게 느껴질지 몰라도 함수에 전달하는 인자의 개수는 함수를 선언할 때 적어준 매개변수의 개수와 다를 수도 있다. 함수에서는 예상하고 있던 개수보다

많건 적건 상관없이 임의의 매개변수를 받아들일 수 있다. 함수를 호출할 때 선언했던 것보다 적은 매개변수를 사용하면 전달되지 않은 매개변수 값은 undefined로 설정된다. 아래 예를 살펴보자.

```
function viewVars (x, y, z) {
  trace ("x is" + x);
  trace ("y is" + y);
  trace ("z is" + z);
}

viewVars(10);   // "x is 10", "y is", "z is"라고 출력된다.
                // y와 z의 값은 undefined 이므로 공백으로 출력될 뿐이다.
```

만약 함수를 호출할 때, 선언할 때 사용한 것보다 많은 인자를 전달하면 예상보다 많이 전달된 인자는 arguments 객체를 이용하여 액세스할 수 있다(초과된 인자는 그 이름이 선언되지 않은 상태이기 때문에 선언할 때 이름이 정해진 매개변수처럼 이름을 이용하여 액세스할 수 없다).

arguments 객체

어떤 함수를 실행하더라도 arguments라는 내장 객체를 이용하면 그 함수와 관련된 세 가지 정보(함수에 전달된 인자의 매개변수 개수, 각 매개변수 값이 저장된 배열, 실행 중인 함수의 이름)를 알아낼 수 있다. arguments 객체는 사실 배열과 다른 속성을 가지고 있는 객체 사이에서 태어난 특별한 잡종에 해당한다.

arguments 배열로부터 매개변수 값을 구하는 방법

arguments 배열을 이용하면 함수를 선언할 때 사용한 선언문에서 그 매개변수를 정의하지 않았더라도 임의의 함수의 매개변수 값을 알아낼 수 있다. 매개변수를 사용하기 위해 다음과 같이 arguments 배열의 인덱스를 조사한다.

```
arguments[n]
```

여기서 n은 사용하고자 하는 매개변수의 인덱스이다. 첫째 매개변수(함수 호출 선언문에서 맨 왼쪽에 있는 매개변수)는 0번 인덱스에 저장되며 arguments[0]이라는

표현을 사용하여 참조할 수 있다. 그 뒤에 나오는 인자들도 순서대로 저장되므로 둘째 인자는 arguments[1], 셋째 인자는 arguments[2],...와 같은 식으로 쓸 수 있다.

함수 내부에서는 다음과 같이 arguments의 원소 개수를 조사하여 현재 실행 중인 함수에 전달된 인자의 개수를 알아낼 수 있다.

```
arguments.length
```

〔예제 9-3〕에 나온 코드를 이용하면 함수에 전달된 모든 매개변수를 확인해 보고 그 결과를 Output 창에 출력할 수 있다.

〔예제 9-3〕 개수를 미리 알지 못하는 매개변수를 화면에 출력하는 방법

```
function showArgs() {
  for (var i = 0; i < arguments.length; i++) {
    trace("Parameter" + (i + 1) + "is" + arguments[i]);
  }
}

showArgs(123, 23, "skip intro");

// 다음과 같이 출력된다.
Parameter 1 is 123
Parameter 2 is 23
Parameter 3 is skip intro
```

arguments 배열을 이용하면 임의의 개수의 매개변수를 받아들일 수 있는 매우 융통성 있는 함수를 만들 수 있다.

스테이지에 있는 모든 중복된 무비 클립을 제거할 수 있는 일반적인 함수는 다음과 같다.

```
function killClip() {
  for (var i = 0; i < arguments.length; i++) {
    arguments[i].removeMovieClip();
  }
}

killClip(clip10, clip5, clip13);
```

연습문제: 앞에서 만들어 본 combine() 함수를 고쳐서 몇 개의 인자라도 받아들일 수 있도록 만들어 보자. 매개변수의 개수를 마음대로 정하면 좋을만한 함수에는 어떤 것들이 있을까? 임의의 숫자 목록을 받아들여 그 평균 값을 계산하는 함수를 만들어 보자(힌트: 모든 인자의 합을 구한 다음 인자의 개수로 나누기만 하면 된다).

