19

디버깅

지금까지 여러 상황에서 사용할 수 있는 테크닉과 문법에 대해 배웠다. 하지만 액션스크립트 프로그램을 짜다 보면 수많은 오류를 접하게 마련이다(특히 처음에는 문법이나 개념 면에서 부족한 점이 많기 때문에 버그가 많이 생긴다). 그렇다고 슬퍼할 필요는 없다. 숙련된 프로그래머들도 디버깅(잘못된 코드를 고치는 작업)을 하는 데 적지 않은 시간을 투자하기 때문이다.

처음에는 버그를 찾아내기 위해 만들어낸 무비를 테스트하는 것이 중요하다. 즉 여러 종류와 여러 버전의 브라우저에서, 그리고 그 무비를 재생시킬 모든 플랫폼에서 무비를 테스트해 보아야 한다. 윈도우 95/98/ME, NT/2000이나 XP와 같은 다양한 버전에서, 그리고 가능하다면 예전 버전의 플래시 플러그인에서도 테스트해보는 것이 좋다. 예전에 나온 플래시 플레이어는 아래 링크에서 구할 수 있다.

http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm

프로그램 테스트나 품질 보장(QA, quality assurance)과 같은 주제는 이 책의 범위를 넘어서므로 자세히 다루지는 않겠다. 다만 테스트 및 QA 절차를 제대로 처리하고 어떤 오류가 있다면 사용자로부터 자세한 버그 리포트(플랫폼, 브라우저 버전, 플래시 플러그인 버전, 오류가 일어나는 과정에 대한 자세한 설명 등)를 받아서 오류가

발생하는 상황을 똑같이 재현할 수 있도록 만들 수 있는 체제를 구축하는 것이 필요하다는 점만 알아두자. 버그 리포팅은 버그를 고치기 위한 첫 번째 조건이다.

디버깅은 프로그래밍에서 필수불가결한 요소이며, 뛰어난 프로그래머와 초보 프로그래머는 디버깅 실력에서 가장 큰 차이가 난다. 초보자들은 처음에 있던 버그가 어쩌다가 없어지면 그냥 기뻐할 뿐이다. 하지만 숙련된 프로그래머들은 그러한 버그가 언젠가는 다시 나타날 것을 미리 알고, 가끔 일어나는 버그라도 자세히 파헤쳐 본다. 초보 프로그래머들은 오류 메시지가 나오면 당황해서 의욕을 상실하는 경향이 있지만, 숙련된 프로그래머들은 오류 메시지를 자세히 살펴보고 자주 발생하는 오류일수록 고치기 쉽다는 것을 안다.

디버깅을 제대로 하려면 논리적으로 확실하게 버그를 인식하고 문제 해결 도구를 제대로 이해해야 한다. 이 장에서는 디버깅 도구를 다루는 기본적인 방법과 코드 문제를 해결하는 일반적인 테크닉에 대해 알아보자. 디버깅은 여러 가지 가설을 세우고 그러한 가설을 조직적으로 검증하는 작업이라는 점을 명심하자. 보통 어떤 문제가 생기면 그 문제의 원인이 또 다른 문제에 있는 경우가 많다. 디버깅을 할 때는 디버깅 도구를 이용하여 모든 것이 처음 의도대로 잘 작동하는지 확인해야 한다. 그렇게 하지 않으면 버그가 일어나는 상황을 파악하고 문제를 해결하는 것이 불가능하다.

디버깅 도구

액션스크립트에서는 다음과 같은 디버깅 도구를 활용할 수 있다.

- trace( ) 함수
- List Variables 명령
- List Objects 명령
- 대역폭 프로파일러
- 디버거

이 도구는 모두 테스트 무비 모드에서 사용할 수 있다. 테스트 무비 모드에 들어가려면 저작 도구에서 Control → Test Movie를 사용하여 무비를 실행하면 된다.

위와 같은 디버깅 도구를 사용하지 않더라도 플래시에서 무비를 내보내거나 Check Syntax를 수행하면 Output 창에서 오류 메시지가 출력된다(Check Syntax는 액션 패널의 오른쪽 위에 있는 화살표 버튼에 있다). 오류 메시지에서는 경우에 따라 오류의 원인을 정확하게 찾아내서 소스 코드 블록에서 문제가 되는 행의 번호까지 출력해 주기도 한다. 오류 메시지에 대한 자세한 설명은 매크로미디어에서 나온 액션스크립트 레퍼런스 가이드에 나와 있다.

