10

이벤트 및 이벤트 핸들러

지금까지 액션스크립트 인터프리터에서 실행시키기 위한 명령어를 만드는 법에 대해 자세히 살펴보았다. 이제 인터프리터에 '어떤' 일을 할지 알려주는 것은 꽤 익숙해졌을 것이다. 그런데 '언제' 그러한 일을 할지는 어떻게 알려줄 수 있을까? 액션스크립트 코드는 저절로 실행되지 않는다. 무엇인가 그 코드를 실행하도록 원인을 제공해야 한다.

그 '무엇'은 무비를 동기화시켜서 순서대로 재생하는 것일 수도 있고 미리 정해진 비동기화된 이벤트일 수도 있다.

동기 코드 실행

무비를 재생하면 타임라인의 플레이헤드가 프레임에서 프레임으로 움직인다. 플레이헤드가 새 프레임으로 들어갈 때마다 인터프리터에서는 그 프레임에 있는 코드를 실행시킨다. 프레임에 있는 코드가 실행되면 화면에 표시되는 내용이 업데이트되고 소리도 재생된다. 그리고 나면 플레이헤드가 다음 프레임으로 이동한다.

예를 들어 무비의 1번 프레임에 코드를 직접 넣으면 그 코드는 1번 프레임에 있는 내용이 화면에 표시되기 전에 실행된다. 같은 무비의 5번 프레임에 있는 키프레임에 다른 코드 블록을 넣으면 1번부터 4번 프레임까지 재생한 다음 5번 프레임에 있는 코드를 실행하고, 그 다음에 5번 프레임이 화면에 표시된다. 1번 프레임과 5번 프레임에서 실행된 코드는 '동기화되어 실행된다'고 할 수 있다. 선형적이고 예측할 수 있는 순서대로 실행되기 때문이다.

무비의 프레임에 들어 있는 코드는 모두 동기화되어 실행된다. gotoAndPlay() 나 gotoAndStop() 명령어 때문에 프레임 번호 순서와 다른 순서로 재생되는 프레임이 있더라도 각 프레임에 있는 코드는 예측할 수 있는 순서로 플레이헤드와 동기화되어 실행된다.

이벤트 기반의 비동기 코드 실행

어떤 코드는 실행되는 순서를 예측할 수 없는 것도 있다. 대신 그러한 코드는 액션스크립트 인터프리터에서 미리 정해진 '이벤트(event)'가 일어났다는 것을 알게 되면 실행된다. 많은 이벤트는 마우스를 클릭하거나 키를 누르는 것과 같은 사용자 행동과 연관되어 있다. 플레이헤드가 새로운 프레임에 들어가면 프레임 코드가 동기화되어 실행되는 것처럼 이벤트는 이벤트가 발생하면 이벤트 기반 코드가 실행된다. 이벤트가 언제든 일어날 수 있기 때문에 이벤트 기반 코드(이벤트가 발생하면 실행되는 코드)는 '비동기적으로 실행된다'라고 말한다.

동기화된 프로그래밍을 하려면 코드가 언제 실행될지 미리 알 수 있어야 한다. 반면에 비동기 프로그래밍을 이용하면 이벤트가 일어날 때마다 동적으로 반응할 수 있다. 비동기 코드를 실행할 수 있다는 점은 액션스크립트와 플래시의 상호작용에서 매우 결정적인 요소이다.

이 장에서는 플래시에서의 비동기 (이벤트 기반) 프로그래밍과 액션스크립트에서 지원하는 다양한 이벤트에 대해 알아보자.

이벤트 유형

이벤트는 크게 두 가지 범주로 나눌 수 있다.

사용자 이벤트

사용자 행동(마우스 클릭이나 키를 누르는 것)

시스템 이벤트

무비를 내부적으로 재생하는 과정에서 일어나는 일(예: 무비 클립이 스테이지에 나타나는 경우 또는 외부 파일로부터 일련의 변수를 로딩하는 경우)

액션스크립트에서는 사용자 이벤트와 시스템 이벤트를 정확히 구분하지 않는다. 무비에 의해 내부적으로 일어나는 이벤트라고 해서 사용자가 마우스를 클릭할 때 일어나는 이벤트와 다르지는 않다. 보통 어떤 무비 클립이 스테이지에서 없어지는 것이 그다지 대단한 이벤트로 느껴지진 않겠지만 시스템 이벤트에 반응할 수 있다는 것은 무비를 제어하는 데 크게 도움이 된다.

액션스크립트 이벤트는 그 이벤트가 속하는 객체를 기준으로 나눌 수도 있다. 모든 이벤트는 플래시 환경에서 어떤 객체와 연관되어 발생한다. 즉 인터프리터에서는 '사용자가 클릭했다' 라고 하지는 않는다. 대신 '사용자가 이 버튼을 클릭했습니다' 또는 '이 무비 클립이 스테이지에 있을 때 사용자가 마우스를 클릭했습니다' 라고 말한다. 또한 '데이터를 받았습니다' 가 아니라 '이 무비 클립에서 어떤 데이터를 받았습니다' 라고 말한다. 어떤 이벤트에 반응하기 위한 코드를 정의할 때는 이벤트와 관련된 객체에서 정의하게 된다.

이벤트를 받아들일 수 있는 액션스크립트 객체는 다음과 같다.

- 무비 클립
- 버튼
- XML 및 XMLSocket 클래스의 객체

이 장 전반에 걸쳐 배우게 되겠지만, 액션스크립트에서는 사실 두 종류의 서로 다른 이벤트를 구현한다. 하나는 무비 클립 및 버튼과 연관된 이벤트이고 나머지는 다른 모든 종류의 객체와 연관된 이벤트이다.

이벤트 핸들러

모든 이벤트에서 코드를 실행하도록 만드는 것은 아니다. 이벤트는 무비에 어떤 영향을 미치지 않고도 정기적으로 일어난다. 예를 들어 어떤 사용자가 한 버튼을 계속 클릭하여 수십개의 이벤트를 발생시킬 수도 있지만, 그러한 클릭을 무시해버릴 수도 있다. 어떻게 그렇게 할 수 있을까? 이벤트가 일어나는 것만으로는 어떤 코드를 실행시킬 수 없고 이벤트에 반응하는 코드를 별도로 만들어야 하기 때문이다. 어떤 이벤트가 일어났을 때 인터프리터에서 어떤 코드를 실행하도록 하려면 특정 이벤트가 일어날 때 해야 할 일을 기술하는 '이벤트 핸들러'라는 것을 추가해야 한다. 이벤트 핸들러라는 이름이 붙은 이유는 이벤트 핸들러에서 무비에서 일어난 이벤트를 '잡거나(catch)' '처리(handle)' 하기 때문이다.

이벤트 핸들러는 특정 이벤트가 발생할 때 자동으로 호출되는 특별한 이름을 가진 함수라고 볼 수 있다. 따라서 이벤트 핸들러를 만드는 것은 몇 가지 차이점을 제외하면 함수를 만드는 것과 매우 흡사하다.

- 이벤트 핸들러에는 keyDown과 같이 미리 정해진 이름이 있다. 이벤트 핸들러의 이름은 프로그래머가 마음대로 정할 수 있는 것이 아니라 〔표 10-1〕과 〔표 10-2〕에 나온 것처럼 미리 정해진 이름을 사용해야 한다.
- 이벤트 핸들러는 function 선언문으로 선언하지 않는다.
- 이벤트 핸들러는 버튼, 무비 클립 또는 객체에 집어넣을 수 있으며 프레임에서는 사용할 수 없다.

대부분의 이벤트는 플래시 5에서 처음 도입되었다. 플래시 4 형식으로 저장하려면 버튼 이벤트 핸들러만 사용하고 플래시 4 플레이어로 주의 깊게 테스트해보아야 한다(플래시 4에서는 버튼 이벤트만 지원된다).

이벤트 핸들러 문법

액션스크립트 이벤트 이름(또한 그에 상응하는 이벤트 핸들러의 이름)은 모두 미리 정해져 있다. 버튼 이벤트 핸들러는 on (eventName)을 이용하여 정의하고 무비 클립 이벤트 핸들러는 onClipEvent (eventName)을 이용하여 정의한다. 여기서 eventName은 처리할 이벤트의 이름이다.

따라서 모든 버튼 이벤트 핸들러(key 매개변수가 필요한 keyPress는 제외)는 다음과 같은 형식으로 쓰인다.

```
on (eventName) {
  statements
}
```

하나의 버튼 이벤트 핸들러로 여러 개의 이벤트를 처리할 수도 있다. 이 때는 각 이벤트를 쉼표로 구분한다. 예를 들면 다음과 같다.

```
on (rollover, rollOut) {
  // rollOver와 rollOut 이벤트가 발생하면 실행할 사용자 정의 함수를 호출한다.
  playRandomSound();
}
```

모든 무비 클립 이벤트 핸들러는 다음과 같은 형식으로 쓰인다.

```
onClipEvent (eventName) {
  statements
}
```

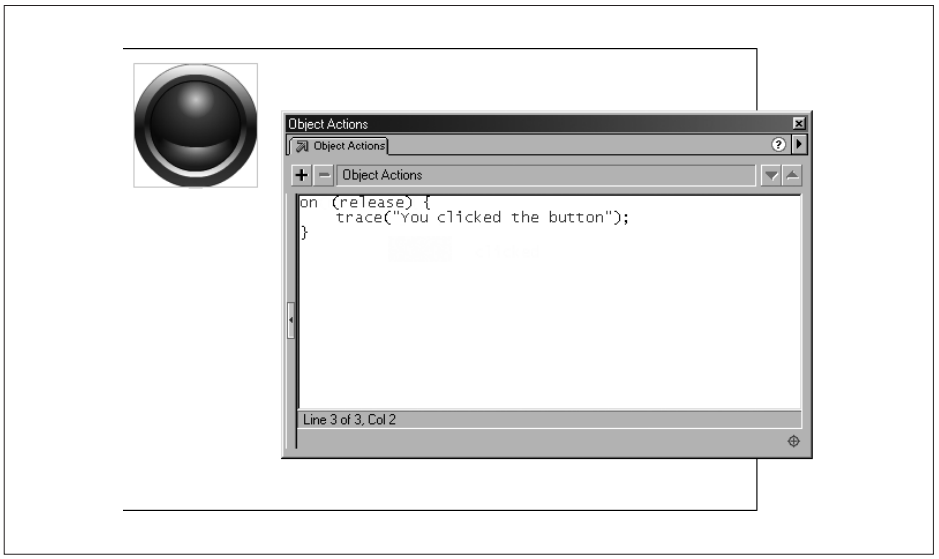
버튼 이벤트 핸들러와는 달리 클립 이벤트 핸들러는 하나의 이벤트만 처리할 수 있다.

이벤트 핸들러 만들기

이벤트 핸들러를 만들 때는 핸들러를 만들고 그 핸들러를 적절한 객체에 집어넣으면 된다. 우선 가장 일반적인 핸들러인 버튼 이벤트 핸들러와 무비 클립 이벤트 핸들러에 대해 알아보자.

버튼과 무비 클립에 이벤트 핸들러 추가하기

버튼이나 무비 클립에 이벤트 핸들러를 추가하려면 핸들러 함수 코드를 원하는 버튼 또는 클립에 추가해야 한다. 이러한 작업을 하려면 〔그림 10-1〕에 나온 것과 같이 플래시 저작 도구에서 스테이지에 있는 객체를 선택하고 Actions 패널에서 적절한 코드를 입력해야 한다.



〔그림 10-1〕 버튼에 이벤트 핸들러를 추가하는 법

버튼과 무비 클립에 간단한 이벤트 핸들러 함수를 추가해 보자. 우선 버튼 이벤트 핸들러를 만들려면 다음과 같이 하면 된다.

