8

순환문

앞 장에서는 조건문을 이용하면 기준 표현식 값이 true인 경우에만 선언문 블록을 실행시킬 수 있다는 것을 배웠다. 반면에 '순환문(loop statement)'에서는 기준 표현식이 true이면 계속해서 순환문 블록을 반복 실행한다.

순환문에는 while, do-while, for, for-in과 같이 다양한 종류가 있다. 앞에 있는 세 종류의 순환문은 문법만 약간 다를 뿐 거의 똑같은 기능을 지닌다. 마지막에 나와 있는 for-in은 객체와 함께 사용하는 특화된 순환문이다. 우선 가장 이해하기 쉬운 while 선언문부터 시작해서 순환문에 대해 알아보기로 하자.

while 루프

구조적으로 볼 때 while 선언문은 if 선언문과 매우 흡사하다. 주 선언문에 주어진 조건이 true인 동안만 실행되는 하위 순환문이 포함된다.

```
while (condition) {
  substatements
}
```

조건이 true이면 substatements가 실행된다. 하지만 if 선언문과는 달리 substatement를 마치고 나면 while 선언문의 시작 부분으로 다시 돌아간다(즉 인터프리터에서 while 선언문의 시작점부터 다시 루프를 돌리게 된다). while 선언문을 두 번째 거칠 때도 첫 번째와 마찬가지로 동작한다. 조건을 확인해 보고 그 조건이 여전히 true이면, substatements를 다시 실행시킨다. 이 과정은 condition이 false가 될 때까지 계속 반복되며, while 선언문이 끝나면 그 뒤에 있는 스크립트가 계속해서 실행된다.

매우 간단한 루프[1]의 예를 들어보자면 다음과 같다.

```
var i = 3;
while (i < 5) {
  trace("x is less than 5");
}
```

이 예제도 문법면에서 본다면 while 선언문을 제대로 사용하긴 했지만 사실은 거의 오류에 해당한다. 그 이유를 알아보기 위해 인터프리터에서 위 예제를 실행하는 과정을 따라가 보자.

우선 while 선언문 앞에 있는 var i = 3에서는 변수 i를 3으로 설정한다. 변수 i는 순환문에서 기준 표현식으로 쓰이므로 이 과정을 순환문 초기화라고도 부른다. 그 다음에는 기준 표현식인 i < 5의 값을 계산하여 while 선언문을 시작한다. i 값은 3이므로 5보다 작고 따라서 기준 표현식은 true가 되기 때문에 순환문에 있는 trace( ) 선언문이 실행된다.

그리고 나서 다시 루프를 돌게 된다. 이번에도 기준 표현식 값을 계산하는데 i 값이 바뀌지 않기 때문에, 기준 표현식은 이번에도 true가 되고 trace( ) 선언문을 다시 실행하게 된다. 그러면 또 루프를 한 번 끝마쳤으므로 다시 루프를 시작하게 된다. 그런데 이번에도 여전히 i 값은 그대로이므로, 기준 표현식은 여전히 true이고 따라서 trace( ) 선언문을 실행시키며 이 과정은 영원히 계속된다. 기준 표현식이 언제나 true이므로 루프를 빠져나갈 방법이 없다. 결국 while 선언문 뒤에 있는 선언

1) 역자주: 원래 루프는 '고리'를 뜻하는데 순환문에서는 반복하여 같은 선언문을 실행하므로 하위 선언문을 한 번 실행할 때마다 고리를 한 바퀴 돌리는 것 같다는 의미로 매번 반복되는 단위를 '루프'라고 부른다. 또한 순환문을 반복하는 것을 '루프를 돌린다'라는 식으로 표현한다. 이 책에서는 순환문과 루프라는 용어를 거의 같은 뜻으로 사용하였다.

문은 실행할 수 없고, 무한 루프에 갇혀버리게 된다. 잠시 후에 배우겠지만, 액션스크립트에서는 무한 루프가 있으면 오류가 발생한다.

위 순환문이 무한 루프가 되는 이유는 기준 표현식에서 쓰이는 변수 값을 바꿔주는 업데이트 선언문이 없기 때문이다. 보통 업데이트 선언문을 실행시키다 보면 결국 기준 표현식이 false가 되어 루프를 끝마치게 된다. 업데이트 선언문을 추가하여 무한 루프가 생기지 않도록 고쳐보자.

```
var i = 3;
while (i < 5) {
  trace("x is less than 5");
  i++;
}
```

여기서는 i++라는 업데이트 선언문을 루프 본체의 맨 뒤에 추가했다. 인터프리터에서 루프를 돌릴 때 trace() 선언문을 실행하고 나서 i++를 실행하여 i 값을 1씩 증가시킨다. 매번 루프를 돌 때마다 i 값이 증가하므로, 루프를 두 번 돌고 나면 i 값이 5가 되어 기준 표현식인 i < 5가 false가 된다. 따라서 루프가 완료된다.

위에서 사용한 루프의 업데이트 선언문에서는 매우 기본적인 작업인 숫자를 세는 역할을 한다. 변수 i(이러한 변수를 카운터라고 부른다)는 예측할 수 있는 수열을 따라 움직인다. 이러한 카운터는 무비 클립을 복사하거나 배열 원소를 액세스하는 작업에 안성맞춤이다. 아래 예에서는 루프를 사용하지 않고 square 무비 클립을 다섯 번 복사한다.

```
// 새로운 무비 클립에 순서대로 이름을 붙이고 각기 다른 레벨에 놓는다.
duplicateMovieClip("square", "square1", 1);
duplicateMovieClip("square", "square2", 2);
duplicateMovieClip("square", "square3", 3);
duplicateMovieClip("square", "square4", 4);
duplicateMovieClip("square", "square5", 5);
```

다음 예에서는 순환문을 이용하여 간단하게 처리한다.

```
var i = 1;
while (i <= 5) {
  duplicateMovieClip("square", "square" + i, i);
  i++;
}
```

square 무비 클립을 100번 정도 복사한다고 할 때 어떤 결과가 나올지는 쉽게 상상할 수 있을 것이다.