하지만 모든 버그가 오류 메시지로 출력되는 것은 아니다. 예를 들어 숫자를 계산했을 때 엉뚱한 결과가 나오는 경우에는 무비가 다운되지는 않지만 이것도 결국은 버그에 해당한다. 오류 메시지에는 두 가지 종류가 있다. 스크립트를 내보낼 때(즉 무비로 저장할 때) 생기는 컴파일 오류 메시지, 플래시 무비를 실행시킬 때 오류가 생기면 나오는 런타임 오류 메시지가 있다. 컴파일 오류는 괄호를 빼먹거나 따옴표를 제대로 적어주지 않는 것과 같이 문법적인 오류가 있을 때 생긴다. 각 명령어별 문법은 레퍼런스에 나와 있고, 액션스크립트의 문법에 대한 전반적인 설명은 '14장. 렉시컬 구조'에서 찾을 수 있다.

런타임 오류의 유형은 매우 다양하며 실행중인 코드에 문제가 있는 경우보다는 그 전에 했던 작업에 다른 문제가 있어서 생기는 경우가 많다. 예를 들어 loadVariables()를 통해 웹 서버로부터 받은 데이터를 사용하는 경우를 생각해 보자. 그 명령어를 처리한 펄 스크립트에서 엉뚱한 데이터를 보내거나 데이터의 형식이 잘못된 경우에는 펄 스크립트를 수정해야 한다. 플래시 스크립트에는 전혀 문제가 없지만 잘못된 값이 들어오기 때문에 문제가 생길 수 있다.

방어적인 프로그래밍 습관을 기른다면 이런 문제를 어느 정도 예방할 수 있다. 문제가 일어날 수 있는 상황을 항상 미리 확인한다면 오류가 생길 가능성을 크게 줄일 수 있다. 이러한 작업을 오류 검사라고 한다(입력 데이터를 확인하는 과정은 데이터 검증이라고 부른다). 예를 들어 퀴즈 문제를 화면에 출력하기 전에 문제가 제대로 로딩되었는지 확인할 수 있다. 또한 각 문제가 화면에 출력할 수 있는 형식으로 되어있는지 미리 확인할 수도 있다. 엉뚱한 데이터가 들어왔다면 오류 메시지를 출력하여 프로그래머 혹은 사용자가 적절한 조치를 취할 수 있도록 하는 것이 좋다.

trace() 함수

액션스크립트에서 버그의 원인을 찾아내는 데 가장 효과적인 도구로 trace() 함수를 들 수 있다. 지금까지 수도 없이 사용했던 이 trace() 함수는 테스트 무비 모드에서 Output 창에 어떤 표현식의 값을 출력하는 매우 간단한 함수이다. 예를 들어 무비에 다음과 같은 코드를 추가해 보자.

```
trace("hello world"3);
```

"hello world"라는 텍스트가 Output 창에 출력된다. 또한 trace() 함수를 이용하여 다음과 같이 변수 값을 확인할 수도 있다.

```
var x = 5;
trace(x);  // Output 창에 5가 출력된다.
```

trace() 함수를 사용하면 변수, 속성이나 객체의 상태를 조사할 수 있고 코드가 얼마나 실행되었는지도 확인할 수 있다. 스크립트의 작업 결과를 일일이 확인하면 어디서 문제가 생겼는지 쉽게 찾아낼 수 있다. 예를 들어 어떤 함수에서 어떤 값을 리턴해야 하는데, trace() 명령어로 확인해 보면 리턴 값이 undefined인 경우도 있다(물론 이런 경우에는 Output 창에 아무것도 출력되지 않는다). 이럴 때는 그 함수를 다시 확인하고 return 명령어를 이용하여 제대로 된 값을 리턴하는지 확인하면 버그를 잡아낼 수 있다.

List Variables 명령

테스트 무비 모드에서 무비를 실행시킬 때 Debug → List Variables 명령을 이용하면 무비에 있는 변수 값을 확인할 수 있다. List Variables를 이용하면 현재 무비에서 활성화되어 있는 변수의 이름과 위치를 알 수 있다. 함수와 무비 클립도 변수에 저장되므로 List Variables 명령을 이용하면 무비에 있는 함수와 무비 클립도 확인할 수 있다.

List Variables 명령을 사용하면 〔예제 19-1〕에 나온 것과 같은 내용이 출력된다. 예를 보면 rate라는 변수가 선언되긴 했지만 그 값이 여전히 undefined라는 것을 알 수 있다. 이런 내용은 trace() 함수로는 알아내기 힘들다. trace()로는 undefined

값을 비어있는 문자열("")로 변환하기 때문에, 비어있는 문자열과 undefined를 구분할 수 없다.

〔예제 19-1〕 List Variables 출력 결과

```
Level #0:
   Variable _level0.$version = "WIN 5,0,30,0"
   Variable _level0.calcDist = [function]
   Variable _level0.deltaX = 194
   Variable _level0.deltaY = 179
   Variable _level0.rate = undefined
   Variable _level0.dist = 264
Movie Clip:  Target="_level0.clip1"
Movie Clip:  Target="_level0.clip2"
```

trace() 함수와 List Variables 명령을 이용하더라도 특정 시간에서의 변수 값만 확인할 수 있다. 때때로 무비가 진행됨에 따라 변수 값이 어떻게 바뀌는지 확인해야 하는 경우도 있고 변수 값을 반복해서 확인해야 하는 경우도 있다. 이런 경우에는 디버거를 이용하면 된다.