1. 새로운 플래시 무비를 시작한다.
2. 버튼을 만들고 그 버튼의 인스턴스를 메인 스테이지에 끌어다 놓는다.
3. 버튼을 선택한 후 Actions 패널에 다음과 같은 코드를 입력한다.

```
on (release) {
  trace("You clicked the button");
}
```

4. Control → Test Movie를 선택한다.

5. 버튼을 클릭한다. 그러면 "You clicked the button"이라는 메시지가 Output 창에 출력된다.

무비가 재생되고 버튼을 눌렀다 때면 인터프리터에서는 release 이벤트를 감지하고 on (release) 이벤트 핸들러를 실행한다. 매번 버튼을 눌렀다 뗄 때마다 Output 창에 "You clicked the button"이라는 메시지가 출력된다.

이제 무비 클립에 조금 다른 이벤트 핸들러를 추가해 보자. 이번에는 다음 과정을 따라해보자.

1. 새로운 플래시 무비를 시작한다.
2. 메인 무비 스테이지에 직사각형을 그린다.
3. Insert → Convert to Symbol을 선택한다.
4. Symbol Properties 대화상자에서 새로운 심볼의 이름을 rectangle이라고 입력하고 Behavior에서 Movie Clip을 선택한다.
5. OK를 클릭하여 rectangle 무비 클립을 만드는 작업을 끝낸다.
6. 스테이지에서 rectangle 클립을 선택한 후 다음과 같은 내용을 Actions 패널에 입력한다.

```
onClipEvent (keyDown) {
  _visible = 0;
}

onClipEvent (keyUp) {
  _visible = 1;
}
```

7. Control → Test Movie를 선택한다.
8. 무비를 클릭하여 키보드 포커스가 그 무비에 있는지 확인한다. 그리고 나서 키를 누른채 기다려 보자. 키를 누르면 rectangle 무비 클립이 사라고 눌렀던 키에서 손을 떼면 rectangle 클립이 다시 나타난다.

핸들러를 호출하는 선언문은 별도로 만들 필요가 없다는 점에 주의하자. 핸들러와 연관된 이벤트가 발생하면 인터프리터에서 자동으로 이벤트 핸들러를 호출한다.

플래시에서는 무비를 실행하는 도중에 액션스크립트를 통해 핸들러를 추가하거나 제거하는 기능을 지원하지 않는다. 이벤트 핸들러는 플래시 저작도구를 이용하여 버튼이나 클립에 할당해야 한다. 따라서 다음과 같은 구문은 사용할 수 없다.

```
myClip.onKeyDown = function () { _visible = 0; };
```

이러한 단점을 극복하는 방법은 '동적 무비 클립 이벤트 핸들러' 절에서 알아보기로 하자.

다른 객체에 이벤트 핸들러 추가하기

무비 클립이나 버튼 외에도 두 개의 내장 객체 클래스(XML과 XMLSocket)에서 이벤트 핸들러를 지원한다. 이러한 객체에서는 이벤트 핸들러가 저작 도구에 있는 어떤 물리적인 대상에 포함되지 않는다. 대신 이벤트 핸들러가 객체 인스턴스에 대한 메소드로 들어간다.

액션스크립트에서는 XML과 XMLSocket 객체에서 이벤트 핸들러로 사용하기 위한 속성 이름이 미리 정해져 있다. 예를 들어 onLoad 속성은 외부 XML 데이터를 가져왔을 때 실행할 핸들러의 이름이 된다.

XML 객체에 대한 onLoad 속성을 설정하려면 다음과 같은 코드를 사용하면 된다.

```
myDoc = new XML();
myDoc.onLoad = function () { trace("all done loading!"); };
```

또는 핸들러 함수를 먼저 정의한 다음 그 함수를 객체의 onLoad 속성에 대입할 수도 있다.

```
function doneMsg () {
  trace("all done loading!");
}
myDoc.onLoad = doneMsg;
```

이런 구문은 〔예제 10-1〕에 나온 것처럼 이벤트 핸들러 속성에 함수를 대입할 수 있는 자바스크립트의 경우와 비슷하다.

[예제 10-1] 자바스크립트 이벤트 핸들러 대입

```
// 자바스크립트의 onload 핸들러에 함수 리터럴을 대입한다.
window.onload = function () { alert("done loading"); };

// 또는 함수를 만들어서 onload 속성에 대입해도 된다.
function doneMsg () {
  alert("done loading");
}
window.onload = doneMsg;
```

플래시 차기 버전에서는 더 많은 액션스크립트 객체에서 객체 속성을 이용하여 이벤트 핸들러를 할당하는 방식을 지원할 가능성이 있으므로, 이러한 스타일에 미리 익숙해지는 것이 좋다. XML이나 XMLSocket 객체를 사용하지 않더라도 HTML 문서와 자바스크립트를 이용하여 이러한 방식으로 핸들러 만드는 연습을 해 볼 수도 있다. 이런 방법은 활용 범위가 매우 넓다는 장점을 가지고 있다. 이렇게 하면 무비를 재생하는 도중에 어떤 이벤트 핸들러 함수라도 손쉽게 새로 할당하거나 제거할 수 있다.

객체에 함수를 추가하는 방법은 '12장. 객체와 클래스'에서 더 자세하게 배울 것이다. XML과 XMLSocket 객체에서 지원하는 이벤트에 대한 정보는 '3부. 레퍼런스'에서 찾아볼 수 있다.

이벤트 핸들러의 생존 기간은 그 이벤트 핸들러가 속해 있는 객체의 생존 기간에 의해 결정된다. 클립이나 버튼이 스테이지에서 사라지거나 XML 또는 XMLSocket 객체가 없어지면 그와 관련된 이벤트 핸들러도 같이 제거된다. 핸들러가 동작하려면 객체가 스테이지에 있거나 타임라인에 존재해야 한다.

이벤트 핸들러 영역

다른 함수에 있는 선언문과 마찬가지로 이벤트 핸들러에 있는 선언문은 미리 정해진 영역 안에서 실행된다. 영역에 의해 인터프리터가 이벤트 핸들러에서 참조한 변수, 하위 함수, 객체, 속성 등을 결정하는 범위가 정해진다. 무비 클립 이벤트, 버튼 이벤트, 다른 객체 이벤트에 대한 이벤트 핸들러 영역을 각각 알아보도록 하자.

무비 클립 이벤트 핸들러 영역

보통 함수와는 달리 무비 클립 이벤트 핸들러에는 지역 영역이라는 것이 정의되어 있지 않다. 클립에 핸들러를 추가하면 핸들러 영역은 이벤트 핸들러 자체에 국한되는 것이 아니라 클립 전체가 된다. 즉 모든 변수를 클립의 타임라인으로부터 알아낸다. 예를 들면 navigation이라는 클립에 enterFrame 이벤트 핸들러를 추가하고 그 핸들러에 trace(x);라는 선언문을 입력하면, 인터프리터에서는 x 값을 navigation의 타임라인에서 검색한다.

```
onClipEvent (enterFrame) {
  trace(x);  // navigation.x의 값이 출력된다.
}
```

검색할만한 지역 영역이 아예 없기 때문에, 인터프리터에서는 지역 영역을 먼저 검색하지 않는다. 핸들러 안에서 var y = 10;이라는 선언문을 실행시켜도 y는 navigation의 타임라인에서 정의된다(일반적인 함수에서 var 키워드를 사용하면 함수 안에서만 쓸 수 있는 지역 변수를 선언하게 된다).

핸들러 선언문을 핸들러 클립의 프레임에 포함된 것처럼 생각하면 클립 이벤트 핸들러의 영역을 따지는 규칙을 기억하기에 편리하다. 예를 들어 xVelocity라는 변수를 포함하고 있는 ball이라는 클립이 있다고 생각해 보자. ball의 이벤트 핸들러 내부에서 xVelocity를 사용할 때는 다음과 같이 그냥 변수 이름만 사용하면 된다.

```
onClipEvent (mouseDown) {
  xVelocity += 10;
}
```

인터프리터에서 이미 우리가 ball 클립에 있는 xVelocity라는 변수를 사용하려고 한다는 것을 알고 있기 때문에, _root.ball.xVelocity와 같이 변수의 경로까지 적어줄 필요는 없다. 속성과 메소드에 대해서도 똑같은 규칙이 적용된다. ball._x가 아니라 _x만, ball.gotoAndStop(5)가 아니라 gotoAndStop(5)만 적어주면 된다. 예를 들면 다음과 같다.

```
onClipEvent (enterFrame) {
  _x += xVelocity;                 // 공을 움직인다.
  gotoAndPlay(_currentframe - 1);  // 간단한 루프 반복
}
```

심지어 다음과 같이 핸들러에서 function 선언문을 이용하여 ball에 포함되는 함수를 정의할 수도 있다.

```
onClipEvent (load) {
  function hideMe() {
    _visibility = 0;
  }
}
```

클립 이벤트 핸들러에 있는 선언문의 영역은 핸들러 함수의 지역 영역이나 클립의 부모 타임라인(클립이 있는 곳 바로 위의 타임라인)이 아니라 클립의 타임라인이라는 점을 잊어버리는 경우도 종종 있다.

예를 들어 ball 클립을 무비의 메인 타임라인에 놓고 (ball의 타임라인이 아닌) 메인 타임라인에서 moveBall()이라는 함수를 정의했다고 하자. 별 생각 없이 ball에 있는 이벤트 핸들러에서 다음과 같이 moveBall()을 호출할 수도 있다.

```
onClipEvent (enterFrame) {
  moveBall();  // ball에는 moveBall()이라는 함수가 없으므로
               // 아무 일도 일어나지 않는다.
               // moveBall() 함수는 _root에서 정의한 함수이다.
}
```

이런 경우에는 다음과 같이 _root를 이용하여 moveBall() 함수가 메인 타임라인에 있다는 것을 알려줘야 한다.

```
onClipEvent (enterFrame) {
  _root.moveBall();  // 이렇게 하면 제대로 동작한다.
}
```

때에 따라 이벤트 핸들러에서 현재 클립 객체를 직접 참조해야 하는 경우도 있다. 이런 경우에는 this 키워드를 사용하면 되는데, 이 키워드는 이벤트 핸들러 안에서 현재 무비 클립을 가리키는 역할을 한다. 따라서 아래와 같은 레퍼런스는 이벤트 핸들러 내부에서 쓰면 똑같은 변수를 가리킨다.

```
this._x  // _x와 똑같다.
_x

this.gotoAndStop(12);  // gotoAndStop(12)와 같다.
gotoAndStop(12);
```

현재 클립의 속성(변수 또는 클립) 이름을 동적으로 생성할 때 this를 가장 많이 사용한다. 아래 코드에서는 ball 클립의 현재 프레임 번호에 따라 ball.stripe1, ball.stripe2, ...와 같은 이름을 가진 클립 중 하나를 재생한다.

```
onClipEvent (enterFrame) {
  this["stripe" + _currentframe].play();
}
```

this라는 키워드는 무비 클립 객체에 대한 직접적인 레퍼런스가 필요한 무비 클립 메소드에서도 자주 쓰인다. 액션스크립트의 전역 함수와 같은 이름을 가진 무비 클립 메소드를 사용할 때는 직접 클립 레퍼런스를 적어줘야 한다. 따라서 다음과 같은 함수를 이벤트 핸들러 내부에서 메소드로 호출할 때는 this 키워드를 사용해야 한다.

duplicateMovieClip()

loadMovie()

loadVariables()

print()

printAsBitmap()

removeMovieClip()

startDrag()

unloadMovie()

예를 들면 다음과 같다.

```
this.duplicateMovieClip("ball2", 1);
this.loadVariables("vars.txt");
this.startDrag(true);
this.unloadMovie();
```

이처럼 이름이 같은 함수 때문에 생기는 문제는 '13장. 무비 클립'의 '메소드와 전역 함수가 겹치는 문제'라는 절에서 조금 더 알아보기로 하자.