루프는 데이터를 다룰 때, 특히 배열에 저장된 데이터를 다룰 때 진가를 발휘한다. [예제 8-1]에는 배열에 있는 모든 원소를 Output 창에 출력하는 예제가 나와 있다. 배열의 첫째 원소는 1번이 아니라 0번부터 시작한다는 점에 주의하자.

[예제 8-1] while 루프를 이용하여 배열을 출력하는 방법

```
var people = ["John", "Joyce", "Margaret", "Michael"];  // 배열 생성
var i = 0;
while (i < people.length) {
  trace("people element " + i + " is " + people[i]);
  i++;
}
```

Output 창에는 다음과 같은 결과가 출력된다.

```
people element 0 is John
people element 1 is Joyce
people element 2 is Margaret
people element 3 is Michael
```

위 예제에서 변수 i는 기준 표현식에서도 쓰이고 배열의 인덱스 번호로도 쓰였는데, 이러한 방법은 매우 흔하게 사용된다. 아래 예에서는 변수 i를 charAt() 함수의 인자로 사용한다.

```
var city = "Toronto";
trace("The letters in the variable 'city' are ");
var i = 0;
while (i < city.length) {
  trace(city.charAt(i));
  i++;
}
```

이 예제를 실행하면 Output 창에 다음과 같은 결과가 출력된다.

```
The letters in the variable 'city' are:
T
o
r
```

```
o
n
t
o
```

루프를 이용하여 어떤 데이터를 분리하는 것 외에도 다음과 같이 배열에 저장된 일련의 단어들을 합쳐서 문장을 만들고 데이터를 합치는 작업을 할 수 있다.

```
var words = ["Toronto", "is", "not", "the", "capital", "of", "Canada"];
var sentence;
var i = 0;
while (i < words.length) {
  sentence += words[i];       // 현재 단어를 sentence에 더한다.

  // 마지막 단어가 아니면
  if (i < words.length - 1) {
    sentence += " ";          // sentence 뒤에 공백을 추가한다.
  } else {
    sentence += ".";          // 마지막 단어라면 뒤에 마침표를 추가한다.
  }
  i++;
}
trace(sentence); // "Toronto is not the capital of Canada."가 출력된다.
```

거의 모든 루프에서 카운터(또는 반복자, 인덱스 변수라고도 부른다)를 사용한다. 카운터를 이용하면 데이터를 순서대로 처음부터 끝까지 액세스할 수 있다. 조금 전에 나온 예에서 보았듯이 조작하려는 배열이나 문자열의 length 속성을 이용하여 카운터의 최대값을 결정할 때는 카운터를 이용하는 것이 매우 편리하다.

또한 루프를 끝마치는 부분이 카운터에 의해 결정되지 않는 경우도 있다. 루프의 기준 표현식이 false가 되기만 하면 루프는 끝나게 된다. 아래 예에서는 플래시 플레이어의 레벨 스택을 검사하여 비어있는 첫 번째 레벨을 찾아낸다.

```
var i = 0;
while (typeof eval("_level" + i) == "movieclip") {
  i++;
}
trace("The first vacant level is" + i);

// 비어있는 레벨을 찾았으므로 그 레벨에 무비를 로드한다.
loadMovie("myMovie.swf", i);
```

순환문 관련 용어

바로 앞 절에서 몇 가지 새로운 용어가 등장했다. 루프를 다룰 때 필요한 용어를 이해할 수 있도록 그 용어에 대해 자세히 알아보자.

초기화

루프의 기준 표현식에서 쓰이는 한 개 이상의 변수를 정의하는 선언문 또는 표현식

기준 표현식

루프 본체에 있는 선언문이 실행되기 위해 만족되어야 하는 조건. '조건(condition)', '기준(test)', '제어부(control)'라는 용어로 불리기도 한다.

업데이트 선언문

기준 표현식으로 돌아가기 전에 기준 표현식에서 쓰이는 변수를 바꿔주는 선언문. 일반적인 업데이트 선언문에서는 루프의 카운터를 증가 또는 감소시킨다.

반복

기준 표현식과 루프 본체를 한 번 실행하는 것을 반복이라고 부른다. 종종 루프라고 부르기도 한다.

다단계 루프

일종의 이차원 데이터같은 것을 다루기 위해 루프 안에 다른 루프가 들어있는 루프를 다단계 루프라고 부른다. 예를 들어 어떤 표에 있는 각 행에 있는 열을 루프를 통해 돌리면서 모든 행을 다시 루프를 통해 돌리면 그 표에 있는 모든 내용을 확인할 수 있다. 바깥쪽 루프, 또는 최상위 루프에서는 각 행을 검사하고 안쪽 루프에서는 주어진 행에 있는 모든 열을 검사한다.

반복자 또는 인덱스 변수

루프를 매번 반복할 때 증가 또는 감소하는 변수. 보통 어떤 숫자를 세거나 데이터를 쭉 훑어볼 때 사용한다. 루프 반복자는 보통 카운터라고도 부른다. 반복자의 이름은 보통 i, j, k 또는 x, y, z와 같은 식으로 붙인다. 다단계 루프에서는 보통 i를 최상위 루프의 반복자로, j를 그 다음 단계에 있는 루프의 반복자로, k는 그 아래 단계 루프의 반복자 같은 식으로 사용한다. 물론 코드의 뜻을 명확하게 하고 싶다면 원하는 이름을 가지는 반복자를 사용해도 된다. 예를 들어 어떤 카

운터가 문자열의 현재 글자를 나타낸다는 것을 확실하게 나타내고 싶다면 charNum과 같은 변수 이름을 사용해도 된다.

루프 본체

루프의 조건이 만족될 때 실행되는 선언문 블록. 루프 본체는 경우에 따라 한 번도 실행되지 않을 수도 있고 수천 번 실행될 수도 있다.