List Objects 명령

List Objects 명령을 사용하면 무비 안에서 정의된 텍스트, 도형, 그래픽, 무비 클립의 목록이 생성된다. 테스트 무비 모드에서 Debug → List Objects를 선택하면 List Objects 명령이 실행된다. List Objects 명령의 결과에는 프로그램 안에 있는 데이터 객체(클래스의 인스턴스)의 목록은 나타나지 않는다는 점에 주의하자. 데이터 객체는 List Variables 명령으로 확인할 수 있다.

〔예제 19-2〕에는 List Objects 명령을 실행시킨 결과의 예가 나와 있다. 편집할 수 있는 텍스트 필드도 따로 일목요연하게 표시되고, 자동으로 이름이 정해진 무비 클립 인스턴스의 이름도 여기서 확인할 수 있다(예: _level0.instance1).

〔예제 19-2〕 List Objects 출력 결과

```
Level #0: Frame=1
   Shape:
   Text: Value = "variables functions clip events startDrag stopDrag Math"
   Text: Value = "this movie demonstrates a little math, variables, movie
clip events"
   Text: Value = "draggable distance"
   Text: Value = "calculator"
   Movie Clip: Frame=1 Target="_level0.instance1"
      Shape:
   Text: Value = "distance between clipstotal:horizontal:vertical:"
   Edit Text: Variable=_level0.dist Text="222"
   Edit Text: Variable=_level0.deltaX Text="174"
   Edit Text: Variable=_level0.deltaY Text="138"
   Movie Clip: Frame=1 Target="_level0.obj1"
      Shape:
   Movie Clip: Frame=1 Target="_level0.obj2"
      Shape:
```

List Objects 명령을 사용할 때도 특정한 순간에 해당하는 값만 알아낼 수 있다. 현재 객체에 저장된 값을 알아내고 싶다면 그때마다 명령을 새로 실행시켜야 한다.

대역폭 프로파일러

대역폭 프로파일러는 다양한 속도로 무비 다운로딩을 시뮬레이션하는 데 쓰인다. 대역폭 프로파일러를 이용하여 무비의 성능을 확인하거나 프리로딩 코드를 테스트하거나 무비를 재생하는 도중에 메인 무비 플레이헤드의 위치를 추적할 수 있다. 대역폭 프로파일러는 다음과 같이 작동시킬 수 있다.

1. 테스트 무비 모드에서 View → Bandwidth Profiler를 선택한다.
2. Debug 메뉴에서 원하는 다운로드 속도를 선택한다.
3. View → Show Streaming을 선택하면 시뮬레이션이 시작된다.

플래시의 성능은 사용하는 구성요소나 플레이어에서 처리해야 하는 렌더링의 복잡도와 같은 다양한 요인에 의해 영향을 받는다. 예를 들어 커다란 비트맵 파일을 사용하거나 곡선이 많이 있는 복잡한 도형을 렌더링한다거나 알파 채널을 과도하게 사용하는 경우에 성능이 크게 떨어질 수 있다. 구성요소를 다운로드하거나 렌더링

에 걸리는 시간 때문에 대역폭이나 액션스크립트에서 실행하는 데 걸리는 프로세서 시간을 크게 잡아먹을 수도 있다. 또한 액션스크립트는 일반적으로 C와 같은 컴파일러 언어보다 훨씬 느리다.

액션스크립트에서는 데이터 업로딩, 다운로딩에 관련된 작업과 여러 번 반복해야 하는 작업(큰 배열을 확인하는 것과 같은 작업)을 처리할 때 가장 시간이 많이 소요된다.

액션스크립트에서 Output 창에 어떤 내용을 출력하는 작업은 시간이 꽤 오래 걸리는 편이다. 무비 자체는 상당히 간단한데도 실행 속도가 너무 느리다는 느낌이 들면 trace() 선언문을 모두 주석처리하거나 테스트 무비 모드가 아닌 일반 모드에서 무비를 재생해 보는 것이 좋다.

최적화된 코드를 작성하는 방법에 대한 내용은 이 책의 주제를 넘어서므로 다음과 같은 간단한 팁만 짚고 넘어가도록 하자.