callee 속성

지금까지 살펴본 것처럼 arguments 배열을 이용하면 함수의 매개변수를 알아낼 수 있다. arguments 객체에 포함된 속성은 callee 하나뿐인데 이 값은 실행하고 있는 함수에 대한 레퍼런스를 리턴한다. 호출하고 있는 함수의 이름을 알고 있는 경우가 대부분이지만 함수 리터럴로 만든 익명 함수를 실행하는 경우에는 callee 속성이 유용하게 쓰일 수도 있다. 〔예제 9-4〕에는 이름이 없이도 재귀호출을 수행하는 함수 리터럴로 만든 함수가 나와 있다. 자세한 내용은 뒷부분에 나오는 '재귀 함수' 부분을 참조하기 바란다.

〔예제 9-4〕 callee를 이용한 카운트다운

```
count = function (x) {
  trace(x);
  if (x > 1) {
    arguments.callee(x - 1);
  }
}
count(25);
```

물론 익명 재귀 함수를 사용하지 않고도 카운트다운을 할 수 있다. 재귀호출을 이용하는 함수의 예는 잠시 후에 알아보기로 하자.

원시 데이터 및 복합 데이터 매개변수

매개변수에 대해 한 가지 더 생각해 봐야 할 것이 있다. 바로 원시 데이터와 복합 데이터를 함수에 전달할 때의 차이점이다.

함수에 원시 데이터 값을 인자로 전달하면 함수에서는 그 데이터의 원본이 아닌 복사본을 전달받는다. 따라서 매개변수 값을 함수 내에서 바꾼다고 해도 함수 밖에

있는 원본 값은 바뀌지 않는다. [예제 9-5]에서 variableName 값은 25로 설정된다. setValue() 함수에서 그 값을 10으로 바꾸더라도 y 값에는 전혀 변화가 없다.

[예제 9-5] 원시 데이터는 값으로 전달된다

```
var y = 25;
function setValue(variableName) {
  variableName = 10;
}
setValue(y);
trace("y is" + y);  // "y is 25"이 출력된다.
```

이처럼 원시 데이터는 값으로 전달된다. 하지만 복합 데이터를 함수의 인자로 전달하면 함수에서는 데이터의 복사본이 아니라 인자의 원본과 같은 데이터를 가리키는 레퍼런스를 전달받게 된다. 따라서 매개변수를 통해 데이터를 변경하게 되면 함수 밖에 있는 데이터인 경우에도 원본 데이터, 즉 그 매개변수가 가리키고 있는 데이터도 변경된다. 따라서 복합 데이터는 레퍼런스로 전달된다.

[예제 9-6]에서는 매개변수인 myArray를 변경하면 함수 밖에 있는 boys 배열도 변경된다. 둘 다 메모리에서 같은 데이터를 가리키고 있기 때문이다.

[예제 9-6] 레퍼런스로 전달된 복합 데이터 인자 변경

```
// 배열 생성
var boys = ["Andrew", "Graham", "Derek"];

// setValue() 함수에서 배열의 첫 번째 원소 값을 설정한다.
function setValue(myArray) {
  myArray[0] = "Sid";        // 배열의 첫 번째 원소 값을 설정한다.
}

// 함수에 배열을 전달한다.
setValue(boys);

// 배열 원소의 값을 체크한다.
trace("Boys:" + boys);       // "Boys: Sid,Graham,Derek"이라고 출력된다.
```

함수 안에서 배열의 각 원소에 원래 있던 값을 지우고 새로운 값을 대입하는 것은 가능하지만 만약 매개변수에 새로운 값을 대입하면 원래 인자와의 관계가 끊어지게 된다. 그 후에 매개변수를 변경시키면 원본 인자에 아무런 변화도 일어나지 않

게 된다. [예제 9-7]에서도 boys 배열을 인자로 전달하지만 myArray 매개변수는 바로 girls로 설정되기 때문에 myArrary를 변경하면 boys 배열이 아니라 girls 배열의 내용이 바뀐다.

