2

변수

보통 스크립트를 사용한 무비에서는 프레임 번호부터 사용자의 암호, 우주선에서 발사한 포탄 어뢰의 속도에 이르기까지 모든 것을 직접 추적하고 조작해야 한다. 그런 정보를 모두 관리하기 위해 그 값들을 액션스크립트의 기본 정보 저장소인 '변수(variable)' 에 저장해야 한다.

변수는 돈 대신 정보(데이터)가 들어 있는 은행 통장이라고 생각할 수 있다. 새로운 변수를 만드는 것은 새로운 계좌를 만드는 것과 같다. 즉 나중에 필요한 것을 저장해 둘 장소를 만드는 것이다. 그리고 모든 은행 계좌에 계좌번호가 있는 것처럼 모든 변수에는 그 변수와 연관된 이름이 있으며, 그 이름을 이용하여 변수에 저장된 데이터를 액세스할 수 있다.

변수를 만들고 나면 우리가 원하는 대로 새로운 데이터를 집어넣을 수 있다. 은행 계좌에 돈을 입금하는 것과 비슷하다고 보면 된다. 또는 변수의 이름을 이용하여 변수에 어떤 값이 들어 있는지 알아낼 수 있다. 마치 은행 계좌의 잔고를 확인하는 것과 비슷하다. 변수가 더 이상 필요 없다면 변수를 지움으로써 은행 계좌를 없애듯이 변수를 없앨 수도 있다.

이 때 가장 주목할만한 기능은 변수를 이용하여 무비가 재생되는 동안 변하거나 계속 새로 계산되는 데이터를 참조하는 기능이다. 은행 계좌에 있는 잔액이 변해도 계좌번호는 그대로 남아 있는 것처럼 그 안에 있는 데이터가 바뀌어도 변수 이름은 바뀌지 않는다. 변화하는 내용물에 대해 고정된 이름을 사용하면 복잡한 계산을 할 수도 있고 카드 게임에서 카드를 추적할 수도 있으며, 방명록을 저장할 수도 있고 변화하는 조건에 따라 플레이헤드를 다른 위치로 옮길 수도 있다.

기대감으로 가득 차서 눈빛이 번뜩이는 독자들도 있을 것이다. 사실 은행에 관한 비유를 읽으면서 독자들이 책을 덮지나 않았을까 걱정된다. 이제 변수를 만드는 것부터 시작해서 변수에 대한 모든 것을 알아보기로 하자.

변수 만들기(선언)

변수를 새로 만드는 것을 '선언(declaration)'이라고 한다. 선언은 은행에 비유하자면 구좌 개설에 해당한다. 변수를 선언하는 순간부터 변수는 그 프로그램 내에 존재하게 된다. 변수를 처음 선언했을 때는 그 변수는 비어 있다. 이렇게 비어 있는 변수에 우리가 새로운 값을 집어넣게 된다. 이 상태에서 변수에는 undefined(데이터가 들어있지 않음을 의미함)라는 특별한 값이 저장되어 있다.

새로운 변수를 선언할 때는 var 선언문을 사용한다. 예를 들면 다음과 같다.

```
var speed;
var bookTitle;
var x;
```

var라는 단어는 인터프리터에 변수를 새로 선언하는 중이며 그 뒤에 오는 speed, bookTitle, x와 같은 텍스트가 새로운 변수의 이름이라는 것을 알려준다. 변수는 키프레임, 버튼, 무비 클립과 같이 코드를 만들 수 있는 곳이라면 어디서든지 선언할 수 있다.

다음과 같이 하나의 var 선언문에서 여러 개의 변수를 동시에 선언할 수도 있다.

```
var x, y, z;
```

하지만 이렇게 하면 각 변수 옆에 주석을 추가하기 힘들어진다.

변수를 만들고 나면 그 변수에 값을 대입할 수 있지만 변수 대입을 배우기 전에 변수 선언을 좀더 자세히 살펴보자.

자동 변수 생성

데이터를 변수에 저장하기 전에 반드시 변수를 먼저 선언해야 하는 언어가 많이 있다. 그러한 언어에서는 변수를 선언하지 않고 사용하면 오류가 발생한다. 하지만 액션스크립트는 그다지 까다롭지 않다. 존재하지 않는 변수에 어떤 값을 대입하면 인터프리터에서 자동으로 새로운 변수를 만들어준다. 은행에 계속 비유하자면 처음으로 돈을 입금할 때 자동으로 새로운 계좌를 만들어주는 것과 같다.

이러한 방법은 편리하긴 하지만 단점도 있다. 변수를 직접 선언하지 않으면 코드를 점검할 때 어떤 기준이 될만한 것이 없다. 게다가 var 선언문을 이용하여 직접 변수를 선언하면 자동으로 변수가 선언되는 것과는 다른 결과가 나올 수도 있다. 이 책에서는 먼저 변수를 선언한 뒤에 사용하는데, 이렇게 하는 것이 더 안전하다.

사용할 수 있는 변수 이름

변수를 만들기 전에 변수 이름은 다음과 같은 조건을 만족시켜야 한다는 점에 주의하자.

- 영문자, 숫자, 밑줄만을 이용한다(공백, 하이픈, 문장 부호 따위는 사용하면 안 된다).
- 문자 또는 밑줄로 시작해야 한다.
- 255자를 넘기면 안 된다(물론 255자를 넘기기는 쉽지 않지만 그렇게 될 것 같다면 변수 이름 붙이는 방법을 바꾸는 게 좋을 것이다).
- 대소문자를 구분한다(실제로 대소문자를 구분하는 것은 아니지만 프로그래밍할 때 대소문자를 일관되게 사용하는 것이 좋다).

다음과 같은 변수 이름은 올바른 변수 이름이다.

```
var first_name;
var counter;
var reallyLongVariableName;
```

하지만 다음과 같은 변수 이름은 옳지 않으며, 이렇게 하면 오류가 발생한다.

```
var 1first_name;                // 숫자로 시작
var variable name with spaces;  // 공백이 들어 있음
var another-illegal-name;       // 하이픈이 들어 있음
```

동적으로 이름이 주어지는 변수 만들기

거의 쓰이지 않지만 프로그래밍을 통해 동적으로 변수 이름을 정할 수도 있다. 표현식으로부터 변수 이름을 만들려면 set 선언문을 이용하면 된다. 예를 들어 아래 코드에서는 player1name이라는 변수에 bruce라는 값을 대입한다.

```
var i = 1;
set ("player" + i + "name", "bruce");
```

나중에 배울 배열이나 객체를 이용하면 동적으로 이름이 주어지는 데이터를 더 편하게 관리할 수 있으므로, 동적으로 이름이 주어지는 변수 이름 대신 배열이나 객체를 이용하는 것이 좋다.

