ACTIONSCRIPT® 3.0 학습

마지막 업데이트 2011 년 5 월 13 일

법적 고지 사항

법적 고지 사항은 http://help.adobe.com/ko_KR/legalnotices/index.html을 참조하십시오.

목차

1장: ActionScript3.0 소개

ActionScript

ActionScript는 Adobe® Flash® Player 및 Adobe® AIR™ 런타임 환경의 프로그래밍 언어입니다. 이를 사용하여 Flash, Flex 및 AIR 내용과 응용 프로그램에서 대화형 작업, 데이터 처리 등 보다 다양한 작업을 수행할 수 있습니다.

ActionScript는 Flash Player 및 AIR에 내장되어 있는 AVM(ActionScript Virtual Machine)에 의해 실행됩니다. ActionScript 코드는 일반적으로 컴파일러에 의해 바이트코드 형식으로 변환됩니다. **바이트코드**는 컴퓨터에서 작성 및 이해되는 프로그래밍 언어의 형식 중 하나입니다. 컴파일러에는 Adobe® Flash® Professional에 내장된 컴파일러 및 Adobe® Flash® Builder™에 내장되어 있으며 Adobe® Flex™ SDK에서 사용 가능한 컴파일러가 있습니다. 바이트코드는 SWF 파일에 포함되며 Flash Player 및 AIR에서 실행됩니다.

ActionScript 3.0에서는 객체 지향 프로그래밍에 대한 기본적인 지식만을 가진 개발자도 손쉽게 사용할 수 있는 강력한 프로그래밍 모델을 제공합니다. 이전 버전의 ActionScript보다 향상된 ActionScript 3.0의 주요 기능을 몇 가지 소개하면 다음과 같습니다.

- 새 바이트코드 명령어 집합을 사용하고 성능이 크게 향상된 새로운 AVM(ActionScript Virtual Machine)인 AVM2
- 이전 버전의 컴파일러보다 심도 있는 최적화를 수행하는 최신 컴파일러 코드 베이스
- 객체에 대한 저수준 제어 및 진정한 객체 지향 모델을 사용하여 확장되고 향상된 API(Application Programming Interface)
- E4X(ECMAScript for XML) 사양(ECMA-357 edition 2)을 기반으로 한 XML API. E4X는 언어의 기본 데이터 유형으로 XML이 추가된 ECMAScript의 언어 확장입니다.
- DOM(Document Object Model) 레벨 3 이벤트 사양을 기반으로 하는 이벤트 모델

ActionScript 3.0의 장점

ActionScript 3.0은 이전 버전의 ActionScript를 능가하는 스크립팅 기능을 가지고 있습니다. 대용량 데이터 세트 및 재사용 가능한 객체 지향 코드 베이스를 통해 매우 복잡한 응용 프로그램을 쉽게 만들 수 있습니다. Adobe Flash Player에서 내용을 실행하는 데 ActionScript 3.0이 반드시 필요한 것은 아닙니다. 그러나 ActionScript 3.0 가상 컴퓨터인 AVM2의 향상된 기능을 활용하려면 ActionScript 3.0을 사용해야 합니다. ActionScript 3.0 코드는 이전 ActionScript 코드보다 실행 속도가 최고 10배 향상되었습니다.

이전 버전의 AVM, 즉 AVM1에서는 ActionScript 1.0 및 ActionScript 2.0 코드를 실행합니다. Flash Player 9 및 10에서는 이전 버전과의 호환성을 위해 AVM1을 지원합니다.

ActionScript 3.0의 새로운 기능

ActionScript 3.0에 ActionScript 1.0 및 2.0과 유사한 여러 가지 클래스와 기능이 포함되어 있지만 ActionScript 3.0은 이전 버전의 ActionScript와 구조적 및 개념적인 면에서 다릅니다. ActionScript 3.0의 향상된 기능에는 객체에 대한 저수준 제어를 강화한 API 및 기본 언어의 새로운 기능이 포함됩니다.

기본 언어 기능

기본 언어에는 명령문, 표현식, 조건문, 반복문 및 유형과 같은 프로그래밍 언어의 기본적인 구성 단위가 정의되어 있습니다. ActionScript 3.0에는 개발 작업의 속도를 향상시킬 수 있는 여러 가지 기능이 포함되어 있습니다.