[예제 9-7] 인자와 매개변수 사이의 연결 끊기

```
// 두 개의 배열을 만든다.
var boys  = ["Andrew", "Graham",  "Derek"];
var girls = ["Alisa",  "Gillian", "Daniella"];

// setValue()에서는 전달된 배열을 무시하고 girls 배열을 수정한다.
function setValue(myArray) {
  myArray = girls;   // myArray에서 boys가 아닌 girls 배열을 가리키게 된다.
  myArray[0] = "Mary";  // girls의 첫 번째 원소를 바꾼다.
}

// setValue() 함수에 boys 배열을 전달한다.
setValue(boys);

trace("Boys:" + boys);   // "Boys: Andrew,Graham,Derek"이 출력된다.
trace("Girls:" + girls); // "Girls: Mary,Gillian,Daniella"가 출력된다.
```

원시 데이터와 복합 데이터에 대한 내용은 '15장. 고급 주제'에서 더 자세히 다룰 것이다.

재귀 함수

함수 본체에서 그 함수 자체의 이름을 사용하여 자기 자신을 호출하는 함수를 '재귀 함수(recursive function)'라고 부른다. 아래 예는 재귀호출의 원리를 보여주는 간단한 예이다. 하지만 (마치 두 개의 거울을 마주 놓으면 똑같은 상이 무한히 많이 생기는 것과 마찬가지로) 코드에서 trouble() 함수를 계속해서 호출하기 때문에 플래시에서 메모리를 모두 소모하여 오류가 발생한다.

```
function trouble() {
  trouble();
}
```

따라서 실제 재귀 함수를 사용할 때에는 주어진 조건이 만족되는 경우에만(그렇게 해야 무한 재귀호출을 막을 수 있다) 자기 자신을 호출하도록 만든다. [예제 9-4]에서는 재귀호출을 이용하여 주어진 숫자부터 1까지 카운트다운을 한다. 물론 이러한 코드는 재귀호출을 사용하지 않고도 구현할 수 있다.

재귀호출을 사용하는 가장 고전적인 것 중 하나로 어떤 숫자의 팩토리얼을 계산하는 것이 있다. 3 팩토리얼(3!이라고 쓴다)은 3*2*1=6이 된다. 5 팩토리얼은 5*4*3*2*1=120이다. [예제 9-8]에 재귀호출을 이용한 팩토리얼 함수가 나와 있다.

[예제 9-8] 재귀호출을 이용한 팩토리얼 계산

```
function factorial(x) {
    if (x < 0) {
      return undefined;  // 오류
    } else if (x <= 1) {
      return 1;
    } else {
      return x * factorial(x-1);
    }
}
trace (factorial(3));  // 6이 출력된다.
trace (factorial(5));  // 120이 출력된다.
```

물론 팩토리얼을 계산하는 방법이 이 방법밖에 없는 것은 아니다. [예제 9-9]에 나온 것처럼 루프를 이용하면 재귀호출을 사용하지 않고도 팩토리얼을 계산할 수 있다.

[예제 9-9] 재귀호출을 이용하지 않은 팩토리얼 계산

```
function factorial(x) {
    if (x < 0) {
      return undefined; // 오류
    } else {
      var result = 1;
      for (var i = 1; i <= x; i++) {
        result = result * i;
      }
      return result;
    }
}
```

〔예제 9-8〕과 〔예제 9-9〕는 같은 문제를 해결하는 두 가지 방법을 보여준다. 재귀적인 방법을 사용하면 '6 팩토리얼은 5 팩토리얼에 6을 곱한 것이고 5 팩토리얼은 4 팩토리얼에 5를 곱한 것이고...' 이런 식으로 6 팩토리얼을 계산한다. 재귀적인 방법을 사용하지 않는 경우에는 1부터 x까지 루프를 돌면서 모든 값을 곱한다.