처음에 변수 선언하기

무비 프리로더 뒤에 오는 첫째 키프레임인 무비의 주요 스크립트 시작 부분에서 변수를 선언하는 습관을 들이도록 하자. 나중에 알아보기 좋도록 각 변수의 용도를 주석으로 달아 놓는 것도 잊지 말자. 잘 짜여진 스크립트라면 보통 다음과 같은 식으로 시작된다.

```
// ^^^^^^^^^^^^^^^^^^^^
// 변수 선언
// ^^^^^^^^^^^^^^^^^^^^
var ballSpeed;   // 공의 속도, 최고 10
var score;       // 현재 점수
```

```
var hiScore;      // 최고 점수(세션간에 저장되지 않는다).
var player1;      // palyer1의 이름, 사용자가 입력함
```

변수를 만들면서 동시에 초기 값을 설정할 수도 있다.

```
var ballSpeed = 5;    // 공의 속도를 5로 설정
var score = 0;        // 현재 점수
var hiScore = 0;      // 최고 점수(세션간에 저장되지 않는다).
```

변수 값 대입

이제 조금 더 흥미로운 내용인 변수에 데이터를 대입하는 방법을 배워보자. 여기서도 은행에 비유하자면 계좌에 돈을 입금하는 것과 같다고 말할 수 있다. 변수에 값을 대입하려면 다음과 같이 하면 된다.

```
variableName = value;
```

여기서 variablename은 변수 이름이고 value는 변수에 저장되는 데이터이다. 이러한 구문을 사용하는 방법은 다음과 같다.

```
bookTitle = "ActionScript: The Definitive Guide";
```

등호의 왼쪽에 있는 bookTitle은 변수의 이름(인식자)이다. 등호 오른쪽에는 ActionScript: The Definitive Guide라는 구문이 있는데, 이것이 변수의 값(저장하고자 하는 데이터)이 된다. 등호는 대입 연산자라고 부른다. 이 연산자는 등호 오른쪽에 있는 것을 왼쪽에 있는 변수에 대입하겠다는 것을 인터프리터에 알려준다. 만약 왼쪽에 있는 변수가 아직 만들어지지 않은 변수라면 플래시에서 알아서 그 변수를 선언한다(하지만 이러한 방법은 그다지 바람직하지 못하다).

변수에 값을 대입하는 예를 두 가지 더 들어보자.

```
speed = 25;
output = "thank you";
```

첫째 예에서는 speed라는 변수에 25라는 정수를 대입한다. 변수에는 텍스트뿐만 아니라 숫자도 저장할 수 있다. 잠시 후에 또 다른 종류의 데이터도 대입할 수 있다는 것을 배울 것이다. 둘째 예에서는 thank you라는 텍스트를 output이라는 변수에 대입한다. 액션스크립트에서는 텍스트 문자열을 나타낼 때 큰따옴표(“”)를 사용한다.

이제 y에 1 + 5라는 표현식 값을 대입하는 조금 더 복잡한 예를 들어보자.

```
y = 1 + 5;
```

y = 1 + 5;라는 선언문이 실행되면 우선 1을 5에 더하여 6을 계산한 다음 y에 6을 대입한다. 오른쪽에 있는 표현식은 왼쪽에 있는 변수에 값을 대입하기 전에 계산된다. 아래 예에서는 y가 포함된 표현식을 다시 다른 변수인 z에 대입한다.

```
z = y + 4;
```

여기서도 등호 오른쪽에 있는 표현식을 먼저 계산하고 그 값을 z에 대입한다. 인터프리터에서는 y의 현재 값을(말하자면 계좌의 잔고를) 구해서 거기에 4를 더한다. y 값은 6이기 때문에 z에는 10을 대입한다.

데이터형에 상관없이 어떤 데이터 숫자, 텍스트 및 다른 데이터형을 사용해도 변수에 데이터를 대입하는 방법은 거의 똑같다. 예를 들면 아직 배열을 배우진 않았지만 다음과 같은 문장이 변수를 대입하는 선언문이라는 것은 쉽게 알 수 있을 것이다.

```
myList = ["John", "Joyce", "Sharon", "Rick", "Megan"];
```

앞에서와 마찬가지로 변수 이름이 왼쪽으로 가고 대입 연산자(등호)가 가운데 있고 대입하려는 값이 오른쪽에 있다.

같은 값을 여러 개의 변수에 한꺼번에 대입하고 싶다면 다음과 같이 연속으로 대입할 수도 있다.

```
x = y = z = 10;
```

변수 대입은 언제나 오른쪽에서 왼쪽으로 진행된다. 위 선언문에서는 10이 z에 대입되고 z 값이 y에 대입되고, y 값이 x에 대입된다.

변수 값 변경 및 검색

변수를 만든 후에는 〔예제 2-1〕에서 볼 수 있듯이 새로운 값을 계속해서 대입할 수 있다.

〔예제 2-1〕 변수 값 바꾸기

```
var firstName;                    // firstName을 선언한다.
firstName = "Graham";             // firstName 값을 설정한다.
firstName = "Gillian";            // firstName 값을 바꾼다.
firstName = "Jessica";            // firstName 값을 다시 바꾼다.
firstName = "James";              // firstName 값을 또 바꾼다.
var x = 10;                       // x를 선언하고 숫자를 대입한다.
x = "loading...please wait...";   // x에 텍스트를 대입한다.
```

x의 데이터형을 숫자에서 텍스트 데이터로 바꿀 때도 단지 원하는 형의 데이터를 대입하기만 하면 된다는 점을 기억해 두자. 프로그래밍 언어에 따라 변수의 데이터형을 바꿀 수 없는 것도 있지만, 액션스크립트에서는 마음대로 바꿀 수 있다.

변수를 만들고 그 변수에 값을 대입하는 것은 변수 값을 나중에 다시 알아낼 수 없다면 무의미한 일이다. 변수에 저장된 값을 구하려면 그냥 변수 값이 필요한 곳에서 변수 이름을 쓰기만 하면 된다. 변수 이름이 나타날 때마다(변수를 선언할 때와 대입 선언문의 왼쪽에 있는 경우는 제외) 그 이름이 변수 값으로 변환된다. 아래 예를 살펴보자.

```
newX = oldX + 5;  // newX에 oldX와 5를 더한 값을 대입
ball._x = newX;   // ball 무비 클립의 수평 위치를 newX 값으로 설정
trace(firstName); // firstName의 값을 Output 창에 출력
```

ball._x에서 ball은 무비 클립 이름이고 ._x는 그 클립의 x축 속성(스테이지에서의 수평 위치)을 의미한다. 속성에 관한 것은 나중에 자세히 배우게 될 것이다. 마지막 줄에 있는 trace(firstName)에서는 스크립트가 실행되는 도중에 변수 값을 화면에 표시해 주는데, 코드를 디버깅할 때 편리하게 쓰인다.