this 키워드를 이용하면 저작도구에서 현재 클립의 이름을 정해주지 않았거나 클립 이름을 모르는 경우에도 현재 클립을 참조할 수 있다는 점을 기억해 두자. 게다가 현재 클립의 이름을 모르는 경우에도 this 키워드만 사용하면 현재 클립을 다른

함수에 레퍼런스로 전달할 수 있다. 아래 예제에 이러한 기능을 사용하여 만든 코드가 나와 있다.

```
// 메인 타임라인의 코드
// 어떤 클립이든 움직일 수 있는 범용 함수
function move (clip, x, y) {
  clip._x += x;
  clip._y += y;
}

// 클립에 들어가는 코드
// 메인 타임라인에 있는 함수를 호출하여 현재 클립을 움직인다.
// 이 때 this 키워드를 이용하여 현재 클립에 대한 레퍼런스를 전달한다.
onClipEvent (enterFrame) {
  _root.move(this, 10, 15);
}
```

플래시 5의 빌드 30에는 버그가 있어서 gotoAndStop()과 gotoAndPlay() 함수를 클립 핸들러 안에서 실행시킬 때 문자열 리터럴 레이블을 사용할 수 없다. 그러한 명령이 나오면 그냥 무시된다. 예를 들면 다음과 같은 코드는 작동하지 않는다.

```
onClipEvent(load) {
 gotoAndStop("intro"); // 플래시 5r30에서는 작동하지 않는다.
}
```

이러한 버그를 피하려면 다음과 같은 클립 레퍼런스를 사용하면 된다.

```
onClipEvent(load) {
 this.gotoAndStop("intro");
}
```

버튼 이벤트 핸들러 영역

버튼 핸들러 영역은 그 버튼이 속해 있는 타임라인 영역에 들어간다. 예를 들어 어떤 버튼을 메인 타임라인에 추가하고 그 버튼에 있는 핸들러에서 speed라는 변수를 선언하면 speed 영역은 메인 타임라인(_root)에 속하게 된다.

```
// 버튼 핸들러에 들어가는 코드
on (release) {
  var speed = 10;  // _root에서 speed라는 변수를 정의한다.
}
```

이와는 달리 메인 타임라인에 ball이라는 무비 클립을 놓고 ball의 핸들러에서 speed라는 변수를 선언하면 speed의 영역은 ball에 속하게 된다.

```
// ball 핸들러에 들어가는 코드
on (load) {
  var speed = 10;  // _root가 아닌 ball에서 speed를 정의한다.
}
```

버튼 핸들러 안에서 this 키워드는 버튼을 포함하고 있는 타임라인을 가리키게 된다.

```
on (release) {
  // 이 버튼을 포함하고 있는 클립의 투명도를 50%로 설정한다.
  this._alpha = 50;
  // 이 버튼을 포함하고 있는 클립을 오른쪽으로 10 픽셀 옮긴다.
  this._x += 10;
}
```

다른 객체 이벤트 핸들러 영역

무비 클립 핸들러 또는 버튼 핸들러와 달리 XML이나 XMLSocket과 같은 내장 클래스의 인스턴스에 들어가는 이벤트 핸들러 영역은 함수 영역과 똑같다. XML 또는 XMLSocket 객체의 이벤트 핸들러의 영역 사슬은 핸들러 함수를 정의할 때 함께 정의되는 영역 사슬과 같다. 게다가 XML 및 XMLSocket 이벤트 핸들러에는 지역 영역도 있다. '9장. 함수'의 '함수 영역' 절에서 설명한 모든 함수 영역 규칙은 버튼 또는 무비 클립을 제외한 객체에 들어가는 이벤트 핸들러 함수에도 그대로 적용된다.

버튼 이벤트

〔표 10-1〕에 버튼에서 사용할 수 있는 다양한 이벤트를 간단하게 정리해 놓았다. 버튼 이벤트를 이용하면 다른 위치로의 이동, 폼, 게임 외의 다양한 인터페이스 요소에서 사용할 수 있는 코드를 손쉽게 만들 수 있다. 각 버튼 이벤트에 대해 알아보고 버튼에서 마우스나 키보드 이벤트에 반응할 수 있도록 프로그램을 짜는 방법을 알아보기로 하자.

〔표 10-1〕에 나온 각 버튼 이벤트는 on (eventName) 형태로 주어지는, 각 이벤트에 상응하는 버튼 이벤트 핸들러에서 처리한다. 예를 들어 press 이벤트는 on (press)로 시작하는 이벤트 핸들러로 처리한다. 단 한 가지 예외가 있다. keyPress 이벤트 핸들러는 on (keyPress key)와 같은 형식으로 시작되는데, 이 때 key에는 그 이벤트 핸들러에서 처리할 키가 들어간다. 버튼 이벤트는 마우스와 상호작용을 하는 버튼에만 전달된다. 여러 개의 버튼이 서로 겹쳐 있는 경우에는 가장 위에 있는 버튼에서 모든 이벤트를 받아들이며, 가장 위에 있는 버튼에 이벤트 핸들러가 없는 경우에도 다른 버튼에서는 주어진 이벤트에 반응할 수 없다. 이 책에서는 버튼이 활성화되기 위해 마우스 포인터가 있어야 할 버튼 영역을 가리킬 때 '히트 영역(hit area)'이라는 용어를 사용하겠다(버튼의 히트 영역은 플래시 저작 도구에서 버튼을 만들 때 그래픽 작업에 의해 정의된다).

〔표 10-1〕 버튼 이벤트

버튼 이벤트 이름	버튼 이벤트가 일어나는 경우
press	포인터가 버튼의 히트 영역에 있을 때 마우스의 주 버튼을 누른 경우. 다른 마우스 버튼은 무시된다.
release	포인터가 버튼의 히트 영역에 있을 때 마우스의 주 버튼을 눌렀다가 떼는 경우
releaseOutside	포인터가 버튼의 히트 영역에 있을 때는 마우스의 주 버튼을 누르고 있다가 포인터가 히트 영역에서 벗어나면 버튼에서 손을 떼는 경우
rollOver	마우스 버튼을 누르지 않은 상태에서 마우스 포인터를 버튼의 히트 영역으로 이동시킨 경우
rollOut	마우스 버튼을 누르지 않은 상태에서 마우스 포인터를 버튼의 히트 영역 밖으로 이동시킨 경우
dragOut	포인터가 버튼의 히트 영역 안에 있는 상태에서 마우스의 주 버튼을 누르고 마우스 버튼을 누른 채로 포인터를 히트 영역 밖으로 움직인 경우

버튼 이벤트 이름	버튼 이벤트가 일어나는 경우
dragOver	포인터가 버튼의 히트 영역 안에 있는 상태에서 마우스의 주 버튼을 누르고 마우스 버튼을 누른 채로 포인터를 히트 영역 밖으로 움직였다가 다시 히트 영역 안으로 끌고 오는 경우
keyPress	주어진 키를 누른 경우. 대부분의 경우에 keyPress 버튼 이벤트보다는 keyDown 클립 이벤트를 주로 사용한다.

press

마우스 클릭을 자세히 보면 두 단계로 진행된다는 점을 알 수 있다. 우선 마우스 버튼을 누르는(press) 단계와 버튼에서 손가락을 떼는(release) 단계로 나눌 수 있기 때문이다. press 이벤트는 마우스 포인터가 히트 영역 안에 있는 상태에서 마우스의 주 버튼[1]을 누르는 경우에 발생한다. 다른 마우스 버튼은 아무리 눌러도 press 이벤트가 발생하지 않는다. 버튼 press 이벤트는 라디오 버튼이나 게임에서의 무기 발사 버튼과 같은 용도에 적합하지만, 마우스 버튼에서 손가락을 떼기 전에 사용자가 생각이 바뀌면 선택을 취소할 수 있도록 하고 싶다면 press 대신 release 이벤트를 이용하는 것이 좋다.

release

release 버튼 이벤트는 다음과 같은 일련의 동작이 일어날 때 발생한다.

1. 마우스 포인터가 버튼의 히트 영역 안에 있다.
2. 마우스 포인터가 버튼의 히트 영역 안에 있는 상태에서 마우스의 주 버튼을 누른다(이 때 press 이벤트가 발생한다).
3. 마우스 포인터가 여전히 원래 버튼의 히트 영역 안에 있는 상태에서 마우스의 주 버튼에서 손가락을 뗀다(이 순간에 release 이벤트가 발생한다).

1) 역자주: 일반적인 경우 마우스의 왼쪽 버튼(왼손잡이용으로 설정한 경우에는 마우스의 오른쪽 버튼)이 마우스의 주 버튼이다. 매킨토시와 같이 버튼이 하나뿐인 경우에는 그냥 그 버튼이 주 버튼이 된다.

press 이벤트 대신 release 이벤트를 사용하는 경우에는 사용자가 마우스 버튼을 누른 후에도 포인터를 버튼 밖으로 옮기고 나서 손가락을 떼면 release 이벤트가 발생하지 않으므로 사용자가 처음에 내렸던 결정을 취소할 수 있다.

releaseOutside

releaseOutside 이벤트는 보통 사용자가 버튼을 클릭했다가 마우스 버튼에서 손가락을 떼기 전에 포인터를 버튼 밖으로 움직여서 버튼을 선택하려다가 만 경우에 발생한다. 이 이벤트는 다음과 같은 일련의 동작이 일어나면 발생한다.

1. 마우스 포인터가 버튼의 히트 영역 안에 있다.
2. 마우스의 주 버튼을 누르고 손가락을 버튼에서 떼지 않는다(이 때 press 이벤트가 발생한다).
3. 마우스 포인터를 버튼의 히트 영역 밖으로 움직인다(이 때 dragOut 이벤트가 발생한다).
4. 원래 버튼의 히트 영역 밖에서 마우스의 주 버튼에서 손가락을 뗀다.

굳이 releaseOut 이벤트를 이용하는 경우는 거의 없을 것이다. 사용자가 버튼을 클릭했다가 마음이 바뀌어서 취소하려고 하는 경우에 보통 releaseOut 이벤트가 발생하기 때문이다.

rollOver

rollOver 이벤트는 마우스 버튼을 누르지 않은 상태에서 마우스 포인터를 히트 영역 안으로 움직일 때 발생한다. 포인터가 버튼 위로 움직일 때 버튼 모양을 바꾸거나 하는 작업은 스크립트를 이용하지 않고 플래시 저작 도구에서 직접 처리하기 때문에 액션스크립트에서 rollOver 이벤트를 사용하는 일은 거의 없다. 버튼을 강조하는 기능을 구현할 때는 저작 도구에서 제공하는 up, over, down 프레임을 이용해야 한다.

플래시 5의 rollOver 이벤트를 이용하면 텍스트 필드 선택사항을 손쉽게 읽어들일 수 있다. 자세한 내용은 3부의 '선택 객체'를 참조하기 바란다.

rollOut

rollOut 이벤트는 rollOver 이벤트와 반대되는 경우에 발생한다. 즉 마우스 버튼을 누르지 않은 상태에서 마우스 포인터를 히트 영역 밖으로 움직이는 경우에 rollOut 이벤트가 발생한다. 버튼 강조 상태는 저작 도구에서 직접 만들기 때문에 액션스크립트에서 별도로 이미지를 바꿀 필요가 없으므로 rollOver와 마찬가지로 rollOut 이벤트도 거의 쓰이지 않는다.

dragOut

dragOut 이벤트는 포인터가 버튼의 히트 영역에서 벗어날 때 마우스 버튼이 눌려 있는 상태에서 발생한다는 점을 제외하면 rollOut 이벤트와 거의 비슷하다. dragOut 이벤트 뒤에는 반드시 releaseOutside 이벤트(사용자가 마우스 버튼에서 손가락을 떼는 경우)나 dragOver 이벤트(사용자가 마우스 버튼을 누른 채로 마우스 포인터를 버튼의 히트 영역으로 다시 끌고 가는 경우)가 뒤따른다.

dragOver

dragOver 이벤트는 그다지 자주 일어나는 이벤트는 아니다. 이 이벤트는 다음과 같은 일련의 동작에 의해 발생한다.