루프 헤더 또는 루프 제어부

순환문 중에서 루프 선언문 키워드(while, for, do-while, for-in)와 루프 제어부를 포함하고 있는 부분. 루프 제어부는 루프의 종류에 따라 조금씩 다르다. for 루프에서는 제어부에서 초기화, 기준 표현식, 업데이트 선언문을 모두 처리한다. while 루프에서는 제어부에 단순히 기준 표현식만 들어있다.

무한 루프

기준 표현식이 절대로 false가 될 수 없기 때문에 무한정 반복되는 루프. '최대 반복 횟수' 부분에서 배우게 되겠지만 액션스크립트에서 무한 루프를 만들면 오류가 발생한다.

do-while 루프

앞에서 본 것처럼 while 선언문을 이용하면 주어진 조건이 true이면 계속해서 코드 블록을 반복할 수 있다. while 루프에서는 루프의 기준 표현식을 처음 확인할 때부터 그 조건이 만족되지 않으면 루프 본체를 한 번도 실행하지 않고 바로 건너뛰게 된다. do-while 선언문을 이용하면 루프 본체가 적어도 한 번은 실행되도록 할 수 있다. do-while 루프의 본체는 처음에 루프를 시작할 때 적어도 한 번은 실행된다. do-while 선언문의 문법은 다음과 같이 while 선언문을 뒤집어 놓은 것과 비슷하다.

```
do {
  substatements
} while (condition);
```

루프는 do라는 키워드로 시작된다. 그 뒤에는 substatements 자리에 루프 본체가 들어간다. 인터프리터에서 do-while 선언문을 처음 실행할 때는 condition으로 주어진 조건을 확인하기 전에 일단 substatements 부분을 실행하게 된다. substatements 블록이 끝나고 나서 condition이 true이면 루프를 다시 돌게 되며 substatements가 다시 실행된다. 이 루프는 condition이 false가 되기 전까지는 계속해서 실행된다. condition 부분을 감싸고 있는 괄호 뒤에 세미콜론을 붙여야 한다는 점에 주의하자.

이런 특징 때문에 어떤 작업을 일단 한 번 수행한 다음 조건에 따라 그 작업을 다시 반복하는 경우에 do-while을 사용하면 편리하다. 〔예제 8-2〕에서는 starParent라는 클립에서 여러 개의 빛나는 별 무비 클립을 복사하여 스테이지의 적당한 위치(난수로 정해지는 위치)에 올려놓는다. 따라서 numSatrs를 0으로 설정하더라도 적어도 한 개의 별은 스테이지에 올라가게 된다.

〔예제 8-2〕 do-while 루프 사용법

```
var numStars = 5;
var i = 1;
do {
  // starParent 클립을 복사한다.
  duplicateMovieClip(starParent, "star" + i, i);

  // 복사한 클립을 스테이지의 임의의 위치에 놓는다.
  _root["star" + i]._x = Math.floor(Math.random() * 551);
  _root["star" + i]._y = Math.floor(Math.random() * 401);
} while (i++ < numStars);
```

코드를 주의 깊게 살펴보면 기준 표현식에서 변수 i를 교묘하게 업데이트한다는 것을 눈치챌 수 있을 것이다. 5장에서 배웠듯이 후치 증가 연산자를 이용하면 먼저 피연산자 값을 리턴한 후에 그 값에 1을 더한다. 순환문을 사용할 때는 증가 연산자를 사용하는 것이 매우 편리하고, 실제 거의 모든 순환문에서는 증가 연산자를 사용하게 된다.

for 루프

for 루프는 문법이 간결하다는 점을 제외하면 while 루프와 똑같다. 가장 눈에 띄는 점은 루프 헤더에 기준 표현식 외에도 초기화 및 업데이트 선언문까지 포함된다는 점이다.

```
for (initialization; condition; update) {
  substatements
}
```

for 루프에서는 루프의 핵심적인 구성요소를 깔끔하게 루프 헤더에 세미콜론으로 구분하여 모아둔다. 초기화 선언문(initialization 부분)은 for 루프에서 첫 번째 반복을 시작하기 전에 한 번만 실행된다. 보통 반복자의 초기 값을 설정하는 내용이 여기에 들어간다. 다른 루프와 마찬가지로 condition 부분이 true이면 substatements가 실행된다. 그렇지 않으면 루프가 끝난다. 각 루프 반복이 끝날 때마다 조건을 다시 확인하기 전에 업데이트 선언문(update 부분)을 실행한다. 아래 예는 for 루프를 이용하여 1부터 10까지 세는 코드이다.

```
for (var i = 1; i <= 10; i++) {
  trace("Now serving number" + i);
}
```

위와 똑같은 내용을 while 루프로 만든 아래 코드를 보면, for 루프가 어떤 식으로 동작하는지 쉽게 이해할 수 있을 것이다.

```
var i = 1;
while (i <= 10) {
  trace("Now serving number" + i);
  i++;
}
```

일단 for 문법에 익숙해지고 나면, for 루프를 쓰면 공간을 절약하는 데도 도움이 되고 루프의 본체와 제어부를 전체적으로 훑어보기도 쉽다는 것을 알게 될 것이다.

for 루프에서 반복자를 여러 개 사용하기

루프에서 하나 이상의 요소를 제어하고 싶다면 두 개 이상의 반복자를 사용할 수도 있다. while 루프에서 여러 개의 반복자를 사용한다면 다음과 같은 형태가 된다.

```
var i = 1;
var j = 10;
while (i <= 10) {
  trace("Going up" + i);
  trace("Going down" + j);
  i++;
  j--;
}
```

쉼표 연산자를 이용하면 for 선언문에서도 똑같은 결과를 얻을 수 있다.

```
for (var i = 1, j = 10; i <= 10; i++, j--) {
  trace("Going up" + i);
  trace("Going down" + j);
}
```

for-in 루프

for-in 루프는 객체의 속성을 열거하기 위해 쓰이는 특화된 루프이다. 프로그래밍을 처음 배우는 독자라면 일단 이 부분은 넘어가고 '12장. 객체와 클래스'를 읽어본 후에 이 내용을 다시 보는 것이 좋다.