변수에 값이 있는지 확인하기

종종 변수에 있는 값을 알아보기 전에 변수에 어떤 값이 들어있는지 확인하고 싶을 때가 있다. 앞에서 배운 것처럼 선언은 했지만 아직 아무런 값도 대입하지 않은 변수에는 undefined라는 특별한 값이 들어 있다. 변수에 어떤 값을 대입한 적이 있는지 확인하려면 아래와 같이 그 변수 값을 undefined라는 키워드와 비교하면 된다.

```
if (someVariable != undefined) {
  // someVariable에 어떤 값이 들어있을 때에만 여기에 있는 코드를 실행한다.
}
```

부등 연산자(!=)는 두 값이 서로 다른지 확인할 때 쓰인다.

값 유형

액션스크립트 프로그래밍에서 쓰이는 데이터에는 다양한 형이 있다. 지금까지는 숫자와 텍스트만 사용했지만 데이터형에는 부울형, 배열, 함수, 객체 같이 다양한 종류가 있다. 각 데이터형을 자세히 살펴보기 전에 우선 변수 사용과 관련된 문제에 대해 짚고 넘어가기로 하자.

자동 유형

모든 액션스크립트 변수에는 임의의 데이터형을 가진 데이터를 저장할 수 있다. 어떻게 보면 별 것 아니라고 생각할 수도 있겠지만 모든 변수에 임의의 데이터형을 가진 데이터를 저장할 수 있는 언어는 생각보다 흔하지 않다. C++나 자바 같은 언어에서는 유형이 정해진 변수를 사용하기 때문에, 모든 변수에 변수를 선언할 때 지정한 한 가지 유형의 데이터만 저장할 수 있다. 액션스크립트 변수는 데이터를 변수에 대입할 때 인터프리터에서 변수의 데이터형을 직접 설정하는 방식으로 자동적으로 형이 결정된다.

액션스크립트에서는 변수에 임의의 데이터형을 저장할 수 있을 뿐만 아니라, 변수의 데이터형을 동적으로 바꿀 수도 있다. 변수에 원래 저장되어 있던 값과 다른 데이터형을 가진 새로운 값을 대입하면 변수의 형이 자동으로 바뀐다. 따라서 액션스크립트에서는 아래와 같은 스크립트를 사용해도 문제가 없다.

```
x = 1;                  // 숫자
x = "Michael";          // 문자열
x = [4, 6, "hello"];    // 배열
x = 2;                  // 다시 숫자
```

C++나 자바 같은 언어에서는 자동으로 새로운 데이터형으로 변경하는 것이 불가능하므로 데이터가 변수의 원래 데이터형으로 변환된다(변환이 불가능한 경우에는 오류가 발생한다). 액션스크립트의 변수는 자동 유형 및 동적 유형 기능 때문에 앞으로 나오는 절에서 설명할 몇 가지 중요한 특징을 지닌다.

값 자동 변환

상황에 따라 액션스크립트에서 특정한 형의 데이터가 필요한 경우가 있다. 만약 필요한 형과 맞지 않는 값을 가진 변수를 사용하면 인터프리터에서 그 데이터를 변환한다. 예를 들어 숫자가 필요한 위치에 텍스트 변수를 사용하면 인터프리터에서는 필요에 따라 그 변수의 텍스트 값을 숫자 값으로 변환한다. [예제 2-2]를 보면 z 값이 2로 설정된다. 이렇게 되는 이유는 뺄셈 연산에서는 숫자가 필요하기 때문에, y 값이 4라는 문자열에서 숫자 4로 변환되어 6(x 값)에서 4를 뺀 2가 z에 저장되기 때문이다.

[예제 2-2] 문자열에서 숫자로 자동 변환

```
x = 6;      // x는 숫자 6
y = "4";    // y는 문자열 "4"
z = x - y;  // 이렇게 하면 z 값이 2가 된다.
```

반대로 숫자 변수를 문자열이 들어갈 자리에 사용하면 인터프리터에서 숫자를 문자열로 변환한다. [예제 2-3]에서 z는 숫자 10이 아니라 문자열 "64"가 된다. x + y에 있는 두 번째 피연산자가 문자열이기 때문이다. 따라서 (+) 연산자는 덧셈이 아

닌 문자열을 합치는 연산을 처리하게 된다. x 값(6)은 문자열 6으로 변환되고 그 문자열이 4(y의 값)와 합쳐져 64라는 결과가 나온다.

〔예제 2-3〕 숫자에서 문자열로 자동 변환

```
x = 6;      // x 는 숫자 6
y = "4";    // y 는 문자열 "4"
z = x + y;  // 이렇게 하면 z 값이 64가 된다.
```

어떤 표현식에서 변수를 사용할 때는 자동 형 변환이 변수에 있는 데이터의 복사본에 대해 이루어지기 때문에, 원래 변수의 데이터형에는 영향을 주지 않는다. 변수 형은 변수에 원래 데이터형과 다른 유형의 데이터를 대입할 때만 바뀐다. 따라서 〔예제 2-2〕와 〔예제 2-3〕을 실행하더라도 y는 그대로 문자열로, x는 그대로 숫자로 남는다.

셋째 줄에 있는 연산자(〔예제 2-2〕에서는 -, 〔예제 2-3〕에서는 +) 때문에, z에 대입되는 값이 크게 달라진다는 점에 주의하자. 〔예제 2-2〕에서는 4라는 문자열이 숫자 4로 바뀌지만 〔예제 2-3〕에서는 반대로 숫자 6이 문자열 6으로 바뀐다. 데이터형 변환 규칙이 + 연산자와 - 연산자에 따라 서로 다르기 때문이다. 데이터 변환 규칙에 대해서는 '3장. 데이터와 데이터형'에서, 연산자에 대해서는 '5장. 연산자'에서 자세히 다룰 것이다.

수동으로 형 검사하기

자동 유형과 값 자동 변환이 편리하긴 하지만 〔예제 2-2〕나 〔예제 2-3〕에서 볼 수 있듯이 예상하지 못한 결과가 나올 수도 있다. 여러 가지 데이터형이 섞인 경우에는 명령을 수행하기 전에 typeof 연산자를 이용하여 변수의 데이터형을 알아보는 것이 좋을 수도 있다.

```
productName = "Macromedia Flash";  // 문자열 값
trace(typeof productName);         // "string"이 출력된다.
```

변수 형을 알고 나면 상황에 따라 적절한 조치를 취할 수 있다. 예를 들어 아래 코드에서는 어떤 변수가 숫자인지 미리 확인하고 있다.

```
if (typeof age == "number"){
  // 계속 해도 괜찮다.
```

```
} else {
  trace ("Age isn't a number");  // 오류 메시지 출력
}
```

typeof 연산자에 대한 자세한 내용은 5장을 참조하기 바란다.