재귀적인 방법과 재귀적이지 않은 방법, 두 가지 중 어떤 방법이 나을지는 주어진 문제에 따라 조금씩 다르다. 어떤 문제는 재귀호출을 사용하면 더 쉽게 해결할 수 있지만 재귀호출을 이용하면 재귀호출을 이용하지 않은 경우보다 계산 속도가 조금 느려진다. 재귀호출은 데이터 구조가 얼마나 여러 단계로 내려갈지 잘 모르는 경우에 가장 적합하다. 예를 들어 어떤 하위 디렉토리에 있는 모든 파일의 목록을 (그 밑에 있는 모든 디렉토리와 모든 파일을 포함하여) 출력하는 경우에는 재귀호출을 사용하지 않으면, 하위 디렉토리가 매우 복잡한 경우에도 제대로 작동하는 해결법을 구현하기가 힘들다. 재귀호출을 사용하는 해결법을 유사코드로 대강 만들어 본다면 다음과 같이 구현할 수 있다.

```
function listFiles (directoryName) {
  do (directoryName에 있는 다음 아이템을 체크) {
    if (이 아이템이 하위 디렉토리인 경우) {
      // 새로운 하위 디렉토리로 내려가서 이 함수를 재귀적으로 호출한다.
      listFiles(subDirectoryName);
    } else {
      // 파일 이름을 출력한다.
      trace (filename);
    }
  } while (아직 체크할 아이템이 있는 경우);
}
```

'3부. 레퍼런스'에서 XML 객체에 대해 배울 때 재귀호출을 이용하여 XML 문서에 있는 모든 요소의 목록을 출력하는 코드를 구현해 보도록 하자.

내부 함수

지금까지 사용자 정의 함수를 만드는 법에 대해 배웠지만 액션스크립트에는 매우 다양한 내부 함수(사람의 언어로 치면 동사에 해당한다)가 포함되어 있다. 데이터

를 조작하는 데 쓰이는 내부 함수는 이미 조금 배웠다. 또한 플래시 무비와 사용자 환경을 제어하는 함수도 조금 배웠다.

예를 들어 무비 클립의 플레이헤드를 조작하고 싶다면 프레임 번호를 인자로 사용하여 gotoAndPlay() 함수를 호출하면 된다.

```
gotoAndPlay(5);
```

프로그래밍을 처음 배우는 독자라면 뭔가 새로운 발견을 한 기분이 들지도 모른다. 앞에서 배운 사용자 정의 함수와 마찬가지로 내부 함수를 호출할 때도 함수 호출 연산자(괄호)와 인자 목록(이 경우에는 숫자 5)을 이용한다는 것을 깨달을 수 있을 것이다. gotoAndPlay()와 같은 내부 함수도 우리가 직접 만드는 함수와 똑같은 방법으로 이용한다. 물론 내부 함수는 사용자 정의 함수와는 다른 일을 처리하는 데 쓰이고 액션스크립트 내부 함수로 할 수 있는 일을 굳이 사용자 정의 함수를 만들어서 할 필요는 없다. 하지만 다른 사용자 정의 함수와 마찬가지로 각 내부 함수에는 이름, 인자, 리턴 값이 있다.

플래시에서는 옛날부터 gotoAndPlay를 '액션'이라고 부르지만 이제는 사실 액션이라기보다는 함수라고 부르는 것이 더 적합하다. 6장에서 플래시 액션 중에는 선언문인 것도 있고 내부 함수인 것도 있다는 것을 배웠다. 이제 선언문과 함수를 모두 자세히 익혔으므로 어떤 것이 선언문이고 어떤 것이 내부 함수인지 확실히 구분할 수 있을 것이다.

몇몇 액션은 함수라고 보면 더 외우기도 쉽고 구문에서 사용하기도 쉬워진다. 예를 들어 플래시 플레이어에 무비를 로딩하려면 loadMovie()라는 함수가 있다는 사실, 그리고 그 함수에서 어떤 인자를 사용하는지를 알아야 한다. 3부를 살펴보면 그러한 정보를 얻을 수 있고 손쉽게 다음과 같은 선언문을 만들 수 있다.

```
loadMovie("myMovie.swf", 1);
```

함수를 사용하는 것에 익숙해지고 나면 매우 간단하다.

액션스크립트의 내부 함수는 매우 다양하다. 내부 함수를 이용하면 무비의 요소를 제어할 수도 있고 사운드 볼륨부터 편집할 수 있는 텍스트 필드에서 선택한 텍스트의 분량에 이르는 거의 모든 정보를 알아내고 바꿀 수도 있다. 액션스크립트의 내부 함수는 3부에 모두 수록해 놓았다. 모든 문법을 외울 필요는 없지만 시간이 날 때마다 가끔씩 3부의 내용을 훑어보고 사용할 수 있는 함수의 유형에 익숙해지도록

노력해 보자.