for-in 루프에서는 주어진 조건이 true인 동안 일련의 선언문을 반복하는 대신 주어진 객체의 각 속성에 대해 한 번씩 하위 선언문을 실행한다. 따라서 for-in 선언문에서는 업데이트 선언문을 사용할 필요가 없다. 루프를 도는 횟수가 주어진 객체에 포함된 속성 개수에 의해 결정되기 때문이다. for-in 루프의 구조는 다음과 같다.

```
for (var thisProp in object) {
  substatements;  // 선언문에서는 보통 thisProp 변수를 사용한다.
}
```

substatements 부분은 object 객체의 각 속성에 대해 한 번씩 실행된다. 제대로 된 객체 이름이라면 모두 object 자리에 쓰일 수 있다. thisProp 자리에는 임의의 변수 또는 인식자 이름을 사용하면 된다. 매번 루프를 돌 때마다 thisProp 변수에 현재 작업중인 객체 속성의 이름이 문자열 형태로 저장된다. 매번 반복할 때마다 이 문자열 값을 이용하여 현재 속성을 액세스하고 조작할 수 있다. for-in 루프를 이용한 가장 간단한 예는 객체에 있는 속성을 모두 출력하는 예제일 것이다. 여기서는 객체를 만들고 for-in 루프를 이용하여 그 객체의 속성을 모두 열거한다.

```
var ball = new Object();
ball.radius = 12;
ball.color = "red";
ball.style = "beach";

for (var prop in ball) {
  trace("ball has the property" + prop);
}
```

prop에는 ball에 들어 있는 속성의 이름이 문자열 형태로 저장되기 때문에, prop을 [] 연산자와 함께 사용하여 그 속성 값을 읽어들일 수 있다.

```
for (var prop in ball) {
  trace("ball." + prop + " is " + ball[prop]);
}
```

for-in 루프를 이용하여 속성을 읽어들이는 방법을 활용하면 타임라인에 있는 모든 무비 클립을 알아낼 수 있다. for-in 루프를 활용하여 무비 클립을 알아내는 방법은 〔예제 3-1〕에서 볼 수 있다.

for-in 루프에서 객체에 있는 속성을 확인하는 순서는 정해져 있지 않으므로 어떤 속성을 먼저 확인할지 알 수 없다. 또한 for-in 선언문에서 반드시 객체에 있는 모든 속성을 확인하는 것도 아니다. 사용자 정의 객체를 사용하는 경우에는 상속된 속성을 포함한 모든 속성을 확인하게 되지만, 내장 객체의 속성 중에는 for-in 선언문에서 그냥 건너뛰는 것도 있다. 예를 들어 내장 객체에 있는 메소드는 for-in 루프에서 검사하지 않는다. for-in 선언문을 이용하여 내장 함수의 속성을 조작하고 싶다면 우선 그 객체에서 사용자가 액세스할 수 있는 속성의 목록을 알아내기 위한 테스트용 루프를 하나 만드는 것이 좋다.

기본값이 정해지지 않은 입력 텍스트 필드는 for-in 루프로 확인할 수 없다. 따라서 입력 텍스트 필드를 그 필드가 포함되어 있는 타임라인에서 보통 변수로 직접 선언하지 않으면 비어있는 텍스트 필드를 찾아내기 위한 폼 검증 코드는 제대로 작동하지 않는다(18장의 '비어있는 텍스트 필드와 for-in 선언문' 참조).

for-in 선언문은 아래와 같이 배열의 원소를 추출할 때도 쓰일 수 있다.

```
for (var thisElem in array) {
  substatements; // 선언문에서 보통 thisElem 변수를 사용하게 된다.
}
```

아래 코드는 배열에 있는 원소의 목록을 출력하는 예제이다.

```
var myArr = [123, 234, 345, 456];
for (var elem in myArr) {
  trace(myArr[elem]);
}
```

루프 중단

간단한 루프에서는 기준 표현식을 통해서만 루프를 멈추게 된다. 간단한 루프에서 기준 표현식이 false가 되면 루프를 멈추게 된다. 하지만 루프가 복잡해지면 기준 표현식 값에 상관없이 루프를 중단해야 하는 경우도 있다. 이런 경우에는 break 및 continue 선언문을 이용하면 된다.

break 선언문

break 선언문은 현재 루프의 실행을 멈춘다. 사용법은 다음과 같이 매우 간단하다.

```
break
```

break 선언문에는 루프의 본체 안에 들어있어야 한다는 것 외에 다른 제약조건은 없다.

break 선언문을 이용하면 어떤 프로세스를 더 이상 수행하지 않아도 될 때 그 프로세스를 멈출 수 있다. 예를 들어 타임라인의 입력 텍스트 변수를 돌면서 사용자 입력 폼을 조사하는 for-in 루프를 만드는 경우를 생각해 보자. 만약 어떤 입력 필드가 비어 있으면 사용자에게 폼을 제대로 채우지 않았다는 경고문을 내보낼 수 있다. 이러한 루프는 break 선언문을 이용하여 중단시킬 수 있다. 〔예제 8-3〕에 그러한 코드가 나와 있다. 이 예제에서는 input01, input02와 같은 이름의 입력 변수를 선언하는 form이라는 무비 클립이 있다고 가정한다.

〔예제 8-3〕 간단한 폼-필드 검증기

```
for (var prop in form) {
  // 이 속성이 입력 텍스트 필드 중 하나인 경우에
  if (prop.indexOf("input") != -1) {
    // 폼의 어떤 입력사항이 비어 있으면 작업을 중단한다.
    if (form[prop] == "") {
      displayMessage = "Please complete the entire form.";
      break;
    }
    // break 명령어가 선언되면 그 뒤에 있는 하위 선언문은
    // 실행되지 않는다.
  }
}
// break 명령을 이용하든 기준 표현식이 false가 되든 for-in 루프가
// 끝나고 나면 이 부분에서 프로그램이 계속된다.
```

무한 루프에서도 break 선언문을 이용하면 루프를 멈출 수 있다. 이렇게 하면 if (condition) break; 문 뒤에 있는 선언문을 실행시키지 않은 채로 앞부분에 있는 코드 블록만 실행할 수도 있다. 이런 기법은 일반적으로 〔예제 8-4〕와 같은 형식으로 구현한다.