변수 영역

지금까지 플래시 문서의 메인 타임라인에 있는 하나의 프레임에서 변수를 만들고 그 값을 검색하는 것에 대해 살펴보았다. 하나의 문서에 여러 개의 프레임과 여러 개의 무비 클립 타임라인이 들어 있으면 변수를 만들고 검색하는 것이 좀더 복잡해진다.

그 이유를 알아보기 위해 몇 가지 시나리오를 생각해 보자.

시나리오 1

x라는 변수를 메인 타임라인의 1번 프레임에서 만드는 경우를 가정해 보자. x를 만든 후에 10을 대입해 보자.

```
var x;
x = 10;
```

그리고 나서 2번 프레임에 아래 코드를 집어넣는다.

```
trace(x);
```

그리고 나서 무비를 실행시켜 보자. Output 창에 무엇이 나타나게 될까? 변수를 만든 위치는 1번 프레임이지만 그 값을 검색하는 위치는 2번 프레임이다. 이런 경우에도 변수가 그대로 있을까? 답은 '그렇다'이다.

타임라인에서 변수를 정의하면 그 변수는 같은 타임라인에 있는 모든 프레임에서 사용할 수 있다.

시나리오 2

시나리오 1과 마찬가지로 x를 만들고 그 값을 설정했다고 가정해 보자. 하지만 이번에는 변수를 설정하는 코드를 1번 프레임에 직접 넣지 않고 1번 프레임에 있는 버튼에 넣는다. 그리고 나서 2번 프레임에 다음 코드를 입력한다.

```
trace(x);
```

이렇게 해도 제대로 작동할까? 물론이다. x가 버튼에 붙어 있고 버튼은 메인 타임라인에 붙어 있기 때문에, 그 변수는 결국 메인 타임라인에 간접적으로 포함되는 셈이다. 따라서 앞에서와 마찬가지로 2번 프레임에서 변수를 사용할 수 있다.

시나리오 3

메인 타임라인의 1번 프레임에서 secretPassword라는 변수를 만드는 경우를 생각해 보자. 무비를 실행할 때 사용자가 무비의 특정 부분을 이용하려면 암호를 맞춰야 한다.

1번 프레임에서 secretPassword를 선언하는 것 외에 사용자가 입력한 암호를 실제 암호와 비교하는 함수도 만든다. 코드는 다음과 같다.

```
var secretPassword;
secretPassword = "yppah";

function checkPassword() {
  if (userPassword == secretPassword) {
    gotoAndStop("accessGranted");
  } else {
    gotoAndStop("accessDenied");
  }
}
```

30번 프레임에서 암호를 입력해야 한다고 가정하자. 사용자는 userPassword라는 입력 텍스트 필드 변수에 암호를 입력한다. 그러면 1번 프레임에 있는 checkPassword() 함수를 이용하여 그 값을 secretPassword 변수와 비교한다. 암호 검사 코드는 1번 프레임에서 정의하지만 userPassword는 30번 프레임에 다다르기 전까

지는 정의되지 않는다. 그렇다면 우리가 checkPassword() 함수를 호출할 때는 userPassword 변수가 존재할까?

이번에도 대답은 '그렇다'이다. userPassword를 checkPassword() 함수보다 나중에 정의하더라도 결국은 그 변수도 같은 타임라인에 속하기 때문이다.

한 타임라인에서 선언한 모든 변수는 그 타임라인이 없어지지만 않는다면 같은 타임라인에 있는 임의의 스크립트에서 사용할 수 있다.

변수 접근성(영역)

앞에서 다룬 세 개의 시나리오는 영역(scope)이라는 주제에 관한 것이다. 변수의 영역은 무비에서 언제, 그리고 어디서 그 변수를 조작할 수 있는지 정해준다. 변수의 영역은 그 변수의 유효 기간과 스크립트에 있는 다른 코드 블록에서의 접근성을 정의한 것이다. 변수 영역을 결정하려면 두 가지 정보가 필요하다. (1) 변수가 얼마나 오랫동안 남아 있을까? (2) 코드의 어느 부분부터 변수 값을 설정하거나 검색할 수 있을까?

전통적인 프로그래밍에서는 변수 영역의 범주를 크게 '전역(global) 변수'와 '지역(local) 변수'로 나눈다. 프로그램 전체에서 사용할 수 있는 변수는 전역 변수(global variable), 프로그램의 제한된 영역에서만 사용할 수 있는 변수는 지역 변수(local variable)라고 부른다. 플래시에서는 일반적인 지역 변수는 지원하지만, 진정한 의미에서의 전역 변수는 지원하지 않는다. 그 이유를 알아보자.

무비 클립 변수

앞에서 본 세 가지 시나리오에서 알 수 있듯이 한 타임라인에서 정의된 변수는 같은 타임라인에 있는 모든 스크립트에서 사용할 수 있다. 프레임 맨 앞이나 맨 뒤에 있는 코드라도 상관없으며, 변수가 프레임에서 선언되었는지 버튼에서 선언되었는지도 상관없다. 하지만 아래에 있는 시나리오 4와 같이 하나의 무비에 여러 개의 타임라인이 있는 경우에는 어떻게 될까?

시나리오 4

정사각형과 원의 두 가지 기본 도형을 각각 무비 클립 심벌로 정의했다고 가정하자.

직사각형 클립 심벌의 1번 프레임에서 x라는 변수 값을 3으로 설정한다.

```
var x;
x = 3;
```

원형 클립 심벌의 1번 프레임에서 y라는 변수 값을 4로 설정한다.

```
var y;
y = 4;
```

각 클립의 인스턴스를 무비 메인 타임라인의 1번 레이어, 1번 프레임에 갖다 놓고 각 인스턴스의 이름을 square와 circle이라고 정한다.

첫째 질문: 다음과 같은 코드를 메인 무비 타임라인의 1번 프레임(square와 circle이 있는 곳)에 덧붙인다면 Output 창에 어떤 내용이 출력될까?

```
trace(x);
trace(y);
```

답: Output 창에 아무것도 나타나지 않는다. x와 y라는 변수는 무비 클립에 있는 타임라인에서 정의된 것일 뿐, 메인 타임라인에서 정의된 것이 아니다.

square나 circle과 같이 무비 클립 타임라인에 들어 있는 변수는 해당 타임라인 영역 안에서만 유효하다. 메인 무비 타임라인과 같이 다른 타임라인에 있는 스크립트에서는 직접 사용할 수 없다.

둘째 질문: trace(x)와 trace(y)라는 선언문을 메인 무비 타임라인의 1번 프레임이 아닌 square 무비 클립의 1번 프레임에 입력하면 Output 창에 어떤 내용이 출력될까?

답: x 값인 3만 출력된다. x는 square의 타임라인에서 정의되었기 때문에 같은 타임라인에 있는 trace( ) 명령어에서 사용할 수 있다. 하지만 y 값은 circle에서 정의되어 있으므로 다른 타임라인에 속하기 때문에 사용할 수 없다.