내부 함수를 사용할 수 있는 범위

내부 함수 중에는 어디서든 사용할 수 있는 것도 있지만 객체를 통해서 호출해야 하는 것도 있다. 어떤 함수가 전역 함수라면 어디에서나 사용할 수 있다. 하지만 어떤 함수가 한 객체의 메소드라면 적절한 객체와 함께 사용해야만 한다. 아직 객체에 대해 자세히 배우지 않았고 대부분의 액션스크립트 내부 함수는 객체 메소드이므로, 내장 메소드에 대한 내용은 12장에서 다시 알아보기로 하자.

객체로서의 함수

액션스크립트에서 함수는 기술적으로 볼 때 내장 객체의 특별한 유형에 속한다. 방금 한 말이 무슨 뜻인지, 그리고 프로그래머가 함수로 할 수 있는 일에 어떤 영향을 끼치는지 알아보기로 하자.

함수에 함수 전달하기

조금 이상하게 들릴지 모르지만 모든 함수는 다음과 같이 다른 함수의 인자로 사용할 수 있다.

```
function1(function2);
```

function2 뒤에 괄호가 없으면 인터프리터에서는 function2()를 호출하는 대신 그 함수의 '객체 레퍼런스'를 function1()의 인자로 전달한다. 즉 function1()에서는 function2()의 리턴 값이 아닌 function2 함수 자체를 전달받게 된다. 객체는 레퍼런스로 전달되기 때문에 함수 인식자를 다른 함수에 전달하더라도 함수가 바뀌거나 하지는 않는다. 이렇게 전달받은 함수는 다음과 같이 실행시킬 수 있다.

```
function doCommand(command) {
  command();      // 전달받은 함수를 실행한다.
}
```

```
// 몇 가지 예제:
doCommand(stop);  // stop()이라는 내부 함수를 전달한다(현재 무비를 정지시킴).
doCommand(play);  // play()라는 내부 함수를 전달한다(현재 무비를 시작시킴).
```

함수도 일종의 객체이므로 다른 데이터와 똑같이 취급할 수 있다. 아래 예제에서는 gp라는 변수에 내부 함수인 gotoAndPlay 함수를 대입하여 함수를 더 간단하게 표현할 수 있도록 만든다.

```
gp = gotoAndPlay;  // gotoAndPlay()를 짤막하게 표현할 수 있는
                   // 함수 이름을 만든다.
gp(25);            // 레퍼런스를 이용하여 gotoAndPlay() 함수를 호출한다.
```

함수를 객체로 간주하여 전달하거나 저장하는 것 외에도 함수의 객체성을 이용하여 다음과 같이 함수에 속성을 추가할 수도 있다.

```
// 함수를 만든다.
function myFunction () {
  trace(myFunction.x);
}

// 함수에 속성을 추가한다.
myFunction.x = 15;

// 함수를 호출하여 속성 값을 알아낸다.
myFunction();  // 15가 출력된다.
```

함수에 속성을 추가하면 타임라인에 변수를 추가하지 않고도 함수 호출 사이에 어떤 정보의 상태를 계속 유지할 수 있다.

함수의 속성을 이용하면 함수를 호출할 때마다 값이 바뀌지 않으면서도 지역 변수 역할을 하는 변수를 만들 수 있다. 이러한 기능은 단일 인식자를 이용하여 함수를 호출해야 하는 경우에 유용하다. 아래 함수는 무비 클립을 복사하여 그 복사된 클립에 고유의 이름과 레이블을 추가하는 일반적인 프로그램의 예이다.

```
makeClip.count = 0;     // makeClip()의 속성을 정의한다(함수는 코드가
                        // 실행되기 전에 정의되므로 makeClip() 함수는
                        // 이미 존재하고 있는 상태이다).
```

```
// 전달된 클립의 복사본을 만들어서 자동으로 이름을 붙여준다.
function makeClip (theClip) {
  // 클립 카운터에 1을 더한다.
  makeClip.count++

  // 클립을 복사해서 고유 이름과 깊이를 대입한다.
   theClip.duplicateMovieClip(theClip._name + makeClip.count,
makeClip.count);
}

makeClip(square);   // makeClip()을 이용하여 square의 복사본을 만든다.

square1._x += 100;  // 복사된 square를 오른쪽으로 옮긴다.
```