- 어떤 작업을 루프 밖에서 처리해도 기능에 별 차이가 없다면 굳이 루프 안에서 그 작업을 처리하지 않는 것이 좋다.
- 이벤트는 루프 안에서 처리하지 말아야 한다. 이벤트가 일어날 때까지 얼마나 기다려야 할지도 잘 모르고 그 이벤트가 아예 일어나지 않을 수도 있는데, 그러한 경우에는 프로그램 속도가 엄청나게 느려지거나 아예 프로그램이 멎어버릴 수도 있다. 루프 안에서 이벤트를 처리하는 대신 on (load)와 같은 이벤트 핸들러를 이용하여 이벤트를 처리해야 한다.
- 가능하다면 코드를 일반화시키자(스마트 클립을 만드는 것도 괜찮다). 이렇게 하면 다운로드 용량을 줄일 수 있다. 예를 들어 거의 똑같은 내용으로 이루어진 5킬로바이트짜리 코드를 두 번 적어주는 대신 일반화된 루틴을 하나 만들고 서로 다른 매개변수로 두 번 호출하면, 다운로드해야 할 파일 크기를 5킬로바이트 줄일 수 있다(코드를 일반화하는 방법은 '9장. 함수'와 '16장. 액션스크립트 저작 환경'의 스마트 클립을 설명하는 부분에서 자세히 다루고 있다).
- 플래시 5 플레이어를 사용하더라도 액션스크립트의 성능을 최적화시켜야 한다면 플래시 5 문법 대신 플래시 4 스타일의 문법을 사용해 보자. 플래시

4 문법을 사용하면 플래시 5 문법을 사용하는 것보다 속도가 더 빨라질 수도 있다. 예를 들어 플래시 4의 substring() 함수는 플래시 5의 substring()이나 substr() 메소드에 비해 더 빠르다. 또한 플래시 5의 점 표기법 대신 플래시 4의 Tell Target을 사용하면 속도를 어느 정도 향상시킬 수 있다.

- File → Publish Settings → Flash → Options → Omit Trace Actions를 선택하여 무비를 저장할 때 trace() 선언문을 자동으로 생략하도록 해놓으면 속도가 빨라진다.
- 무비에 원래 있던 클립을 움직이는 것보다는 새로운 클립을 추가하거나 원래 있던 클립을 제거하는 작업이 더 느리다. 가능하다면 무비의 자원을 재활용하자.
- 일반적인 플래시 최적화 기법은 http://www.macromedia.com/support/flash/publishexport/stream_optimize/stream_optimize.html에 자세히 나와 있다.

디버거

디버거를 이용하면 무비에 있는 속성, 객체, 변수 값을 체계적으로 액세스하거나 실행 중에 변수의 값을 바꿀 수 있기 때문에 디버깅할 때 매우 중요한 도구로 쓰인다.

디버거를 사용하려면 플래시 저작 도구에서(테스트 무비 모드가 아닌 상태에서) Control → Debug Movie를 선택하면 된다. 다음과 같은 조건이 만족된다면 웹 브라우저에서도 디버거를 사용할 수 있다.

- 무비를 만든 사람이 그 무비를 저장할 때 디버깅 기능을 포함시켜 저장한 경우
- 무비를 실행시키는 데 사용한 플레이어가 디버깅 가능한 플레이어인 경우
- 디버깅할 때 플래시 저작 도구가 실행 중인 경우

브라우저에서 디버깅할 수 있는 기능을 포함시켜 무비를 저장할 때는 File → Publish Settings → Flash → Debugging Permitted를 선택하면 된다. 비밀번호를 설정해 두면 비밀 번호를 입력해야만 디버깅 기능을 이용할 수 있다. 디버깅이 가능한 플레이어를 웹 브라우저에 설치하려면 플래시를 설치할 때 /Players/Debug/ 폴더에

저장되는 인스톨 프로그램을 사용하면 된다. 무비를 볼 때 디버깅 기능을 활성화시키려면 무비에서 오른쪽 버튼을 클릭(매킨토시에서는 Ctrl-클릭)한 후 Debugger를 선택하면 된다.

모든 버전의 플래시 플레이어에 디버깅 플레이어가 있는 것은 아니다. 디버깅 플레이어의 최신 버전은 http://www.macromedia.com/support/flash에서 찾을 수 있다.

디버거의 상단부를 차지하는 디스플레이 목록에는 무비 클립 계층이 나온다. 특정 무비 클립의 속성과 변수를 조사하려면 디스플레이 목록에서 원하는 무비 클립을 선택하면 된다. 디버거의 하단부에는 선택한 클립의 속성과 변수를 자동으로 업데이트하면서 보여주는 Properties, Variables, Watch라는 세 개의 탭이 있다. 어떤 속성이나 변수 값을 설정하려면, 그 값을 더블클릭하고 새로운 데이터를 입력하면 된다. 몇 개의 아이템만 따로 모아서 보고 싶다면, Properties나 Variables 탭에서 아이템을 선택하고 디버거의 오른쪽 위에 있는 화살표 버튼에서 Add Watch를 선택하면 된다. 선택한 모든 변수와 속성은 Watch 탭에 추가된다(이렇게 하면 서로 다른 무비 클립의 변수를 동시에 조사할 수 있다).

