13

무비 클립

모든 플래시 문서에는 도형, 텍스트와 같은 시각적인 요소들을 포함하는 '스테이지(Stage)'와 스테이지의 내용 변화 기준이 되는 메인 타임라인이 하나씩 들어간다. 스테이지, 즉 '메인 무비(main movie)'에는 무비 클립, 또는 줄여서 클립이라고도 부르는 독립적인 '서브무비(submovie)'를 포함시킬 수 있다. 무비 클립은 각각 독립된 타임라인과 '캔버스(canvas, 스테이지는 메인 무비의 캔버스이다)'를 가진다. 다른 클립을 포함하는 클립은 '호스트 클립(host clip)' 또는 '부모 클립(parent clip)'이라고 부른다.

하나의 플래시 문서에서는 서로 연관된 무비 클립들을 조직화할 수 있다. 예를 들어 메인 무비에 산악지형의 경관을 집어 넣었다고 생각해 보자. 여기에 캐릭터를 포함하는 무비 클립을 따로 추가하여 메인 무비 위에서 움직이게 하면 캐릭터가 걸어가는 것처럼 보이도록 만들 수 있다. 또한 캐릭터 클립에 또 다른 무비 클립을 포함시켜 캐릭터의 눈만 따로 깜빡이도록 할 수도 있다. 만화 캐릭터에 있는 각 요소를 한꺼번에 재생시키면 하나의 무비로 만든 것처럼 보이게 된다. 또한 캐릭터가 정지하면 눈이 깜빡이게 만든다거나 캐릭터가 움직이기 시작하면 다리가 걷는 모양으로 움직이도록 하는 것처럼 각 컴포넌트를 서로 연동시킬 수도 있다.

액션스크립트를 이용하면 여러 개의 무비 클립을 세밀하게 조절하여 원하는 대로 클립을 움직이거나 멈추게 할 수도 있고 플레이헤드를 고유의 타임라인 안에서 움직일 수도 있으며, 프로그램을 이용하여 속성(크기, 회전 각도, 투명도, 스테이지에서의 위치 등)을 바꾸거나 실제 프로그래밍 객체로 다룰 수도 있다. 무비 클립은 액션스크립트 언어의 정형적인 컴포넌트로, 플래시에서 프로그램을 이용하여 생성된 내용물을 만드는 원재료로 쓰일 수 있다. 예를 들어 무비 클립을 탁구 게임의 공 또는 탁구채로 사용할 수도 있고 제품 카탈로그로 쓰이는 웹사이트에서 주문용 폼으로 사용할 수도 있으며, 애니메이션에서 배경 음향을 담기 위한 컨테이너로 사용할 수도 있다. 이 장의 마지막 부분에서는 무비 클립을 시계바늘로 사용하는 법, 그리고 객관식 퀴즈의 답으로 사용하는 법을 알아볼 것이다.

무비 클립의 객체성

플래시 5부터 무비 클립을 '12장. 객체와 클래스'에서 배운 객체와 같이 사용할 수 있는 기능이 지원된다. 클립을 읽어 들여 속성을 설정할 수도 있고 클립의 내장 메소드 또는 사용자가 만든 메소드를 호출할 수도 있다. 하지만 다른 객체와는 달리 클립에 대해 어떤 연산을 수행하면 그 결과가 플레이어의 화면에 눈으로 보이는 요소로 나타날 수도 있고 소리로 나타날 수도 있다.

무비 클립은 정확하게 말하자면 객체의 일종은 아니다. MovieClip이라는 클래스도 없는 데다가 코드에서 객체 리터럴을 이용하여 무비 클립의 인스턴스를 만들 수도 없다. 무비 클립이 객체가 아니라면 도대체 무엇일까? 무비 클립은 'movieclip'이라는 객체와 비슷한 데이터형에 속한다(실제로 무비 클립에 대해 typeof 연산자를 실행시키면 "movieclip"이라는 문자열이 리턴된다). 무비 클립과 실제 객체의 주된 차이점은 할당(생성) 및 제거 방법에 있다. 자세한 내용은 '15장. 고급 주제'를 참고하기 바란다. 이러한 기술적인 차이점을 제외하면 무비 클립은 거의 언제나 일반적인 객체와 똑같이 다룰 수 있다.

그렇다면 이런 무비 클립의 '객체성'을 이용하여 액션스크립트에서 무엇을 할 수 있을까? 가장 주목할만한 것은 클립을 제어하고 그 속성을 조사하는 것이다. 무비 클립을 조절할 때 다음 예제처럼 내장 메소드를 직접 이용할 수 있다.

```
eyes.play();
```

다른 객체의 속성을 액세스하는 것과 마찬가지로 점 연산자(.)를 이용하여 무비 클립의 속성을 알아내거나 설정할 수 있다.

```
ball._xscale = 90;
var radius = ball._width / 2;
```

무비 클립에 있는 변수가 바로 그 클립의 속성이므로 점 연산자를 이용하여 변수 값을 설정하거나 알아낼 수 있다.

```
myClip.myVariable = 14;
x = myClip.myVariable;
```

서브무비 클립은 그 부모 무비 클립의 객체 속성인 것처럼 다룰 수 있다. 따라서 서브무비를 액세스할 때도 점 연산자를 사용하면 된다.

```
clipA.clipB.clipC.play();
```

현재 가리키고 있는 클립을 포함하는 부모 클립을 참조하려면 _parent라는 예약어를 사용하면 된다.

```
_parent.clipC.play();
```

클립을 객체로 다루면 문법을 간단하게 만들 수 있고 재생 과정을 유연하게 제어할 수 있다는 장점이 있다. 하지만 클립을 객체로 다룰 때 클립을 데이터로 다룰 수 있게 된다는 점도 간과할 수 없다. 무비 클립을 배열의 원소 또는 어떤 변수에 저장하거나 클립에 대한 레퍼런스를 함수 인자로 전달하는 것도 가능하다. 아래 예제는 클립을 화면의 특정 위치로 옮기는 함수 코드이다.

```
function moveClip (clip, x, y) {
  clip._x = x;
  clip._y = y;
}
moveClip(ball, 14, 399);
```

이 장에서는 무비 클립을 데이터 객체로 간주하여 레퍼런스를 이용하거나 무비 클립을 제어하고 조작하는 방법에 대해 자세히 알아보겠다.

무비 클립 유형

모든 무비 클립이 똑같은 것은 아니다. 플래시에서 사용할 수 있는 무비 클립은 다음과 같이 세 가지 유형으로 분류할 수 있다.

- 메인 무비
- 일반 무비 클립
- 스마트 클립

위와 같은 공식적인 분류 외에도 일반 무비 클립을 용도에 따라 다음과 같은 하위 카테고리로 분류할 수 있다.

- 프로세스 클립
- 스크립트 클립
- 연결 클립
- 씨앗 클립

위와 같은 카테고리는 액션스크립트에서 사용하는 공식 용어는 아니지만 무비 클립을 이용하는 프로그램을 만들 때 유용하게 사용할 수 있다. 이제 각 무비 클립 타입을 자세히 알아보자.

메인 무비

플래시 문서의 메인 무비에는 반드시 기본적인 타임라인과 스테이지가 포함된다. 메인 무비는 다른 모든 무비 클립을 포함하는 플래시 문서의 모든 내용물의 기반을 이룬다. 메인 무비는 메인 타임라인, 메인 무비 타임라인, 메인 스테이지, 또는 루트라는 명칭으로 부르기도 한다.

메인 무비를 조작하는 방법은 전체적으로 일반 무비 클립을 다루는 방법과 비슷하지만 다음과 같은 면에서 차이가 난다.

- 메인 무비는 .swf 파일에서 제거할 수 없다(물론 플래시 플레이어에서 .swf 파일 자체를 제거할 수는 있다).

- 메인 무비에 대해서는 duplicateMovieClip(), removeMovieClip(), swapDepths() 무비 클립 메소드를 호출할 수 없다.
- 메인 무비에 이벤트 핸들러를 만들 수 없다.
- 메인 무비는 전역 속성인 _root와 _leveln을 이용하여 참조할 수 있다.

하나의 .swf 파일에는 메인 무비를 하나밖에 만들 수 없지만 플래시 플레이어에서는 여러 개의 .swf 파일을 동시에 사용할 수 있다는 점을 기억해 두자. 앞으로 배우게 될 loadMovie() 및 unloadMovie() 함수를 통해 levels 스택에 여러 개의 .swf 문서(따라서 여러 개의 메인 무비)를 동시에 로드할 수 있다.

일반 무비 클립

'일반 무비 클립(regular movie clip)'은 컨텐츠를 담기 위한 가장 일반적이면서 기초적인 컨테이너이다. 일반 무비 클립에는 시각적인 요소와 소리를 저장할 수 있으며 이벤트 핸들러를 통하여 사용자가 입력하는 내용 또는 무비의 재생에 대해 반응할 수도 있다. DHTML을 사용하는 데 익숙한 자바스크립트 프로그래머라면 메인 무비를 HTML 문서 객체로, 일반 무비 클립을 문서의 레이어 객체로 생각하면 이해하기 수월할 것이다.

스마트 클립

플래시 5부터 도입된 '스마트 클립(smart clip)'은 저작 도구에서 클립 속성을 사용자가 직접 조절하는 데 사용하는 그래픽 사용자 인터페이스가 포함된 일반 무비 클립의 일종이다. 스마트 클립은 보통 전문 프로그래머들이 클립 코드의 동작 원리를 자세히 모르는 플래시 초보자들도 클립의 특성을 원하는 대로 바꿀 수 있도록 하기 위한 용도로 쓰인다. 스마트 클립은 '16장. 액션스크립트 저작 환경'에서 자세히 다룬다.

프로세스 클립

'프로세스 클립(process clip)' 은 특별한 내용을 담기 위한 클립이 아닌 코드 블록을 반복해서 실행하기 위한 목적으로 만드는 무비 클립을 뜻한다. 프로세스 클립은 enterFrame 이벤트 핸들러, 또는 '8장. 순환문' 의 '타임라인 루프와 클립 이벤트 루프' 에서 배운 타임라인 루프를 이용하여 만들 수 있다.

프로세스 클립은 액션스크립트에서 자바스크립트 윈도우 객체의 setTimeout(), setInterval() 메소드를 대체하는 것이라고 봐도 무방하다.

스크립트 클립

'스크립트 클립(script clip)' 은 프로세스 클립과 마찬가지로 특별한 내용을 담기 위한 클립이 아니라 어떤 변수를 추적하거나 스크립트를 실행하기 위한 무비 클립이다. 예를 들어 키보드 또는 마우스 이벤트를 감지하기 위한 이벤트 핸들러가 필요할 때 스크립트 클립을 이용할 수 있다.

연결 클립

'연결 클립(linked clip)' 은 무비 라이브러리로부터 연결된 무비 클립이다. 보내기 또는 가져오기 설정은 라이브러리에 나와 있는 각 무비 클립의 연결 옵션에서 알아낼 수 있다. 연결 클립은 앞으로 배우게 될 attachMovie() 클립 메소드를 이용하여 라이브러리 심볼로부터 직접 클립의 인스턴스를 동적으로 생성할 때 흔히 사용한다.

씨앗 클립

플래시 5에서 attachMovie() 메소드가 도입되기 전까지는 원래 있던 '씨앗 클립(seed clip)' 이라고 부르는 무비 클립을 기반으로 새로운 무비 클립을 만들기 위해 duplicateMovieClip()이라는 함수를 사용했다. 씨앗 클립은 duplicateMovieClip()을 통해 다른 클립에 복사하기 위한 용도만으로 스테이지에 붙어 있는 클립이다. attachMovie()가 도입되면서 씨앗 클립을 더 이상 사용할 필요가 없어졌다. 하지만

클립을 복사하는 과정에서 이벤트 핸들러 및 변환 기능을 그대로 유지하려는 경우에는 여전히 씨앗 클립과 duplicateMovieClip()을 사용한다.

내용물을 동적으로 생성하기 위해 duplicateMovieClip()을 많이 사용하는 무비에서는 무비 캔버스 외각에 여러 개의 씨앗 클립을 사용하는 경우가 흔히 있다. 이때 씨앗 클립은 클립을 복사하기 위한 용도로만 쓰이기 때문에 스테이지 위에 그대로 드러나지는 않는다.

무비 클립 생성

무비 클립을 다루는 방법은 일반적인 데이터 객체를 다루는 법과 같다. 점 연산자를 이용하여 속성을 설정하고 함수 호출 연산자(괄호)를 이용하여 메소드를 호출하며 무비 클립을 변수, 배열 원소, 객체 속성과 같은 것으로 저장할 수도 있다. 하지만 무비 클립을 만드는 방법은 객체를 만드는 방법과 다르다. 객체 리터럴을 이용하여 객체를 기술하는 것처럼 무비 클립을 코드를 통해 기술할 수는 없다. 또한 다음과 같이 무비 클립 생성자 함수를 이용하여 무비 클립을 만들 수도 없다.

```
myClip = new MovieClip();  // 좋은 생각이긴 하지만 이렇게 하면 안 된다.
```

대신 무비 클립은 저작도구를 이용하여 사용자가 직접 만들어야 한다. 일단 클립을 만들고 나면 duplicateMovieClip()이나 attachMovie()와 같은 명령어를 이용하여 새로운 독립적인 클립 복사본을 만들 수 있다.

무비 클립 심벌과 인스턴스

모든 객체 인스턴스가 하나 이상의 클래스에 기반을 두는 것과 마찬가지로 모든 무비 클립 인스턴스는 '심벌(symbol)' 또는 '정의(definition)'라고도 부르는 템플릿 무비 클립을 기반으로 한다. 무비 클립의 심벌은 클립의 내용과 구조에 대한 모델 역할을 한다. 특정한 클립 객체를 만들기 전에 반드시 무비 클립 심벌이 필요하다. 심벌을 이용하면 무비에서 렌더링할 클립을 수동 또는 프로그래밍을 통해 자동으로 만들 수 있다.

스테이지에서 렌더링되는 무비 클립은 '인스턴스(instance)'라고 부른다. 인스턴스는 액션스크립트를 이용하여 조작할 수 있는 개별 클립 객체를 의미하며, 심벌은 모든 실제 무비 클립의 인스턴스를 만들어내기 위한 틀에 해당한다. 무비 클립 심벌은 플래시 저작 도구를 이용하여 만든다. 비어있는 심벌을 새로 만들 때는 다음과 같은 과정을 거치게 된다.

1. Insert → New Symbol 선택. Symbol Properties 대화상자가 나타난다.
2. Name 필드에 심벌의 이름을 입력한다.
3. Movie Clip 라디오 버튼을 선택한다.
4. OK를 클릭한다.

일반적으로 다음 단계에서는 무비 클립에서 사용할 내용을 바탕으로 심벌의 캔버스와 타임라인을 채우게 된다. 일단 심벌을 만들고 나면 그 심벌은 라이브러리에 들어가며 나중에 실제 무비 클립 인스턴스를 만들 때 사용된다. 하지만 스테이지에 이미 존재하는 도형이나 객체의 그룹을 무비 클립 심벌로 변환할 수도 있다. 이 작업은 다음과 같이 처리하면 된다.

1. 원하는 도형과 객체를 선택한다.
2. Insert → Convert to Symbol을 선택한다.
3. Name 필드에 심벌 이름을 입력한다.
4. Movie Clip 라디오 버튼을 선택한다.
5. OK를 클릭한다.

새로운 무비 클립 심벌을 만들기 위해 사용한 도형과 객체는 새로운 클립의 익명 인스턴스로 치환된다. 이렇게 하면 새로 만든 무비 클립 심벌이 라이브러리에 포함되며 나중에 객체의 인스턴스를 만들 때 이 심벌을 사용할 수 있다.

인스턴스 만들기

무비 클립 심벌을 기반으로 새로운 인스턴스를 만드는 방법에는 세 가지가 있다. 이 중 두 가지 방법은 프로그래밍을 통한 방법이고 나머지 하나는 플래시 저작 도구에서 직접 수동으로 만드는 방법이다.