1. 마우스 포인터가 버튼의 히트 영역 안으로 들어간다(rollOver 이벤트가 발생한다).
2. 마우스의 주 버튼을 누른 채로 손가락을 떼지 않는다(press 이벤트가 발생한다).
3. 마우스 포인터를 버튼의 히트 영역 밖으로 움직인다(dragOut 이벤트가 발생한다).
4. 마우스 포인터를 버튼의 히트 영역 안으로 다시 움직인다(dragOver 이벤트가 발생한다).

따라서 dragOver 이벤트는 사용자가 마우스 버튼을 계속 누른 상태에서 마우스 포인터를 히트 영역 밖으로 움직였다가 다시 히트 영역 안쪽으로 움직였다는 것을 의미한다. 마우스 포인터가 버튼의 히트 영역에서 나왔다가 다시 그 안으로 들어갈

때 마우스 버튼을 누른 상태이면 rollOver 이벤트가 아닌 dragOver 이벤트가 발생한다는 점에 주의하자.

keyPress

keyPress 이벤트는 마우스 이벤트와 관련된 것이 아니라 특정 키를 누르는 것에 의해 생기는 이벤트이다. 이 이벤트를 지금 다루는 이유는 액션스크립트의 다른 버튼 이벤트 핸들러와 마찬가지로 on (eventName)과 같은 구조를 가지기 때문이다. 이 이벤트 핸들러에서는 그 이벤트를 발생시키는 키를 지정해 주어야 한다.

```
on (keyPress key) {
  statements
}
```

여기서 key는 그 이벤트와 관련된 키를 나타내는 문자열이다. 이 문자열에는 키에 나와 있는 글자("s"나 "S" 등) 또는 "<Keyword>"와 같은 형식으로 표현된 키워드가 들어갈 수 있다. 각 핸들러에서는 하나의 키밖에 지정할 수 없다. keyPress를 이용하여 여러 키를 감지하려면 다음과 같이 여러 개의 keyPress 이벤트 핸들러를 만들면 된다.

```
// "a" 키를 감지한다.
on (keyPress "a") {
  trace("The 'a' key was pressed");
}

// Enter 키를 감지한다.
on (keyPress "<Enter>") {
  trace("The Enter key was pressed");
}

// 아래쪽 화살표 키를 감지한다.
on (keyPress "<Down>") {
  trace("The Down Arrow key was pressed");
}
```

Keyword 자리에 사용할 수 있는 값들은 다음과 같다(F1부터 F12까지 기능키는 keyPress에서는 지원하지 않기 때문에 Key 객체를 이용해야 한다).

```
<Backspace>
<Delete>
<Down>
<End>
<Enter>
<Home>
<Insert>
<Left>
<PgDn>
<PgUp>
<Right>
<Space>
<Tab>
<Up>
```

플래시 4에서는 keyPress를 이용하는 방법이 키보드를 감지할 수 있는 유일한 방법이었다. 플래시 5 이후에서는 Key 객체를 무비 클립 이벤트인 keyDown 및 keyUp(잠시 후에 배울 것이다)과 함께 사용하여 키보드를 훨씬 다양하게 제어할 수 있다. keyPress 이벤트를 이용하는 경우에는 한 번에 하나의 키를 누르는 것만 감지할 수 있지만, Key 객체를 이용하면 여러 개의 키를 한꺼번에 누르는 것도 감지할 수 있다.

무비 클립 이벤트 개요

무비 클립 이벤트는 마우스 클릭에서 데이터 다운로드에 이르기까지 플래시 플레이어 전반에 걸쳐 다양하게 일어난다. 클립 이벤트는 사용자 입력 이벤트와 무비 재생 이벤트의 두 가지 범주로 크게 나눌 수 있다. 사용자 입력 이벤트는 마우스나 키보드와 관련되어 있고 무비 재생 이벤트는 플래시 플레이어에서 프레임을 렌더링하는 것이나 무비 클립의 시작과 끝, 데이터 로딩 따위와 연관되어 있다.

사용자 입력 클립 이벤트는 앞에서 다룬 버튼 이벤트와 그 기능 면에서 약간 겹치는 부분이 있다. 예를 들어 어떤 클립의 mouseDown 이벤트 핸들러는 버튼의 press 이벤트 핸들러와 마찬가지로 마우스 버튼이 눌리는 것을 감지할 수 있다. 하지만 무비 클립 이벤트는 버튼 이벤트처럼 히트 영역같은 것에 얽매여 있지 않으며 마우스 포인터의 모양에도 영향을 미치지 않는다.

〔표 10-2〕에 요약해 놓은 액션스크립트 무비 클립 이벤트에 대해 자세히 알아보자. 우선 무비 재생 이벤트(enterFrame, load, unload, data)에 대해 배우고 나서 사용자 입력 이벤트(mouseDown, mouseUp, mouseMove, keyDown, keyUp)에 대해 알아보기로 하자. 각 클립 이벤트는 onClipEvent (eventName)과 같은 형식으로 시작하는 그 이벤트에 상응하는 클립 이벤트 핸들러에서 처리한다. 예를 들어 enterFrame 이벤트는 onClipEvent (enterFrame)으로 시작하는 이벤트 핸들러로 처리한다. load, unload, data 이벤트를 제외한 모든 무비 클립 이벤트는 사용자가 다른 무비 클립(또는 무비 클립이 아닌 곳)을 클릭하더라도 스테이지에 있는 모든 무비 클립으로 전달된다.

〔표 10-1〕 무비 클립 이벤트

클립 이벤트 이름	클립 이벤트가 발생하는 경우
enterFrame	플레이헤드가 프레임에 들어갈 때(플래시 플레이어에서 프레임을 렌더링하기 전)
load	클립이 처음 스테이지에 등장할 때
unload	클립이 스테이지에서 사라질 때
data	클립의 변수 로딩이 완료되었을 때, 또는 로딩된 무비의 일부가 클립에 로딩될 때
mouseDown	클립이 스테이지에 있는 상태에서 마우스의 주 버튼을 누를 때(다른 버튼에는 반응하지 않음)
mouseUp	클립이 스테이지에 있는 상태에서 마우스의 주 버튼에서 손가락을 뗄 때
mouseMove	클립이 스테이지에 있는 상태에서 마우스 포인터가 조금이라도 움직일 때(마우스가 클립 위에 있지 않아도 상관없음)
keyDown	클립이 스테이지에 있는 상태에서 키를 누를 때
keyUp	클립이 스테이지에 있는 상태에서 눌렀던 키에서 손가락을 뗄 때

무비 재생 무비 클립 이벤트

다음 이벤트는 플래시를 로딩하고 무비를 재생하는 과정에서 사용자가 아무것도 하지 않더라도 발생되는 이벤트이다.

enterFrame

스크립트를 실행하기 위해 비어있는 무비로 루프를 돌려야 한다면, enterFrame을 대신 사용할 수 있다. enterFrame 이벤트는 무비에서 각 프레임이 지나갈 때마다 발생한다. 예를 들어 다음과 같은 코드를 무비 클립에 집어넣으면 그 클립은 프레임마다 10픽셀씩 커진다.

```
onClipEvent (enterFrame) {
  _height += 10;
  _width += 10;
}
```

(앞에서 배웠듯이 _height와 _width 속성은 enterFrame 이벤트 핸들러를 포함하고 있는 클립 영역에 속하므로, _height와 _width 앞에 클립 인스턴스 이름을 붙이지 않아도 된다)

enterFrame 이벤트는 enterFrame 핸들러가 들어있는 클립의 플레이헤드가 멈추더라도 각 프레임이 렌더링되기 전에 매번 발생한다. 따라서 enterFrame 이벤트는 계속해서 발생한다.

어떤 플래시 무비라도 플래시 플레이어에서 재생되면 화면에서 아무것도 움직이지 않거나 무비의 플레이헤드가 한 프레임에서 정지되어 있더라도 계속해서 실행된다. 개별 무비 클립의 enterFrame 핸들러는 그 클립이 재생 중이거나 정지해 있거나 상관없이 클립이 스테이지에 있는 한 계속 실행된다. gotoAndStop() 함수를 호출하여 클립의 플레이헤드를 다른 곳으로 옮기더라도 각 프레임을 지날 때마다 클립의 enterFrame 이벤트 핸들러는 여전히 실행된다. stop() 함수를 이용하여 무비 전체의 모든 플레이헤드가 멈추더라도 모든 클립에 있는 enterFrame 이벤트 핸들러는 계속해서 실행된다.

enterFrame 이벤트는 일반적으로 무비 클립의 상태를 계속해서 업데이트하는데 사용된다. 하지만 enterFrame 이벤트 핸들러가 그 핸들러를 포함하고 있는 클립에 직접 적용되어야 하는 것은 아니다. enterFrame을 코드를 반복 실행하기 위한 용도로 프레임이 하나뿐인 비어있는 클립에서 사용할 수도 있다. 클립 이벤트 루프(또는 간단하게 프로세스라고도 부름)는 '8장. 순환문'의 '타임라인 루프와 클립 이벤트 루프'에서 이미 살펴보았다.

enterFrame 이벤트 핸들러에 있는 코드는 그 핸들러를 포함하고 있는 클립의 타임라인에 들어 있는 다른 어떤 코드보다도 먼저 실행된다.

조금 더 욕심을 부리면 enterFrame을 이용하여 클립을 다양한 형태로 변화시킬 수도 있다. 뒤에 나오는 〔예제 10-7〕에서는 조금 전에 나왔던 클립 확대 코드를 확장하여 클립의 크기가 커졌다 작아졌다 하는 기능을 구현한다.

load

load 이벤트는 무비 클립이 탄생하는 순간(즉 무비 클립이 스테이지에 처음 나타날 때)에 발생한다. 무비 클립은 다음 중 한 가지 방법을 통해 스테이지에 등장한다.

- 플레이헤드가 저작 도구에서 정해준 위치에 클립의 새로운 인스턴스를 만들어주는 키프레임으로 들어갈 때
- duplicateMovieClip( ) 함수를 통해 다른 클립으로부터 새로운 클립을 복사할 때
- attachMovie( ) 함수를 이용하여 프로그래밍을 통해 클립을 스테이지에 추가할 때
- loadMovie( ) 함수를 이용하여 외부의 .swf 파일을 클립으로 로딩할 때
- unloadMovie( ) 함수를 이용하여 클립의 내용을 언로딩할 때(클립의 내용물이 사라지면 그 자리를 메꾸기 위한 비어있는 클립이 클립으로 로딩되기 때문에 load 이벤트가 발생한다)

load 이벤트 핸들러의 본체는 무비 클립이 처음 나타나는 타임라인에 있는 다른 모든 코드가 실행된 후에 실행된다.

load 이벤트 핸들러는 클립의 변수를 초기화하거나 초기 설정 과정(동적으로 생성된 클립의 크기나 위치를 조절하는 작업 등)을 처리하기 위한 목적으로 쓰이기도 한다. 또한 load 핸들러를 이용하여 무비 클립이 자동으로 재생되는 것을 막을 수도 있다.

```
onClipEvent (load) {
  stop();
}
```

특정 클립이 있어야 제대로 실행되는 함수를 호출하는 경우에도 load 이벤트 핸들러를 활용할 수 있다.

load 이벤트는 새로운 무비 클립을 만드는 duplicateMovieClip() 함수와 함께 사용할 때 특히 유용하다. 〔예제 10-2〕에서는 하나의 load 이벤트 핸들러를 연속적으로 이용하여 star 클립으로 만들어진 화면을 만든다. load 핸들러는 star 클립을 복사할 때마다 함께 복사되므로 자기 자신을 계속해서 복사하게 된다. 100번째 클립이 만들어지면 이 과정이 멈춘다. 〔예제 10-2〕에서 쓰인 .fla 파일은 온라인 코드 창고에서 구할 수 있다.