플래시 디버거를 사용하는 방법에 대한 상세한 정보를 원한다면 액션스크립트 레퍼런스 가이드에서 'Troubleshooting ActionsScript' 부분을 참조하기 바란다. 레퍼런스 가이드 책이 없다면 http://www.macromedia.com/support/flash 또는 플래시 저작도구의 Help 메뉴에서도 레퍼런스 가이드의 내용을 볼 수 있다.

디버깅 방법론

코드 디버깅과 관련된 몇 가지 기법을 간략하게 알아보자. 디버깅은 크게 세 단계로 구분할 수 있다.

- 문제를 확인하고 그러한 문제가 일어나는 상황을 알아내는 단계
- 문제의 원인을 찾아내는 단계
- 문제를 해결하는 단계

버그 확인

프로그래밍 도중에 코드에서 문제를 발견하는 경우도 많이 있다. 코드를 만들어서 무비를 테스트했을 때 바로 무비가 제대로 작동하지 않는다는 것을 발견하는 경우에는 바로 문제점을 찾아서 해결해야 한다.

버그는 일찍 발견할수록 좋다. 사실 코드를 짠다는 것은 계속해서 코드를 작성하고 그 코드를 테스트하는 작업이라고 할 수 있다. 코드를 몇 줄 작성하고 무비를 내보내고 그 코드가 제대로 작동하는지 확인하고 또 코드를 만들고 다시 테스트하는 작업을 반복해야 한다. 프로그램 전체를 테스트해보기 전에 프로그램의 각 구성요소가 제대로 작동하는지 먼저 확인해야 한다. 복잡한 코드를 만들 때도 틈틈이 코드를 테스트하는 작업을 빼먹지 말아야 한다.

자기 눈에 버그가 보이지 않는다고 해서 자신의 프로그램이 완벽하다고 생각하면 큰 오산이다. 프로젝트를 계획할 때는 다른 사용자들이 테스트하는 데 걸리는 시간을 꼭 생각해보아야 한다. 특히 고객에게 제공할 프로그램이나 돈을 받고 판매할 목적으로 프로그램을 만드는 경우에는 이러한 점을 무시할 수 없다. 앞에서 말했던 것처럼 잘못된 데이터가 들어왔을 때 생길 수 있는 문제를 미연에 방지할 수 있는 오류 검사 코드를 만드는 것도 잊지 말자. 예를 들어 정수 인자가 들어와야 하는 함수를 만드는 경우에는 typeof 연산자를 이용하여 입력된 매개변수의 데이터형이 올바른지 확인하는 것이 좋다. 또한 아주 큰 값, 작은 값, 음수, 0과 같은 극한 상황도 일일이 테스트해보는 것이 좋다.

어떤 문제를 찾았을 때 그러한 문제를 똑같이 재현할 수 있는 최단 과정을 찾아내는 것도 매우 중요한 일이다. '프로그램을 한 시간 정도 돌렸는데 다운되었습니다' 같은 버그 리포트는 문제를 해결하는 데 거의 도움이 되지 않는다. 유용한 버그 리포트라면 아래 버그 리포트의 예와 같이 오류를 똑같이 재현할 수 있는 최단 과정이 포함되어야 한다.

1. 연도에 0을 입력한다.
2. Calculate 버튼을 클릭한다.
3. 결과 필드에 가격이 아닌 "NaN"이 출력된다.

버그의 원인 찾아내기

일단 버그를 확인하고 나면 본격적으로 버그를 해결하기 위한 작업을 시작해야 한다. 무엇보다도 버그의 원인을 먼저 찾아야 하는데, 이 작업을 하다 보면 정말 한참을 거슬러 올라가야 하는 경우도 있다. 버그는 수십 년 동안 잘못된 식습관 때문에 생기는 심장 마비와 같다. 심장 마비는 가장 눈에 띄게 드러나는 증상이긴 하지만 뭔가 그 이전에 잘못된 문제들을 먼저 해결해야 한다. 대부분의 버그는 잘못된 가정 때문에 생긴다. 변수의 이름을 제대로 입력했다고 생각할지 모르지만 그렇지 않은 경우도 많이 있고, 텍스트 필드에 숫자 데이터가 저장된다고 생각했는데 그렇지 않을 수도 있다. 여러 곳에서 trace() 선언문을 사용해 보거나 디버거를 사용하거나 List Variables 명령을 이용하면 프로그래머가 생각했던 것과 인터프리터에서 실제 코드를 해석하는 방법이 차이가 나는 부분을 찾아낼 수 있다.