〔예제 8-4〕 무한 루프에서 빠져나오는 법

```
while (true) {
  // 앞부분의 선언문
  if (condition) break;
  // 그 뒤에 나올 선언문
}
```

continue 선언문

continue 선언문은 현재 루프 반복을 멈추게 된다는 점에서 break 선언문과 비슷하다. 하지만 break와는 달리 루프를 그냥 끝내는 것이 아니라 루프의 시작 부분으로 돌아가서 루프를 계속 진행한다. continue 선언문의 문법은 다음과 같이 매우 간단하다.

```
continue
```

어떤 종류의 루프에서 사용하든 continue 선언문은 현재 실행중인 루프 본체 부분을 멈추고 다음 반복을 수행하는데, 다시 시작하는 지점은 루프 선언문의 종류에 따라 약간씩 다르다. while 루프와 do-while 루프에서는 루프를 계속해서 실행하기 전에 기준 표현식을 검사한다. 하지만 for 루프에서는 기준 표현식을 검사하기 전에 루프 업데이트를 먼저 처리한다. 또한 for-in 루프에서는 조사하고자 하는 객체의 다음 속성에 대해 루프를 계속해서 반복한다.

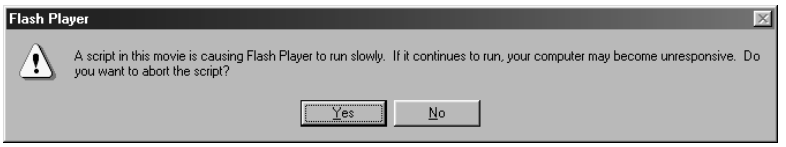
continue 선언문을 이용하면 특정한 상황에서 루프 본체의 일부를 실행하지 않고 건너뛰도록 만들 수 있다. 예를 들어 아래 예에서는 투명하지 않은 무비 클립 인스턴스를 모두 스테이지의 왼쪽 끝으로 옮기고 투명한 인스턴스에 대해서는 루프 본체를 건너뛴다.

```
for (var prop in _root) {
  if (typeof _root[prop] == "movieclip") {
    if (_root[prop]._alpha < 100) {
      continue;
    }
    _root[prop]._x = 0;
  }
}
```

최대 반복 횟수

앞에서 설명한대로 액션스크립트에서는 무한 루프를 실행할 수 없다. 플래시 5 플레이어에서는 루프를 15초까지만 실행시킬 수 있다. 그 시간동안 가능한 반복 횟수는 루프 안에서 처리하는 작업과 컴퓨터의 속도에 따라 달라진다. 프로그램을 안전하게 만들려면 몇 초(사실 컴퓨터 입장에서 보면 매우 긴 시간이다) 이상 실행되는

루프도 만들지 않는 것이 좋다. 대부분의 루프는 수 밀리초 안에 끝난다. 루프를 실행하는 데 시간이 오래 걸린다면(예를 들면 단어 섞기 게임처럼 수백 개의 문자열을 처리해야 하는 경우에는 시간이 오래 걸린다), 다음 절에서 배우는 내용을 이용하여 타임라인 루프를 이용하는 코드로 고치는 것이 좋다. 타임라인 루프를 이용하면 화면에 스크립트가 얼마나 진행되었는지를 보여줄 수 있으므로, [그림 8-1]에 나온 것과 같은 오류 메시지가 출력되는 것을 방지할 수 있다.



[그림 8-1] 루프에 문제가 있으므로 실행을 중단한다.

플래시 5 플레이어에서 하나의 루프가 15초 이상 실행되면 무비에 있는 스크립트에서 무비 재생이 지연되고 있다는 경고문이 나온다. 이 때 사용자가 스크립트가 실행될 때까지 기다릴지 아니면 스크립트를 중단시킬지 결정할 수 있다.

플래시 4 플레이어는 더 엄격한 기준을 적용하여 200,000번 이상 루프를 반복할 수 없게 되어 있다. 만약 200,000번 이상 실행되면 아무런 경고 없이 모든 스크립트가 중단된다.

특히 주의할 점: 스크립트가 15초 이상 실행될 때 경고 메시지가 나온다고 해도 바로 스크립트가 중지되는 것은 아니다. 사용자가 '예'를 선택해야만 무비에 있는 모든 스크립트가 멈추게 된다.

타임라인 루프와 클립 이벤트 루프

지금까지 알아본 모든 루프는 인터프리터에서 코드 블록을 반복하여 실행하도록 만드는 역할을 한다. 대부분의 루프는 이러한 '액션스크립트 선언문' 유형에 해당한다. 하지만 플래시의 플레이헤드를 타임라인에서 순환시키는 타임라인 루프를 만드는 것이 더 나은 경우도 있다. 그렇게 하려면 일련의 선언문을 임의의 프레임에 추가하고 다음 프레임에 바로 앞 프레임으로 움직이는 gotoAndPlay() 함수를 붙이면 된다. 무비가 실행되면 플레이헤드가 그 두 프레임 사이에서 계속 순환하게 된다. 따라서 첫 번째 프레임에 있는 코드가 반복적으로 실행된다.

다음 단계를 거치면 간단한 타임라인 루프를 만들 수 있다.

1. 새로운 플래시 무비를 시작한다.
2. 1번 프레임에 다음과 같은 선언문을 추가한다.

```
trace("Hi there! Welcome to frame 1");
```

3. 2번 프레임에 다음과 같은 선언문을 추가한다.

```
trace("This is frame 2");
gotoAndPlay(1);
```

4. Control → Test Movie를 선택한다.

무비를 테스트하면 다음과 같이 텍스트가 끊임없이 출력된다.

```
Hi there! Welcome to frame 1
This is frame 2
Hi there! Welcome to frame 1
This is frame 2
```

타임라인 루프를 이용하면 보통 루프로는 할 수 없는 다음과 같은 일을 처리할 수 있다.