〔예제 10-2〕 load 이벤트를 이용하여 별로 가득찬 화면 만들기

```
onClipEvent (load) {
  // 현재 클립을 임의의 위치로 옮긴다.
  _x = Math.floor(Math.random() * 550);
  _y = Math.floor(Math.random() * 400);

  // 이전 클립의 비율을 그대로 상속하지 않도록 클립 비율을 리셋한다.
  _xscale = 100;
  _yscale = 100;

  // 50 퍼센트에서 150 퍼센트 사이의 임의의 값을 이용하여
  // 현재 클립의 크기를 조절한다.
  randScale = Math.floor(Math.random() * 100) - 50;
  _xscale += randScale;
  _yscale += randScale;

  // 아직 100번째 별을 만들지 못했다면 또 다른 클립을 복사한다.
  if (_name != "star100") {
    nextStarNumber = number(_name.substring(4, _name.length)) + 1;
    this.duplicateMovieClip("star" + nextStarNumber, nextStarNumber);
  }
}
```

unload

unload 이벤트는 load 이벤트와 반대되는 이벤트로 무비 클립이 완료되면 발생하는 이벤트이다. 즉 클립이 스테이지에 존재하는 마지막 프레임이 완료되자마자(하지만 다음 프레임으로 들어가기 전에) 발생한다.

다음과 같은 상황에서 무비 클립의 unload 이벤트가 발생한다.

- 플레이헤드가 클립이 존재하는 기간의 마지막 프레임에 올 때
- removeMovieClip( ) 함수(attachMovie( ) 또는 duplicateMovieClip( ) 함수로 만든 클립을 제거하는 함수)로 클립을 제거했을 때
- unloadMovie( ) 함수를 이용하여 전에 로딩했던 .swf 파일을 제거했을 때
- 클립에 외부 .swf 파일을 로딩했을 때

마지막에 나와 있는 상황은 조금 이상해 보일지 모르지만 사실 플래시에 무비가 로딩되는 방법을 생각해 보면 당연히 그렇게 된다는 것을 알 수 있다. 무비 클립에 .swf 파일을 로딩하면 그 클립에 원래 들어있던 내용이 없어지므로 unload 이벤트가 발생하는 것이다.

아래 예에서는 loadMovie( ) 함수와 연관된 load 및 unload 이벤트를 설명하고 있다.

1. 플래시 저작 도구에서 무비의 메인 타임라인에 있는 1번 프레임의 스테이지에 비어있는 무비 클립을 추가한다. 그 클립의 이름을 emptyClip이라고 하자.
2. 메인 타임라인의 5번 프레임에서 emptyClip.loadMovie("test.swf");라는 코드를 이용하여 test.swf라는 무비를 emptyClip으로 로딩한다.
3. Control → Play movie를 선택하여 무비를 재생한다.

이렇게 하면 다음과 같은 결과가 나온다.

1. 1번 프레임: emptyClip 클립이 나타나면서 load 이벤트가 발생된다.
2. 5번 프레임: 두 단계에 걸쳐 loadMovie( ) 함수가 실행된다.

a. test.swf를 로딩할 수 있도록 emptyClip의 빈 자리를 차지하고 있던 내용물을 제거한다. 이 때 unload 이벤트가 발생한다.

b. test.swf라는 무비가 로딩되면서 load 이벤트가 발생한다.

unload 이벤트는 뒷정리용 코드(스테이지를 정리하거나 프로그램 환경을 리셋하는 코드)를 시작할 때 주로 쓰인다. 또한 unload 핸들러를 이용하면 무비 클립이 끝나고 난 뒤에 어떤 다른 작업(다른 무비를 재생하는 것 등)을 처리할 수 있다.

data

data 이벤트는 무비 클립에 외부 데이터가 로딩되었을 때 발생한다. data 이벤트는 로딩되는 데이터 종류에 따라 두 가지의 서로 상이한 상황에서 발생한다. 각 경우를 따로 생각해 보자.

loadVariables()에서 data 이벤트 핸들러 사용법

loadVariables( ) 함수를 이용하여 서버에 일련의 변수를 요청하면 그 변수의 정보를 활용하기 전에 우선 그러한 변수가 완전히 로딩될 때까지 기다려야 한다(3부 참조).

무비 클립에서 로딩된 변수의 마지막 부분을 받고 나면 data 이벤트가 발생되어 로딩된 값이 필요한 코드를 실행해도 좋다는 것을 알려준다.

예를 들어 방문객이 의견을 입력하고 그러한 의견을 서버에 저장하는 방명록 무비가 있다고 생각해 보자. 사용자가 방문객이 남긴 의견을 볼 때는 loadVariables( ) 를 이용하여 서버에 그러한 정보를 요청한다. 하지만 사용자 의견을 화면에 표시하기 전에 요청한 데이터를 모두 사용할 수 있는지 확인하기 전까지는 로딩 화면을 보여준 채로 기다려야 한다. data 이벤트 핸들러를 이용하면 데이터 로딩이 완료되는 것을 알 수 있기 때문에 방문객의 의견을 안전하게 출력할 수 있다.

〔예제 10-3〕은 방명록 코드의 일부를 간단하게 정리해둔 것으로 loadVariables( ) 함수와 함께 data 이벤트 핸들러를 이용하는 방법을 보여준다. 이 예에서는 한 버튼에서 텍스트 파일로부터 두 개의 URL로 인코딩된 변수를 읽어들여 무비 클립으로 로딩한다. 이 무비 클립에는 변수가 모두 로딩된 후에 실행하는 data 이벤트 핸들러가 들

어있다. 핸들러에 있는 코드는 data 이벤트가 일어나기 전까지(즉 데이터를 모두 받기 전까지)는 실행되지 않으므로, 변수를 화면에 출력해도 괜찮다는 것을 알 수 있다.

〔예제 10-3〕 data 이벤트를 기다리는 예제

```
// guestbook.txt 파일의 내용
name=judith&message=hello

// 클립에 있는 버튼
on (release) {
  this.loadVariables("guestbook.txt");
}

// 클립에 있는 핸들러
onClipEvent (data) {
  trace(name);
  trace(message);
}
```

'17장. 플래시 폼'에서 플래시 폼을 만드는 법에 대해 배울 때 data 이벤트를 조금 더 자세히 알아보자.

loadMovie()와 data 이벤트 핸들러

data 이벤트를 활용하는 두 번째 용도는 loadMovie() 함수를 이용하여 외부의 .swf 파일을 무비 클립으로 로딩하는 것과 관련되어 있다. .swf 파일을 호스트 클립으로 로딩하면 파일을 모두 로딩하지 않은 경우에도 바로 파일이 재생된다. 하지만 이렇게 무조건 로딩하는 것이 항상 바람직한 것은 아니다. 재생을 시작하기 전에 .swf 파일을 모두 또는 일정 비율 이상 로딩해야 하는 경우도 있다. data 이벤트 핸들러와 프리로딩 코드를 이용하면 그렇게 만들 수 있다.

data 이벤트는 호스트 무비 클립에서 외부의 .swf 파일의 일부를 받을 때마다 발생하는 이벤트이다. 이 때 '일부'의 의미는 생각보다 복잡하다. data 이벤트가 발생하려면, 마지막 data 이벤트가 발생한 후, 또는 .swf 파일 로딩을 시작한 후에 외부 .swf 파일 중 적어도 하나의 프레임이 로딩되어야 한다(그동안 .swf 파일로부터 두 개 이상의 프레임을 로딩할 수도 있지만 data 이벤트가 발생할 수 있는 최소한의 프레임 개수는 한 개다).

data 이벤트 핸들러의 실행은 플레이어에서 프레임을 렌더링하는 것과 밀접하게 연관되어 있다. 각 프레임이 렌더링될 때마다 인터프리터에서는 로딩된 외부 .swf 파일의 일부분 가운데 data 이벤트 핸들러가 있는 클립에 로딩된 것이 있는지 확인한다. 만약 외부 .swf 파일 중 일부가 그러한 클립에 로딩되었다면, 그리고 새로 로딩된 부분이 적어도 한 프레임 이상이라면 data 이벤트 핸들러가 실행된다. 이러한 과정은 (플레이헤드가 멈추더라도) 한 프레임을 렌더링할 때 한 번만 발생한다.

data 이벤트는 매 프레임을 기준으로 발생하므로 프레임 속도가 빠른 무비일수록 프리로더의 모양이 더 부드럽다. .swf 파일의 로딩 상황을 더 자주 업데이트하기 때문이다.

loadMovie() 작업을 수행하는 중에 일어나는 data 이벤트의 정확한 개수는 로딩하는 .swf 파일의 내용 분포와 연결 속도에 따라 달라진다. 한 프레임짜리 .swf 파일은 파일 크기가 아무리 커도 한 번에 하나의 data 이벤트만 발생시킨다. 반면에 100개의 프레임이 있는 .swf 파일은 무비의 프레임 속도나 각 프레임의 바이트 크기, 네트워크 연결 속도 등에 따라 달라질 수 있지만 최대 100개의 data 이벤트까지 발생시킬 수 있다. 프레임이 크고 연결이 느릴 때 더 많은 data 이벤트가 발생된다(최대 한 프레임에 한 번까지). 프레임이 작고 연결 속도가 빠르면 data 이벤트 횟수가 줄어든다(플레이어에서 두 프레임을 렌더링하는 동안 100 프레임이 모두 전송되면 data 이벤트가 한 번 밖에 발생하지 않는다).

어떻게 data 이벤트 핸들러를 이용하여 프리로더를 만들 수 있을까? loadMovie() 함수를 호출하여 data 이벤트가 발생되면 외부 .swf 파일을 다운로드하는 중이라는 것을 알 수 있다. 따라서 data 이벤트 핸들러 내부에서 파일을 재생하기 전에 파일이 충분히 다운로드되었는지 확인할 수 있다. 이러한 작업은 〔예제 10-4〕에 나온 것처럼 getBytesLoaded()와 getBytesTotal() 함수를 이용하여 처리할 수 있다(무비 클립 속성인 _framesloaded와 _totalframes를 이용해도 된다).

〔예제 10-4〕에서는 무비를 로딩하는 중간에 다운로드 상황을 보여준다. 완전히 다운로드하기 전에 자동으로 재생되는 것을 방지하려면 로딩 중인 .swf 파일의 첫 번째 프레임에서 stop() 함수를 호출해야 한다는 점에 주의하자. 온라인 코드 창고에서 〔예제 10-4〕를 변형시킨 코드를 구할 수 있다.

〔예제 10-4〕 data 이벤트 프리로더

```
onClipEvent (data) {
    trace("data received");           // 프리로더 시작

    // 데이터 전송을 나타내는 불을 켠다.
    _root.transferIndicator.gotoAndStop("on");

    // 로딩이 끝나면 데이터 전송을 나타내는 불을 끄고 무비를 재생한다.
    if (getBytesTotal() > 0 && getBytesLoaded() == getBytesTotal()) {
      _root.transferIndicator.gotoAndStop("off");
      play();
    }

    // _root에 있는 텍스트 필드 변수에 로딩과 관련된 자세한 내용을 표시한다.
    _root.bytesLoaded = getBytesLoaded();
    _root.bytesTotal = getBytesTotal();
      _root.clipURL = _url.substring(_url.lastIndexOf("/") + 1,
_url.length);
}
```

사용자 입력 무비 클립 이벤트

나머지 무비 클립 이벤트는 사용자와의 상호작용과 연관되어 있다. 사용자 입력 클립 이벤트가 발생하면 스테이지에 있는 모든 클립(다른 클립 안에 들어 있는 클립도 상관없다)에서 그 이벤트를 받아들인다. 따라서 마우스를 클릭하거나 움직이거나 키보드의 키를 누를 때, 한꺼번에 여러 개의 클립이 반응할 수도 있다.