이제 액션스크립트에서 전역 변수를 지원하지 않는 이유를 이해할 수 있을 것이다. 전역 변수는 프로그램 전체에서 사용할 수 있어야 하는데, 플래시에서는 개별 타임라인에 들어 있는 변수는 같은 타임라인에 있는 스크립트에서만 직접 사용할 수 있다. 플래시에 있는 모든 변수는 타임라인에서 정의되므로 무비에 있는 모든 스크립트에서 직접 사용할 수 있는 변수는 만들 수 없다. 따라서 어떤 변수도 전역 변수라고 부를 수 없다.

혼동을 피하기 위해 타임라인에 포함되는 변수를 '타임라인 변수(timeline variable)' 또는 '무비 클립 변수(movie clip variable)'라고 부르기로 하자. 하지만 Object 클래스를 이용하면 전역 변수 비슷한 것을 만들 수 있다. 모든 타임라인에서 사용할 수 있는 변수를 만들려면 다음과 같은 선언문을 이용하면 된다.

```
Object.prototype.myGlobalVariable = myValue;
```

예를 들면 다음과 같다.

```
Object.prototype.msg = "Hello world";
```

이러한 테크닉과 원리에 대한 내용은 '12장. 객체와 클래스'의 '상속 사슬의 끝' 절에서 자세히 다룬다.

다른 타임라인에 있는 변수 사용하기

어떤 타임라인에 들어 있는 변수를 다른 타임라인에 있는 스크립트에서 직접 사용할 수는 없지만, 간접적으로 사용하는 것은 가능하다. 다른 타임라인에 변수를 만들거나 사용하고 그 변수에 값을 대입할 때는 자바, C++, 자바스크립트 같은 객체 지향 프로그래밍 언어에서 흔히 쓰이는 표기법인 점 구문(dot syntax)을 사용하면 된다. 다른 타임라인에 있는 변수를 나타낼 때 사용하는 일반적인 점 구문은 다음과 같다.

```
movieClipInstanceName.variableName
```

즉 그 변수를 포함하고 있는 클립의 이름 뒤에 점을 붙이고 그 뒤에 변수 이름을 붙이면, 다른 타임라인에 있는 변수를 참조할 수 있다. 예를 들어 앞에서 살펴본 시

나리오의 경우를 생각해보면 메인 타임라인에서 square 클립에 있는 변수 x를 다음과 같이 사용할 수 있다.

```
square.x
```

또한 circle 클립에 있는 변수 y는 다음과 같이 참조할 수 있다.

```
circle.y
```

메인 무비 타임라인에서 위와 같은 레퍼런스를 사용하면 다음과 같은 방법으로 square에 있는 변수에 새로운 값을 대입하거나 저장되어 있는 값을 알아낼 수 있다.

```
square.z = 5;       // sqaure에 있는 z에 5를 대입한다.
var mainZ;          // 메인 타임라인에 mainZ라는 변수를 만든다.
mainZ = square.z;   // mainZ에 square에 있는 z 값을 대입한다.
```

하지만 (클립 이름).(변수 이름)을 이용한 것만으로는 circle 클립에서 square 클립에 있는 변수를 이용할 수 없다. circle에 있는 프레임에서 square.x를 사용하면 인터프리터에서는 circle 클립 안에 있는 square라는 클립을 찾으려고 하는데 square는 메인 타임라인에 있는 클립이기 때문이다. 따라서 circle 클립에서 square 클립을 포함하고 있는 타임라인(이 경우에는 메인 타임라인)을 가리킬 수 있는 메커니즘이 필요하다. 이러한 메커니즘은 두 개의 특별한 속성인 _root와 _parent를 이용하여 구현할 수 있다.

_root와 _parent 속성

_root 속성은 무비의 메인 타임라인을 직접 참조하기 위한 레퍼런스이다. 무비 클립 구조에서 어느 위치에 있더라도 _root를 이용하면 메인 무비 타임라인에 있는 변수를 참조할 수 있다.

```
_root.mainZ     // 메인 타임라인에 있는 mainZ라는 변수를 이용한다.
_root.firstName // 메인 타임라인에 있는 firstName이라는 변수를 이용한다.
```

_root를 이용하여 무비의 다단계 구조를 거쳐 임의의 무비 클립 인스턴스를 참조하는 것도 가능하다. 예를 들어 메인 무비 타임라인에 있는 square 클립 안에 있는 변수 x는 다음과 같은 식으로 참조할 수 있다.

```
_root.square.x
```

이렇게 하면 무비 내의 어떤 클립에서든지 원하는 변수를 참조할 수 있다. 레퍼런스가 메인 무비 타임라인인 _root에서부터 시작하기 때문이다. shapes라는 인스턴스의 타임라인에 들어 있는 triangle이라는 인스턴스의 area라는 변수는 다음과 같이 참조할 수 있다.

```
_root.shapes.triangle.area
```

_root 키워드로 시작하는 변수에 대한 레퍼런스는 '절대 레퍼런스(absolute reference)'라고 부른다. 무비에서 고정되어 있고 바뀌지 않는 지점인 메인 타임라인에서 시작하여 변수의 위치를 기술하는 방식이기 때문이다.

하지만 무비의 메인 타임라인을 참조하지 않고 다른 타임라인에 있는 변수를 참조하고 싶을 때도 있다. 이럴 때는 현재 무비 클립 인스턴스를 포함하고 있는 타임라인을 가리키는 _parent 속성을 이용하면 된다. 예를 들어 square 클립에 붙어 있는 코드에서 square를 포함하고 있는 타임라인에 있는 변수를 참조할 때는 다음과 같이 쓰면 된다.

```
_parent.myVariable
```

_parent라는 키워드로 시작되는 레퍼런스는 '상대 레퍼런스(relative reference)'라고 부른다. 현재 클립의 위치를 기준으로 상대적인 위치를 가리키는 방식이기 때문이다.

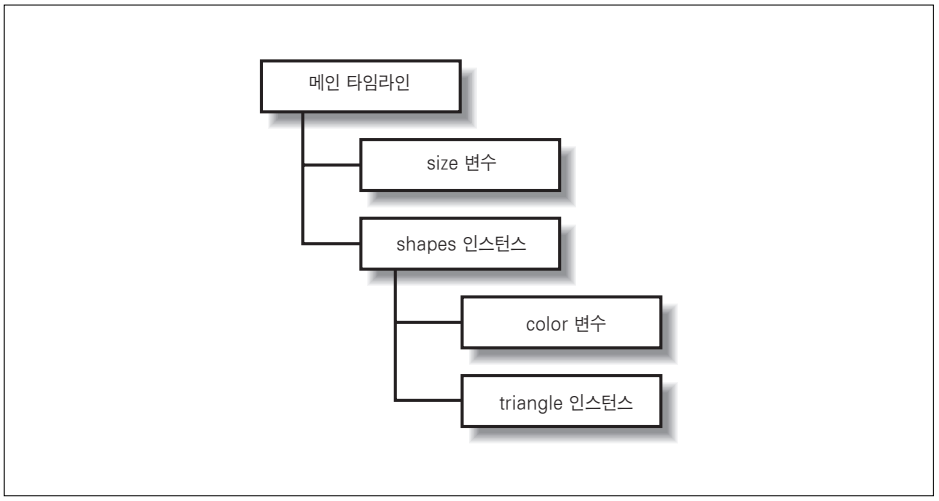
앞에서 사용한 예제를 다시 생각해 보자. 무비의 메인 타임라인에서 정의한 size라는 변수가 있다고 가정하자. 메인 무비 타임라인에 shapes라는 이름의 클립 인스턴스를 만들고 shapes 타임라인에 color라는 변수를 정의한다. 또 〔그림 2-1〕에 나온 것과 같이 shapes 타임라인에 triangle이라는 클립을 만든다.