- 오류 없이 어떤 코드 블록도 무한히 여러 번 실행시킬 수 있다.
- 루프를 반복하는 중간에 스테이지를 업데이트하는 코드 블록을 실행시킬 수 있다.

타임라인 루프의 두 번째 기능에 대해 조금 더 자세히 알아보자. 어떤 프레임 코드를 실행시키더라도 스크립트가 끝나기 전에는 무비 스테이지가 업데이트되지 않는다. 즉 일반적인 루프 선언문을 이용하는 것만으로는 시각적, 또는 청각적인 작업을 반복적으로 수행할 수 없다. 작업 결과가 루프를 반복하는 도중에 실제로 반영되지 않기 때문이다. 예를 들어 무비 클립의 위치를 옮기려면 스테이지를 업데이트해야 한다. 따라서 일반적인 루프 선언문으로는 프로그래밍을 통해 무비 클립을 움직일 수 없다.

다음 코드를 이용하면 ball 무비 클립을 스테이지에서 수평으로 움직일 수 있을 거라는 생각이 들지도 모른다.

```
for (var i = 0; i < 50; i++) {
  ball._x += 10;
}
```

개념으로만 본다면 위 루프 선언문을 통한 접근법은 문제가 전혀 없다. ball의 위치를 조금씩 움직이면 그 클립이 움직이는 것처럼 보이기 때문이다. 하지만 사실 ball의 _x 값이 바뀔 때마다 ball 클립이 움직이지는 않는다. 대신 스크립트가 완료된 뒤에 ball 클립이 한꺼번에 500픽셀(10픽셀씩 50번 더했으므로 500픽셀이 된다) 오른쪽으로 움직이는 결과가 발생한다.

매번 ball._x += 10;이라는 선언문이 실행될 때마다 스테이지가 업데이트되게 하려면 다음과 같은 타임라인 루프를 이용하면 된다.

```
// 1번 프레임의 코드
ball._x += 10;

// 2번 프레임의 코드
gotoAndPlay(1);
```

플래시에서는 두 프레임 사이를 움직일 때마다 스테이지를 업데이트하므로 ball이 움직이는 것처럼 보인다. 하지만 타임라인 루프는 그 부분의 타임라인을 완전히 독점하게 된다. 이 루프가 실행되는 동안에는 그 타임라인에 있는 다른 내용은 재생할 수 없다. 더 좋은 방법은 타임라인 루프를 비어 있는 프레임 두 개짜리 무비 클립에 집어넣는 방법이다. 다른 애니메이션에 필요한 타임라인을 독점하지 않고도 루프가 반복되는 사이에 스테이지가 업데이트되는 기능을 활용할 수 있다.

비어있는 클립의 타임라인 루프 만들기

다음 단계를 거치면 비어있는 클립의 타임라인 루프를 만들 수 있다.

1. 새로운 플래시 무비를 시작한다.
2. 원 모양이 들어 있는 ball이라는 이름의 무비 클립 심벌을 만든다.
3. 메인 스테이지에서 Layer 1의 이름을 ball로 바꾼다.
4. ball 레이어에 ball 심벌의 인스턴스를 추가한다.
5. ball 클립의 인스턴스 이름을 ball로 설정한다.
6. Insert → New Symbol을 선택하여 비어 있는 무비 클립 심벌을 만든다.
7. 그 클립 심벌의 이름을 process로 설정한다.
8. process 클립의 1번 프레임에 다음과 같은 코드를 추가한다.

```
_root.ball._x += 10;
```

9. process 클립의 2번 프레임에 다음과 같은 코드를 추가한다.

```
gotoAndPlay(1);
```

10. 메인 무비 타임라인으로 돌아가서 scripts라는 레이어를 만든다.
11. scripts 레이어에 process 심벌의 인스턴스를 추가한다.
12. 새로 추가한 인스턴스의 이름을 processMoveBall로 설정한다.
13. Control → Test Movie를 선택한다.

이렇게 하면 processMoveBall 인스턴스에서 ball을 포함하고 있는 메인 타임라인이 재생되는 것을 방해하지 않으면서도 ball을 움직일 수 있다.

12번 단계는 꼭 필요한 것은 아니지만, 그렇게 해두면 루프를 더욱 자유롭게 제어할 수 있다. 타임라인 루프 인스턴스에 이름을 정해 두면 다음과 같은 방법으로 그 인스턴스를 멈추거나 재생시킴으로써 루프를 시작하거나 멈출 수 있다.

```
processMoveBall.play();
processMoveBall.stop();
```

위 예제에서 루프를 작동시키려면 processMoveBall과 ball은 모두 메인 타임라인에 들어가야 한다. 위 코드를 조금 더 쉽게 다른 곳에서 사용하고 싶다면 process에서 ball 클립에 대한 상대 레퍼런스를 사용하면 된다.

```
_parent.ball._x += 10;
```

임의의 타임라인에서 공을 제어하려면 다음과 같이 ball에 대한 절대 레퍼런스를 이용해야 한다.

```
_root.ball._x += 10;
```

타임라인 루프는 하나의 프레임에서는 루프를 돌 수가 없다. 즉 어떤 무비의 5번 프레임에 gotoAndPlay(5)라는 선언문이 있으면 그 함수는 그냥 무시된다. 플레이어에서는 플레이헤드가 이미 5번 프레임에 있다는 것을 알고 있기 때문에 이러한 함수를 실행하지 않는다.

온라인 코드 창고에서 타임라인 루프 및 비어있는 클립의 타임라인 루프의 예제 .fla 파일을 구할 수 있다.