수동으로 만드는 방법

플래시 저작 도구에서 라이브러리를 이용하여 수동으로 무비 클립 인스턴스를 만들 수 있다. 라이브러리에서 스테이지로 무비 클립 심벌을 드래그하면 새로운 인스턴스가 생성된다. 이렇게 만든 인스턴스의 이름은 인스턴스 패널을 통해 직접 지정해야 한다(인스턴스 이름에 대해서는 잠시 후에 배우기로 하자). 플래시에서 무비 클립을 이용한 작업을 한 번도 해보지 않았다면 매크로미디어 플래시 도움말에서 '심벌 및 인스턴스 사용법'을 참조하기 바란다.

duplicateMovieClip()을 이용하여 만드는 방법

액션스크립트를 이용하여 이미 플래시 무비의 스테이지에 존재하는 인스턴스를 복사할 수 있다. 그리고 나면 그 독립적인 복사본은 전혀 다른 클립으로 간주할 수 있다. 수동으로 만든 클립이나 프로그래밍을 통해 만든 클립 모두 복사가 가능하다. 따라서 복사본의 복사본을 만드는 것도 가능하다.

일반적으로 duplicateMovieClip()을 이용하여 인스턴스를 만드는 데는 크게 두 가지 방법이 있다.

- 다음과 같은 문법을 이용하여 duplicateMovieClip()을 전역 함수로 호출할 수 있다.

```
duplicateMovieClip(target, newName, depth);
```

  이 때 target은 복사하고자 하는 인스턴스의 이름을 나타내는 문자열, newName 인자는 새로운 인스턴스의 이름을 지정해주는 문자열이며 depth는 프로그램을 통해 만든 클립 스택 중에서 새로운 인스턴스를 집어넣고자 하는 위치를 나타내는 정수이다.

- 이미 존재하는 인스턴스의 메소드로 duplicateMovieClip()을 호출하는 것도 가능하다.

```
myClip.duplicateMovieClip(newName, depth);
```

  여기서 myClip은 복사하려는 클립의 이름이며 newName과 depth는 앞에서 설명한 것과 같은 의미를 지닌다.

duplicateMovieClip()을 통해 만들어진 인스턴스는 씨앗 클립 바로 위에 놓이게 된다. 따라서 복사가 끝나고 나면 다음과 같이 복사한 클립을 새로운 위치로 옮겨야 한다.

```
ball.duplicateMovieClip("ball2", 0);
ball2._x += 100;
ball2._y += 50;
```

액션스크립트나 플래시 저작 도구를 통해 변환된 씨앗 클립(색 변경, 회전, 크기 조절 등을 거친 씨앗 클립)을 복사한 인스턴스는 씨앗 클립의 초기 변환 내용을 그대로 계승한다. 그 뒤로는 씨앗 클립을 변환시키더라도 복사된 인스턴스에는 영향을 미치지 않는다. 이와 마찬가지로 각 인스턴스는 각각 다르게 변환할 수 있다. 예를 들면 씨앗 클립을 45도 회전시킨 후에 인스턴스를 복사하면 인스턴스의 초기 회전 각도는 45도가 된다.

```
seed._rotation = 45;
seed.duplicateMovieClip("newClip", 0);
trace(newClip._rotation); // 45가 출력된다.
```

그리고 나서 인스턴스를 10도 회전시키면 그 인스턴스의 회전 각도는 55도가 되지만 씨앗 클립의 회전 각도는 여전히 45도로 남아 있다.

```
newClip._rotation += 10;
trace(newClip._rotation); // 55가 출력된다.
trace(seed._rotation);    // 45가 출력된다.
```

여러 개의 인스턴스를 한꺼번에 복사하고 나서 각 복사본을 조금씩 변환시키면 특이한 복합 변환도 만들어낼 수 있다(〔예제 10-2〕의 load 이벤트에서 이러한 기법을 사용한다).

액션스크립트에서 duplicateMovieClip()을 이용하여 클립을 복사하면 무비에서 클립을 일일이 직접 만드는 것에 비해 다음과 같은 장점이 있다.

- 프로그램의 실행 과정을 기준으로 스테이지에 클립이 나타나는 때를 정확하게 제어할 수 있다.
- 프로그램의 실행 과정을 기준으로 스테이지에서 클립이 없어지는 때를 정확하게 제어할 수 있다.

- 다른 클립 복사본에 대해 상대적인 클립 복사본 레이어 깊이를 할당할 수 있다 (무비의 레이어 스택을 변경할 수 없었던 플래시 4에서는 매우 중요한 문제이다).
- 클립의 이벤트 핸들러를 복사할 수 있다.

이러한 기능을 이용하면 무비 내용을 프로그램을 통해 더 세밀하게 제어할 수 있다. 예를 들어 우주선 게임을 만든다면 미사일 발사 버튼을 누를 때 미사일 무비 클립을 복사하여 만들면 된다. 그리고 나서 프로그램을 통해 미사일 클립을 이동시키고 무비에 있는 장애물 뒤로 넘어가면 적기와 부딪치고 나서 그 미사일을 없애버리면 된다. 수동으로 클립을 만들면 이러한 기능을 활용할 수 없다. 이 경우에는 타임라인을 바탕으로 클립이 생기는 시점과 없어지는 시점을 미리 정해야 하며, 플래시 4에서는 클립의 레이어를 변경하는 것도 불가능하다.

attachMovie()를 이용하여 만드는 방법

attachMovie() 메소드는 duplicateMovieClip() 메소드와 마찬가지로 새로운 무비 클립 인스턴스를 만드는 메소드이다. 하지만 이미 생성해 둔 인스턴스를 사용하지 않고 무비의 라이브러리에 있는 심벌로부터 직접 인스턴스를 만들어낸다는 점에서 duplicateMovieClip()과 다르다. attachMovie()를 이용하여 심벌의 인스턴스를 만들려면 먼저 라이브러리로부터 심벌을 꺼내야 한다.

1. Library에서 원하는 심벌을 선택한다.
2. Library의 Options 메뉴에서 Linkage를 선택한다. 그러면 Symbol Linkage Properties 대화상자가 나타난다.
3. Export This Symbol 라디오 버튼을 선택한다.
4. Identifier 필드에 클립 심벌의 이름을 입력한다. 아무 문자열이나 입력해도 되지만(종종 심벌 이름을 그대로 사용하기도 한다), 다른 클립 심벌의 이름과 겹치면 안 된다.
5. OK를 클릭한다.

클립 심벌을 꺼내고 나면 다음과 같은 구문을 이용하여 attachMovie()를 호출하면 원래 있던 클립에 해당 심벌의 새로운 인스턴스를 추가할 수 있다.

```
myClip.attachMovie(symbolIdentifier, newName, depth);
```

여기서 myClip은 새로운 인스턴스를 추가하려는 클립의 이름이다. myClip을 생략하면 현재 클립(attachMovie() 구문이 포함되어 있는 클립)에 새로운 인스턴스가 추가된다. symbolidentifier 인자는 라이브러리의 Linkage 옵션에서 Identifier 필드에 입력했던 심벌 이름으로, 인스턴스를 만들기 위해 사용할 심벌의 이름을 나타내는 문자열이다. newName은 새로 생성되는 인스턴스의 이름을 나타내는 문자열이고 depth는 호스트 클립의 레이어 스택 중에서 새로운 인스턴스가 들어갈 자리를 나타내기 위한 정수 값이다.

다른 클립에 인스턴스를 추가할 때 그 인스턴스는 클립의 레이어 스택 중에서 클립 한 가운데 위치하게 된다(클립 스택에 관한 것은 잠시 후에 배우게 될 것이다). 인스턴스를 문서의 메인 무비에 추가할 때는 좌표가 (0, 0)인 스테이지의 왼쪽 위 구석에 놓는다.

인스턴스 이름

인스턴스를 만들면 그 인스턴스의 이름을 할당해야 나중에 그 인스턴스를 참조할 수 있다. 일반적인 객체와의 차이점을 주의 깊게 살펴보자. 일반적인 데이터 객체(무비 클립 제외)를 만들 때는 다음과 같이 객체를 변수나 다른 데이터 컨테이너에 대입해야 그 객체를 계속 유지하고 그 이름을 이용하여 나중에 다시 참조할 수 있다.

```
new Object();                 // 객체가 생성되고 나서 바로 사라지기
                              // 때문에 나중에 다시 사용할 수 없다.
var thing = new Object();     // 객체 레퍼런스가 thing에 저장된다.
                              // 나중에 thing으로 참조할 수 있다.
```

무비 클립 인스턴스는 어떤 변수에 저장하지 않아도 참조할 수 있다. 일반적인 데이터 객체와는 달리 클립 인스턴스는 그 인스턴스를 만들기만 하면 인스턴스 이름을 통해 액션스크립트에서 액세스할 수 있다.

```
ball._ y = 200;
```

각 클립의 인스턴스 이름은 내장된 속성인 _name에 저장되며 나중에 그 값을 읽어오거나 바꿀 수도 있다.

```
ball._name = "circle";  // ball의 이름을 circle로 바꾼다.
```

인스턴스의 _name 속성을 바꾸면 그 뒤로는 바뀐 이름을 이용하여 인스턴스를 참조해야 한다. 예를 들어 위와 같은 코드를 실행하고 나면 ball 레퍼런스는 사라지므로 circle이라는 이름을 이용하여 그 인스턴스를 참조해야 한다.

인스턴스 이름을 정하는 방법은 그 인스턴스를 만드는 방법에 따라 달라진다. 프로그래밍을 통해 생성된 인스턴스는 그 인스턴스를 만드는 함수에서 이름을 결정한다. 인스턴스를 수동으로 만들 때는 일반적으로 다음과 같이 저작 도구에서 Instance 패널을 통해 인스턴스 이름을 직접 지정해야 한다.

1. 스테이지에서 인스턴스를 선택한다.
2. Modify → Instance를 선택한다.
3. Name 필드에 인스턴스 이름을 입력한다.

수동으로 만든 클립에 인스턴스 이름이 없다면 실행 중에 플래시 플레이어에서 자동으로 이름을 정해준다. 자동으로 주어진 인스턴스 이름은 instance1, instance2, instance3,...instancen과 같은 식으로 이름이 주어지지만 이러한 이름에서 클립의 내용을 유추하는 것은 불가능하다(또한 자동으로 생성된 이름을 어림짐작하여 맞춰야 한다).

인스턴스 이름은 인식자이므로 '14장. 렉시컬 구조' 에 나온 인식자를 만드는 규칙에 따라 이름을 정해야 한다. 가장 주의해야 할 점은 인스턴스 이름은 숫자로 시작하면 안 되고 하이픈이나 공백이 들어갈 수 없다는 점이다.

외부 무비 가져오기

하나의 문서에서 무비 클립 인스턴스를 만드는 법은 이제 모두 배웠지만, 플래시 플레이어에서는 동시에 여러 개의 .swf 문서를 화면에 표시할 수 있다. loadMovie()(전역 함수나 무비 클립 메소드 중 어느 것을 사용해도 상관없다)를 이용하면 외부의 .swf 파일을 플레이어로 가져올 수 있으며, 클립 인스턴스 또는 베이스 무비에서 몇 레벨 위를 지정하여(즉 베이스 무비보다 앞쪽) 그 위치에 놓을 수 있다. 여러 컨텐츠를 서로 다른 파일에서 관리하면 다운로드 과정을 정확하게 제어할 수

있다. 예를 들어 주 이동 메뉴와 다섯 개의 서브섹션이 있는 무비가 있다고 가정하자. 사용자가 다섯 번째 섹션으로 이동하려면 첫 번째부터 네 번째 섹션까지 모두 다운로드해야 한다. 하지만 각 섹션을 별도의 .swf 파일에 저장하면 순서에 상관없이 필요한 파일만 다운로드하면 되므로 사용자가 각 섹션을 직접 사용할 수 있다.

외부 .swf 파일을 어떤 레벨로 가져오면 그 파일의 메인 무비 타임라인이 그 레벨의 루트 타임라인이 되며 그 레벨에 전에 있던 모든 무비를 대신하게 된다. 이와 마찬가지로 외부 무비를 클립으로 가져오면 새로 가져온 무비의 메인 타임라인이 원래 있던 클립의 타임라인을 대신하여 원래 있던 클립의 그래픽, 사운드, 스크립트가 모두 없어지게 된다.

duplicateMovieClip()과 마찬가지로 loadMovie()도 독립적인 함수로 쓰일 수도 있고 인스턴스 메소드로 쓰일 수도 있다. loadMovie()를 독립적으로 사용하는 경우에는 다음과 같은 식으로 사용한다.

```
loadMovie(URL, location)
```

URL은 가져올 외부 .swf 파일의 주소이고, location 매개변수는 새로운 .swf 파일이 들어갈 (즉 새로 가져온 무비가 원래 있던 것을 대체할) 현존하는 클립이나 문서 레벨의 경로를 나타내는 문자열이다. 예를 들면 다음과 같다.

```
loadMovie("circle.swf", "_level1");
loadMovie("photos.swf", "viewClip");
```

무비 클립 레퍼런스는 문자열로 사용하면 경로로 변환되므로 location 자리에도 "_level1" 대신 _level1과 같이 무비 클립 레퍼런스를 직접 사용해도 된다. 하지만 레퍼런스를 사용할 때는 주의를 기울여야 한다. 주어진 문자열이 유효하나 클립을 가리키지 않는다면, loadMovie() 함수에서 예상하지 못한 행동(현재 타임라인으로 외부 .swf 파일을 가져온다)을 할 수 있기 때문이다. 자세한 내용은 '3부. 레퍼런스' 또는 이 장의 뒷부분에 있는 '메소드와 전역 함수가 겹치는 경우'를 참조하기 바란다.

loadMovie()을 클립 메소드로 사용하는 경우에는 다음과 같은 문법을 사용한다.

```
myClip.loadMovie(URL);
```

loadMovie()를 무비 클립의 메소드로 사용하면 외부 .swf 파일을 myClip으로 가져오는 것으로 가정하기 때문에, 독립적인 loadMovie() 함수와는 달리 location

매개변수를 사용하지 않아도 된다. 따라서 URL 매개변수를 이용하여 가져올 .swf 파일의 경로만 제공하면 된다. 물론 URL에는 로컬 파일 이름을 사용해도 된다.

```
viewClip.loadMovie("photos.swf");
```

무비를 로딩하여 클립 인스턴스에 집어넣으면 로딩된 무비는 그 클립의 속성(클립의 크기, 회전 각도, 색 변환 등)을 그대로 받아들인다.

loadMovie()를 메소드 형태로 사용하려면 myClip에 들어갈 클립이 존재해야 한다. 예를 들어 다음과 같이 circle.swf를 불러오려고 한다면 _level1이 비어있을 때는 오류가 발생한다.

```
_level1.loadMovie("circle.swf");
```

무비 가져오기 실행 순서

loadMovie() 함수는 선언문 블록에 그 함수를 호출하는 선언문이 나올 때 바로 실행되는 것이 아니다. 이 함수는 그 블록에 있는 모든 선언문이 실행된 다음에야 실행된다.

어떤 무비를 플레이어로 가져오는 loadMovie()를 호출하는 블록 안에서는 새로 가져온 외부 무비의 속성이나 메소드를 사용할 수 없다.

loadMovie()에서는 외부 파일(보통 네트워크를 통한다)을 가져오므로 그 실행 과정이 비동기식이다. 즉 loadMovie()는 파일 전송 속도에 따라 언제 완료될지 모른다. 따라서 외부에서 가져온 무비를 액세스하기 전에 그 무비를 플레이어로 가져오는 과정이 완료되었는지 확인해야 한다. 이러한 작업은 보통 프리로더(어떤 작업을 하기 전에 파일이 얼마나 전송되었는지 확인하는 코드)라는 것을 이용하여 처리한다. 프리로더는 _totalframes와 _framesloaded 무비 클립 속성과 getBytesLoaded(), getBytesTotal() 무비 클립 메소드를 이용하여 만든다. 코드 샘플은 3부에서 찾을 수 있다. data 클립 이벤트를 이용하여 프리로더를 만드는 방법은 〔예제 10-4〕에서 볼 수 있다.

loadMovie()와 attachMovie() 함께 사용하기

loadMovie()를 이용하여 클립 인스턴스에 외부 .swf 파일을 가져오면 attachMovie()를 이용하여 그 클립에 새로운 인스턴스를 추가할 수 없게 된다. 어떤 클립에 외부 .swf 파일을 가져오면 그 클립에는 그 클립을 가져온 라이브러리에서 무비를 가져올 수 없다. 예를 들어 movie1.swf에 clipA라는 인스턴스가 들어있고 clipA에 movie2.swf를 추가하면 movie1.swf의 라이브러리에 있는 인스턴스를 clipA에 더 이상 부착할 수 없게 된다.