아래 코드에는 버그가 있다. status를 "equal"이라고 설정하는데, 이 과정에 문제가 있다.

```
var x = 11;
isTen(x);
function isTen(val) {
  if (val = 10) {
    status = "equal";
  }
}
```

코드의 어느 부분이 잘못되었는지 알아내려면 '코드에서 이런 작업을 해야지'라고 생각한 것과 실제 코드에서 작업을 처리한 결과를 하나씩 차근차근 비교해 보면 된다.

```
// x를 11로 설정해야 한다.
var x = 11;

// 실제로 x 값이 11인지 확인해 보자.
trace(x);     // 11이 제대로 출력된다.

// 여기서는 isTen() 함수를 호출해야 한다.
isTen(x);

// 이제 함수가 시작되는 부분이다.
function isTen(val) {
```

```
// 함수가 제대로 호출되었는지 직접 확인해 보자.
trace("isTen was called");   // "isTen was called"가 제대로 출력된다.

// 매개변수가 제대로 전달되었는지 확인해 보자.
trace("val is " + val);    // "val is 11"이라고 출력되었으므로 괜찮다.
```

여기서 잠깐 생각해 볼 것이 있다. 이제 코드를 거의 다 살펴본 셈인데 지금까지는 모두 예상대로 진행되었다. 변수도 제대로 설정되었고 isTen() 함수도 제대로 호출되었으며 매개변수도 문제 없이 전달받았다.

분명히 실행될 것이라고 생각했던 코드가 실행되지 않기 때문에 생기는 오류도 적지 않다. trace() 선언문을 이용하면 어떤 코드가 실행되는지 확인할 수 있다.

지금까지는 문제가 없었으므로 그 뒤에 있는 if (val = 10)이나 그 뒤에 있는 텍스트 필드 대입 부분인 status = 'equal'에 문제가 있으리라는 것을 쉽게 알 수 있다. 이제 trace()를 이용하여 if문 안에 있는 표현식의 값을 직접 출력해 보자(분명히 true 또는 false가 출력되어야 한다).

```
trace(val = 10);
```

드디어 찾았다. Output 창에는 true도 false도 아닌 10이 출력된다.

이 부분을 살펴보면 if문 안의 조건식이 비교문이 아니라 대입 선언문이라는 점을 알 수 있다. 동치 비교 연산자에서 등호(=)를 하나 빼먹었기 때문에 이러한 버그가 생긴 것이다. if(val = 10)이 아니라 if(val == 10)이라고 적어야 한다.

물론 모든 버그가 방금 예로 든 경우처럼 쉽게 해결되는 것은 아니다(물론 이러한 실수는 프로그래밍을 하다 보면 흔히 하는 실수 중 하나이다). 하지만 더 복잡한 버그도 결국 이러한 방법으로 해결할 수 있다. 곳곳에서 trace() 함수를 호출하여 무비 코드의 진행 상황을 확실히 파악하고 디버거를 활용하면 어떤 복잡한 버그라도 해결할 수 있다.

일반적인 버그의 원인

〔표 19-1〕에 액션스크립트에서 자주 접하게 되는 버그의 유형을 정리해 놓았다.

〔표 19-1〕 액션스크립트에서 자주 범하게 되는 실수

문제점	내용
코드가 엉뚱한 곳에 있는 경우	모든 코드는 무비 클립, 프레임 또는 버튼에 집어넣어야 한다. 액션 패널의 타이틀을 확인하여 코드를 원래 의도한 위치에 입력하고 있는지 알아보자. 프레임에 코드를 추가할 때는 액션 패널의 타이틀에 Frame Actions라고 나오고, 무비 클립이나 버튼에 코드를 추가할 때는 액션 패널 타이틀에 Object Actions라고 나온다. 스크립트를 특정 프레임에 추가해야 한다면 코딩을 시작하기 전에 타임라인에서 프레임을 제대로 선택했는지 확인하고 코드를 삽입하고자 하는 프레임에 키프레임이 있는지도 확인해야 한다. 무비 클립이나 버튼에 코드를 추가할 때는 스테이지에서 객체를 제대로 선택했는지 확인해야 한다. 코드를 어디에 입력했는지 확인하고 싶다면 Movie Explorer(Window → Movie Explorer 선택)를 이용하면 된다.

이벤트 핸들러가 없는 경우	무비 클립이나 버튼에 삽입된 코드는 반드시 어떤 이벤트 핸들러 안에 들어가야 한다. 무비 클립의 경우에는 다음과 같은 이벤트 핸들러를 사용한다.
	`onClipEvent (event) {`
	`  // 선언문`
	`}`
	버튼에서 사용하는 이벤트 핸들러는 다음과 같은 형태로 구성된다.
	`on (event) {`
	`  // 선언문`
	`}`
	event 자리에는 처리할 이벤트의 이름이 들어간다. 이벤트 핸들러가 없으면 "Statement must appear within on handler" 또는 "Statement must appear within onClipEvent handler"라는 오류가 발생한다(10장 참조).