플래시 5 클립 이벤트 루프

타임라인 루프로도 원하는 작업을 할 수 있지만 그다지 깔끔한 방법은 아니다. 플래시 5에서는 무비 클립에 있는 이벤트 핸들러를 이용하여 타임라인 루프를 이용하는 것과 똑같은 효과를 얻으면서도 더 유연한 코드를 만들 수 있다(여기에 있는 예제를 그냥 따라해 보거나 무비 클립 이벤트 핸들러에 대한 자세한 내용을 원한다면 '10장. 이벤트와 이벤트 핸들러' 를 참고하기 바란다).

enterFrame 이벤트 핸들러를 무비 클립에 추가하면 무비에서 어떤 프레임을 지나갈 때마다 코드 블록을 실행하도록 할 수 있다. 한 프레임짜리 비어있는 클립에 enterFrame 이벤트 핸들러를 추가하면(타임라인 루프를 이용하는 경우와 마찬가지로) 매번 스테이지를 업데이트하면서 코드 블록을 반복하여 실행시킬 수 있다. 아래에 나온 각 단계를 따라해 보자.

1. 앞 절에 나온 부분의 1단계부터 7단계까지 처리한다.
2. 메인 스테이지에 scripts라는 레이어를 새로 만든다.
3. scripts 레이어에 process 클립의 인스턴스를 추가한다.
4. process 인스턴스를 선택하고 다음과 같은 코드를 추가한다.

```
onClipEvent(enterFrame) {
  _root.ball._x += 10;
}
```

5. Control → Test Movie를 선택한다.

이렇게 하면 ball 인스턴스가 스테이지 위에서 움직이게 된다.

클립 이벤트 루프를 이용하면 코드를 무비 클립 안에 넣지 않아도 되고 타임라인 루프에서처럼 두 개의 프레임을 사용하지 않아도 된다. 클립 이벤트 루프의 모든 액션은 하나의 이벤트 핸들러에서 일어난다. 하지만 방금 살펴본 클립 이벤트 예제에도 단점은 있다. 루프를 일단 시작하고 나면 프로그램을 통해 그 루프를 멈추거나 다시 시작할 수 없다는 점이다. 방금 만든 루프를 멈추는 방법은 비어있는 키프레임을 이용하여 물리적으로 process 인스턴스를 제거하는 방법밖에 없다.

임의로 시작하거나 정지시킬 수 있는 이벤트 루프를 만들려면 이벤트 루프가 들어있는 비어있는 클립이 들어 있는 비어있는 클립을 만들어야 한다. 그렇게 하면 루프를 시작하거나 멈출 때마다 패키지 전체를 동적으로 추가하거나 제거할 수 있다. 물론 약간 복잡하긴 하지만 그렇게 하면 꽤 유연한 프로그램을 만들 수 있다. 다음 단계를 거치면 된다.

1. '비어있는 클립의 타임라인 루프 만들기'에 나온 1단계부터 5단계까지 과정을 처리한다.
2. Insert → New Symbol을 두 번 선택하여 두 개의 비어있는 무비 클립 심벌을 만든다.
3. 클립 심벌의 이름을 각각 process와 eventLoop로 설정한다.
4. 라이브러리에서 process 클립을 선택하고 나서 Options → Linkage를 선택한다. 그러면 Symbol Linkage Properties 대화상자가 나타난다.

5. Export This Symbol을 선택한다.

6. Identifier 필드에 processMoveBall이라고 입력하고 OK를 클릭한다.

7. process 클립의 1번 프레임에서 eventLoop의 인스턴스를 스테이지에 끌어 놓는다.

8. eventLoop 인스턴스를 선택하고 다음과 같은 코드를 입력한다.

```
onClipEvent(enterFrame) {
  _parent._parent.ball._x += 10;
}
```

9. 메인 무비 타임라인으로 돌아와서 1번 프레임에 다음과 같은 코드를 입력한다.

```
attachMovie("processMoveBall", "processMoveBall", 5000);
```

10. 이벤트 루프를 멈추고 싶다면 다음과 같은 선언문을 이용하면 된다.

```
_root.processMoveBall.removeMovieClip();
```

11. Control → Test Movie를 선택한다.

이렇게 하면 ball 인스턴스가 스테이지를 가로질러 움직이며, 9번, 10번 단계에 나온 attachMovie()와 removeMovieClip() 함수를 이용하면 언제든지 애니메이션을 시작하거나 멈추게 할 수 있다.

온라인 코드 창고에 가면 일반적인 클립 이벤트 루프, 사용자가 제어할 수 있는 클립 이벤트 루프의 예제를 구할 수 있다.

쉽게 이식할 수 있는 이벤트 루프

앞에서 살펴본 클립 이벤트 루프에는 스테이지의 ball 인스턴스 위치를 업데이트하는 다음과 같은 코드가 들어 있다.

```
onClipEvent(enterFrame) {
  _parent._parent.ball._x += 10;  // ball의 위치를 업데이트한다.
}
```

이런 식으로 해도 제대로 동작하긴 하지만 코드가 조금 지저분하다. 클립 이벤트에 의미 있는 코드를 덧붙이는 방법으로 코드 베이스를 분산시켜 필요한 논리와 특징을 여러 곳에 나누어 놓을 수 있다. 제작 과정에서 코드를 사용할 수 있도록 하고 나중에 이벤트 루프에서 코드를 재사용하기 쉽게 하려면 함수를 호출하는 수밖에 없다. 따라서 앞 예제에서 했던 것처럼 ball 클립을 직접 움직이는 대신 다음과 같이 ball 클립을 움직이는 함수를 호출해야 한다.

```
onClipEvent(enterFrame) {
  _parent._parent.moveBall();
}
```

사용자 정의 함수인 moveBall()은 processMoveBall 클립을 포함하고 있는 타임라인에서 다음과 같이 정의한다.

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

함수와 코드의 이식성에 관한 것은 '9장. 함수'에서 자세히 알아보자.

간단한 애플리케이션에서는 앞에서 배운 비어있는 클립의 이벤트 루프를 사용하는 것이 나을 수도 있다. 경우에 따라서 조작하고자 하는 클립에 직접 이벤트 루프를 추가할 수도 있다. ball 예제의 경우 다음과 같은 코드를 ball 인스턴스에 바로 추가하면 비어있는 클립을 별도로 만들지 않아도 된다.

```
onClipEvent(enterFrame) {
  _x += 10;
}
```

이러한 방법은 매우 편리하긴 하지만 그다지 확장성이 좋지 않고 첫째 예제와 마찬가지로 사용자가 임의로 시작하거나 중단시킬 수 없다는 단점이 있다.