왜 이렇게 될까? 그 이유는 attachMovie() 메소드는 하나의 문서에 대해서만 작동하기 때문이다. 즉 한 문서의 라이브러리에서 다른 문서로 인스턴스를 부착할 수 없다. .swf 파일을 클립으로 가져오면 그 클립에 새로운 문서를 추가하는 것이므로 새로운(다른) 라이브러리도 추가된다. 그 다음에 원래 문서에서 클립에 인스턴스를 추가하려고 하면 클립의 라이브러리가 원본 문서의 라이브러리와 다르기 때문에 작업을 처리할 수 없다. 하지만 unloadMovie()를 이용하여 클립에 있는 문서를 제거하면 그 클립의 문서 라이브러리에서 무비를 클립에 추가할 수 있다.

마찬가지로 loadMovie()를 이용하여 .swf 파일을 클립으로 불러오면 duplicateMovieClip()을 이용하여 그 클립을 복사할 수 없다.

무비와 인스턴스 스택 순서

플레이어에서 화면에 표시되는 모든 무비 클립 인스턴스와 외부에서 가져온 무비는 카드를 쌓아놓는 것처럼 시각적으로 볼 수 있는 스택 순서를 따르게 된다. 플레이어에서 인스턴스나 외부에서 가져온 .swf 파일이 겹치게 되면, 한 클립(둘 중 높은 것)이 다른 클립(둘 중 낮은 것)을 덮게 된다. 이 원리는 상당히 간단하지만 모든 인스턴스와 모든 .swf 파일을 담고 있는 메인 스택은 사실 여러 개의 작은 서브스택으로 나누어진다. 우선 이러한 서브스택을 각각 살펴본 후에 그러한 서브스택이 합쳐져 메인 스택을 이루는 방법을 알아보자(여기서 말하는 스택은 '11장. 배열'에 나온 LIFO 스택 또는 FIFO 스택과는 전혀 상관없다).

내부 레이어 스택

플래시 저작 도구에서 만들어진 인스턴스는 내부 레이어 스택이라고 부르는 스택에 들어간다. 이 스택의 순서는 무비의 타임라인에 있는 실제 레이어에 의해 결정된다. 서로 다른 타임라인 레이어에 있는 두 개의 수동으로 생성되는 인스턴스가 서로 겹치면 위쪽에 있는 레이어가 아래쪽에 있는 레이어를 덮어서 가리게 된다.

또한 하나의 타임라인 레이어에 여러 개의 클립이 있을 수도 있기 때문에 내부 레이어 스택에 있는 각 레이어는 그 레이어만의 미니스택을 가지게 된다. 타임라인의 같은 레이어에 있는 서로 겹치는 클립의 순서는 저작 도구에서 Modify → Arrange 명령어를 이용하여 지정할 수 있다.

플래시 5부터는 두 인스턴스가 같은 타임라인에 있다면(즉 두 클립의 _parent 속성이 같다면), swapDepths() 메소드를 이용하여 내부 레이어 스택에 있는 두 개의 인스턴스의 위치를 서로 바꿀 수 있다. 플래시 5 이전에는 내부 레이어 스택을 액션스크립트로 변경할 수 있는 방법이 없었다.

프로그래밍을 통해 생성된 클립 스택

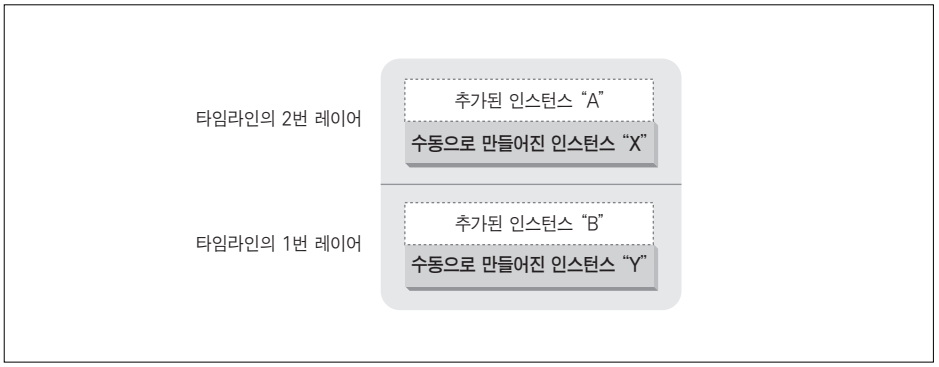
프로그래밍을 통해 생성된 인스턴스는 내부 레이어 스택에 들어가는 수동으로 생성된 인스턴스와는 별도의 스택에 들어간다. 각 인스턴스에는 별도의 프로그래밍을 통해 생성된 클립 스택이 있어서 duplicateMovieClip()과 attachMovie()를 이용하여 생성된 클립은 그 클립 스택으로 들어간다. 이러한 클립의 스택 순서는 그 클립을 만든 방법에 따라 달라진다.

attachMovie()를 이용하여 생성된 클립이 스택에 추가되는 과정

attachMovie()를 이용하여 만든 새로운 인스턴스는 스택에서 그 인스턴스가 부착된 클립의 위로 올라간다(즉 원래 스택보다 위로 올라간다). 예를 들어 무비의 내부 레이어 구조에 X와 Y라는 두 개의 클립이 있고 X는 Y보다 위에 있는 레이어에 속한다고 가정해 보자. 또한 새로운 클립 A를 X에, B를 Y에 추가한다고 가정해 보자.

```
x.attachMovie("A", "A", 0);
y.attachMovie("B", "B", 0);
```

이와 같은 상황이라면 [그림 13-1]에서 볼 수 있듯이 클립이 스택에 들어가는 순서는 A, X, B, Y가 된다.

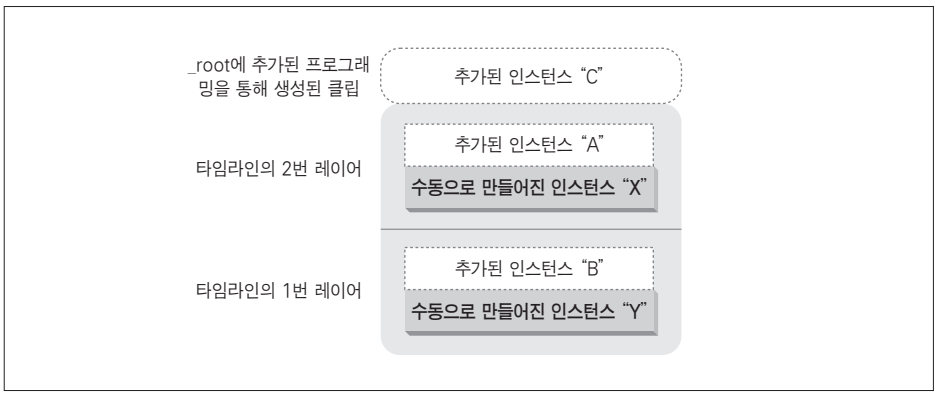


[그림 13-1] 인스턴스 스택의 예

어떤 클립이 생성되고 나면 그 클립에서도 프로그래밍을 통해 만들어진 클립을 더 받아들일 수 있도록 별도의 공간을 제공한다. 즉 추가된 클립에도 다른 클립을 추가할 수 있다.

플래시 문서의 _root 무비에 부착된 클립은 _root 무비의 프로그래밍을 통해 생성된 클립 스택에 들어가며, 그 스택에 다른 프로그래밍을 통해 생성된 내용이 있다고 하더라도 _root 무비에 있는 다른 모든 클립보다 위에 나타나게 된다.

앞에서 사용한 예를 확장해 보자. X, Y, A, B 클립이 들어있는 무비의 _root에 C라는 클립을 추가하면 C 클립은 다른 모든 클립보다 위로 올라간다.



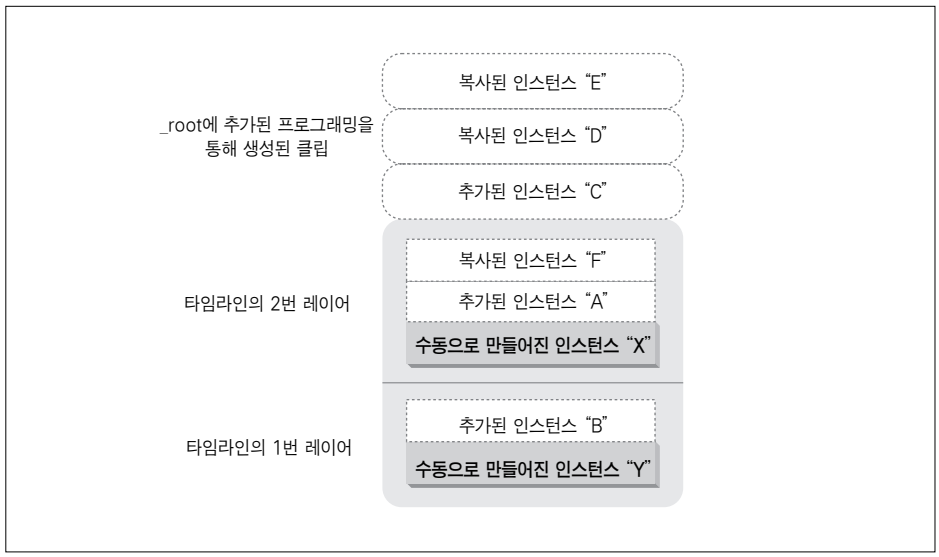
[그림 13-2] _root에 추가된 클립을 보여주는 인스턴스 스택

duplicateMovieClip()을 이용하여 생성된 클립이 스택에 추가되는 과정

duplicateMovieClip()을 이용하여 생성된 각 인스턴스는 인스턴스의 씨앗 클립이 생성된 방법에 따라 프로그램 스택에 대입된다.

- 인스턴스의 씨앗 클립이 수동으로 만들어졌다면(또는 수동으로 만들어진 클립을 duplicateMovieClip()으로 복사한 것이라면) 새로운 인스턴스는 _root 위의 스택으로 들어간다.
- 인스턴스의 씨앗 클립이 attachMovie()를 이용하여 만든 것이라면 새로운 인스턴스는 씨앗 클립의 스택에 들어간다.

앞에서 사용한 예제를 이용하여 어떤 식으로 작동하는지 확인해보자. 클립 X(수동으로 만들어진 클립)를 복사하여 클립 D를 만들면, 클립 D는 클립 C가 들어있는 _root 위의 스택으로 들어간다. 마찬가지로 클립 D(수동으로 만들어진 클립 X에서 나온 클립)를 복사한 클립 E를 만들면, 클립 E도 C, D와 마찬가지로 _root 위의 스택으로 들어간다. 하지만 클립 A(attachMovie()로 만들어진 클립)를 복사하여 클립 F를 만들면, F는 클립 A와 마찬가지로 X 위의 스택으로 들어간다. [그림 13-3]을 보면 좀더 쉽게 이해할 수 있을 것이다.



[그림 13-3] 여러 복사된 클립을 보여주는 인스턴스 스택

프로그래밍을 통해 생성된 클립 스택에서 인스턴스 깊이를 지정하는 방법

〔그림 13-3〕에서 클립 C, D, E 또는 클립 A, F의 스택 순서가 어떻게 결정되는지 궁금할 것이다. 프로그래밍을 통해 생성된 클립의 스택 순서는 attachMovie( ) 또는 duplicateMovieClip( ) 함수에 전달되는 depth 인자에 의해 결정되며, swapDepths( ) 함수를 이용하여 언제든지 바꿀 수 있다. 모든 프로그래밍을 통해 생성된 클립의 depth(z-인덱스라고도 부른다)는 프로그래밍을 통해 생성된 클립의 특정 스택에서의 위치를 결정한다.

클립의 depth는 임의의 정수로 지정할 수 있으며, 아래에서 위로 올라가게 되어 있다. 즉 -1은 0보다 낮고 1은 0보다 높으며(즉 더 위에 있다), 2는 그보다 높은 식으로 올라가게 되어 있다. 두 개의 프로그래밍을 통해 생성된 클립이 스크린에서 같은 위치에 있다면, depth 값이 더 큰 클립이 더 작은 클립보다 위에 렌더링된다.

레이어에는 한 번에 하나의 클립밖에 들어갈 수 없다. 따라서 이미 다른 클립이 들어있는 레이어에 어떤 클립을 놓으면 원래 있던 클립이 지워지고 새 클립이 자리를 차지하게 된다.

클립의 깊이에는 간격이 있어도 상관없다. 세 개의 클립을 하나는 깊이 0으로, 다른 하나는 깊이 500으로, 또 다른 하나는 깊이 1000으로 지정해도 괜찮다. 깊이를 지정할 때 depth 값들이 서로 차이가 난다고 해서 속도가 저하되거나 메모리를 많이 소모하게 되는 것은 아니다.

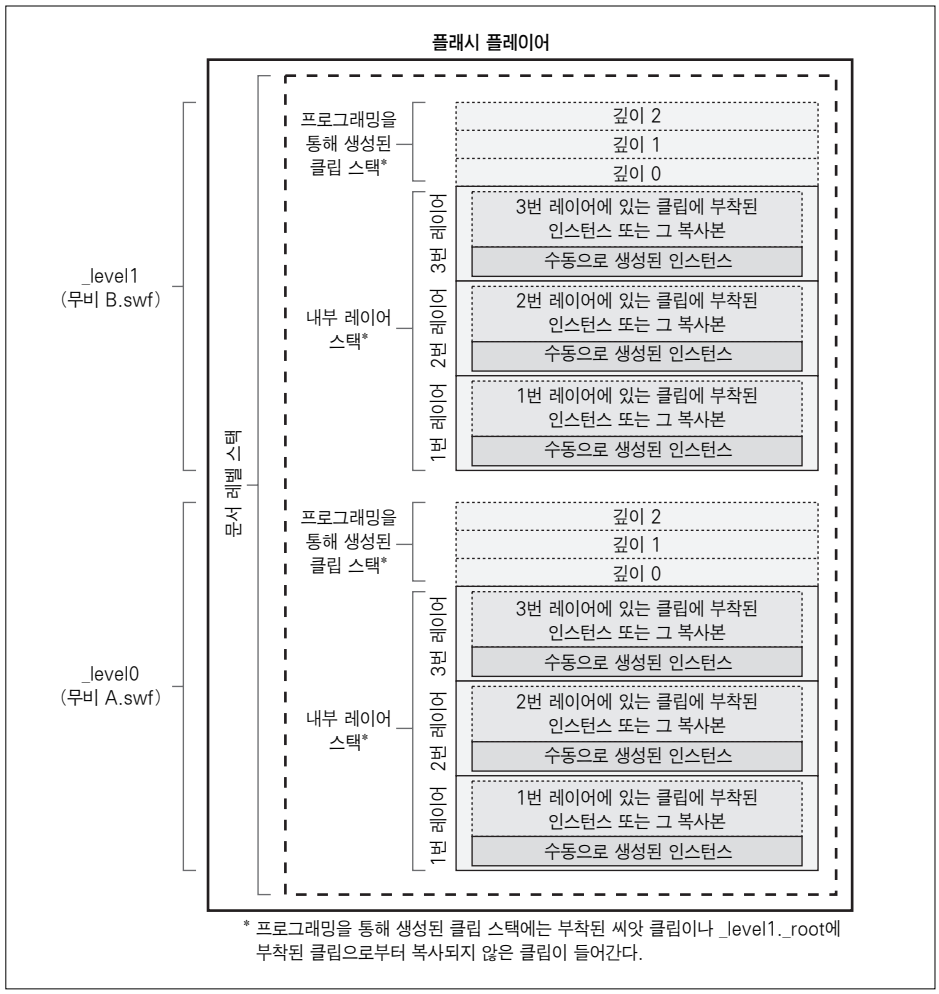
.swf 문서 "_level" 스택

내부 레이어 스택과 프로그래밍을 통해 생성된 클립 스택 외에 인스턴스가 아닌 loadMovie( )를 통해 플레이어에 로딩된 전체 .swf 파일이 겹치는 순서를 결정하는 문서 스택(또는 레벨 스택)이 있다.