코드 집중화

함수의 가장 중요한 기능을 하나만 꼽으라고 한다면 코드를 재사용할 수 있다는 점일 것이다. 즉 무비의 어느 곳에서나 실행시킬 수 있는 재사용 가능한 코드를 만들 수 있다. 함수는 원격에서 실행할 수 있기 때문에 한 장소에 모아서 저장해 두면 코드를 손쉽게 관리하고 업그레이드할 수 있다.

예를 들어 똑같은 기능을 가진 세 개의 버튼이 서로 다른 세 무비 클립에 들어간다고 가정해 보자. 코드를 세 군데 모두 따로 적는 것보다는 메인 타임라인에 있는 함수에 코드를 집어넣고 그 함수를 각 버튼에서 필요할 때마다 호출하는 것이 더 낫다. 이렇게 하면 시간도 절약되고 오류가 발생할 가능성도 줄일 수 있으며 한꺼번에 세 버튼의 기능을 모두 업데이트하기도 편하다. 코드를 한 곳으로 집중시켜 놓았기 때문에 테스트하기도 더 좋고 문제가 생겼을 때도 해결하기가 훨씬 쉬워진다. 만약 버튼 중 하나는 제대로 작동하지 않는데 나머지 두 개는 제대로 작동하고 있다면, 함수의 문제가 아니라 그 함수를 호출하는 부분에 문제가 있을 가능성이 높다.

객관식 퀴즈 다시 보기

〔예제 1-1〕에서는 프로그래밍을 처음 배우는 독자들을 위해 간단한 스크립트가 포함된 무비(객관식 퀴즈)를 만들어 보았다. 이제 이 퀴즈를 다시 살펴보고 코드를 집중화하는 것에 대해 알아보기로 하자.

이번에 만들 퀴즈의 레이어 구조는 전에 사용했던 것과 똑같다. 여기서는 첫 번째 프레임, quizEnd 프레임 및 버튼에 있는 코드만 고쳐보도록 하자.

퀴즈의 원본과 개정된 버전의 .fla 파일은 모두 온라인 코드 창고에서 구할 수 있다.

퀴즈 코드를 함수로 만들기

처음에 객관식 퀴즈를 만들 때는 무비 전반에 걸쳐 코드를 여기저기에 입력했다. 퀴즈와 관련된 논리는 세 군데에 나와 있다. 우선 첫 번째 프레임에서 퀴즈를 초기화하고 버튼에서는 사용자가 입력한 답을 추적하고 quizEnd 프레임에서는 사용자의 점수를 계산한다. 이제 이러한 작업을 모두 첫 번째 프레임에서 처리하여 퀴즈와 관련된 논리식을 한 군데로 집중시켜 보도록 하자. 퀴즈를 초기화할 때는 첫 번째 버전에서와 마찬가지로 간단한 선언문을 이용할 것이다. 하지만 사용자가 입력한 답을 추적하고 사용자의 점수를 계산하는 부분에서는 각각 answer()와 gradeUser()라는 함수를 이용해 보자.

〔예제 9-10〕에 퀴즈의 첫 번째 프레임에 들어갈 코드가 나와 있다. 이 부분에 퀴즈를 실행하는 데 필요한 논리 표현식이 모두 들어간다. 우선 코드를 자세히 읽어보고 차근차근 분석해 보자.