잘못된 무비 클립 레퍼런스	무비에 포함되지 않은 무비 클립을 참조하거나 무비 클립에 대한 레퍼런스의 형식이 잘못된 경우. 모든 인스턴스에 이름이 있는지, 인스턴스 이름이 레퍼런스 이름과 같은지 확인해야 한다. 제대로 된 무비 클립 레퍼런스에 대한 정보는 13장 '다중 인스턴스 참조하기'에 나와 있다.

문제점	내용
예상과 다른 형 변환	데이터 변환 결과가 예상한 것과 다른 경우. 예를 들어 3 + "4"의 연산 결과는 7이 아니라 "34"이다. 또한 "true"라는 문자열을 부울 값으로 변환하면 false가 된다. 3장에 나온 형 변환 규칙을 확실히 익혀두자. 데이터형은 typeof 연산자로 확인할 수 있다.
세미콜론을 빠뜨린 경우	세미콜론을 사용하지 않아서 선언문이 어디에서 끝나는지 알 수 없는 경우. 세미콜론 사용법은 14장에 자세히 설명되어 있다.
따옴표 문제	문자열에 이스케이프 처리를 제대로 하지 않은 따옴표를 넣어서 문자열 리터럴이 제대로 인식되지 않은 경우(4장 '문자열 리터럴' 참조)
텍스트 필드 데이터를 잘못 사용한 경우	텍스트 필드를 문자열이 아닌 숫자 등의 다른 데이터형으로 사용한 경우. 사용자 입력 텍스트 필드는 언제나 문자열 값이며 다른 형으로 사용할 때는 직접 변환해야 한다.
영역 문제	변수, 속성, 클립 또는 함수를 잘못된 영역에서 참조하는 경우. 예를 들어 클립 핸들러에 있는 선언문에서 그 클립의 부모 타임라인에 있는 함수를 호출하려는 경우에 버그가 생긴다. 10장의 '이벤트 핸들러 영역'과 2장의 '변수 영역', 9장의 '함수 사용 범위와 유효 기간' 참조
전역 함수와 메소드를 혼동하는 경우	전역 함수 중에는 무비 클립 메소드와 이름이 같은 것도 있는데, 이로 인해 문제가 생기는 경우도 있다. 13장의 '메소드와 전역 함수가 겹치는 문제' 참조
로딩이 끝나지 않은 경우	로딩이 끝나지 않은 상태에서는 클립, 속성, 함수 또는 변수에 대한 레퍼런스가 제대로 작동하지 않는다. MovieClip._framesloaded 속성을 확인하여 필요한 내용이 모두 로딩되었는지 확인해야 한다.
대소문자 구분	액션스크립트에서도 몇몇 키워드에서는 대소문자를 구분한다. onClipEvent를 실수로 onclipevent라고 쓰면 onClipEvent라는 내장 이벤트 핸들러 키워드가 아닌 onclipevent라는 사용자 정의 함수를 사용하는 것으로 인식한다. 따라서 onClipEvent 선언문 블록의 시작 부분에 있는 { 기호에서 오류가 생긴다(onclipevent라는 함수를 호출하는 경우라면 뒤에 { 대신 세미콜론이 들어가야 하기 때문이다). 14장의 '대문자와 소문자' 참조

버그 고치기

정말 손쉽게 버그를 고칠 수 있는 경우도 있다. 예를 들어 문자열을 따옴표로 감싸지 않아서 오류가 생겼다면 따옴표만 추가하면 버그를 고칠 수 있다.

더 복잡한 프로그램에서는 버그를 고치는 일이 꽤 힘든 작업이 될 수도 있다. 버그를 고치기 힘들다면 다음과 같은 점들을 고려해 보자.

- 코드를 새로 쓰는 것을 두려워하지 말자. 코드가 너무 복잡한 경우에는 아예 프로그램 체계를 다시 구상하여 처음부터 새로 코딩하는 것이 가장 좋은 해결책일 수도 있다. 코드를 다시 짜는 작업은 코드를 처음 짤 때보다는 시간도 적게 걸리고 힘도 덜 든다(실제로 퀘이크 3에서는 퀘이크 2에서 썼던 엔진을 그대로 쓰지 않고 완전히 새로 만든 엔진을 사용했다). 물론 새로 만든 코드라고 버그가 없는 것은 아니다. 하지만 데드라인이 임박한 경우에는 그냥 코드를 포기해 버리는 것보다는 우선 괜찮은 부분은 그대로 두고 문제가 있는 부분만 찾아서 고치는 것이 좋다.