플래시 플레이어에 로딩된 첫 번째 .swf 파일은 문서 스택의 가장 낮은 레벨(전역 속성인 _level0으로 표시함)로 들어간다. 다른 .swf 파일을 플레이어에서 로딩할 때 문서 스택에서 _level0 위의 레벨에 할당하면 원본 문서 앞에 놓을 수 있다. 레벨 스택에서 더 높은 레벨에 있는 문서에 있는 모든 내용은 각 문서 안의 무비 클립 스택 순서와는 상관없이 더 낮은 레벨에 있는 문서보다 위에 놓인다.

프로그래밍을 통해 생성된 클립 스택에서 한 레이어에 하나의 클립만을 넣을 수 있는 것과 마찬가지로 문서 스택에서도 한 레벨에 하나의 문서만을 넣을 수 있다. .swf 파일을 다른 문서가 들어 있는 레벨에 집어넣으면 그 레벨에 원래 들어있던 문서는 없어지고 새로운 문서가 그 자리를 차지한다. 예를 들어 _level0에 새로운 .swf 파일을 로딩하면 원본 문서를 대체할 수 있다. _level1에 새로운 .swf 파일을 로딩하면 _level0에 있는 문서가 보이지 않게 되긴 하지만, 그 문서가 플레이어에서 제거되는 것은 아니다.

〔그림 13-4〕에 플래시 플레이어에서 관리하는 다양한 스택 사이의 관계를 요약해 놓았다.



〔그림 13-4〕 플래시 플레이어 무비 스택

스택과 실행 순서

무비 클립에 레이어를 지정하는 것과 타임라인 레이어는 코드 실행 순서에 영향을 준다. 이 때 적용되는 규칙은 다음과 같다.

- 서로 다른 타임라인 레이어에 있는 프레임에 속한 코드는 언제나 위에서 아래의 순서로 실행된다.
- 수동으로 만든 인스턴스가 처음 로딩되면 그 타임라인에 있는 코드와 load 이벤트 핸들러가 플래시 문서의 Publish Settings에 있는 Load Order 설정에 따라 실행된다. 이 설정에서는 아래에서 위로(Bottom Up) 향하는 순서가 기본이고, 이 외에 위에서 아래로 가는 순서(Top Down)를 선택할 수도 있다.

  예를 들어 top과 bottom(레이어 스택에서 top은 bottom보다 위에 있다)이라는 두 개의 레이어가 있는 타임라인을 가정해 보자. 클립 X는 top 레이어에, 클립 Y는 bottom 레이어에 넣는다. 문서의 Load Order가 Bottom Up으로 설정되어 있으면 클립 Y의 코드가 클립 X의 코드보다 먼저 실행된다. 하지만 Load Order가 Top Down으로 되어 있으면 클립 X에 있는 코드가 클립 Y에 있는 코드보다 먼저 실행된다. 이러한 실행 순서는 X와 Y가 처음으로 등장하는 프레임에서만 적용된다.
- 일단 로딩된 후에는 무비의 모든 인스턴스가 실행 순서에 추가된다. 이 실행 순서는 로딩 순서와 정반대이다. 즉 무비에 마지막으로 추가된 인스턴스에 있는 코드가 언제나 가장 먼저 실행된다.

이러한 규칙을 활용할 때는 항상 주의를 기울여야 한다. 레이어는 변경되기 쉬우므로 레이어 사이의 상대적인 위치에 의존하는 코드를 만드는 것은 피해야 한다. 스택에서의 클립 실행 순서에 의존하지 않고 안전하게 실행할 수 있는 코드를 만들 수 있도록 노력해야 한다. 이와 같은 실행 스택에 의한 문제는 모든 코드를 코드가 들어있는 각 타임라인의 맨 위에 있는 scripts 레이어에 집어넣으면 어느 정도 해결할 수 있다.

인스턴스 및 메인 무비 참조

앞에서 플래시 플레이어에 있는 무비 클립 인스턴스와 외부 .swf 파일을 만들고 레이어를 정하는 방법을 배웠다. 액션스크립트에서 효과적으로 인스턴스나 외부 .swf 파일을 제어하려면 그러한 내용물을 참조할 수 있어야 한다.

인스턴스와 메인 무비는 다음과 같은 네 가지 일반 상황에서 참조하게 된다.

- 클립이나 무비의 속성을 알아내거나 설정할 때
- 클립이나 무비의 메소드를 만들거나 호출할 때
- 클립이나 무비에 어떤 함수를 적용할 때
- 변수에 저장하거나 함수를 호출할 때 인자로 전달하는 것과 같이 클립이나 무비를 데이터로 조작할 때

클립 인스턴스나 무비 클립을 참조하는 상황은 꽤 간단한 편이지만 실제로 이러한 대상을 참조하는 방법은 매우 다양하다. 이 절의 나머지 부분에서는 액션스크립트에서 인스턴스나 무비를 참조하는 도구에 대해 살펴보자.

인스턴스 이름을 이용하는 법

앞에서 인스턴스 이름을 이용하여 무비 클립을 참조하는 방법을 배웠다. 예를 들면 다음과 같다.

```
trace(myVariable);  // 변수를 참조한다.
trace(myClip);      // 무비 클립을 참조한다.
```

위 예제와 같이 인스턴스를 직접 참조하려면 인스턴스가 코드가 포함된 타임라인 안에 있어야 한다. 예를 들어 문서의 메인 타임라인에 clouds라는 인스턴스가 있다면 메인 타임라인에 있는 코드에서 clouds를 참조할 때는 다음과 같이 하면 된다.

```
// 인스턴스의 속성을 설정한다.
clouds._alpha = 60;
// 인스턴스의 메소드를 호출한다.
clouds.play();
// 인스턴스를 다른 연관된 인스턴스의 배열에 집어넣는다.
var background = [clouds, sky, mountains];
```

참조하려는 인스턴스가 코드와 같은 타임라인에 없다면 '중복 인스턴스 참조하기'에 나오는 더 복잡한 문법을 사용해야 한다.

현재 인스턴스 또는 무비 참조하기

클립을 참조할 때 반드시 인스턴스의 이름을 이용해야 하는 것은 아니다. 인스턴스의 타임라인에 있는 어떤 프레임에 들어있는 코드에서는 그 인스턴스의 속성과 메소드를 인스턴스 이름 없이도 직접 참조할 수 있다.

예를 들어 cloud라는 클립의 _alpha 속성을 설정하려면, cloud 타임라인에 있는 어떤 프레임에 다음과 같은 코드를 추가하면 된다.

```
_alpha = 60;
```

마찬가지로 cloud 타임라인에 있는 프레임에서 cloud의 play() 메소드를 호출하고 싶다면 다음과 같이 간단하게 처리할 수 있다.

```
play();
```

이러한 방법은 메인 무비의 타임라인을 포함하여 모든 타임라인에서 사용할 수 있다. 예를 들어 아래 코드를 생각해 보면 만약 두 코드가 모두 플래시 문서의 메임 타임라인에 있는 프레임에 들어있다면, 그 두 선언문은 똑같은 의미를 가지게 된다. 첫 번째 코드에서는 메인 무비를 간접적으로 참조하고 두 번째 코드에서는 _root 전역 속성을 통해 메인 무비를 직접 참조한다.

```
gotoAndStop(20);
_root.gotoAndStop(20);
```

'10장. 이벤트와 이벤트 핸들러'에서 배웠듯이 인스턴스의 이벤트 핸들러에서도 타임라인 코드와 마찬가지로 속성과 메소드를 직접 참조할 수 있다. 예를 들어 cloud에 다음과 같은 이벤트 핸들러를 추가할 수 있다. 이 핸들러에서는 cloud 인스턴스를 직접 참조하지 않고도 그 인스턴스에 있는 속성을 설정하고 메소드를 호출할 수 있다.

```
onClipEvent (load) {
  _alpha = 60;
  stop();
}
```

하지만 모든 메소드가 무비 클립에 대한 간접적인 레퍼런스를 통해 쓰일 수 있는 것은 아니다. 전역 함수와 같은 이름을 가지는 무비 클립 메소드(duplicateMovieClip()이나 unloadMovie()와 같은 함수)를 호출할 때는 직접 인스턴스를 참조해야 한다. 따라서 약간이라도 의심스럽다면 직접 참조하는 것이 좋다. 메소드와 전역 함수간의 문제에 대해서는 '메소드와 전역 함수가 겹치는 문제'에서 더 자세히 알아보기로 하자.

this 키워드를 이용한 자가 참조

어떤 타임라인에 있는 프레임이나 이벤트 핸들러와 같은 곳에서 현재 인스턴스를 직접 참조하려는 경우에는 this 키워드를 사용한다. 예를 들어 아래 두 선언문은 cloud 인스턴스의 타임라인에 있는 프레임에 들어가면 똑같은 의미를 지니게 된다.

```
_alpha = 60;       // 현재 타임라인에 대한 간접적인 참조
this._alpha = 60;  // 현재 타임라인에 대한 직접적인 참조
```

클립을 바로 참조할 수 있는 상황에서도 this 키워드를 이용하여 클립을 참조하는 데는 두 가지 이유가 있다. 직접적인 인스턴스 레퍼런스 없이 사용하면 인터프리터에서 몇 가지 무비 클립 메소드를 전역 변수로 잘못 해석할 수 있다. this 레퍼런스를 생략하면 인터프리터에서는 같은 이름을 가지는 전역 함수를 호출하는 것으로 착각하고 그 함수에 대상 무비 클립 인자가 없다고 오류 메시지를 내보낸다. 이러한 문제를 해결하려면 다음과 같이 this 키워드를 사용해야 한다.

```
this.duplicateMovieClip("newClouds", 0);  // 인스턴스에 있는 메소드를
                                          // 호출한다.

// this 레퍼런스를 빠뜨리면 오류가 생긴다.
duplicateMovieClip("newClouds", 0);  // 실수!
```

this를 이용하면 무비 클립에 대해 동작하는 함수에 현재 타임라인에 대한 레퍼런스를 전달하는 것도 간단하게 처리할 수 있다.

```
// 클립을 조작하는 함수
function moveTo (theClip, x, y) {
  theClip._x = x;
  theClip._y = y;
}
```

```
// 현재 타임라인에 대해 moveTo() 함수를 호출한다.
moveTo(this, 150, 125);
```

객체 지향 프로그래밍을 많이 사용한다면 인스턴스와 무비를 참조할 때 this 키워드를 사용하는 경우 조심해야 한다. 사용자 정의 메소드나 객체 생성자에서 쓰이는 this는 지금 배운 것과 달라서 현재 타임라인에 대한 레퍼런스가 아니다. 자세한 내용은 12장을 참조하기 바란다.

다중 인스턴스 참조

이 장의 도입부에서 배웠듯이 무비 클립 인스턴스는 다른 인스턴스 안에 들어갈 수도 있다. 즉 클립의 캔버스에 다른 클립의 인스턴스가 들어가고, 그 안에도 또 다른 클립이 들어갈 수 있다. 예를 들어 우주선 게임에서 spaceship(우주선) 클립에는 blinkingLights(깜빡거리는 불) 클립이나 burningFuel(연료 연소) 클립의 인스턴스가 들어갈 수 있다. 또는 어떤 캐릭터의 face(얼굴) 클립에 eyes(눈), nose(코), mouth(입) 클립이 따로 들어갈 수도 있다.

앞에서 하드 드라이브에 있는 하위 디렉토리를 위아래로 움직이는 것처럼 클립 인스턴스의 계층에서 위아래로 움직이는 방법을 간략하게 살펴보았다. 이 부분을 조금 더 자세히 알아보고 몇 가지 예를 더 생각해 보자.

우선 현재 인스턴스 안에 들어있는 클립 인스턴스를 참조하는 방법을 알아보자. 어떤 클립이 다른 클립의 타임라인에 있으면 그 클립의 속성이 되므로, 다른 객체 속성을 액세스하는 방법(점 연산자 이용)을 이용하여 액세스하면 된다. 예를 들어 clipA의 캔버스에 clipB가 들어있다고 가정하자. clipA의 타임라인에서 clipB를 액세스하려면 clipB를 직접 참조하면 된다.

```
clipB._x = 30;
```

이제 clipB에 clipC라는 다른 인스턴스가 들어있는 경우를 생각해 보자. clipA의 타임라인에서 clipC를 참조하려면 다음과 같이 clipC를 clipB의 속성으로 이용해야 한다.

```
clipB.clipC.play();
clipB.clipC._x = 20;
```

멋지지 않은가? 이러한 식으로 무한히 많은 인스턴스를 겹쳐 놓을 수 있다. 다른 클립의 타임라인에 들어있는 모든 클립 인스턴스는 호스트 클립의 속성이 되기 때문에 각 인스턴스를 점 연산자로 구분하면 계층을 쭉 따라 내려갈 수 있다.

```
clipA.clipB.clipC.clipD.gotoAndStop(5);
```

인스턴스 계층을 따라 내려가는 것을 알아보았으므로 이제 현재 인스턴스를 포함하고 있는 무비나 인스턴스를 참조하는 법을 알아보자. 앞에서 본 것처럼 모든 인스턴스에는 _parent라는 내장 속성이 있어서 그 인스턴스 바로 위의 클립이나 메인 무비를 참조할 수 있다. _parent 속성은 다음과 같이 사용한다.

```
myClip._parent
```

메인 타임라인에 clipA가 있고 clipA에는 clipB가, clipB에는 clipC가 들어있는 예를 이용하여 _parent와 점 연산자로 클립 계층에 들어있는 다양한 클립을 참조하는 법을 알아보자. 아래 코드가 clipB의 타임라인에 있는 프레임에 포함된다고 가정하자.

```
_parent          // clipA의 레퍼런스
this             // clipB(현재 클립)의 레퍼런스
this._parent     // clipA의 레퍼런스

// 조금 더 복잡한 것도 해보자.
_parent._parent  // clipA의 부모에 대한 레퍼런스
                 // 이 경우에는 메인 타임라인을 참조한다.
```

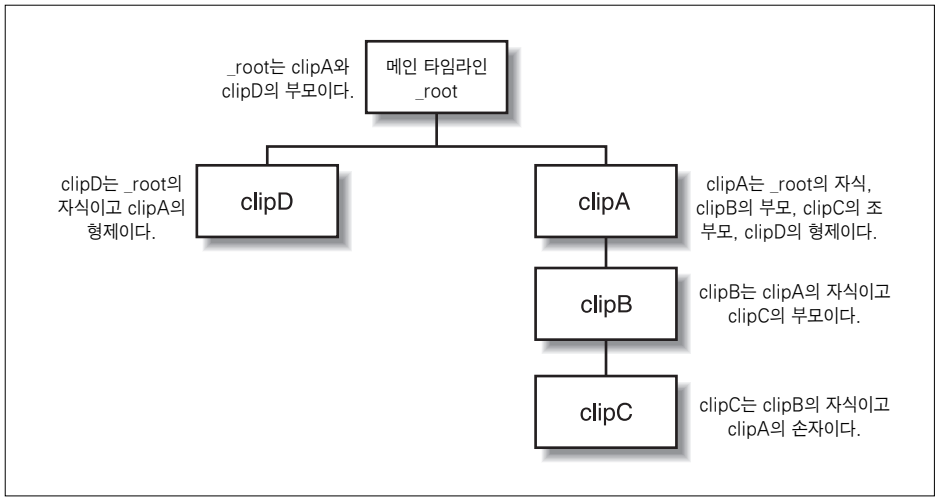
clipB보다 위쪽에 있는 클립을 참조할 때 clipC로 내려갔다가 다시 _parent 속성을 이용하여 올라가는 것도 틀린 표현은 아니지만 굳이 그렇게 빙빙 돌아갈 필요는 없다. 즉 아래 예와 같은 표현은 굳이 사용할 필요가 없지만, 이러한 참조 방법의 유연성을 보여주기 위해 여기에 적어 보았다.

```
clipC._parent    // clipB(현재 타임라인)를 돌려서 참조한 것
clipC._parent._parent._parent  // 메인 타임라인을 돌려서 참조한 것
```

클립 계층을 내려갈 때 점 연산자를 이용하는 방법, 클립 계층을 따라 올라갈 때 _parent 속성을 이용하는 방법을 확실히 익혀두자. 이러한 표현이 생소하다면 플래시에서 clipA, clipB, clipC와 같은 계층을 직접 만들어 보고, 위 예에 나온 코드를

활용해 보는 것도 좋다. 인스턴스를 적절히 참조할 줄 아는 것은 좋은 액션스크립트 프로그래머가 되기 위한 기본 요건 중 하나이다.