특정 클립 근처에 마우스 포인터가 있을 때 반응하는 코드를 만들고 싶다면, 이벤트 핸들러에서 클립에 대한 상대적인 마우스 포인터의 위치를 확인하도록 만들면 된다. 잠시 후에 〔예제 10-9〕에서 볼 수 있겠지만 hitTest()라는 내장 함수를 이용하면 마우스 클릭이 특정 영역 안에서 일어났는지 간단하게 확인할 수 있다.

mouseDown

press 버튼 이벤트와 마찬가지로 mouseDown 이벤트는 마우스 버튼을 누르는 것을 감지한다. mouseDown 이벤트는 마우스 포인터가 스테이지의 어느 위치에 있더라도 마우스의 주 버튼을 누를 때마다 발생한다.

버튼 이벤트인 press와는 달리 mouseDown은 특정 버튼의 히트 영역과는 상관이 없다. [예제 10-8]에서 볼 수 있듯이 mouseUp과 mouseMove 이벤트, 그리고 Mouse.hide() 메소드와 mouseDown 이벤트를 같이 사용하면 사용자 정의 마우스 포인터를 구현할 수 있다.

mouseUp

mouseUp 이벤트는 mouseDown 이벤트와 반대되는 이벤트이다. 이 이벤트는 마우스 포인터가 스테이지의 임의의 지점에 있을 때 마우스의 주 버튼에서 손가락을 뗄 때마다 발생한다. mouseDown과 마찬가지로 이벤트 결과를 나타내려면 마우스 버튼에서 손가락을 뗄 때 스테이지에 mouseUp 핸들러가 들어있는 클립이 있어야 한다. mouseUp, mouseDown, mouseMove 이벤트를 이용하면 (버튼과 마찬가지로) 마우스 포인터의 모양에 영향을 끼치지 않고 마우스의 움직임에 반응하는 다양한 프로그램을 만들 수 있다.

mouseMove

mouseMove 이벤트를 이용하면 마우스 포인터의 위치 변화를 알아낼 수 있다. 마우스가 움직일 때마다 프로세서에서 새로운 이벤트를 발생시킬 수 있는 최대 속도로 mouseMove 이벤트를 발송한다. mouseMove 이벤트 결과를 나타내려면 마우스가 움직일 때 스테이지에 mouseMove 핸들러가 들어있는 클립이 있어야 한다.

mouseMove 이벤트는 정지 상태에 있는 애플리케이션을 다시 시작하거나 마우스가 움직인 경로를 표시하거나 사용자 정의 포인터를 만드는 경우에 유용하다. 이와 관련된 코드는 [예제 10-8]에 나와 있다.

keyDown

keyDown과 keyUp 이벤트는 mouseDown과 mouseUp 이벤트를 키보드에서 적용한 것과 비슷하다. 이 두 이벤트를 이용하면 키보드를 통해 사용자와 상호작용하는 코드를 만들 수 있다. keyDown 이벤트는 키보드에 있는 키를 누를 때 발생한다. 키를 계속 누르고 있으면 OS와 키보드 설정에 따라 조금씩 다르긴 하지만 keyDown 이벤트가 계속해서 발생할 수 있다. keyPress 버튼 이벤트와는 달리 keyDown 클립 이벤트는 특정 키만이 아니라 임의의 키를 누를 때도 발생한다.

keyDown 이벤트를 감지하려면 키를 눌렀을 때 스테이지에 keyDown 이벤트 핸들러가 들어있는 무비 클립이 있어야 한다. 아래 코드는 키가 눌린 것을 감지하는 이벤트 핸들러이다.

```
onClipEvent (keyDown) {
  trace("Some key was pressed");
}
```

keyDown 핸들러에서는 어떤 키가 눌렸는지 알 수 없다. 사용자가 아무 키나 누르면 다음 단계로 넘어가도록 하는 경우에는 어떤 키를 눌러도 상관없다. 하지만 보통 특정 키에 고유 기능을 부여하는 경우가 많다. 예를 들어 여러 키를 이용하여 우주선을 움직이는 방향을 조절하는 것도 그와 같은 경우에 해당한다.

keyDown 이벤트가 발생했을 때 어떤 키가 눌렸는지 알아내려면 키보드 상태를 알려주는 내장 객체인 Key 객체를 이용하면 된다. 우리가 필요로 하는 정보의 유형은 우리가 원하는 상호작용에 따라 달라진다. 예를 들어 게임의 경우에는 여러 개의 키를 연속으로 누르더라도 연속적으로, 그리고 즉각적으로 반응해야 한다. 반면에 위치를 이동하기 위한 인터페이스에서는 하나의 키만을 감지해도 된다(예를 들면 슬라이드 쇼 프레젠테이션의 경우 스페이스바와 같은 것).

Key 객체에서는 사용자가 마지막으로 누른 키와 현재 누르고 있는 키에 대한 정보를 구할 수 있다. 키보드 상태는 Key 객체에 있는 네 개의 메소드를 이용하여 확인할 수 있다.

```
Key.getCode()          // 마지막으로 누른 키의 키코드 (10진수)
Key.getAscii()         // 마지막으로 누른 키의 ASCII 값 (10진수)
Key.isDown(keycode)    // 주어진 키를 누르고 있는 상태이면 true를 리턴한다.
Key.isToggled(keycode)// Caps Lock이나 Num Lock이 걸려 있는지 확인한다.
```

〔예제 10-5〕는 마지막으로 누른 키의 ASCII 값을 알려주는 keyDown 핸들러이다.

〔예제 10-5〕 마지막으로 누른 키 확인

```
onClipEvent (keyDown) {
  // 마지막으로 누른 키의 ASCII 값을 구해서 문자로 변환한다.
  lastKeyPressed = String.fromCharCode(Key.getAscii());
  trace("You pressed the '" + lastKeyPressed + "' key.");
}
```

〔예제 10-6〕은 마지막으로 누른 키가 위쪽 화살표인지 확인하는 keyDown 핸들러이다.

〔예제 10-6〕 위쪽 화살표 키 감지

```
onClipEvent (keyDown) {
  // 마지막으로 누른 키가 위쪽 화살표 키인지 확인한다.
  // 위쪽 화살표는 Key.UP 속성으로 표기한다.
  if (Key.getCode() == Key.UP) {
    trace("The up arrow was the last key depressed");
  }
}
```

키보드 상태를 알아내는 방법에는 여러 가지가 있으므로 용도에 가장 적합한 방법을 선택하여 사용하면 된다. 예를 들어 Key.getAscii( ) 메소드는 마지막으로 누른 키와 연결된 문자에 해당하는 ASCII 값을 리턴하는데, 각 언어별 키보드 배치에 따라 그 결과가 조금씩 다를 수 있다(영문 키보드의 경우에는 키보드에 있는 문자와 숫자의 배치가 표준으로 정해져 있다). 반면에 Key.getCode( ) 메소드에서는 키보드에 있는 특정 키(그 키의 문자가 아닌 키 자체)와 연결된 값을 리턴한다. 그 키가 나타내는 문자와는 상관없이 서로 붙어 있는 키 네 개를 이용하여 위치를 이동하는 데 사용하는 것처럼 전 세계 어느 곳에서나, 또는 어떤 플랫폼에서나 실행시킬 수 있는 프로그램을 만들 때는 Key.getCode( )를 사용하는 것이 더 낫다. 자세한 정보를 원한다면 3부에 있는 'Key 객체'를 읽어보기 바란다.

온라인 코드 창고에서 keyDown과 keyUp을 이용하는 .fla 샘플 파일을 다운로드할 수 있다.

키와 관련된 동작에 반응하는 이벤트 핸들러는 플래시 플레이어에 마우스 포커스가 있을 때만 실행된다. 무비의 키 관련 핸들러가 활성화되려면 사용자가 무비의 스테이지를 미리 클릭해야 한다. 무비에서 키보드에 의해 제어되는 부분에 들어가기 전에 사용자가 버튼을 클릭하도록 하는 방법도 괜찮은 방법이다.

특수 키 처리

(웹 브라우저의 플러그인이 아닌) 독립적인 플래시 플레이어에 있는 메뉴 명령어(Open, Close, Fullscreen과 같은 것)가 나타나지 않도록 하려면 무비 시작 부분에 다음과 같은 코드를 추가하면 된다.

```
fscommand("trapallkeys", "true");
```

위와 같은 명령을 사용하면 Projector에서 Esc 키를 눌러도 전체화면 모드를 빠져나갈 수가 없다. Projector에서 Esc 키를 감지하려면 다음과 같은 코드를 사용하면 된다.

```
onClipEvent (keyDown) {
  if (Key.getCode() == Key.ESCAPE) {
    // Esc 키를 눌렀을 때 실행할 코드
  }
}
```

하지만 모든 브라우저에서 Esc 키를 감지할 수 있는 것은 아니다. 게다가 Alt 키 또는 윈도우의 Alt-Tab, Ctrl-Alt-Del 키를 사용하지 못하게 하는 방법도 없다.

탭 키가 눌리는 것을 감지하려면 다음과 같은 핸들러가 들어있는 버튼을 만들면 된다.

```
on (keyPress "<Tab>") {
  // 탭 키에 반응한다.
}
```

독립적인 플레이어에서는 다음과 같이 클립 이벤트 핸들러를 이용하여 탭 키를 감지할 수도 있다.

```
onClipEvent (keyDown) {
  if (Key.getCode() == Key.TAB) {
```

```
    // 탭 키에 반응한다.
  }
}
```

어떤 브라우저에서는 탭 키를 감지할 때 keyPress 버튼 이벤트만 사용할 수 있는 경우도 있고 keyPress 버튼 이벤트와 keyUp 클립 이벤트를 결합시켜야 하는 수도 있다. 다음 예에서는 우선 keyPress를 이용하여 탭 키를 감지하고 keyUp 핸들러에서 탭 키를 눌렀을 때 실행할 코드를 처리한다. 인터넷 익스플로러에서는 탭 키의 Key.getCode() 값이 탭 키에서 손가락을 뗄 때만 설정되기 때문에, keyDown 이벤트는 사용하지 않는다.

```
// 메인 타임라인에 있는 버튼의 코드
on (keyPress "<Tab>") {
  // 별 필요 없는 변수를 설정한다.
  foo = 0;
}

// 메인 타임라인에 있는 무비 클립의 코드
onClipEvent (keyUp) {
  if (Key.getCode() == Key.TAB) {
    // _levle0에 있는 myTextField로 커서를 옮긴다.
    Selection.setFocus("_level0.myTextField");
  }
}
```

보통 폼에 있는 특정 텍스트 필드의 입력 위치로 움직일 때 탭 키를 이용한다. 자세한 내용을 원한다면 3부의 'Selection.setFocus() 메소드'에 있는 예제를 참조하기 바란다.