타임라인 루프와 클립 이벤트 루프, 프레임 속도

타임라인 루프와 클립 이벤트 루프는 매 프레임마다 한 번씩 반복되므로 그 실행 속도가 결국 무비의 프레임 속도에 의해 결정된다. 타임라인 루프 또는 이벤트

루프를 이용하여 어떤 객체를 화면 위에서 움직일 때는 프레임 속도를 증가시키면 애니메이션 속도도 같이 빨라진다.

앞 예제에서 ball 클립의 움직임을 조절할 때는 간접적으로 프레임 속도에 대해 상대적으로 공의 속도를 지정하였다. 우리가 사용한 코드는 '프레임이 지나갈 때마다 ball을 10픽셀 오른쪽으로 움직인다' 라는 의미를 가지기 때문이다.

```
_ball += 10;
```

따라서 ball의 움직임은 프레임 속도에 의해 결정된다. 초당 12프레임의 속도로 무비를 재생한다면 ball 클립은 초당 120픽셀씩 움직일 것이다. 만약 무비를 초당 30프레임의 속도로 재생하면 ball 클립은 초당 300픽셀씩 움직인다.

스크립트의 애니메이션 속도를 결정할 때는 아이템이 이동할 거리를 무비의 프레임 속도에 따라 계산하는 것이 좋다. 만약 무비의 속도가 초당 20프레임이고 아이템을 초당 100픽셀씩 움직이고 싶다면, 객체의 속도를 프레임당 5픽셀(5픽셀 * 초당 20프레임 = 초당 100픽셀)로 맞추는 게 좋다. 하지만 이러한 방법에는 두 가지 심각한 문제가 있다.

- 아이템 속도를 결정할 때 프레임 속도에 의존하면 프레임 속도를 변경하기가 너무 힘들어진다. 프레임 속도를 바꾸면 모든 아이템 속도를 다시 계산하고 그 값에 맞춰서 코드를 변경해야 하기 때문이다.

- 플래시 플레이어에서는 항상 플래시 저작도구에서 지정한대로 무비 재생 속도를 맞추지 않는다. 종종 속도가 느려지는 경우가 많이 있다. 무비를 재생하는 컴퓨터에서 지정된 프레임 속도를 맞출 수 없다면 무비 속도가 떨어지게 된다. 이러한 속도 저하는 시스템에 걸린 부하에 따라 달라진다. 만약 다른 프로그램이 실행되고 있거나 플래시에서 프로세서를 많이 사용하는 작업을 처리한다면, 잠깐 동안 프레임 속도가 떨어졌다가 잠시 후에 원래대로 빨라지는 경우도 있다.

  다음 사이트에서 구할 수 있는 속도 측정 소프트웨어를 이용하여 직접 위와 같은 현상을 확인할 수 있다.

  http://www.moock.org /webdesign/flash/actionscript/fps-speedometer

상황에 따라 재생 속도가 달라지는 것이 그럭저럭 참을 만할 경우도 있다. 하지만 정확성이 요구되거나 액션 게임에서 재빠른 반응이 필요한 경우에는 객체를 움직이는 거리를 프레임 속도 대신 실제 경과된 시간을 이용하여 계산하는 것이 좋다. [예제 8-5]에는 시간을 기반으로 한 애니메이션(즉 ball의 속도가 프레임 속도에 의해 결정되지 않는 애니메이션)을 간략하게 구현해 본 코드가 나와 있다. 이렇게 해서 만든 무비에는 세 개의 프레임과 두 개의 레이어가 있는데, 한 레이어에는 ball 인스턴스가 들어 있고 다른 레이어에는 스크립트가 포함되어 있다.

[예제 8-5] 프레임 속도가 아닌 시간을 기반으로 움직일 거리를 계산하는 코드

```
// 1번 프레임의 코드
var distancePerSecond = 50;  // 초당 움직일 픽셀 수
var now = getTimer();        // 현재 시각
var then = 0;                // 마지막 프레임을 렌더링한 시각
var elapsed;                 // 프레임을 렌더링하는 시간 간격(밀리초 단위)
var numSeconds;              // elpased를 초 단위로 나타낸 값
var moveAmount;              // 각 프레임당 움직일 거리

// 2번 프레임의 코드
then = now;
now = getTimer();
elapsed = now - then;
numSeconds = elapsed / 1000;
moveAmount = distancePerSecond * numSeconds;
ball._x += moveAmount;

// 3번 프레임의 코드
gotoAndPlay(2);
```

이처럼 시간을 기반으로 하는 애니메이션은 프레임 속도가 갑자기 바뀌면 뭔가 걸리는 것처럼 부드럽지 않게 움직일 수도 있다. 이런 경우에는 프레임 두 개만이 아니라 여러 프레임 사이의 평균을 재서 시간 간격을 측정하면 조금 더 부드럽게 처리할 수 있다.

앞으로 배울 내용

지금까지 꽤 많은 내용을 배웠다. 이 장을 마지막으로 액션스크립트 선언문을 모두 배움으로써 중대한 이정표를 세우게 되는 셈이다. 즉 변수, 데이터, 데이터형, 표현식, 연산자, 그리고 선언문을 모두 배웠다. 언어의 이러한 구성요소는 모든 스크립트의 기초가 되는 내용이다. 지금까지 배운 내용을 전부, 아니면 거의 모두 이해했다면 액션스크립트를 쓸 줄 안다고 말할 수 있다.

1부의 나머지 부분에서는 스크립트를 더욱 강력하게 만드는 법을 배울 것이다. 이식성이 좋은 코드를 만드는 방법, 코드를 동작시키는 이벤트를 만드는 법, 복잡한 데이터를 관리하는 법, 프로그램을 통해 무비 클립을 조작하는 법과 같은 고급 주제를 다룰 것이다. 이러한 테크닉을 이용하면 다양한 고급 응용 예제도 만들 수 있다.