클립 계층은 가계도와 비슷하다. 일반적인 가계도의 경우에는 각 자손에 대해 아버지와 어머니가 있지만 클립 계층도에는 아버지와 어머니가 따로 구분되지 않는다(즉 성이 하나밖에 없는 가계도라고 생각하면 된다). 각 가구에는 부모가 하나씩 있고 그 부모 밑에는 여러 자식이 있을 수 있다. 모든 클립(트리의 노드)에는 부모(그 클립을 포함하는 클립)가 하나밖에 없지만 하나의 클립에는 여러 자식(그 클립에 포함된 클립)이 있을 수 있다. 물론 그 부모 클립에도 부모 클립은 하나뿐이므로 모든 클립에서 조부모는 하나뿐이다(사람처럼 조부모가 네 명이 되지 않는다). 〔그림 13-5〕를 참조하기 바란다.



〔그림 13-5〕 샘플 클립 계층

따라서 가계도에서 아무리 많이 내려가더라도 내려간 만큼 다시 올라가면 시작한 위치로 다시 돌아가게 된다. 따라서 쓸데없이 계층을 따라 내려갔다가 다시 올라올 필요가 없다. 하지만 계층을 따라 올라갔다가 다른 경로를 따라 내려가는 것은 유용할 수도 있다. 예를 들어 메인 타임라인에 clipD와 clipA가 있다고 가정하면, 둘 다 메인 타임라인에 속하므로 메인 타임라인이 _parent가 되어 두 클립은 서로 형제 관계를 이룬다. 이러한 경우에는 clipB에 있는 코드에서 다음과 같은 식으로 clipD를 참조하는 것이 가능하다.

```
_parent._parent.clipD    // 메인 타임라인(clipA의 _parent)의 자식이므로
                         // clipA와는 형제 관계를 이룬다.
```

메인 타임라인에는 _parent 속성이 없다(메인 무비는 모든 클립 계층에서 가장 위에 있으므로 다른 어떤 타임라인에도 속하지 않는다). 따라서 _root._parent를 참조하면 undefined가 리턴된다.

_root와 _leveln을 이용하여 메인 무비 참조하기

현재 클립을 기준으로 클립 계층의 위아래로 움직이는 방법을 배웠으므로 이제 절대 경로를 통해 움직이는 방법, 그리고 플레이어의 문서 스택에서 다른 레벨에 있는 다른 문서 사이를 움직이는 방법에 대해 알아보기로 하자. 이러한 기법을 변수와 함수에 적용시키는 방법은 예전에 배웠다. 여기서는 이 기법을 이용하여 무비 클립을 제어하는 것에 대해 알아보기로 하자.

_root를 이용하여 현재 레벨의 메인 무비 참조하기

어떤 인스턴스가 클립 계층에서 여러 단계 아래로 내려가 있다면 _parent 속성을 여러 번 사용하면 메인 무비 타임라인까지 거슬러 올라갈 수 있다. 하지만 액션스크립트에 내장된 전역 속성인 _root(메인 무비 타임라인에 대한 단축 레퍼런스)를 이용하면 복잡하게 _parent 속성을 여러 번 사용하지 않아도 된다. 예를 들어 다음과 같은 코드를 사용하면 메인 무비를 재생할 수 있다.

```
_root.play();
```

_root 속성은 클립 계층의 어떤 위치에 대한 절대 레퍼런스라고 부른다. 현재 클립을 기준으로 하는 _parent나 this 속성과는 달리 _root 속성은 어떤 클립에서 참조하더라도 같은 위치를 가리키기 때문이다. 아래 세 가지 표현은 모두 똑같은 클립을 가리킨다.

```
_parent._root
this._root
_root
```

따라서 현재 클립이 클립 계층에서 어느 위치에 있는지 모를 때는 _root를 이용해야 한다. 예를 들어 다음과 같이 circle이 메인 무비 타임라인의 자시이고 square가 circle의 자식인 경우를 생각해 보자.

```
메인 타임라인
  circle
    square
```

이제 다음과 같은 스크립트가 circle과 square에 모두 들어있다고 가정하자.

```
_parent._x += 10  // 이 클립의 부모 클립을 오른쪽으로 10픽셀 이동시킨다.
```

위 코드를 circle에서 실행시키면 메인 무비가 오른쪽으로 10픽셀 움직인다. 또한 이 코드를 square에서 실행시키면 메인 무비가 아닌 circle이 오른쪽으로 10픽셀 움직인다. 어떤 위치에서 실행하더라도 메인 무비를 오른쪽으로 10픽셀 움직이게 하는 스크립트를 만들고 싶다면 다음과 같이 하면 된다.

```
_root._x += 10   // 메인 무비를 오른쪽으로 10 픽셀 이동시킨다.
```

게다가 _parent 속성은 메인 타임라인에서는 사용할 수 없다. 하지만 위와 같이 _root를 이용하여 만든 스크립트는 메인 타임라인에 있는 프레임에서 실행시켜도 제대로 동작한다.

_root 속성을 일반적인 인스턴스 레퍼런스와 함께 사용하면 클립 계층을 따라 여러 단계 아래로 내려갈 수도 있다.

```
_root.clipA.clipB.play();
```

_root로 시작하는 레퍼런스는 어떤 문서에서 사용하더라도 똑같은 위치를 가리킨다. 이렇게 하면 복잡하게 계층 관계를 따질 필요도 없다.

_leveln을 이용한 플레이어의 다른 문서 참조

플래시 플레이어의 문서 스택에 여러 개의 .swf 파일이 들어 있다면 액션스크립트에 내장된 전역 속성인 _level0부터 _leveln까지의 레퍼런스를 이용하여 다양한 문서의 메인 무비 타임라인을 참조할 수 있다. 여기서 n은 참조하고자 하는 문서의 레벨을 나타낸다.

_level0은 문서 스택에서 가장 낮은 레벨에 있는 문서를 나타낸다(더 위 레벨에 있는 문서가 앞쪽에 렌더링된다). loadMovie()를 이용하여 무비를 _level0으로 로딩하지만 않는다면 _level0에는 플레이어가 시작될 때 처음에 가져온 무비가 들어간다.

플레이어 문서 스택의 3번 레벨에 있는 문서의 메인 무비 타임라인을 재생하고 싶다면 다음과 같은 코드를 이용하면 된다.

```
_level3.play();
```

_root 속성과 마찬가지로 _leveln 속성은 점 연산자를 이용하여 일반적인 레퍼런스와 함께 쓸 수 있다.

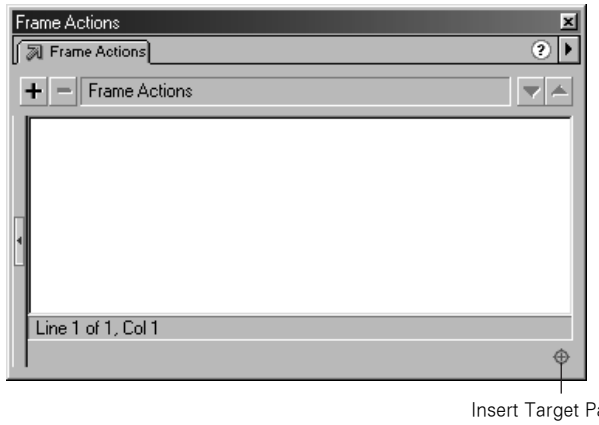
```
_level1.clipA.stop();
```

또한 _leveln 속성에 대한 레퍼런스도 _root 속성과 마찬가지로 절대 레퍼런스라고 부른다. 문서의 어느 위치에서 실행시키더라도 같은 대상을 가리키기 때문이다.

하지만 _leveln과 _root의 의미는 약간 다르다. _root 속성은 현재 문서의 레벨과는 상관없이 언제나 현재 문서의 메인 타임라인을 가리키지만, _leveln 속성은 특정 문서 레벨의 메인 타임라인에 대한 레퍼런스이기 때문이다. 예를 들어 myMovie.swf라는 파일에 _root.play()라는 코드를 집어넣는다고 가정해 보자. myMovie.swf를 5번 레벨로 로딩하면 그 코드에서는 _level5의 메인 무비 타임라인을 재생한다. 이와는 달리 myMovie.swf에 _leve2.play()라는 코드를 집어넣으면 이 코드에서는 _level5가 아닌 _level2의 메인 무비 타임라인을 재생한다. 물론 2번 레벨 안에서는 _root와 _level2가 같은 의미를 가진다.

Insert Target Path를 이용하여 인스턴스 레퍼런스 만들기

무비의 인스턴스 구조가 아주 복잡해지면 무비 클립이나 메인 무비에 대한 레퍼런스를 만드는 작업이 엄청나게 힘들어질 수도 있다. 여러 클립의 계층을 모두 기억할 수도 없는 일이고 복잡한 계층을 알아보기 위해 저작도구에서 귀찮게 여러 클립을 선택하고 편집해야 하는 경우도 있다. 액션스크립트 편집기에서는 일일이 수동으로 하지 않아도 시각적인 도구를 통해 클립 레퍼런스를 만들어낼 수 있는 Insert Target Path 도구([그림 13-6])를 제공한다.



〔그림 13-6〕 Insert Target Path 버튼

Insert Target Path를 사용하려면 다음과 같은 단계를 거치면 된다.

1. 클립 레퍼런스를 집어넣을 곳으로 커서를 움직인다.
2. 〔그림 13-6〕에 나온 Insert Target Path 버튼을 클릭한다.
3. Insert Target Path 대화상자가 나오면 참조하고 싶은 클립을 선택한다.
4. _root에서 시작하는 절대 경로를 이용할지(Absolute 선택), 아니면 코드가 들어있는 클립을 기준으로 하는 상대 경로를 이용할지(Relative 선택) 선택한다.
5. 플래시 4 형식으로 저장할 계획이라면 플래시 4와의 호환성을 위해 Slashes 옵션을 선택한다(기본으로 선택되는 Dots 옵션을 이용하여 만든 레퍼런스는 플래시 4에서는 사용할 수 없다). 〔표 2-1〕 참조

하지만 Insert Target Path에서는 클립 계층의 위에 있는 클립의 레퍼런스를 만들 수 없다. 즉 현재 클립을 포함하는 클립(_root에서 시작해서 아래로 내려가는 경우는 제외)은 이 도구를 이용하여 참조할 수 없다. 클립 계층을 거슬러 올라가는 레퍼런스를 만들려면 _parent 속성을 이용하여 적절한 레퍼런스를 수동으로 입력해야 한다.

클립 객체에 대한 동적 레퍼런스

일반적으로 프로그래밍을 할 때는 자신이 조작하고 있는 무비나 인스턴스의 이름은 알고 있겠지만, 이름을 모르는 클립을 제어해야 하는 경우도 있다. 예를 들어 루프를 이용하여 여러 클립을 동시에 축소하거나 매번 클릭할 때마다 서로 다른 클립을 참조하는 버튼을 만들어야 하는 경우도 있다. 이러한 작업을 처리하려면 실행 중에 동적으로 클립 레퍼런스를 만들어야 한다.

배열 원소 액세스 연산자 사용하기

'5장. 연산자'와 '12장. 객체와 클래스'에서 보았듯이 어떤 객체의 속성은 점 연산자 또는 배열 원소 액세스 연산자인 []를 이용하여 알아낼 수 있다. 예를 들면 아래 두 선언문은 똑같은 의미를 가진다.

```
myObject.myProperty = 10;
myObject["myProperty"] = 10;
```

배열 원소 액세스 연산자에는 점 연산자에는 없는 중요한 기능이 한 가지 있다. 인식자가 아닌 문자열 표현식을 이용하여 속성을 참조할 수 있다는 점이다([] 연산자에서는 인식자를 사용하면 안 된다). 예를 들어 다음과 같은 문자열을 합치는 표현식을 사용하더라도 myProperty라는 속성을 참조할 수 있다.

```
myObject["myProp" + "erty"];
```

똑같은 기법을 이용하여 동적으로 인스턴스나 무비에 대한 레퍼런스를 만들 수 있다. 클립 인스턴스는 그 부모 클립의 속성으로 저장된다는 점을 알고 있을 것이다. 앞에서는 점 연산자를 이용하여 그러한 인스턴스 속성을 참조하였다. 예를 들어 메인 타임라인에서 clipA라는 인스턴스 안에 있는 clipB라는 인스턴스를 다음과 같이 참조할 수 있다.

```
clipA.clipB;         // clipA 안에 있는 clipB를 참조한다.
clipA.clipB.stop();  // clipB의 메소드를 호출한다.
```

인스턴스도 속성이기 때문에 [] 연산자를 이용하여 참조하더라도 전혀 문제가 없다.

```
clipA["clipB"];         // clipA에 있는 clipB를 참조한다.
clipA["clipB"].stop();  // clipB의 메소드를 호출한다.
```

[] 연산자를 이용하여 clipB를 참조할 때는 clipB의 이름을 인식자가 아닌 문자열로 제공해야 한다. 이러한 문자열 레퍼런스로는 제대로 된 문자열이 결과로 나오는 표현식이라면 어떤 표현식이라도 사용할 수 있다. 예를 들어 아래와 같이 문자열 병합 연산을 이용하여 clipB에 대한 레퍼런스를 만들 수도 있다.

```
var clipCount = "B";
clipA["clip" + clipCount];         // Refer to clipB inside clipA
clipA["clip" + clipCount].stop();  // Invoke a method on clipB
```

순서대로 이름이 붙어있는 클립이 있다면 다음과 같이 동적으로 클립 레퍼런스를 참조할 수도 있다.

```
// clip1, clip2, clip3, clip4를 정지시킨다.
for (var i = 1; i <= 4; i++) {
  _root["clip" + i].stop();
}
```

이러한 기능을 활용하면 다양한 작업을 처리할 수 있다.

데이터 컨테이너에 클립 레퍼런스 저장하기

이 장을 시작하는 부분에서 언급했듯이 무비는 사실 액션스크립트의 데이터 객체의 일종이다. 무비 클립 인스턴스에 대한 레퍼런스를 변수, 배열 원소, 객체 속성에 저장할 수 있다.

앞에서 다룬 문서의 메인 타임라인에 있는 중첩된 인스턴스 계층(clipC는 clipB에 속하고 clipB는 clipA에 속하는 계층) 예제를 떠올려보자. 이처럼 다양한 클립을 데이터 컨테이너에 저장하면 그러한 클립들을 클립에 대한 레퍼런스를 직접 사용하지 않아도 데이터 컨테이너를 이용하여 동적으로 제어할 수 있다. 메인 타임라인에 들어가는 코드인 〔예제 13-1〕에서도 데이터 컨테이너를 이용하여 인스턴스를 저장하고 제어한다.

〔예제 13-1〕 배열과 원소에 클립 레퍼런스 저장하기

```
var x = clipA.clipB;  // clipB에 대한 레퍼런스를 x에 저장한다.
x.play();             // clipB를 재생한다.

// 클립을 배열 원소로 저장해 보자.
```

```
var myClips = [clipA, clipA.clipB, clipA.clipB.clipC];
myClips[0].play();    // clipA를 재생한다.
myClips[1]._x = 200;  // clipB를 스테이지의 왼쪽에서 200픽셀 위치에 놓는다.

// 루프를 이용하여 배열에 있는 모든 클립을 정지시킨다.
for (var i = 0; i < myClips.length; i++) {
  myClips[i].stop();
}
```

클립 레퍼런스를 데이터 컨테이너에 저장하면 문서의 클립 계층을 몰라도, 그리고 클립 계층에 영향을 미치지 않고도 클립을 조작(재생, 회전, 정지 등)할 수 있다.

for-in 루프를 이용한 무비 클립 액세스

8장에서 for-in 루프를 이용하여 각 객체의 속성에 대해 루프를 돌리는 방법을 배웠다. for-in 루프의 반복자 변수는 자동으로 객체의 모든 속성을 순환하므로 루프에 있는 선언문이 각 속성마다 한 번씩 실행된다.

```
for (var prop in someObject) {
  trace("the value of someObject." + prop + " is " + someObject[prop]);
}
```

〔예제 13-2〕에 for-in 루프를 이용하여 주어진 타임라인에 있는 모든 클립을 조작하는 예제가 나와 있다.