런타임 예외

ActionScript 3.0에서는 이전 버전의 ActionScript보다 많은 오류 상황을 보고합니다. 일반적인 오류 상황에 런타임 예외가 사용되면서 디버깅이 편리해지고 오류를 철저히 처리하는 응용 프로그램을 개발할 수 있게 되었습니다. 런타임 오류를 통해 소스 파일 및 줄 번호 정보가 포함된 스택 추적을 확인하여 오류를 빠르고 정확하게 찾아낼 수 있습니다.

런타임 유형

ActionScript 3.0에서는 유형 정보가 런타임에 보존됩니다. 이 정보는 런타임에 유형을 확인하여 시스템의 유형 안전을 향상시키는 데 사용됩니다. 또한 유형 정보는 변수를 기본 시스템 표현으로 나타내는 데 사용되므로 성능을 향상시키고 메모리 사용량을 줄여 줍니다. 이에 비해 ActionScript 2.0에서 유형 약어는 주로 개발자가 참조하는 데 사용되며 모든 값의 유형은 런타임에 동적으로 지정됩니다.

봉인 클래스

ActionScript 3.0에서는 봉인 클래스 개념이 도입되었습니다. 봉인 클래스에는 컴파일 타임에 정의된 속성 및 메서드의 고정된 집합만 있으며 속성 및 메서드를 더 이상 추가할 수 없습니다. 런타임에 클래스를 변경할 수 없도록 함으로써 컴파일 타임에 보다 엄격한 확인 절차가 수행되어 더욱 견고한 프로그램을 만들 수 있게 됩니다. 또한 각 객체 인스턴스에 내부 해시 테이블이 필요하지 않으므로 사용 가능한 메모리가 증가합니다. dynamic 키워드를 사용하여 동적 클래스를 정의할 수도 있습니다. ActionScript 3.0의 모든 클래스는 기본적으로 봉인되지만 dynamic 키워드를 사용하여 동적으로 선언할 수 있습니다.

메서드 클로저

ActionScript 3.0에서는 메서드 클로저를 원본 객체 인스턴스에 바인딩할 수 있습니다. 이 기능은 이벤트 처리에 유용합니다. ActionScript 2.0에서는 메서드 클로저가 어떠한 객체 인스턴스에서 추출되었는지 기억하지 못하기 때문에 메서드 클로저가 호출되면 예기치 못한 비헤이비어가 발생합니다.

E4X(ECMAScript for XML)

ActionScript 3.0은 최근 ECMA-357로 표준화된 E4X(ECMAScript for XML)를 구현합니다. E4X는 XML을 조작하기 위한 자연스럽고 쉬운 언어 구문 집합을 제공합니다. 기존 XML 구문 분석 API와 달리 E4X에서의 XML은 언어의 기본 데이터 유형처럼 동작합니다. E4X는 필요한 코드 양을 현저하게 줄여 XML을 조작하는 응용 프로그램을 효율적으로 개발할 수 있습니다.

ECMA E4X 사양을 보려면 www.ecma-international.org를 방문하십시오.

일반 표현식

ActionScript 3.0에는 일반 표현식에 대해 기본적인 지원이 포함되어 있어 문자열을 빠르게 검색하고 조작할 수 있습니다. ActionScript 3.0은 ECMAScript(ECMA-262) Edition 3 언어 사양에 정의되어 있는 일반 표현식을 지원합니다.

네임스페이스

네임스페이스는 선언의 가시성을 제어하는 데 사용되는 기존 액세스 지정자(public, private, protected)와 유사합니다. 네임스페이스는 사용자 정의 액세스 지정자로 작동하며 네임스페이스의 이름은 임의로 지정할 수 있습니다. 네임스페이스는 충돌을 방지하기 위해 URI(Universal Resource Identifier)와 함께 사용되며 E4X를 사용하여 작업하는 경우 XML 네임스페이스를 나타내는 데 사용될 수도 있습니다.

새 프리미티브 유형

ActionScript 3.0에는 숫자 유형으로 Number, int 및 uint 세 가지가 있습니다. Number는 배정밀도, 부동 소수점 숫자를 나타냅니다. int 유형은 ActionScript 코드에서 CPU의 신속한 정수 계산 기능을 이용할 수 있는 부호 있는 32비트 정수입니다. int 유형은 정수가 사용되는 루프 카운터 및 변수에 적합합니다. uint 유형은 RGB 색상 값, 바이트 수 등에 적합한 부호 없는 32비트 정수 유형입니다. 이에 비해 ActionScript 2.0에는 숫자 유형으로 Number 하나만 있습니다.