〔예제 9-10〕 객관식 퀴즈, 버전 2

```
// 첫 번째 문제에서 무비를 멈춘다.
stop ();

//  메인 타임라인 변수를 초기화한다.
var displayTotal;          // 사용자의 점수를 출력하기 위한 텍스트 필드
var numQuestions = 2;      // 퀴즈 문제 번호
var q1answer;              // 문제 1에 대한 사용자의 답
var q2answer;              // 문제 2에 대한 사용자의 답
```

```
var totalCorrect = 0;      // 정답 개수
var correctAnswer1 = 3;    // 문제 1의 정답
var correctAnswer2 = 2;    // 문제 2의 정답

//  사용자의 답을 등록하는 함수
function answer (choice) {
  answer.currentAnswer++;
  set ("q" + answer.currentAnswer + "answer", choice);
  if (answer.currentAnswer == numQuestions) {
    gotoAndStop ("quizEnd");
  } else {
    gotoAndStop ("q" + (answer.currentAnswer + 1));
  }
}

//  사용자의 점수를 계산하는 함수
function gradeUser() {
  // 사용자의 답 중 정답의 개수를 센다.
  for (i = 1; i <= numQuestions; i++) {
    if (eval("q" + i + "answer") == eval("correctAnswer" + i)) {
      totalCorrect++;
    }
  }

  // 온스크린 텍스트 필드에 사용자의 점수를 출력한다.
  displayTotal = totalCorrect;
}
```

가장 먼저 할 일은 자동으로 다음 문제로 넘어가지 않도록 stop() 함수를 이용하여 무비를 멈추는 일이다. 그리고 나서 퀴즈의 타임라인 변수를 초기화한다. displayTotal, totalCorrect, q1answer, q2answer는 첫 번째 버전에서 이미 사용한 변수이다. 이번 버전에서는 numQuestions(answer()와 gradeUser() 함수에서 사용할 변수), correctAnswer1과 correctAnswer2(퀴즈 점수를 매길 때 사용할 변수)가 새로 추가되었다.

변수를 초기화하고 나면 사용자의 정답을 기록하고 플레이헤드를 다음 문제로 옮기기 위한 answer() 함수를 만든다. answer() 함수에서는 각 문제에 대해 사용자가 고른 답의 번호인 choice라는 하나의 인자를 받아들인다. 따라서 함수를 선언하는 부분은 다음과 같은 내용으로 시작한다.

```
function answer (choice) {
```

매번 사용자가 답을 할 때마다 이 함수에서는 문제 번호를 추적하는 currentAnswer 변수 값을 증가시킨다.

```
answer.currentAnswer++;
```

그리고 나서 사용자가 답한 문제에 해당하는 동적으로 이름이 정해진 타임라인 변수에 사용자가 선택한 답을 대입한다. 이 때 currentAnswer 속성 값을 이용하여 새로 만드는 타임라인 변수의 이름(q1answer, q2answer...)을 결정한다.

```
set ("q" + answer.currentAnswer + "answer", choice);
```

사용자가 선택한 답을 변수에 저장했을 때 마지막 문제를 처리했을 경우에는 퀴즈의 끝부분으로 이동한다. 마지막 문제가 아닌 경우에는 "q" + (answer.currentAnswer +1)이라는 레이블이 붙은 프레임으로 이동하여 다음 문제를 화면에 표시한다.

```
if (answer.currentAnswer == numQuestions) {
  gotoAndStop ("quizEnd");
} else {
  gotoAndStop ("q"+ (answer.currentAnswer + 1));
}
```

이렇게 하면 사용자가 문제의 답을 선택했을 때 필요한 작업을 처리할 수 있다. answer() 함수에서는 퀴즈에 있는 어떤 문제에 대한 답도 처리할 수 있다. 이제 점수를 매기기 위한 함수인 gradeUser()를 만들어 보자.

gradeUser() 함수에는 인자가 없다. 이 함수에서는 사용자가 선택한 답을 정답과 맞춰보고 사용자의 점수를 화면에 출력한다. 각 문제의 정답을 확인할 때는 for 루프를 이용하는데, 이 때 퀴즈에 있는 문제의 개수만큼 for 루프를 반복하게 된다.

```
for (i = 1; i <= numQuestions; i++) {
```

루프 안에서는 비교 표현식을 이용하여 사용자가 선택한 답과 정답을 비교한다. eval() 함수를 이용하여 사용자가 선택한 답과 정답을 비교할 수 있다. 두 변수가 같다면 totalCorrect 값을 증가시킨다.

```
if (eval("q" + i + "answer") == eval("correctAnswer" + i)) {
  totalCorrect++;
}
```