〔예제 13-2〕 타임라인에 있는 무비 클립 찾아내기

```
for (var property in myClip) {
  // myClip의 property 변수가 무비 클립인지 확인한다.
  if (typeof myClip[property] == "movieclip") {
    trace("Found instance: " + myClip[property]._name);

    // 클립을 조작한다.
    myClip[property]._x = 300;
    myClip[property].play();
  }
}
```

for-in 루프를 이용하면 특정 클립 인스턴스나 메인 무비에 있는 클립을 액세스하는 것이 매우 편리해진다. for-in을 이용하면 클립의 이름을 알거나 모르거나 상관없이, 클립이 자동으로 만들어졌는지 프로그램을 통해 만들어졌는지에 상관없이 어떤 타임라인에 있는 어떤 클립이라도 제어할 수 있다.

〔예제 13-3〕에는 앞 예제를 재귀적으로 고친 함수가 있다. 이 함수에서는 타임라인의 모든 클립 인스턴스를 찾아내고 모든 중첩된 타임라인에 있는 클립 인스턴스도 찾아낸다.

〔예제 13-3〕 타임라인의 모든 무비 클립을 재귀적으로 찾는 법

```
function findClips (myClip, indentSpaces) {
  // 각 계층마다 자식 클립을 한 칸 들여쓰기 위해 공백을 추가한다.
  var indent = " ";
  for (var i = 0; i < indentSpaces; i++) {
    indent += " ";
  }
  for (var property in myClip) {
    // myClip의 property 변수가 무비 클립인지 확인한다.
    if (typeof myClip[property] == "movieclip") {
      trace(indent + myClip[property]._name);
      // 이 클립이 다른 클립의 부모 클립인지 확인한다.
      findClips(myClip[property], indentSpaces + 4);
    }
  }
}
findClips (_root, 0); // 메인 타임라인에서 시작하여 모든 클립 인스턴스를 찾는다.
```

함수 재귀호출에 관한 자세한 내용은 '9장. 함수'의 '재귀 함수'를 참조하기 바란다.

_name 속성

'인스턴스 이름' 절에서 배웠듯이 모든 인스턴스의 이름은 _name이라는 내장 속성에 문자열 형태로 저장된다. 〔예제 13-2〕에서 보았듯이 이 속성을 이용하여 현재 클립이나 인스턴스 계층에 있는 다른 클립의 이름을 알아낼 수 있다.

```
_name;              // 현재 인스턴스의 이름
_parent._name       // 현재 클립을 포함하고 있는 클립의 이름
```

_name 속성은 클립의 인식자에 따른 조건 연산을 수행할 때 유용하게 쓰인다. 예를 들어 다음과 같은 코드를 이용하면 seedClip이라는 클립이 로딩되면 그 클립을 복사할 수 있다.

```
onClipEvent (load) {
  if (_name == "seedClip") {
    this.duplicateMovieClip("clipCopy", 0);
  }
}
```

seedClip이라는 이름을 직접 확인하기 때문에 무한 재귀호출을 방지할 수 있다. 위와 같은 조건문이 없으면 load 핸들러에서 새로 복사한 클립에 의해 다른 클립이 다시 복사되는 식으로 같은 작업을 반복하기 때문이다.

_target 속성

모든 무비 클립에는 _target이라는 속성이 내장되어 있다. 이 속성에는 지금은 쓰이지 않는 플래시 4의 '슬래시' 표기법을 이용하여 클립의 절대경로를 나타내는 문자열이 저장된다. 예를 들어 clipB가 clipA에 들어있고 clipA는 메인 타임라인에 있다면, 각 클립의 _target 속성은 다음과 같다.

```
_root._target                  // "/"가 저장되어 있다.
_root.clipA._target            // "/clipA"가 저장되어 있다.
_root.clipA.clipB._target      // "/clipA/clipB"가 저장되어 있다.
```

targetPath() 함수

targetPath( ) 함수는 클립의 절대 경로를 점 표기법으로 나타낸 문자열을 리턴한다. targetPath( ) 함수는 _target 속성을 대신해서 플래시 5에서 새로 만들어진 함수이다. 이 함수의 형식은 다음과 같다.

```
targetPath(movieClip)
```

여기서 movieClip은 절대 경로를 알아내고자 하는 클립의 인식자이다. 앞에서 사용한 클립들을 이용한 예는 다음과 같다.

```
targetPath(_root);            // "_level0"이 저장되어 있다.
targetPath(_root.clipA);      // "_level0.clipA"가 저장되어 있다.
targetPath(_root.clipA.clipB);// "_level0.clipA.clipB"가 저장되어 있다.
```

_name 속성에서는 클립의 이름만 알 수 있지만 targetPath() 함수를 이용하면 클립의 전체 경로를 알 수 있다(파일 이름만 있는 경우와 파일의 전체 경로 사이의 차이점과 비슷하다고 생각하면 된다). 따라서 targetPath()를 이용하여 이름뿐만 아니라 위치를 기반으로 클립을 제어하는 코드를 만들 수도 있다. 예를 들어 targetPath()를 확인하여 그 내용물이 들어있는 위치에 따라 색을 설정하는 범용 이동 버튼을 만들 수도 있다. targetPath()를 실제로 사용하는 예는 3부의 'Selection.setSelection() 메소드' 예제로 나와 있다.

Tell Target은 어디로?

플래시 4에서는 Tell Target이 무비 클립을 참조하는 가장 대표적인 도구이다. 하지만 안타깝게도 Tell Target은 플래시 5에서 훨씬 세련된 객체 모델을 도입하면서 이제는 더 이상 쓰이지 않게 되었다. Tell Target 함수는 이제 사용하지 않는 것이 좋다. 물론 Tell Target 함수를 플래시 4 형식 코드로 사용해도 되긴 하지만, 조만간 Tell Target 함수가 아예 없어질 수도 있다.

Tell Target을 이용하여 closingSequence라는 이름의 인스턴스를 재생하는 다음과 같은 코드를 살펴보자.

```
Begin Tell Target ("closingSequence")
  Play
End Tell Target
```

플래시 5에서는 다음과 같이 closingSequence 인스턴스의 play() 메소드를 호출하면 훨씬 더 간편하면서 가독성도 높아진다.

```
closingSequence.play();
```

Tell Target에서 다음과 같이 한 블록에서 하나의 인스턴스에 대해 여러 작업을 처리할 수도 있다.

```
Begin Tell Target ("ball")
  (Set Property: ("ball", x Scale) = "5")
```

```
    Play
End Tell Target
```

플래시 5에서는 '6장. 선언문'에서 배운 with() 선언문을 이용하면 비슷한 작업을 할 수 있으며, 이 방법이 권장된다.

```
with (ball) {
  _xscale = 5;
  play();
}
```

이제는 더 이상 쓰이지 않는 플래시 4 액션스크립트와 플래시 5에서 권장하는 방법에 대한 자세한 내용은 '부록 C. 하위 호환성'을 참조하기 바란다.

메인 무비 및 클립 인스턴스 제거

지금까지 무비 클립을 만들고 참조하는 법을 배웠다. 이제 그러한 클립을 제거하는 방법을 알아보자.

인스턴스나 무비를 만드는 방법에 따라 나중에 그 인스턴스 또는 무비를 제거하는 방법이 달라진다. unloadMovie()나 removeMovieClip()을 이용하여 직접 무비나 인스턴스를 제거할 수도 있다. 또는 원래 있던 클립 대신 다른 클립을 로딩하거나 추가하거나 복사해 넣으면 원래 있던 클립을 간접적으로 제거할 수 있다. 각 기법에 대해 자세히 살펴보기로 하자.

인스턴스와 레벨에 대해 unloadMovie()와 함께 사용하는 법

내장된 unloadMovie() 함수를 이용하면 어떤 클립 인스턴스나 메인 무비도(수동으로 만든 무비나 loadMovie(), duplicateMovieClip(), attachMovie() 등으로 프로그램을 통해 만든 무비 모두) 제거할 수 있다. 다음과 같이 전역 함수 또는 메소드로 호출하면 된다.

```
unloadMovie(clipOrLevel);   // 전역 함수
clipOrLevel.unloadMovie();  // 메소드
```

전역 함수로 사용할 때는 clipOrLevel 자리에 제거할 클립이나 레벨의 절대 경로를 나타내는 문자열이 들어간다. 자동 값 변환 기능이 있기 때문에 clipOrLevel에 무비 클립 레퍼런스를 사용해도 된다(문자열 위치에 들어가면 무비 클립은 경로를 나타내는 문자열로 변환된다). 메소드 형태로 사용할 때는 clipOrLevel에 무비 클립 객체에 대한 레퍼런스가 들어가야 한다. unloadMovie()의 작동 과정은 그 대상이 레벨인지 아니면 인스턴스인지에 따라 달라진다.

레벨에 대해 unloadMovie() 사용하기

unloadMovie()를 문서 스택에 있는 레벨(_level0, _level1, _level2 등)에 사용하면 대상 레벨을 완전히 제거하고 그 레벨에 들어있던 무비도 완전히 없어진다. 그리고 레벨에 대한 레퍼런스는 모두 undefined가 된다. unloadMovie() 함수는 문서 레벨을 제거하는 데 가장 많이 쓰인다.

```
unloadMovie("_level1");
_level1.unloadMovie();
```

인스턴스에 대해 unloadMovie() 사용하기

unloadMovie()를 어떤 인스턴스(수동으로 만들었거나 프로그램을 통해 만들었거나 상관없다)에 사용하면, 클립의 내용은 지워지지만 클립 자체는 지워지지 않는다. 클립의 타임라인과 캔버스는 사라지지만 그 껍데기는 스테이지에 남는다. 이 껍데기는 removeMovieClip()를 호출할 때까지(또는 인스턴스가 포함된 프레임이 종료될 때까지) 참조할 수 있다. 또한 그 껍데기에 있는 클립 이벤트 핸들러는 여전히 작동한다.

이렇게 인스턴스의 일부만 지워지는 성질을 이용하면 특이한 작업을 처리할 수 있다. 즉 loadMovie()와 unloadMovie()를 이용하여 마음대로 클립의 내용을 바꿀 수 있는 범용 컨테이너를 만들 수 있다. 예를 들어 clipA라는 인스턴스에서 다음과 같은 함수들을 호출하는 것도 가능하다(물론 실전에서는 다음과 같은 선언문에 프리로더 코드를 넣는 것이 일반적이다).

```
clipA.loadMovie("section1.swf"); // clipA에 문서를 불러온다.
clipA.unloadMovie();             // 문서를 없애고 clipA는 그대로 남겨둔다.
clipA.loadMovie("section2.swf"); // clipA에 다른 문서를 불러온다.
```

이러한 방법을 사용할 때는 한 가지 주의할 점이 있다. unloadMovie()를 인스턴스에 사용하면, 인스턴스에 포함된 클립의 모든 사용자 정의 속성이 없어진다. _x나 _alpha와 같은 물리적인 속성은 남아있지만 사용자 정의 변수나 함수는 모두 사라진다.

unloadMovie()를 전역 함수로 사용할 때 실제로 존재하지 않는 클립이나 레벨 인스턴스를 인자로 전달하면 unloadMovie( ) 함수를 호출한 위치의 클립 자체가 언로딩된다.

예를 들어 _level1이 정의되지 않았을 때 _level0의 메인 타임라인에 있는 코드에서 다음과 같은 코드를 사용하면 _level0이 언로딩된다.

```
unloadMovie(_level1);
```

물론 이렇게 되는 데는 논리적인 이유가 있다. 하지만 자세한 내용은 '메소드와 전역 함수가 겹치는 문제'에서 다루기로 하겠다. 이러한 문제는 unloadMovie()의 clipOrLevel 인자를 지정할 때 문자열을 사용하거나 언로딩하기 전에 clipOrLevel이 정말 존재하는지 미리 확인하여 방지할 수 있다. 아래에 그러한 방법을 사용한 예가 있다.

```
unloadMovie("_level1");  // clipOrLevel을 문자열로 지정한 경우
if (_level1) {           // 레벨이 존재하는지 직접 확인하는 경우
  unloadMovie(_level1);
}
```

removeMovieClip()을 이용하여 인스턴스 제거하기

프로그램을 통해 추가하였거나 복사한 인스턴스를 플레이어에서 제거할 때는 removeMovieClip()을 사용할 수도 있다. removeMovieClip()은 복사된 인스턴스 혹은 추가된 인스턴스에만 쓸 수 있다는 점에 주의하자. 수동으로 만든 인스턴스 혹은 메인 무비는 제거할 수 없다. unloadMovie()와 마찬가지로 removeMovieClip()도 전역 함수 또는 메소드 형태로 사용할 수 있다(문법은 조금 다르지만 결과는 똑같다).

```
removeMovieClip(clip)    // 전역 함수
clip.removeMovieClip( )  // 메소드
```

전역 함수 형태로 사용하면 clip에는 제거할 클립을 나타내는 문자열이 들어간다. 자동 값 변환 기능이 있기 때문에 clip 자리에 무비 클립 레퍼런스를 그냥 넣어도 된다(무비 클립을 문자열 자리에 사용하면 자동으로 경로를 나타내는 문자열로 변환된다). 메소드 형태로 사용할 때는 clip이 무비 클립 객체에 대한 레퍼런스여야 한다.

unloadMovie()와는 달리 removeMovieClip()을 통해 인스턴스를 제거하면 클립 객체가 완전히 없어지므로 클립의 껍데기나 클립 및 그 속성의 흔적이 전혀 남지 않는다. 따라서 clip.removeMovieClip()이라는 코드를 실행시키고 나면 clip을 참조했을 때 undefined가 리턴된다.

수동으로 생성된 인스턴스를 수동으로 제거하기

플래시 저작 도구에서 수동으로 만든 인스턴스는 그 생존 기간이 제한되어 있다. 플레이헤드가 그 인스턴스를 포함하지 않는 키프레임으로 들어가면 제거되기 때문이다. 따라서 수동으로 만든 무비 클립은 비어있는 키프레임을 두려워하면서 살아야 하는 운명을 지니고 있다.

무비 클립이 타임라인에서 사라지면 그 클립은 더 이상 데이터 객체가 아니다. 그 안에서 정의했던 모든 변수, 함수, 메소드, 속성이 사라진다. 따라서 클립의 정보나 함수가 그대로 남아있길 원한다면 클립을 수동으로 제거할 때 주의를 기울여야 하고 그 클립을 포함하고 있는 프레임의 수명을 클립의 정보가 필요한 동안은 계속 연장시켜야 한다(사실 이런 것에 신경을 쓰고 싶지 않다면 가장 오래 사용할 코드는 메인 무비 타임라인의 프레임에 집어넣는 것이 좋다). 클립을 타임라인에 남겨둔 채로 그 클립을 숨기려면 클립을 스테이지의 가시 영역 밖에 놓고 _visible 속성을 false로 설정하기만 하면 된다. 클립의 _x 속성을 매우 큰 값 양수 또는 매우 큰 음수로 설정하면 클립을 메모리에서 제거하지 않고도 그 클립을 숨길 수 있다.

내장 무비 클립 속성

내장 속성이 거의 없는 Object 클래스의 범용 객체와는 달리 각 무비 클립에는 다양한 내장 속성이 들어있다. 이러한 속성은 클립의 물리적인 특징을 나타내므로 그러한 특징을 바꾸고 싶다면 내장 속성을 적절히 변경하면 된다. 이러한 방법은 액션스크립트 프로그래머들이 사용하는 핵심 도구 중 하나이다.