Ctrl-F와 같은 단축키 형태의 키 조합을 감지하려면 enterFrame 핸들러와 Key.isDown() 메소드를 이용하면 된다.

```
onClipEvent (enterFrame) {
  if (Key.isDown(Key.CONTROL) && Key.isDown(70)) {
   // Ctrl-F에 반응하는 부분
  }
}
```

엔터 키를 감지하려면 다음과 같이 버튼 핸들러를 사용하거나

```
on (keyPress "<Enter>") {
  // 엔터 키에 반응한다(예: 폼 전송).
}
```

아니면 다음과 같이 keyDown 핸들러를 이용하면 된다.

```
onClipEvent (keyDown) {
  if (Key.getCode() == Key.ENTER) {
    // 엔터 키에 반응한다(예: 폼 전송).
  }
}
```

기능 키(F1, F2,...)나 숫자 키패드에 있는 키와 같이 다른 특수 키를 감지하는 방법은 3부의 'Key 객체'와 'Key.getCode() 메소드'에서 자세히 다룬다.

keyUp

keyUp 이벤트는 키를 눌렀다가 뗄 때 발생한다. keyUp 이벤트는 게임 프로그래밍에서 필수적인 요소이다. keyDown 이벤트를 이용해서 켰던 어떤 기능을 keyUp 이벤트를 이용하여 끄는 경우가 많기 때문이다(대표적인 예로는 우주선 게임에서 특정 키를 누르고 있는 동안 추진력을 강화시키는 경우를 생각할 수 있다). 플래시 저작 도구에서는 스페이스바를 잠시 누르고 있으면 Hand 도구로 전환되고 스페이스바에서 손을 떼면 다시 원래 사용하던 도구로 돌아간다. 이런 식으로 임시 메뉴와 같은 것을 애플리케이션에서 화면에 표시했다가 다시 사라지게 할 수도 있다.

keyDown 이벤트와 마찬가지로 keyUp 이벤트에서도 유용한 정보를 알아내려면 Key 객체를 사용해야 한다.

```
onClipEvent (keyUp) {
  if (!Key.isDown(Key.LEFT)) {
    trace("The left arrow is not depressed");
  }
}
```

Key.isDown() 메소드를 이용하면 언제나 모든 키의 상태를 알 수 있으므로 enterFrame 이벤트 루프를 이용하여 특정 키를 눌렀는지 알 수 있다. 하지만 키보

드를 계속해서 감시하는 것은 keyDown 이벤트가 이벤트 핸들러를 실행시키는 것처럼 사용자가 어떤 키를 눌렀다는 것을 알아낼 때까지 기다리는 것보다는 비효율적이다.

결국 어떤 방법을 사용할지는 만들고 있는 시스템의 유형에 따라 달라진다. 게임과 같이 계속해서 변화하는 시스템에서는 매 프레임마다 메인 게임 루프를 돌기 때문에 키보드를 계속해서 감시하는 것이 좋다. 즉 루프를 돌리면서 Key 객체를 확인해주기만 하면 된다. 예를 들면 다음과 같다.

```
// 비어있는 클립의 코드
// 이렇게 하면 게임 프로세스가 계속 실행된다.
onClipEvent (enterFrame) {
  _root.mainLoop();
}

// 메인 타임라인에 있는 게임의 핵심 코드
// 프레임마다 한 번씩 실행된다.
function mainLoop () {
  if (Key.isDown(Key.LEFT)) {
    trace("The left arrow is depressed");
    // 우주선을 왼쪽으로 회전시킨다.
  }

  // 다른 키를 확인해보고 게임에 필요한 작업을 처리한다.
}
```

하지만 정적인 인터페이스 환경에서는 특정한 키 조합(여러 키를 한꺼번에 누르는 것)을 감지하는 경우를 제외하면, enterFrame 루프를 이용하여 키보드를 체크하지 않아도 된다. 매번 키를 누를 때, 키에서 손을 뗄 때마다 한 번씩 실행되는 keyDown 및 keyUp 이벤트 핸들러만 사용해도 된다. keyUp과 keyDown 이벤트 핸들러를 사용할 때는 어떤 주어진 시각에 키가 눌린 상태인지 몰라도 상관없다. 따라서 사용자가 프레임 사이에서 키에서 손을 떼더라도 키가 눌리는 것을 정확하게 감지할 수 있으므로 키를 한 번만 눌렀을 때도 같은 키를 두 번 조사할 필요가 없다. 하지만 어떤 경우에든 keyDown이나 keyUp 이벤트 핸들러에서 마지막으로 누른 키를 확인하기 위해 Key.getCode() 또는 Key.getAscii() 메소드를 사용한다.

실행 순서

어떤 무비에서는 코드가 여러 타임라인과 여러 클립 이벤트 핸들러에 퍼져 있는 경우도 있다. 따라서 한 프레임에서 여러 코드 블록을 실행해야 하는 경우도 그리 드물지는 않다(어떤 코드는 이벤트 핸들러에 있고 어떤 코드는 클립 타임라인에 있는 프레임에, 어떤 코드는 플레이어에 있는 문서의 메인 타임라인에 있을 수도 있다). 이런 경우에는 여러 코드가 실행되는 순서가 꽤 복잡해지고 그 순서가 프로그램의 기능에 큰 영향을 미칠 수 있다. 무비에 있는 여러 타임라인과 비교해서 이벤트 핸들러가 실행되는 순서를 확실하게 알아두면 예기치 못한 상황이 발생할 가능성도 줄어들고 코드를 원하는 대로 실행시킬 수 있다.

비동기 이벤트 핸들러는 무비의 타임라인에 있는 코드와는 독립적으로 실행된다. 예를 들어 버튼 이벤트 핸들러는 그 핸들러에서 처리하는 이벤트가 발생하면 바로 실행된다. 또한 mouseDown, mouseUp, mouseMove, keyDown, keyUp 이벤트도 이벤트가 발생하는 대로 바로 실행된다.

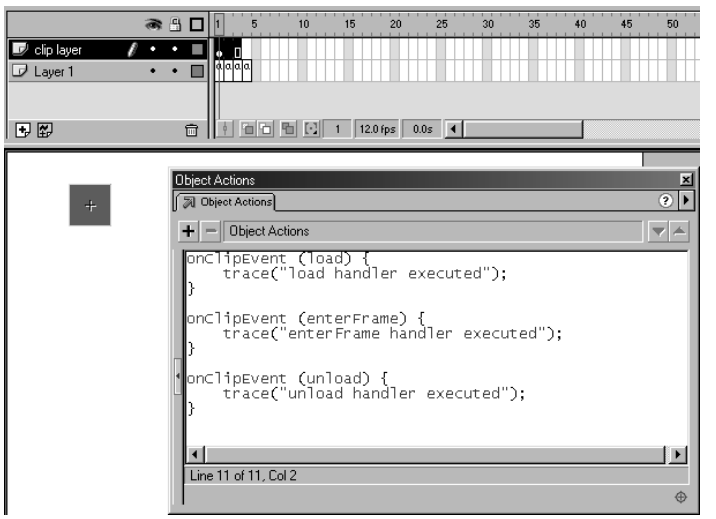
하지만 무비 재생 이벤트는 〔표 10-3〕에 나온 것처럼 무비가 진행됨에 따라 순서대로 실행된다.

〔표 10-3〕 무비 클립 이벤트 핸들러 실행 순서

이벤트 핸들러	실행되는 때
load	클립이 스테이지에 나타난 첫 번째 프레임에서 부모 타임라인의 코드가 실행된 후에 실행된다. 하지만 프레임이 렌더링되기 전에, 그리고 클립 내부 코드가 실행되기 전에 실행된다.
unload	클립이 스테이지에서 없어진 후 첫 번째 프레임에서 부모 타임라인의 코드가 실행되기 전에 실행된다.
enterFrame	클립이 스테이지에 있는 동안 두 번째 이후의 모든 프레임에서 실행된다. 부모 타임라인의 코드나 클립 내부 코드가 실행되기 전에 실행된다.
data	클립에서 데이터를 받으면 어떤 프레임에서라도 실행된다. data 이벤트가 발생하면 클립 내부 코드나 enterFrame 코드가 실행되기 전에 실행된다.

〔표 10-3〕에 나온 규칙은 실질적인 예제와 함께 알아보면 더 쉽게 이해할 수 있다. 메인 타임라인에 네 개의 키프레임이 있는 한 개의 레이어를 가지고 있는 무비가 있다고 가정하자. 각 키프레임에 코드를 추가하고 1번 프레임부터 3번 프레임에 걸쳐서 무비 클립이 들어가는 두 번째 레이어를 만든다(4번 프레임은 포함하지 않는

다). 그리고 이 클립에 load, enterFrame, unload 핸들러를 추가한다. 마지막으로 클립 안에 각각 코드 블록을 포함하고 있는 세 개의 키프레임을 추가한다. 이러한 무비를 만들면 〔그림 10-2〕와 같은 모양이 된다.



〔그림 10-2〕 코드 실행순서를 테스트하기 위한 무비

무비를 재생하면 다음과 같은 순서로 실행된다.

```
========1번 프레임=======
1) 메인 타임라인 코드가 실행된다.
2) load 핸들러가 실행된다.
3) 1번 프레임의 클립 내부 코드가 실행된다.

========2번 프레임=======
1) enterFrame 핸들러가 실행된다.
2) 2번 프레임의 클립 내부 코드가 실행된다.
3) 메인 타임라인 코드가 실행된다.

========3번 프레임=======
1) enterFrame 핸들러가 실행된다.
2) 3번 프레임의 클립 내부 코드가 실행된다.
3) 메인 타임라인 코드가 실행된다.
```

```
========4번 프레임=======
1) unload 핸들러가 실행된다.
2) 메인 타임라인 코드가 실행된다.
```

이 무비 예제의 실행 순서를 보면 이벤트 핸들러를 이용한 코드를 만들 때 지켜야 할 중요한 규칙을 이해할 수 있다.

- load 핸들러에 있는 코드는 내부 클립 코드보다 먼저 실행되므로 어떤 클립의 1번 프레임에서 바로 사용할 변수를 초기화할 때는 load 핸들러를 사용할 수 있다.
- 프레임에 무비 클립의 인스턴스가 만들어지기 전에 그 프레임에 있는 코드가 실행된다. 따라서 어떤 클립 다음에 나오는 프레임이 처음으로 스테이지에 나타나기 전에는 그러한 변수를 클립의 load 핸들러에서 선언한 경우에도 무비 클립에 있는 사용자 정의 변수나 함수를 그 부모 타임라인에 있는 코드에서는 사용할 수 없다.
- enterFrame 이벤트는 load나 unload 이벤트와는 달리 같은 프레임에서 두 번 이상 발생하지 않는다. 어떤 클립이 스테이지에 나타났다가 바로 사라지는 경우에는 load 또는 unload 이벤트를 대신 사용해야 한다.
- 각 프레임에서 클립의 enterFrame 핸들러에 있는 코드는 클립의 부모 타임라인에 있는 코드가 실행되기 전에 실행된다. 따라서 enterFrame 핸들러를 이용하면 하나의 프레임에서 그 클립의 부모 타임라인의 속성을 변경하고 새로운 값을 그 타임라인의 코드에서 곧장 사용할 수 있다.

클립 이벤트 핸들러 복사

duplicateMovieClip() 함수를 이용하여 무비 클립을 복사하면 무비 클립 이벤트 핸들러도 함께 복사된다. 예를 들어 다음과 같이 load 이벤트 핸들러를 가지고 있는 square라는 무비 클립이 스테이지에 있다고 가정하자.

```
onClipEvent (load) {
  trace("movie loaded");
}
```

square를 복사하여 square2라는 클립을 만들면 어떻게 될까?

```
square.duplicateMovieClip("square2", 0);
```

square를 복사할 때 load 핸들러도 square2로 복사되기 때문에, square2가 생기면 load 핸들러가 실행되어 Output 창에 "movie is loaded"라는 내용이 출력된다. 이와 같이 핸들러가 자동으로 복사되는 것을 이용하면 매우 강력한 기능을 가진 재귀 함수를 만들 수 있다. [예제 10-2]에 이러한 기능을 활용한 간단한 예가 나와 있다.

updateAfterEvent를 이용한 화면 갱신

조금 전에 '실행 순서' 절에서 배웠듯이 mouseDown, mouseUp, mouseMove, keyDown, keyUp 이벤트 핸들러는 이벤트가 발생하면 즉시 실행된다. 심지어 프레임을 렌더링하는 도중에 이벤트가 발생하더라도 이벤트 핸들러는 바로 실행된다.