루프가 종료되면 totalCorrect에 사용자가 선택한 답 중 정답의 개수가 저장된다. 동적 텍스트 필드인 displayTotal에 totalCorrect를 대입하면 그 숫자를 화면에 표시할 수 있다.

```
displayTotal = totalCorrect;
```

자, 이처럼 함수를 만들고 퀴즈 변수를 초기화하는 것만으로 퀴즈 시스템에 있는 모든 논리적인 계산 부분을 처리할 수 있다. 대부분의 코드를 하나의 프레임으로 모아 두면 퀴즈가 진행되는 과정을 매우 간단하게 이해할 수 있다는 것을 느낄 것이다. 이제 퀴즈의 적당한 위치에서 필요한 함수를 호출하기만 하면 된다.

퀴즈 함수 호출

사용자가 답을 선택하기 위해 클릭하는 버튼에서 answer() 함수를 호출하고 quizEnd 프레임에서 gradeUser() 함수를 호출하도록 만들어 보자.

1번 프레임의 첫 번째 답변 버튼에 다음과 같은 코드를 추가한다.

```
on (release) {
  answer(1);
}
```

두 번째 버튼에는 다음과 같은 코드를 추가한다.

```
on (release) {
  answer(2);
}
```

그리고 세 번째 버튼에는 다음과 같은 코드를 추가한다.

```
on (release) {
  answer(3);
}
```

그리고 나서 q2 프레임에 있는 세 개의 버튼에도 똑같은 코드를 추가한다. 예전에 사용한 방법보다 훨씬 세련된 방법이라는 느낌이 들 것이다. 각 버튼을 처리하기 위한 코드를 깔끔하게 함수로 모아놓고 나면 버튼 코드를 정말 간단하게 만들 수 있다. 또한 사용자의 답변을 기록하는 부분과 퀴즈를 진행시키는 부분에서 이전 버전

에 있던 중복된 코드를 없앨 수 있다. 게다가 새로운 버튼을 추가하는 경우에도 answer() 함수에 전달하는 숫자만 바꿔주면 된다. answer() 함수에서 자동으로 처리해주기 때문에 마지막 문제라고 해서 특별히 신경을 쓸 필요도 없다.

버튼 코드를 모두 추가했다면 quizEnd 프레임에 다음과 같은 선언문을 추가하여 퀴즈가 끝날 때 사용자의 점수를 계산하도록 하자.

```
gradeUser();
```

이번에도 점수를 매기는 기능을 다른 함수들과 함께 무비의 첫 번째 프레임에 집어넣어서 시스템에서 메커니즘과 그 메커니즘을 활용하는 부분을 분리하였다. 이렇게 하면 프레임과 버튼 구성이 복잡한 경우에도 메커니즘을 놓치지 않을 수 있다(플래시 무비를 만들다 보면 종종 프레임과 버튼이 복잡해져서 헤매는 수가 있다).

퀴즈를 직접 테스트해보고 제대로 작동하는지 확인해 보자. 퀴즈 문제를 추가하거나 제거할 때 어떤 일이 생기는지 직접 실행해 보고 또 다른 아이디어가 떠오른다면 직접 코드를 고쳐보자. 앞으로 새로운 내용을 배울 때도 새로 배운 테크닉을 퀴즈 예제에 어떻게 적용할 수 있을지 생각해 보는 습관을 기르도록 하자. 화면에 출력되는 시각적인 부분을 어떻게 고칠 수 있을까? 사용자가 보기에는 똑같더라도 코드를 더 융통성 있게, 그리고 우아하게 만들려면 어떻게 해야 할까? 이러한 점을 항상 생각해 보자.

앞으로 배울 내용

이 장에서는 액션스크립트에서 가장 강력한 도구 중 하나인 함수에 대해 배웠다. 다음 장에서는 함수를 약간 변형시켜서 인터프리터에서 자동으로 실행시키는 '이벤트 핸들러(event handler)'라는 것에 대해 알아보기로 하자. 이벤트 핸들러는 상호작용에서 핵심적인 구성요소 중 하나이다.