모든 내장 무비 클립 속성 이름은 사용자 정의 속성과 구분할 수 있도록 밑줄로 시작한다. 내장 속성의 형식은 다음과 같다.

```
_property
```

내장 속성 이름은 소문자로 적어야 한다. 하지만 액션스크립트에서는 인식자의 대소문자를 구분하지 않기 때문에 속성 이름을 대문자로 적는다고 프로그램이 실행되지 않는 것은 아니다(하지만 그다지 권장할 만한 방법은 아니다).

내장 속성에 대한 자세한 정보는 3부에 나와 있으므로 여기서는 자세히 설명하지 않겠다. 하지만 속성의 이름과 간단한 내용은 알고 넘어갈 수 있도록 〔표 13-1〕에 내장 무비 클립 속성과 그 기능에 대한 간단한 설명이 들어 있는 목록을 수록해 놓았다.

〔표 13-1〕 내장 무비 클립 속성

속성 이름	속성 내용
_alpha	투명도
_currentframe	플레이헤드의 위치
_droptarget	드래그된 클립을 드롭한 클립이나 무비의 경로
_framesloaded	다운로드된 프레임 수
_height	물리적인 높이(픽셀 수로 나타내며 원래 심벌의 크기와는 상관없다)
_name	클립의 인식자. 문자열로 리턴된다.
_parent	클립을 포함하고 있는 타임라인의 객체 레퍼런스
_rotation	회전 각도(도 단위)
_target	클립의 전체 경로(슬래시 표기법 사용)
_totalframes	타임라인에 있는 프레임 수
_url	.swf 파일의 네트워크 위치
_visible	무비 클립이 화면에 표시되었는지를 나타내는 부울 값
_width	물리적인 너비(픽셀 수로 나타내며 원래 심벌의 크기와는 상관없다)

속성 이름	속성 내용
_x	스테이지의 왼쪽 끝을 기준으로 한 수평 위치(픽셀 단위)
_xmouse	클립의 좌표계에서 마우스 포인터의 수평 위치
_xscale	원래 심벌(무비의 경우에는 메인 타임라인)의 크기에 대한 수평방향 크기의 비율
_y	스테이지의 위쪽 끝을 기준으로 한 수직 위치(픽셀 단위)
_ymouse	클립의 좌표계에서 마우스 포인터의 수직 위치
_yscale	원래 심벌(무비의 경우에는 메인 타임라인)의 크기에 대한 수직방향 크기의 비율

인스턴스나 메인 무비에는 색 속성이 바로 포함되지 않는다. 색은 속성을 통해 제어하지 않고 클립의 색을 제어하기 위한 객체를 Color 클래스를 이용하여 만든다. Color 객체의 메소드를 이용하면 특정 클립의 RGB 값을 알아내거나 설정할 수 있고 변환할 수도 있다. 자세한 내용은 3부의 'Color 클랩스'에 나와 있다.

무비 클립 메소드

12장에서는 '메소드(method)'라고 부르는 특별한 속성 유형을 배웠다. 메소드는 객체에 붙어 있는 함수를 말한다. 메소드는 그 메소드를 포함하고 있는 객체를 조작하거나 그 객체와 상호작용을 하거나 그 객체를 제어하는 데 주로 쓰인다. 프로그래밍 통해 무비 클립을 제어할 때는 내장된 무비 클립 메소드를 사용하면 된다. 무비 클립의 개별 인스턴스나 라이브러리 심벌에서 직접 무비 클립 메소드를 정의할 수도 있다.

무비 클립 메소드 만들기

무비 클립에 새로운 메소드를 추가하려면 클립의 타임라인(또는 이벤트 핸들러 중 하나)에서 함수를 정의하거나 클립의 속성에 함수를 대입하면 된다. 예를 들면 다음과 같다.

```
// 클립의 타임라인에서 함수를 정의하여 메소드를 만든다.
function halfSpin() {
  _rotation += 180;
}
```

```
// 클립의 속성에 함수 리터럴을 대입하여 메소드를 만든다.
myClip.coords = function() { return [_x, _y]; };
// 아래의 메소드는 클립을 사용자가 원하는 대로 변환한다.
myClip.myTransform = function () {
  _rotation += 10;
  _xscale -= 25;
  _yscale -= 25;
  _alpha -= 25;
}
```

무비 클립 메소드 호출하기

무비 클립에 있는 메소드를 호출하는 방법은 다른 객체의 메소드를 호출하는 방법과 똑같다. 다음과 같이 클립 이름 뒤에 메소드 이름을 적으면 된다.

```
myClip.methodName();
```

메소드에 인자가 필요하다면 다음과 같이 호출할 때 인자를 전달하면 된다.

```
_root.square(5);   // square() 메소드에 5를 인자로 전달한다.
```

앞에서 배웠듯이 클립의 타임라인이나 이벤트 핸들러에서 메소드를 호출할 때는 대부분의 메소드를 인스턴스 인식자를 표기하지 않아도 현재 클립에서 직접 호출할 수 있다.

```
square(10);    // 현재 클립의 square()라는 사용자 정의 메소드를 호출한다.
play();        // 현재 클립의 play() 내장 메소드를 호출한다.
```

하지만 인스턴스 인식자가 필요한 내장 메소드도 있다(‘메소드와 전역 함수가 겹치는 문제’ 참조).

내장 무비 클립 메소드

범용 Object 클래스의 모든 멤버 객체에는 toString()과 valueOf()라는 내장 메소드가 있다. 이와 마찬가지로 다른 클래스에도 멤버 객체에서 사용할 수 있는 내장 메소드가 있다. Date 객체에는 getHours() 메소드, Color 객체에는 setRGB() 메소

드, Array 객체에는 push( )와 pop( ) 메소드 등이 들어있다. 무비 클립도 다르지 않다. 무비 클립에는 무비 클립의 외형이나 기능을 제어하거나 특성을 확인하거나 새로운 무비 클립을 만드는 작업에 사용할 수 있는 일련의 내장 메소드가 들어있다. 무비 클립 메소드는 액션스크립트의 핵심 기능 중 하나이다. 〔표 13-2〕에 3부에서 자세히 다룰 무비 클립 메소드를 대략적으로 소개한 내용이 나와 있다.

〔표 13-2〕 내장 무비 클립 메소드

메소드 이름	메소드 설명
attachMovie( )	새로운 인스턴스를 만든다.
duplicateMovieClip( )	인스턴스의 복사본을 만든다.
getBounds( )	클립이 차지하고 있는 시각적인 영역을 기술한다.
getBytesLoaded( )	인스턴스 또는 무비의 다운로드된 바이트 수를 리턴한다.
getBytesTotal( )	인스턴스 또는 무비의 물리적인 바이트 크기를 리턴한다.
getURL( )	외부 문서(주로 .html 파일)를 브라우저로 불러온다.
globalToLocal( )	메인 스테이지 좌표를 클립 좌표로 변환한다.
gotoAndPlay( )	플레이헤드를 새로운 프레임으로 이동시키고 무비를 재생한다.
gotoAndStop( )	플레이헤드를 새로운 프레임으로 이동시키고 무비를 멈춘다.
hitTest( )	어떤 점이 클립 내부에 있는지 표시한다.
loadMovie( )	외부 .swf 파일을 플레이어로 가져온다.
loadVariables( )	외부 변수를 클립이나 무비로 가져온다.
localToGlobal( )	클립 좌표를 메인 스테이지 좌표로 변환한다.
nextFrame( )	플레이헤드를 한 프레임 앞으로 이동한다.
play( )	클립을 재생한다.
prevFrame( )	플레이헤드를 한 프레임 뒤로 이동한다.
removeMovieClip( )	복사된 또는 추가된 인스턴스를 삭제한다.
startDrag( )	인스턴스나 무비를 스테이지상에서 마우스 포인터를 따라 움직이도록 만든다.
stop( )	인스턴스 또는 무비의 재생을 중지시킨다.
stopDrag( )	현재 진행 중인 모든 드래깅 작업을 끝마친다.
swapDepths( )	인스턴스 스택에서 인스턴스의 레이어를 변경한다.
unloadMovie( )	문서 레벨 또는 호스트 클립에서 인스턴스나 메인 무비를 제거한다.
valueOf( )	인스턴스의 절대 경로를 점 표기법을 이용하여 문자열로 표현한 값을 리턴한다.

메소드와 전역 함수가 겹치는 문제

이 장에서 여러 번 언급했듯이 무비 클립 메소드 중에는 전역 함수와 이름이 같은 것도 있다. 플래시 저작 도구에서도 이러한 사실을 확인할 수 있다. Actions 패널을 열고 전문가 모드로 되어 있는 것을 확인한 후 Actions 폴더를 살펴보자. gotoAndPlay(), gotoAndStop(), nextFrame(), unloadMovie()를 포함한 다양한 액션 목록이 나올 것이다. 이러한 액션은 무비 클립 메소드에도 포함되어 있다. 이렇게 액션이 겹치는 것은 단순히 범주를 분류하는 과정의 문제가 아니다. 이러한 액션은 전역 함수이므로 같은 이름을 가지는 무비 클립 메소드와는 완전히 별개로 생각해야 한다.

따라서 다음과 같은 선언문을 실행하면

```
myClip.gotoAndPlay(5);
```

gotoAndPlay()라는 메소드를 액세스하는 셈이 되지만 다음과 같은 선언문을 실행하면

```
gotoAndPlay(5);
```

gotoAndPlay()라는 전역 함수를 액세스하게 된다. 이 두 명령은 이름이 똑같지만 그 기능은 다르다. gotoAndPlay() 전역 함수는 현재 인스턴스 또는 무비에 작용한다. 하지만 gotoAndPlay() 메소드는 그 메소드를 호출한 클립 객체에 작용한다. 대부분의 경우에 이러한 미묘한 차이점은 그다지 중요하지 않다. 하지만 같은 이름을 가지는 메소드와 전역 함수가 생각보다 큰 차이를 나타내는 경우도 있다.

전역 함수 중에는 target이라는 그 함수를 적용할 클립을 나타내는 매개변수가 필요하지 않은 것도 있다. 하지만 같은 이름을 가진 메소드에서는 이러한 target 매개변수가 필요하지 않다. 메소드는 자동으로 그 메소드를 호출한 클립에 적용되기 때문이다. 예를 들어 메소드 형태로 unloadMovie()를 호출할 때는 다음과 같이 사용한다.

```
myClip.unloadMovie();
```

unloadMovie()를 메소드로 사용하면 인자를 전달하지 않아도 자동으로 myClip에 대해 그 메소드가 적용된다. 하지만 전역 함수 형태로 사용할 때는 다음과 같이 해야 한다.

```
unloadMovie(target);
```

전역 함수에서는 제거할 무비 클립을 지정해주는 target 매개변수가 필요하다. 왜 이런 것이 문제가 될까? 첫 번째로 unloadMovie() 전역 함수를 호출할 때 togoAndPlay()에 아무런 매개변수도 적어주지 않는 것처럼 아무런 인자도 넘겨주지 않는다면, 현재 문서를 제거할 수 있다고 생각할 수도 있기 때문이다.

```
unloadMovie();
```

하지만 위와 같은 선언문을 사용한다고 해서 현재 클립이 제거되지는 않는다. 단지 "Wrong number of parameters" 오류가 발생할 뿐이다. 전역 함수의 target 매개변수가 문제를 일으킬 수 있는 두 번째 이유는 조금 더 복잡하고 예상하지 못했을 경우에는 그 결과를 추적하기도 어렵다. target 매개변수를 필요로 하는 전역 함수에 target 클립을 제공하려면 그 함수의 대상이 되는 클립을 나타내는 경로의 문자열 또는 클립 레퍼런스를 전달해야 한다. 예를 들면 다음과 같다.

```
unloadMovie(_level1);    // 클립 레퍼런스
unloadMovie("_level1");  // 문자열
```

클립 객체에 대한 레퍼런스를 문자열이 들어갈 자리에 넣으면 자동으로 무비 클립 경로로 변환되므로 그냥 레퍼런스를 사용해도 상관없다. 이 정도면 그다지 복잡한 문제가 아니다. 하지만 target 매개변수가 비어있는 문자열이거나 undefined 값을 가지면 그 함수는 현재 타임라인을 대상으로 작동한다. 예를 들면 다음과 같다.

```
unloadMovie(x);   // x가 존재하지 않는다면 그 값이 undefined가 되므로
                  // 함수가 현재 타임라인에 작동된다.

unloadMovie("");  // target이 비어있는 문자열이므로 함수가 현재
                  // 타임라인에 작동된다.
```

이렇게 되면 전혀 예상치 못한 결과가 나올 수도 있다. 다음과 같이 존재하지 않는 레벨을 참조하는 경우를 생각해 보자.

```
unloadMovie(_level1);
```

_level1이 비어있다면 인터프리터에서는 이 레퍼런스가 선언되지 않은 변수인 것으로 판단한다. 따라서 undefined가 되므로 이 함수는 _level1이 아닌 현재 타임라인을 대상으로 실행된다. 그렇다면 이러한 문제는 어떻게 해결할까? 해결책에는 몇 가지가 있다. 함수를 실행하기 전에 target이 실제로 존재하는지 확인할 수도 있다.

```
if (_level1) {
  unloadMovie(_level1);
}
```

또는 target을 언제나 문자열 경로로만 표기할 수도 있다. 문자열에 표시된 경로가 실제 클립이 아니라면 이 함수는 그냥 종료된다.

```
unloadMovie("_level1");
```

때에 따라 숫자를 사용하는 함수를 사용할 수도 있다.

```
unloadMovieNum(1);
```

또는 전역 함수 대신 메소드만 사용하면 이러한 문제를 모두 피해갈 수 있다.

```
_level1.unloadMovie();
```

이러한 문제를 일으킬만한 메소드(전역 함수)는 다음과 같다(같은 이름을 가지는 플래시 5 액션스크립트 전역 함수에서는 target 매개변수가 필요하다).

duplicateMovieClip()

loadMovie()

loadVariables()

print()

printAsBitmap()

removeMovieClip()

startDrag()

unloadMovie()

어떤 무비에서 설명하기 힘든 문제가 생긴다면 위 목록을 확인하여 전역 함수를 잘못 사용한 부분이 없는지 알아보는 것이 좋다. 클립 레퍼런스를 target 매개변수 자리에 사용할 때에는 문법을 꼼꼼히 확인해보자.

무비 클립 응용 예제

지금까지 무비 클립 프로그래밍의 기초에 대해 배웠다. 이제 지금까지 배운 내용을 모두 모아서 두 개의 프로그램을 만들어 보자. 이 두 예제는 서로 크게 다르긴 하지만 무비 클립을 기본적인 내용물 컨테이너로 사용하는 법을 보여준다는 점은 서로 비슷하다.

클립을 이용하여 시계 만들기

이 장에서는 attachMovie()를 이용하여 무비 클립을 만드는 법과 점 연산자를 이용하여 무비 클립 속성을 설정하는 법을 배웠다. 이와 같이 간단한 도구와 Date, Color 클래스를 활용하면 시침, 분침, 초침이 모두 있는 시계를 만들 수 있다.

우선 다음과 같은 단계를 통해 시계의 문자판, 시계 바늘 모양을 만들어 보자(이렇게 만든 시계의 일부분을 메인 스테이지에 놓지 않는다. 시계 자체는 액션스크립트를 통해 생성된다).

1. 새로운 플래시 무비를 시작한다.
2. 100픽셀짜리 검은색 원으로 된 clockFace라는 무비 클립 심벌을 만든다.
3. 50픽셀 길이의 수직 방향 붉은 선으로 된 hand라는 무비 클립 심벌을 만든다.
4. hand의 선을 선택하고 Window → Panels → Info를 선택한다.
5. 선의 x 좌표를 0으로, y 좌표를 -50으로 설정하여 선의 아래쪽 끝을 클립의 한 가운데 놓는다.

이제 clockFace와 hand 심벌을 외부로 보내서 그 인스턴스를 무비에 동적으로 추가할 수 있도록 만들자.

1. Library에서 clockFace 클립을 선택하고 Options → Linkage를 선택한다. Symbol Linkage Properties 대화상자가 나타난다.
2. Export This Symbol을 선택한다.
3. Identifier 상자에서 clockFace라고 입력한 후 OK를 클릭한다.