triangle 타임라인에 있는 코드에서(shapes 클립에 있는) color 변수 값을 출력할 때는 다음과 같이 메인 타임라인에서 시작하는 절대 레퍼런스를 사용해도 된다.

```
trace(_root.shapes.color);
```

하지만 이렇게 하면 코드가 메인 무비 타임라인에 얽매이게 된다. 코드를 더 유연성 있게 만들고 싶다면 _parent 속성을 이용하여 상대 레퍼런스를 만들면 된다.

```
trace(_ parent.color);
```



〔그림 2-1〕 무비 클립 계층구조의 예

_root를 이용하는 첫째 방법에서는 위에서 아래로 내려가는 방식을 사용하므로, 메인 타임라인에서 시작하여 무비 클립 계층구조를 따라 내려가서 color 변수를 찾는다. _parent를 사용하는 둘째 방법에서는 밑에서 위로 올라가는 방식을 사용하므로, trace( ) 선언문이 들어있는 클립(triangle 클립)에서 시작하여 클립 구조를 한 단계 올라가서 color 변수를 찾는다.

클립 계층에서 두 단계를 올라가서 메인 타임라인에 있는 size 변수를 이용하기 위해 _parent를 두 번 연속으로 사용해도 된다. triangle에서 다음과 같은 코드를 사용하면 메인 타임라인의 size를 참조하게 된다.

```
trace(_parent._parent.size);
```

_parent 속성을 두 번 사용하면 두 단계를 올라가므로 이 경우에는 무비의 메인 타임라인으로 간다.

변수에 접근하는 방법은 무비 클립 심벌의 인스턴스를 다른 타임라인으로 옮겼을 때 어떤 식으로 동작하기를 원하는지에 따라 달라질 수 있다. triangle 예제에서 만약 color에 대한 레퍼런스가 언제나 shapes 클립에서 정의된 color 변수를 가리키기를 원한다면, _root로 시작하는 구문을 사용하여 shapes에 있는 color에 대한 고정된 레퍼런스를 이용하는 것이 낫다. 하지만 triangle이 들어 있는 타임라인에 따라

상대적인 위치에 있는 color 변수를 참조하고 싶다면, _parent로 시작하는 구문을 이용하는 것이 낫다.

서로 다른 문서 단계의 변수 사용하기

_root 속성은 현재 단계(즉 현재 사용중인 문서)의 메인 무비 타임라인을 가리키지만, 플래시 플레이어에서는 문서 스택(document stack)에 여러 개의 문서를 한꺼번에 담아둘 수 있다. 플레이어 문서 스택에 로딩된 임의의 무비 메인 타임라인은 _leveln(n은 무비가 위치하고 있는 레벨 번호)을 이용하여 참조할 수 있다. 레벨 번호는 0에서 시작하므로 _level0, _level1, _level2, _level3 같은 식으로 쓸 수 있다. 여러 개의 무비를 로딩하는 것과 관련된 내용은 '13장. 무비 클립'에 나와 있다. 다른 레벨에 있는 변수는 다음과 같은 식으로 참조하면 된다.

```
_level1.firstName       // level1의 메인 타임라인에 있는 firstName
_level4.ball.area       // level4의 메인 타임라인에 있는 ball 클립의 area
_level0.guestBook.email // level10의 타임라인에 있는 guestBook 클립의 email
```

점 구문을 이용하여 여러 무비 클립 인스턴스 타임라인에 걸쳐 변수를 이용할 때는 스테이지에서 클립 인스턴스에 이름을 제대로 붙였는지, 코드에서 변수 이름을 정확하게 입력했는지 확인하도록 하자. 인스턴스에 이름이 없으면 코드 자체가 문법적으로 완벽하다고 하더라도 제대로 동작하지 않게 된다. 인스턴스의 이름이 없거나 이름을 잘못 적는 일로 인해 자주 문제가 발생한다.

플래시 4와 플래시 5의 변수 사용 문법

/square:area와 같이 플래시 4 스타일의 슬래시와 콜론을 이용하는 구조는 플래시 5에서는 점 구문으로 바뀌었다. 이 방법이 변수와 타임라인을 참조하기 훨씬 좋기 때문이다. 예전에 쓰던 문법은 이제는 더 이상 쓰이지 않기 때문에 새로 만드는 프로그램에서는 쓰지 않는 것이 좋다. 〔표 2-1〕에 플래시 4와 플래시 5에서 변수를 참조하는 방법을 비교해 놓았다. 문법적인 차이점에 대한 자세한 내용은 '부록 C. 하위 호환성'을 참조하기 바란다.

〔표 2-1〕 플래시 4와 플래시 5의 변수 사용 문법

플래시 4 문법	플래시 5 문법	참조하는 내용
/	_root	무비의 메인 타임라인
/:x	_root.x	무비의 메인 타임라인에 있는 변수 x
/clip1:x	_root.clip1.x	무비의 메인 타임라인에 있는 clip1의 변수 x
/clip1/clip2:x	_root.clip1.clip2.x	메인 무비 타임라인의 인스턴스 clip1에 포함된 인스턴스 clip2에 있는 변수 x
../	_parent	현재 클립 바로 위에 있는 타임라인(현재 클립 타임라인*의 한 단계 위)
../:x	_parent.x	현재 클립 바로 위에 있는 타임라인에 있는 변수 x
../../:x	_parent._parent.x	현재 클립을 포함하는 타임라인을 포함하는 타임라인(현재 클립 타임라인에서 두 단계 위)에 있는 변수 x
clip1:x	clip1.x	현재 타임라인에 들어 있는 인스턴스인 clip1에 들어 있는 변수 x
clip1/clip2:x	clip1.clip2.x	현재 타임라인에 포함된 clip1에 포함된 clip2에 들어 있는 변수 x
_level1:x	_level1.x	레벨 1에 로딩된 무비의 메인 타임라인에 있는 변수 x
_level2:x	_level2.x	레벨 2에 로딩된 무비의 메인 타임라인에 있는 변수 x

* 현재 클립 타임라인은 변수 레퍼런스를 포함하고 있는 코드가 포함된 타임라인을 뜻한다.