이처럼 즉시 실행되기 때문에 반응성이 뛰어날 수도 있지만 상황에 따라 엉뚱한 결과가 나올 수도 있다. mouseDown, mouseUp, mouseMove, keyDown, keyUp 이벤트 핸들러 때문에 생기는 시각적인 결과는 다음 프레임을 렌더링하기 전까지는 렌더링되지 않는다. 이러한 사실을 확인해 보기 위해 프레임 속도가 초당 1 프레임인 한 프레임짜리 무비를 만들어서 다음과 같은 코드가 들어 있는 무비 클립을 스테이지에 넣어 보자.

```
onClipEvent (mouseDown) {
  _x += 2;
}
```

그리고 나서 무비를 재생시키고 최대한 빨리 마우스를 클릭해보자. 사용자가 클릭한 것이 모두 기록되긴 하지만 무비 클립은 1초에 한 번씩만 움직인다는 것을 알 수 있다. 따라서 프레임 사이에서 여섯 번을 클릭하면 다음 프레임이 화면에 나타날 때 클립이 한꺼번에 12픽셀을 움직이고, 세 번 클릭한다면 한꺼번에 6픽셀을 움직인다. mouseDown 이벤트 핸들러가 실행될 때마다 그 내용은 기록이 되지만 다음 프레임이 렌더링되어야 그 결과가 화면에 출력된다. 이러한 특성 때문에 원하는 움직임을 제대로 구현하지 못할 수도 있다.

다행히도 다음 프레임이 나올 때까지 기다리지 않고도 사용자 입력 이벤트 핸들러에서 일어나는 시각적인 변화를 바로 렌더링할 수 있는 방법이 있다. 그냥 다음과 같이 이벤트 핸들러 안에서 updateAfterEvent() 함수를 사용하면 된다.

```
onClipEvent (mouseDown) {
  _x += 2;
  updateAfterEvent();
}
```

updateAfterEvent() 함수는 mouseDown, mouseUp, mouseMove, keyDown, keyUp 이벤트 핸들러에서만 사용할 수 있다. 사용자 입력과 관련된 부드럽고 즉각적인 시각 효과를 만들 때에는 이 함수가 필수적이다. [예제 10-8]에서는 사용자 정의 포인터가 부드럽게 움직일 수 있도록 updateAfterEvent()를 이용한다. 하지만 버튼 이벤트에서는 updateAfterEvent() 함수를 사용하지 않아도 된다. 버튼은 프레임 중간에서도 원래 업데이트되기 때문이다.

코드 재사용

버튼 이벤트와 무비 클립 이벤트를 사용할 때는 9장에서 배운 코드 집중화 원칙을 잊지 말도록 하자. 언제나 불필요하게 무비의 여러 요소에서 코드를 중복하여 만들거나 코드가 섞이지 않도록 주의해야 한다. 프로그래밍을 하다가 두 개 이상의 이벤트 핸들러에 똑같은 코드를 입력한다면 그 코드를 객체에 바로 집어넣지 않는 것이 좋다. 코드를 일반화시켜 객체에서 빼내고 그 코드를 무비 코드를 모아두는 다른 곳에 넣는 것이 좋다(보통 메인 타임라인에 코드를 넣는 것이 좋다).

버튼이나 클립 핸들러에 선언문을 직접 집어넣는 것은 그다지 좋은 방법이 아니다. 코드를 함수로 만들고 그 함수를 핸들러에서 부르면 코드를 재사용하기도 좋고 나중에 그 코드를 찾을 때도 훨씬 편리하다. 특히 버튼의 경우에는 이렇게 하는 것이 매우 유용하다(필자는 버튼에 함수 호출 선언문 외에는 다른 선언문을 거의 집어넣지 않는다). 무비 클립의 경우에는 조금 더 신중하게 판단해야 한다. 상황에 따라 클립에 코드를 직접 집어넣으면 더 깔끔하고 완벽한 코드 구조를 만드는 데 도움이 될 수도 있기 때문이다. 자신의 상황과 능력에 맞게 적당한 균형을 찾을 때까지 다양한 접근법을 시도해 보는 것이 좋다. 코드 중복 및 재사용 문제를 염두에 두고 프로그래밍을 하면 큰 도움이 될 것이다.

코드를 직접 버튼에 추가하는 것과 버튼에서 함수를 호출하는 것 사이의 차이점은 9장의 '코드 집중화' 예제에서 쉽게 알 수 있다.

동적 무비 클립 이벤트 핸들러

이 장의 맨 앞부분에서 플래시에는 두 가지 종류의 이벤트(무비 클립이나 버튼에 들어가는 이벤트와 XML이나 XMLSocket과 같은 다른 데이터 객체에 들어가는 이벤트)가 있다는 것을 배웠다. 데이터 객체에 대한 이벤트 핸들러를 만들려면 핸들러 함수 이름을 객체의 속성에 대입해야 한다. 함수를 동적으로 추가하는 방법을 다시 떠올려 보자.

```
myXMLDoc.onLoad = function () { trace("all done loading!"); };
```

동적으로 함수를 대입하면 무비를 재생하는 동안 핸들러의 기능을 변경할 수 있다. 핸들러의 기능을 변경하려면 다음과 같이 핸들러 속성에 새로운 함수를 대입하기만 하면 된다.

```
myXMLDoc.onLoad = function () { gotoAndPlay("displayData"); };
```

또는 다음과 같이 핸들러를 없앨 수도 있다.

```
myXMLDoc.onLoad = function () { return; };
```

하지만 안타깝게도 무비 클립이나 버튼의 이벤트는 그다지 융통성이 없어서 무비를 재생하는 도중에 변경하거나 제거할 수 없다. 게다가 무비 클립 이벤트 핸들러는 어떤 무비의 메인 무비 타임라인에 추가할 수 없다. 즉 무비의 _root 클립에는 이벤트 핸들러를 추가할 수가 없다.

이러한 한계를 극복하기 위해(enterFrame 이벤트와 사용자 입력 이벤트의 경우에) 동적으로 이벤트 핸들러를 제거하거나 변경하는 것과 비슷한 효과를 볼 수 있도록 비어있는 무비 클립을 이용한다. 비어있는 무비 클립을 이용하면 _root 레벨의 이벤트와 비슷한 기능을 구현할 수 있다. 이러한 테크닉은 사실 8장에서 다음과 같은 방법으로 이벤트 루프를 만들 때 이미 배웠다.

1. process라는 이름의 비어있는 무비 클립을 만든다.
2. process 안에 eventClip이라는 비어있는 클립을 하나 더 만든다.
3. eventClip에 원하는 이벤트 핸들러를 추가한다. eventClip의 핸들러에 있는 코드에서는 다음과 같이 process 클립의 호스트 타임라인을 대상으로 잡아야 한다.

```
onClipEvent (mouseMove) {
  _ parent._ parent.doSomeFunction();
}
```

4. process를 attachMovie() 함수에서 사용할 수 있도록 Library에서 process를 선택하고 Options → Linkage를 선택한다. Linkage를 Export This Symbol로 설정하고 적당한 인식자(예: "mouseMoveProcess")를 할당한다.
5. 마지막으로 이벤트 핸들러와 연결하려면 attachMovie()를 이용하여 process 클립을 적당한 타임라인에 추가한다.
6. 핸들러와 이벤트의 연결을 끊고 싶다면 removeMovieClip() 함수를 이용하여 process 클립을 제거한다.

enterFrame 이벤트에서 이러한 기법을 사용하는 법은 8장의 '플래시 5 클립 이벤트 루프'에 자세히 설명되어 있다.

이벤트 핸들러 응용

몇 가지 실전 예제를 통해 액션스크립트 이벤트와 이벤트 핸들러에 관한 설명을 마칠까 한다. 여기에 나오는 애플리케이션은 매우 간단하긴 하지만, 이 예제들을 통해 이벤트 기반 프로그래밍의 폭넓은 응용범위에 대해 감을 잡을 수 있을 것이다. 마지막 두 예제는 온라인 코드 창고에서 다운로드할 수 있다.

〔예제 10-7〕은 클립을 줄였다 키웠다 하는 예제이다.

〔예제 10-7〕 무비 클립 크기를 변동시키는 예제

```
onClipEvent (load) {
  var shrinking = false;
  var maxHeight = 300;
  var minHeight = 30;
}

onClipEvent (enterFrame) {
  if (_height < maxHeight && shrinking == false) {
    _height += 10;
    _width  += 10;
  } else {
    shrinking = true;
  }

  if (shrinking == true) {
    if (_height > minHeight) {
      _height -= 10;
      _width  -= 10;
    } else {
      shrinking = false;
      _height += 10;      // 한 사이클을 건너뛰지 않도록
      _width  += 10;      // 값을 증가시킨다.
    }
  }
}
```

〔예제 10-8〕에서는 보통 쓰이는 시스템 포인터를 숨기고 마우스의 위치를 클립이 따라다니도록 만들어서 사용자 정의 마우스 포인터와 같은 효과를 보여준다. 이 예제에서 mouseDown과 mouseUp 핸들러에서는 마우스 포인터의 크기를 약간 바꿔서 마우스가 클릭되는 것을 알려준다.

〔예제 10-8〕 사용자 정의 마우스 포인터

```
onClipEvent (load) {
  Mouse.hide();
}

onClipEvent (mouseMove) {
  _x = _root._xmouse;
```

```
  _ y = _root._ymouse;
  updateAfterEvent();
}

onClipEvent (mouseDown) {
  _width  *= .5;
  _height *= .5;
  updateAfterEvent();
}

onClipEvent (mouseUp) {
  _width  *= 2;
  _height *= 2;
  updateAfterEvent();
}
```

마지막으로 〔예제 10-9〕는 액션스크립트 무비 클립 이벤트 핸들러의 강력한 기능을 보여주는 예로, 무비 클립을 사용자 정의 버튼으로 바꾸는 프로그램이다. 여기서는 mouseMove를 이용하여 마우스가 움직이는 것을 확인하고 mouseDown과 mouseUp을 이용하여 버튼을 클릭하는 것을 감지하고 hitTest() 함수를 이용하여 간단하게 마우스의 위치를 파악한다. 이 예제에서는 핸들러를 포함하고 있는 클립에 up, down, over(각각 버튼이 눌리지 않은 상태, 눌린 상태, 버튼 위에 마우스 포인터가 있는 상태를 나타냄)라는 레이블을 가진 세 개의 키프레임이 있다고 가정한다.

〔예제 10-9〕 무비 클립 버튼

```
onClipEvent (load) {
  stop();
}

onClipEvent (mouseMove) {
  if (hitTest(_root._xmouse, _root._ymouse, true) && !buttonDown) {
    this.gotoAndStop("over");
  } else if (!hitTest(_root._xmouse, _root._ymouse, true) && !buttonDown)
{
    this.gotoAndStop("up");
  }
  updateAfterEvent();
}
```

```
onClipEvent (mouseDown) {
  if (hitTest(_root._xmouse, _root._ymouse, true)) {
    buttonDown = true;
    this.gotoAndStop("down");
  }
  updateAfterEvent();
}

onClipEvent (mouseUp) {
  buttonDown = false;
  if (!hitTest(_root._xmouse, _root._ymouse, true)) {
    this.gotoAndStop("up");
  } else {
    this.gotoAndStop("over");
  }
        {_Index  {_EndRange_}event  handlers:applications  of_}
updateAfterEvent();
}
```

앞으로 배울 내용

선언문, 연산자, 함수, 그리고 이벤트 및 이벤트 핸들러까지 배웠으니 이제 액션스크립트의 내부 도구 작동법을 모두 배운 셈이다. 액션스크립트를 완벽하게 이해할 수 있도록 앞으로 세 장에 걸쳐 아주 중요한 데이터형인 배열, 객체, 무비 클립에 대해 살펴볼 것이다.