- 문제가 되는 부분을 서로 다른 테스트 무비로 분산해 놓는다. 시스템의 각 부분을 완전히 분리시켜서 작업한 다음에 전체를 하나로 모아보자.

- 주변 사람에게 코드를 봐 달라고 부탁한다. 다른 프로그래머들을 두려워하지 말자. 숙련된 프로그래머들도 1년 전에 쓴 코드를 다시 보면 쑥스러움을 느끼는 경우가 많다.

- '부록 A. 참고 자료'에 적어놓은 곳에서 도움을 얻어보자. 예를 들어 Flash Coders라는 액션스크립트와 관련된 내용만을 다루는 메일링 리스트를 활용하는 것도 좋은 방법이다.

프로그래밍 기법에 대한 조언이 필요하다면 켄트 벡의 『*Extreme Programming Explained*』(Addison Wesley)와 스티브 맥코넬의 『*Code Complete*』(Microsoft Press)를 읽어보면 크게 도움이 될 것이다.

앞으로 배울 내용

이제 이것으로 액션스크립트에 대한 내용은 모두 끝났다. 하지만 이제 독자들이 새로운 내용을 배워가야 할 차례이다. '1부. 액션스크립트 기초'와 '2부. 액션스크립트 응용'을 읽는 동안 액션스크립트로 말을 하는 방법을 배웠을 것이다. 이제 지금까지 배운 내용을 실제 프로젝트에 적용시키는 일만 남았다. 이제 독자 자신만의 긴 여행을 떠나기 전에 다음과 같은 당부의 말을 전할까 한다.

- 다른 모든 기술이 그렇듯이 프로그래밍을 배우는 것도 어떤 행사처럼 한 번에 끝나는 것이 아니라 계속해서 배워 나가는 과정이다. 프로그래밍을 하면 할수록 새로운 것을 배우게 된다. 1, 9, 11, 13장에서 예제로 사용했던 객관식 퀴즈 예제만 해도 네 번이나 프로그램을 새로 고쳤다. 매번 프로그램을 고칠 때마다 새로운 접근법을 적용하고 기능을 추가하면서 전에는 몰랐던 것을 배울 수 있었다. 화가가 같은 소재로 다른 그림을 그릴 때마다 새로운 것을 배울 수 있듯이 비슷한 애플리케이션을 여러 번 만들다 보면 새로운 내용을 배울 수 있다.

- 3부에는 액션스크립트의 내장 함수, 속성, 클래스와 객체에 대한 자세한 설명이 나와 있다. 거기에 수록된 내용을 잘 익히도록 하자. 3부를 처음부터 끝까지 꼼꼼하게 읽어볼 필요는 없다고 하더라도 액션스크립트의 어떤 기능이 있는지 알아보기 위해서라도 전체적으로 한 번 훑어보는 것이 좋다.

- 이 책의 내용을 틈틈이 다시 읽어보도록 하자. 지금보다 더 많은 것을 알고 있는 상태에서 이 책을 다시 본다면 새롭게 느끼는 것도 많게 마련이고 이 책에서 개념적으로만 이해했던 내용도 실전 경험과 연결하여 이해할 수 있게 될 것이다. 3부는 언제나 작업을 할 때마다 옆에 두고 사전을 찾아보는 기분으로 필요한 부분을 찾아서 읽어보는 것이 좋다.

- 아이디어와 각종 문제에 대한 해결책, 그리고 무엇보다도 개발자들끼리 코드를 공유할 수 있는 플래시 개발자 커뮤니티를 활용하자. 그러한 커뮤니티에서 될 수 있는 한 많은 내용을 배우도록 노력하자. 이해하기 힘든 내용이 있다면 이 책의 내용을 다시 읽어보면 도움이 될 것이다. 매크로미디어 사이트에 가면 플래시와 관련된 웹사이트 및 메일링 리스트에 대한 정보를 얻을 수 있다(www.macromedia.com/support/flash/ts/documents/flash_websites.htm).

- 이 책 한 권만으로 액션스크립트에 대한 모든 내용을 배우려고 하지 말자. 액션스크립트를 다양한 각도에서 바라볼 수 있도록 부록 A나 서문에서 언급한 것과 같은 다른 책이나 사이트를 참조하다 보면 더 많은 것을 배울 수 있다. http://www.moock.org/moockmarks에서 플래시와 관련된 다양한 참고 자료를 얻을 수 있다. http://www.moock.org/webdesign/books에 수록된 플래시 서적에 대한 리뷰도 참고해 보기 바란다.
- 마지막으로 http://www.moock.org/asdg에 있는 이 책의 홈페이지에 종종 들러서 코드 샘플, 기술 관련 노트나 새로운 주제에 대한 논의를 훑어보기 바란다.

부디 더 많은 독자들이 플래시 액션스크립트 프로그래밍을 즐길 수 있기를 바란다.