무비 클립 변수의 생존 기간

앞에서 변수 영역에 의해 두 가지 질문에 답할 수 있다는 것을 배웠다. (1) 변수가 얼마나 오랫동안 남아 있을까? (2) 코드의 어느 부분부터 변수 값을 설정하거나 검색할 수 있을까? 무비 클립 변수의 경우 둘째 질문에 대한 답은 이제 알고 있으리라 생각한다. 하지만 첫째 질문의 답은 아직 잘 모른다. 마지막 변수 코딩 시나리오를 통해 그 문제를 해결해 보자.

시나리오 5

두 개의 키프레임이 있는 새로운 무비를 만든다고 가정하자. 1번 프레임에 ball이라는 클립 인스턴스를 추가한다. ball 타임라인에서 radius라는 변수를 만든다. 메인 타임라인의 2번 프레임은 비어 있는 채로 둔다(여기에는 ball 인스턴스가 없다).

메인 무비 타임라인의 1번 프레임에서는 다음과 같은 코드를 이용하여 radius 값을 알아낼 수 있다.

```
trace(ball.radius);
```

질문: 만약 위와 같은 코드를 메인 타임라인의 1번 프레임에서 2번 프레임으로 옮겨 놓는다면 무비를 재생할 때 Output 창에 어떤 내용이 출력될까?

답: 아무것도 보이지 않는다. ball 클립을 2번 프레임의 메인 타임라인에서 없애면 그 과정에서 모든 변수가 제거된다.

무비 클립 변수는 그 변수가 들어 있는 클립이 스테이지에 있는 동안만 존재할 수 있다. 플래시 문서의 메인 타임라인에서 정의한 변수는 각 문서에서 계속해서 사용할 수 있지만, 플레이어에서 문서를 닫고 나면 모두 사라진다(unloadMovie( ) 함수를 이용하거나 같은 레벨에 다른 무비를 로딩하는 경우에 상관없이 같은 결과가 나타난다).

다양한 타임라인에서 여러 개의 프레임에 걸쳐 있는 무비 클립을 포함하는 무비 스크립트를 만들 때는 변수의 생존 기간이 매우 중요하다. 어떤 클립에 있는 변수를 사용하기 전에 사용하려는 클립이 타임라인에 존재하는지 반드시 확인하도록 하자.

지역 변수

무비 클립 변수는 무비 클립 영역 안에서 사용할 수 있으며, 그 변수가 속해 있는 무비 클립을 사용중이라면 언제든지 사용할 수 있다. 경우에 따라 필요 이상으로 변수가 오래 남아 있는 경우도 있다. 임시로 쓸 변수가 필요한 경우를 위해 액션스크립트에서는 지역 변수를 제공하는데, 지역 변수는 일반적인 무비 클립 변수에 비해 짧은 기간 동안만 남아 있는다.

지역 변수는 함수 또는 예전에 쓰이던 플래시 4 스타일의 서브루틴에서 쓰인다. 함수나 서브루틴을 사용해 본 적이 없다면, 이 절의 나머지 부분은 일단 그냥 넘어가고 '9장. 함수'를 마친 후에 다시 읽어보는 것이 낫다.

함수에서는 종종 함수 밖에서는 필요 없는 변수를 사용하는 경우가 있다. 예를 들어 주어진 배열의 모든 원소를 화면에 표시하는 함수를 생각해 보자.

```
function displayElements(theArray) {
  var counter = 0;
  while(counter < theArray.length) {
    trace("Element " + counter + ": " + theArray[counter]);
    counter++;
  }
}
```

counter라는 변수는 배열을 표시하기 위해 필요하지만 그 작업이 끝나고 나면 쓸모가 없어진다. 이 변수를 타임라인에 그냥 남겨 둬도 되지만 다음과 같은 두 가지 이유에서 그렇게 하지 않는 것이 좋다. 첫째, counter 변수가 계속 남아 있으면 무비가 끝날 때까지 계속 메모리를 차지하게 된다. 둘째, counter를 함수 밖에서도 사용할 수 있다면 같은 이름을 가진 다른 변수와 충돌할 수도 있다. 따라서 dispalyElements() 함수가 끝나고 나면 counter 변수가 없어지도록 하는 것이 좋다.

함수가 끝날 때 counter 변수도 자동으로 없어지게 하려면 그 변수를 지역 변수로 정의하면 된다. 무비 클립 변수와 달리 지역 변수는 그 변수를 정의한 함수가 완료되면 인터프리터에 의해 자동으로 제거된다.

어떤 변수를 지역 변수로 지정하려면 앞에 나온 displayElements() 예제에서 한 것처럼 함수 내에서 var 키워드를 이용하여 선언하면 된다.

이 때 주의할 점이 있다. 만약 var 선언문을 함수 밖에서 사용한다면 지역 변수가 아닌 일반적인 타임라인 변수가 만들어진다. [예제 2-4]에서 볼 수 있듯이 var 선언문의 위치에 따라 결과가 크게 달라진다.

함수 안에 있는 변수가 반드시 지역 변수일 필요는 없다. var 키워드를 생략하면 함수 안에서도 무비 클립 변수를 만들거나 그 값을 변경할 수 있다. 변수를 만들 때 var 키워드를 사용하지 않고 그냥 변수에 어떤 값을 대입하면 플래시에서는 상황에 따라 그 변수를 지역 변수가 아닌 것으로 간주한다. 아래와 같이 함수 안에 있는 변수 대입 선언문을 생각해 보자.

```
function setHeight(){
  height = 10;
}
```

height = 10;이라는 선언문에 의한 결과는 height가 지역 변수인지, 아니면 무비 클립 변수인지에 따라 달라진다. 만약 height가 전에 선언된 지역 변수라면(물론 위에 있는 예제의 경우에는 그렇지 않다), height = 10;이라는 선언문에서는 단순히 지역 변수의 값을 변경하게 된다. 만약 위 예제처럼 height라는 이름을 가진 지역 변수가 없다면 인터프리터에서는 height라는 이름의 무비 클립 변수(지역 변수가 아닌 변수)를 만들고 그 값을 10으로 설정한다. 지역 변수가 아닌 경우에는 height라는 변수는 함수가 끝나고 난 뒤에도 그대로 남아 있게 된다.

〔예제 2-4〕에 지역 변수와 지역 변수가 아닌 변수의 사용법을 정리해 보았다.