4. 1에서 3단계를 반복하여 hand 클립도 내보낸다. 이 때에는 hand를 인식자로 사용한다.

이제 시계의 문자판과 바늘을 모두 만들었으므로 무비에 추가하기만 하면 된다. 이제 시계의 부품들을 스테이지에 올려놓고 매 초 시간이 지날 때마다 위치를 변경시켜주는 스크립트를 만들어 보자.

1. 〔예제 13-4〕에 나온 스크립트를 메인 타임라인의 Layer 1의 1번 프레임에 추가한다.

2. Layer 1의 이름을 scripts로 바꾼다.

우선 〔예제 13-4〕를 훑어보자. 잠시 후에 그 내용을 자세하게 알아볼 것이다.

〔예제 13-4〕 아날로그 시계

```
// 시계 문자판과 바늘을 만든다.
attachMovie("clockFace", "clockFace", 0);
attachMovie("hand", "secondHand", 3);
attachMovie("hand", "minuteHand", 2);
attachMovie("hand", "hourHand", 1);

// 시계 문자판의 위치와 크기를 지정한다.
clockFace._x = 275;
clockFace._y = 200;
clockFace._height = 150;
clockFace._width = 150;

// 시계 바늘의 위치와 크기, 색을 지정한다.
secondHand._x = clockFace._x;
secondHand._y = clockFace._y;
secondHand._height = clockFace._height / 2.2;
secondHandColor = new Color(secondHand);
secondHandColor.setRGB(0xFFFFFF);
minuteHand._x = clockFace._x;
minuteHand._y = clockFace._y;
minuteHand._height = clockFace._height / 2.5;
hourHand._x = clockFace._x;
hourHand._y = clockFace._y;
hourHand._height = clockFace._height / 3.5;
```

```
// 매 프레임마다 바늘을 회전시킨다.
function updateClock() {
  var now = new Date();
  var dayPercent = (now.getHours() > 12 ?
                    now.getHours() - 12 : now.getHours()) / 12;
  var hourPercent = now.getMinutes()/60;
  var minutePercent = now.getSeconds()/60;
  hourHand._rotation = 360 * dayPercent + hourPercent * (360 / 24);
  minuteHand._rotation = 360 * hourPercent;
  secondHand._rotation = 360 * minutePercent;
}
```

코드가 꽤 길다. 이제 이 코드를 자세히 분석해 보자.

우선 clockFace 클립을 먼저 추가하고 깊이를 0으로 지정한다(시계 바늘보다 문자판이 더 밑에 와야 하므로).

```
attachMovie("clockFace", "clockFace", 0);
```

그리고 나서 hand 심벌의 인스턴스 세 개를 추가한다. 각 인스턴스의 이름은 secondHand(초침), minuteHand(분침), hourHand(시침)이다. 각 바늘은 메인 타임라인 위에 프로그램을 통해 만든 각각의 레이어에 들어간다. secondHand(깊이 3)가 가장 위에, minuteHand(깊이 2)가 그 바로 밑에, 그리고 hourHand(깊이 1)는 그 밑으로 들어간다.

```
attachMovie("hand", "secondHand", 3);
attachMovie("hand", "minuteHand", 2);
attachMovie("hand", "hourHand", 1);
```

그리고 시계를 스테이지의 왼쪽 위에 놓는다. clockFace 클립은 스테이지의 가운데에 놓고 _height와 _width 속성을 이용하여 그 크기를 키운다.

```
clockFace._x = 275;          // 수평 위치 설정
clockFace._y = 200;          // 수직 위치 설정
clockFace._height = 150;     // 높이 설정
clockFace._width = 150;      // 너비 설정
```

secondHand 클립을 시계 위로 옮기고 clockFace 클립의 반지름과 거의 비슷한 길이로 만든다.

```
// secondHand를 clockFace 위에 놓는다.
secondHand._X = clockFace._x;
secondHand._y = clockFace._y;
// secondHand의 크기를 설정한다.
secondHand._height = clockFace._height / 2.2;
```

hand 심벌의 선은 빨간색이므로 모든 hand 인스턴스는 빨간색이다. secondHand를 눈에 잘 띄도록 Color 클래스를 이용하여 흰색으로 바꾸자. 따라서 흰색을 나타내는 16진수 값인 0xFFFFFF를 사용한다(색을 조작하는 방법에 대한 자세한 내용은 3부의 'Color 클래스' 참조).

```
// secondHand에 적용시킬 새로운 Color 객체를 만든다.
secondHandColor = new Color(secondHand);
// secondHand에 흰 색을 적용한다.
secondHandColor.setRGB(0xFFFFFF);
```

이제 secondHand에 했던 것과 똑같은 방법으로 minuteHand와 hourHand의 위치, 크기를 설정한다.

```
// minuteHand를 clockFace 위에 놓는다.
minuteHand._x = clockFace._x;
minuteHand._y = clockFace._y;
// minuteHand를 secondHand보다 조금 작게 만든다.
minuteHand._height = clockFace._height / 2.5;
// hourHand를 clockFace 위에 놓는다.
hourHand._x = clockFace._x;
hourHand._y = clockFace._y;
// hourHand를 가장 짧은 길이로 설정한다.
hourHand._height = clockFace._height / 3.5;
```

이제 현재 시각에 맞추어 시계 바늘이 돌아가게 만들어야 한다. 하지만 회전 작업을 하는 것이 아니라 시간에 따라 시계가 계속 움직이도록 해야 하므로, 같은 작업을 여러 번 반복해야 한다. 따라서 시계 바늘을 회전시키는 코드를 updateClock() 이라는 함수로 만들어 계속해서 그 함수를 호출하도록 한다.

```
function updateClock() {
  // 현재 시각을 저장한다.
  var now = new Date();
  // getHours()는 24시간을 기준으로 작동한다. 현재 시각이 12시 이후이면
  // 12를 빼서 12시간 기준의 시계에 맞추도록 한다.
```

```
    var dayPercent = (now.getHours() > 12 ?
                      now.getHours() - 12 : now.getHours()) / 12;
    // 지금이 몇 분인지 확인하여 60분에 대한 비율을 알아낸다.
    var hourPercent = now.getMinutes()/60;
    // 지금이 몇 초인지 확인하여 60초에 대한 비율을 알아낸다.
    var minutePercent = now.getSeconds()/60;
    // 각 바늘을 적절한 각도만큼 회전시킨다.
    hourHand._rotation = 360 * dayPercent + hourPercent * (360 / 24);
    minuteHand._rotation = 360 * hourPercent;
    secondHand._rotation = 360 * minutePercent;
}
```

updateClock()에서는 먼저 현재 시각을 구해서 그 값을 저장한다. 이 때는 Date 클래스의 인스턴스를 만들고 그 인스턴스를 now라는 지역 변수에 저장한다. 그리고 나서 각 바늘이 돌아간 비율(피자를 자르는 것과 비슷하다)을 계산한다. 현재 시각은 언제나 12보다 작은 숫자여야 하고 분과 초는 60보다 작아야 한다. 이러한 비율을 기준으로 각 바늘의 _rotation 값을 설정한다. hourHand의 경우에는 시뿐만 아니라 분도 고려하여 회전시킨다.

이제 시계 프로그램이 거의 완성되었다. 매 프레임을 지날 때마다 다음과 같은 방법으로 updateClock() 함수를 호출하기만 하면 된다.

1. scripts 레이어에 두 개의 키프레임을 추가한다.
2. 2번 프레임에 updateClock();이라는 코드를 추가한다.
3. 3번 프레임에 gotoAndPlay(2);라는 코드를 추가한다.

이제 무비를 테스트해서 시계가 제대로 작동하는지 알아보자. 제대로 작동하지 않는다면 온라인 코드 창고에 있는 .fla 파일과 비교해 보거나 〔예제 13-4〕에 나온 코드를 제대로 입력했는지 확인해 본다. 방금 만든 시계 프로그램을 확장할 수 있는 방법을 생각해 보자. 또한 메인 타임라인 루프(2번과 3번 프레임)를 클립 이벤트 루프로 바꾸는 방법도 생각해 보자. 이 시계를 스마트 클립으로 바꿔서 다른 플래시 무비에 추가하는 것은 어떨까? clockFace에 동적으로 시, 분 표시를 하는 것은 어떨까?

마지막 퀴즈

드디어 1장에서 시작한 객관식 퀴즈 문제의 마지막 버전을 만들어 볼 때가 되었다. 이번 버전에서는 무비 클립을 이용하여 모든 퀴즈 문제와 답을 동적으로 생성하므로, 문제 수를 무한히 많이 추가시킬 수 있고 사용자가 프로그램을 조절하기도 쉽다. 사실 프로그래머가 아닌 사람이라도 사용할 수 있도록 퀴즈 전체를 스마트 클립으로 만드는 것도 그다지 어렵진 않다.

퀴즈의 코드는 〔예제 13-5〕에 나와 있으며 온라인 코드 창고에서도 구할 수 있다. 이제는 퀴즈가 완전히 동적으로 생성되므로 99%의 코드를 모두 하나의 프레임에 넣을 수 있다. 각 타임라인에 문제를 집어넣을 필요도 없다(네트워크를 통해 재생할 때 사용하기 위한 프리로더만 추가한다면 거의 완벽한 플래시 무비가 된다). 여기서는 #include 문을 이용하여 외부 텍스트 파일에서 코드 블록을 불러들인다. #include 문에 대한 자세한 내용은 3부의 '액션스크립트 코드를 외부화하기'에 나와 있다. 연습삼아 새로운 객체를 만들고 그 객체를 문제 배열에 저장하여 새로운 문제를 추가해 보자.

마지막 퀴즈의 코드는 그리 길진 않지만 매우 중요한 기법들로 가득 차 있다. #include 문을 제외하면 다른 기법들은 모두 이전에 배운 적이 있는 것들이다. 여기서는 실전에서 그러한 기법을 어떻게 적용할 수 있는지 알 수 있다. 코드의 주석을 자세히 읽어보자. 이 퀴즈 코드를 제대로 이해할 수 있다면 액션스크립트를 이용하여 고급 프로그램을 만들 수 있는 준비가 다 되었다고 볼 수 있다.

이 퀴즈 코드에 대한 더 자세한 설명은 다음 URL에서 볼 수 있다.

http://www.moock.org/webdesign/lectures/ff2001sfWorkshop

〔예제 13-5〕 객관식 퀴즈, 마지막 버전

```
// 메인 타임라인의 1번 프레임에 들어갈 코드
//  무비를 정지시킨다.
stop();

//  메인 타임라인 변수를 초기화한다.
var displayTotal;                // 사용자의 최종 점수를 출력하기 위한
                                 // 텍스트 필드
var totalCorrect = 0;            // 정답 개수
var userAnswers = new Array();   // 사용자가 입력한 답을 기록하기 위한 배열
```

```
var currentQuestion = 0;          // 사용자가 풀고 있는 문제 번호

// 문제 객체 배열이 저장된 소스 파일을 가져온다.
#include "questionsArray.as"
// 이 예제 뒤에 있는 설명 참조

// 퀴즈 시작
makeQuestion(currentQuestion);

// Question() 생성자
function Question (correctAnswer, questionText, answers) {
  this.correctAnswer = correctAnswer;
  this.questionText = questionText;
  this.answers = answers;
}

// 각 문제를 화면에 표시하는 함수
function makeQuestion (currentQuestion) {
  // 스테이지에서 전 문제를 지운다.
  questionClip.removeMovieClip();

  // 주 문제 클립을 만든다.
  attachMovie("questionTemplate", "questionClip", 0);
  questionClip._x = 277;
  questionClip._y = 205;
  questionClip.qNum = "question\n  " + (currentQuestion + 1);
  questionClip.qText = questionsArray[currentQuestion].questionText;

  // 문제 클립에 각 답의 클립을 만든다.
  for (var i = 0; i < questionsArray[currentQuestion].answers.length; i++) {
    // 라이브러리로부터 연결된 answerTemplate 클립을 추가한다.
    // 문제에 쓰일 일반적인 버튼 및 텍스트 필드가 들어있다.
    questionClip.attachMovie("answerTemplate", "answer" + i, i);
    // 답 클립을 문제 다음 줄에 놓는다.
    questionClip["answer" + i]._y += 70 + (i * 15);
    questionClip["answer" + i]._x -= 100;
    // 이 문제의 답 배열의 원소를 이용하여 답 클립의 텍스트 필드를
    // 설정한다.
    questionClip["answer" + i].answerText =
      questionsArray[currentQuestion].answers[i];
  }
}
```

```
// 사용자가 입력한 답을 등록하는 함수
function answer (choice) {
  userAnswers.push(choice);
  if (currentQuestion + 1 == questionsArray.length) {
    questionClip.removeMovieClip();
    gotoAndStop ("quizEnd");
  } else {
    makeQuestion(++currentQuestion);
  }
}

// 사용자의 점수를 계산하는 함수
function gradeUser() {
  // 정답 개수를 센다.
  for (var i = 0; i < questionsArray.length; i++) {
    if (userAnswers[i] == questionsArray[i].correctAnswer) {
      totalCorrect++;
    }
  }
  // 온스크린 텍스트 필드에 사용자의 점수를 표시한다.
  displayTotal = totalCorrect + "/" + questionsArray.length;
}

// 동적으로 생성된 답 버튼에 들어갈 코드
// 답 클립은 동적으로 생성되며 각 클립의 이름은 "answer0", "answer1",...
// "answern"과 같은 식으로 정해진다. 각 답 클립에는 그 클립을 클릭했을 때
// 답 클립의 이름을 확인하여 사용자가 고른 답의 번호를 알아낼 수 있는
// 버튼이 있다.
on (release) {
  // 클립 이름의 앞부분에 있는 "answer"를 잘라낸다.
  choice = _name.slice(6, _name.length);
  _root.answer(choice);
}

// quizEnd 프레임에 들어갈 코드
gradeUser();
```

questionsArray.as 파일의 내용은 다음과 같다.

```
// questionsarray.as 파일에 들어갈 코드
// ------------------------------------------------
// 객관식 퀴즈의 문제와 답이 되는 문제 객체의 배열이
```

```
// 여기에 들어간다. 다음과 같은 예를 기준으로 새로운
// 문제 객체를 만들 수 있다.

/************** 문제 객체의 예 ***************
  // Question 생성자를 호출할 때에는 세 개의 인자를 사용한다.
  //   정답의 번호(0부터 시작함)
  //   문제 텍스트를 포함하는 문자열
  //   객관식 답이 저장된 배열
  new Question
  (
    1,
    "question goes here?",
    ["answer 1", "answer 2", "answer 3"]
  )
*****************************************************/
// 마지막 객체를 제외한 모든 객체 뒤에 쉼표를 추가하는 것을 잊지 말자.
questionsArray = [new Question (2,
  "Which version of Flash first introduced movie clips?",
  ["version 1", "version 2", "version 3",
  "version 4", "version 5", "version 6"]),

  new Question (2,
    "When was ActionScript formally declared a scripting language?",
    ["version 3", "version 4", "version 5"]),

  new Question (1,
    "Are regular expressions supported by Flash 5 ActionScript?",
    ["yes", "no"]),

  new Question (0,
    "Which sound format offers the best compression?",
    ["mp3","aiff", "wav"]),

  new Question (1,
    "True or False: The post-increment operator (++) returns the
    value of its operand + 1.",
    ["true", "false"]),

  new Question (3,
    "Actionscript is based on...",
    ["Java", "JavaScript", "C++", "ECMA-262", "Perl"])];
```

앞으로 배울 내용

이제 더 배울 것이 그다지 많이 남지 않았다. 객체와 무비 클립을 완전히 이해하고 나면 대부분의 액션스크립트 프로젝트를 직접 공략할 수 있다. 하지만 몇 가지 흥미로운 내용들이 조금 더 남아 있다. 다음 장에서는 렉시컬 구조(액션스크립트 문법에 대한 자질구레한 내용들)에 대해 배울 것이다. 그 뒤에는 다양한 고급 주제를 알아볼 것이다. 그 뒤에는 '2부. 액션스크립트 응용' 및 '3부. 레퍼런스'가 있다.