〔예제 2-4〕 지역 변수와 지역 변수가 아닌 변수

```
var x = 5;                          // 새로운 지역 변수가 아닌 변수. x의 값은 5
                                       이다.
function variableDemo(){
   x = 10;                          // 지역 변수가 아닌 변수 x. 값이 10이 된다.
   y = 20;                          // 새로운 지역 변수가 아닌 변수 y. 값은 20
                                       이다.
   var z = 30;                      // 새로운 지역 변수 z. 값은 30이다.
   trace(x + "," + y + "," + z); // 모든 값들을 Output 창에 출력한다.
}
variableDemo();// 위의 함수를 호출하면 10,20,30 이 출력된다.
trace(x);      // 10이 출력됨(함수 내에서 새로운 값을 대입하면 그 값이 유지된다)
trace(y);      // 20이 출력됨(y는 지역 변수가 아니므로 그대로 남아 있다)
trace(z);      // 아무 것도 출력되지 않는다(지역 변수인 z는 이미 사라졌다).
```

조금 혼동되기도 하고 그다지 좋지 않은 방법이긴 하지만, 같은 스크립트에서 이름은 같지만 영역이 다른 지역 변수와 지역 변수가 아닌 변수를 동시에 사용하는 것도 가능하다. 〔예제 2-5〕에서 그러한 경우를 볼 수 있다.

```
var myColor = "blue";
function hexRed(){
   var myColor = "#FF0000";
   return myColor;
}
trace(hexRed()); // #FF0000이 출력된다(지역 변수 myColor).
trace(myColor);  // "blue"가 출력된다(지역 변수인 myColor를 #FF0000으로
                 // 설정하더라도 지역 변수가 아닌 변수는 영향을 받지 않는다).
```

서브루틴에서의 지역 변수

이식성이 좋은 코드 모듈을 만들 때는 함수를 사용하는 것이 좋지만, 플래시 5에서도 플래시 4 스타일의 서브루틴(subroutine)을 지원하기는 한다. 플래시 4에서는 레이블이 있는 프레임에 코드 블록을 덧붙여서 서브루틴을 만들 수 있다. 그렇게 만들어 두면 Call 액션을 통해 서브루틴을 외부에서 실행시킬 수 있다. 하지만 플래시 4에서는 서브루틴에서 선언한 모든 변수가 지역 변수가 아니기 때문에, 그 변수를 정의한 타임라인이 남아 있는 동안 그 변수들이 모두 그대로 남아 있게 된다. 플래시 5에서는 함수에서 사용한 것과 같은 방법으로 var 선언문을 이용하여 서브루틴에서도 지역 변수를 만들 수 있다. 하지만 서브루틴에서 var를 이용하여 정의한 변수는 서브루틴을 Call 함수를 이용하여 실행한 경우에만 지역 변수가 된다. 만약 서브루틴 프레임의 스크립트가 플레이헤드가 그 프레임에 들어갈 때 자동으로 실행되는 경우에는 var 선언문을 이용하더라도 일반적인 타임라인 변수가 된다. 하지만 이런 기능이 지원된다고 하더라도 서브루틴보다는 더 최근에 나온 방법인 함수 및 함수에 포함된 지역 변수를 사용하는 것이 좋다.

몇 가지 응용 예제

지금까지 변수에 대한 이론적인 내용을 꽤 많이 배웠다. 이러한 개념을 실전에 응용해 보자. 아래에 세 개의 변수와 관련된 예제를 수록하였다. 코드에 대한 자세한 설명은 코드 주석을 참조하기 바란다.

〔예제 2-6〕에서는 무비의 플레이헤드를 임의의 위치로 옮긴다.

〔예제 2-6〕 무비의 플레이헤드를 현재 타임라인에 있는 임의의 위치로 옮긴다.

```
var randomFrame;            // 임의로 만들어낸 프레임 번호를 저장하는 변수
var numFrames;              // 타임라인의 전체 프레임 번호를 저장하는 변수
numFrames = _totalframes;   // numFrames에 _totalframes 속성을 저장한다.

// 임의의 프레임을 선택한다.
randomFrame = Math.floor(Math.random() * numFrames + 1);

gotoAndStop(randomFrame);   // 플레이헤드를 선택한 프레임으로 보낸다.
```

〔예제 2-7〕에서는 두 클립 사이의 거리를 알아낸다. 이 예제를 실제로 응용한 프로그램은 온라인 코드 창고에서 구할 수 있다.

〔예제 2-7〕 두 무비 클립 사이의 거리를 계산한다.

```
var c;          // circle 클립 객체에 대한 레퍼런스
var s;          // square 클립 객체에 대한 레퍼런스
var deltaX;     // c와 s의 수평방향 거리
var deltaY;     // c와 s의 수직방향 거리
var dist;       // c와 s 사이의 전체 거리

c = _root.circle;          // circle 클립에 대한 레퍼런스를 구한다.
s = _root.square;          // square 클립에 대한 레퍼런스를 구한다.
deltaX = c._x - s._x;      // 클립 사이의 수평 거리를 구한다.
deltaY = c._ y - s._ y;    // 클립 사이의 수직 거리를 구한다.

// 거리 = (deltaX의 제곱 더하기 deltaY의 제곱).
dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));

// 깔끔하게 레퍼런스를 이용하면 코드를 읽기가 훨씬 편하다.
dist = Math.sqrt(((_root.circle._x - _root.square._x) * (_root.circle._x -
_root.square._x))  +  ((_root.circle._ y  -  _root.square._ y)  *
(_root.circle._ y -
_root.square._ y)));
```

〔예제 2-8〕에서는 화씨 온도와 섭씨 온도를 변환한다. 온라인 코드 창고에서 실행해 볼 수 있는 버전을 구할 수 있다.

〔예제 2-8〕 화씨/섭씨 온도 변환

```
var fahrenheit;          // 화씨 온도
var celsius;             // 섭씨 온도
var convertDirection;    // 변환하고자 하는 온도 표기법
                         // "fahrenheit(화씨)", "celsius(섭씨)" 중 한 값을
                         // 가질 수 있다.
fahrenheit = 451;                // 화씨 온도를 설정한다.
celsius = 20;                    // 섭씨 온도를 설정한다.
convertDirection = "celsius";    // 이 경우에는 섭씨로 변환

if (convertDirection == "fahrenheit") {
   result = (celsius * 1.8) + 32;   // 섭씨 온도 계산
```

```
        // 결과 출력
        trace (celsius + "degrees Celsius is" + result + "degrees Fahrenheit.");

    } else if (convertDirection == "celsius") {
        result = (fahrenheit - 32) / 1.8;   // 화씨 온도 계산
        // 결과 출력
        trace (fahrenheit+"degrees Fahrenheit is"+result+"degrees Celsius.");

    } else {
        trace ("Invalid conversion direction.");
    }
```

앞으로 배울 내용

이제 변수에 정보를 저장하는 것은 모두 배웠으므로 변수에 저장할 내용물인 데이터에 대해 배워 보자. 앞으로 나올 세 장에서는 데이터가 무엇인지, 데이터를 어떻게 조작하는지, 그리고 그런 작업이 왜 액션스크립트에서 필수적인 부분이 되는지 배울 것이다.