API 기능

ActionScript 3.0의 API에는 저수준에서 객체를 제어할 수 있는 여러 가지 클래스가 포함되어 있습니다. 언어의 구조는 이전 버전보다 직관적으로 설계되었습니다. 클래스가 너무 많아 여기에서 모두를 자세하게 다룰 수 없지만 클래스 간의 중요한 차이점에 대해서는 알고 있는 것이 좋습니다.

DOM3 이벤트 모델

DOM3(Document Object Model Level 3) 이벤트 모델에서는 이벤트 메시지를 생성하고 처리하는 표준 방식을 제공합니다. 이 이벤트 모델은 응용 프로그램 내에서 객체 간 상호 작용 및 통신을 허용하고, 객체의 상태를 유지하고, 변경 사항에 대응하도록 설계되어 있습니다. ActionScript 3.0 이벤트 모델은 World Wide Web Consortium DOM 레벨 3 이벤트 사양을 기반으로 하며 이전 버전의 ActionScript에서 제공되는 이벤트 시스템보다 명확하고 효율적인 메커니즘을 제공합니다.

이벤트 및 오류 이벤트는 flash.events 패키지에 있습니다. Flash Professional 구성 요소와 Flex 프레임워크에 동일한 이벤트 모델이 사용되므로 전체 Flash 플랫폼에서 일관된 이벤트 시스템이 구현됩니다.

표시 목록 API

표시 목록, 즉 응용 프로그램의 시각적인 요소가 모두 포함되어 있는 트리에 액세스하는 데 사용할 수 있는 API는 시각적인 프리미티브 값을 처리하는 클래스로 구성되어 있습니다.

Sprite 클래스는 사용자 인터페이스 구성 요소 같은 시각적 요소의 기본 클래스로 설계된 경량 구성 단위입니다. Shape 클래스는 벡터의 모양을 그대로 나타냅니다. 이러한 클래스는 new 연산자를 사용하여 무리 없이 인스턴스화될 수 있으며 언제든지 동적으로 다른 클래스의 하위 클래스가 될 수 있습니다.

이 경우 수준 관리는 자동으로 이루어집니다. 객체의 쌓이는 순서를 지정하고 관리할 수 있는 메서드가 제공됩니다.

동적 데이터 및 내용 처리

ActionScript 3.0에는 응용 프로그램에서 에셋 및 데이터를 로드하고 처리할 수 있는 메커니즘이 포함되어 있습니다. 이 메커니즘은 직관적이며 API 전체에서 일관성을 유지합니다. Loader 클래스는 SWF 파일과 이미지 에셋을 로드할 수 있는 단일 메커니즘 및 로드된 내용에 대한 세부 정보에 액세스할 수 있는 방법을 제공합니다. URLLoader 클래스는 데이터 기반 응용 프로그램에서 텍스트와 이진 데이터를 로드할 수 있는 별도의 메커니즘을 제공합니다. Socket 클래스는 포맷에 관계없이 서버 소켓에서 이진 데이터를 읽고 쓸 수 있는 방법을 제공합니다.

저수준 데이터 액세스

다양한 API에서 데이터에 대한 저수준 액세스를 제공합니다. URLStream 클래스는 데이터를 다운로드하는 동안 해당 데이터를 원시 이진 데이터로 액세스할 수 있게 합니다. ByteArray 클래스를 사용하면 이진 데이터에 대한 읽기, 쓰기 및 작업 기능을 최적화할 수 있습니다. 사운드 API의 SoundChannel 및 SoundMixer 클래스를 통해 사운드를 보다 정밀하게 제어할 수 있습니다. 보안 API에서는 보안 오류를 해결할 수 있도록 SWF 파일 또는 로드된 내용의 보안 권한에 대한 정보를 제공합니다.

텍스트를 사용한 작업

ActionScript 3.0에는 모든 텍스트 관련 API를 묶은 flash.text 패키지가 포함되어 있습니다. ActionScript 2.0의 TextFormat.getTextExtent() 메서드를 대체하는 TextLineMetrics 클래스는 텍스트 필드 내의 텍스트 행에 대한 자세한 메트릭을 제공합니다. TextField 클래스에는 텍스트 필드에 있는 텍스트 행 또는 단일 문자에 대한 특정 정보를 제공하는 저수준 메서드가 여러 개 포함되어 있습니다. 예를 들어 getCharBoundaries() 메서드는 문자의 테두리 상자를 나타내는 사각형을 반환합니다. getCharIndexAtPoint() 메서드는 지정된 위치에 있는 문자의 인덱스를 반환하고, getFirstCharInParagraph() 메서드는 단락에서 첫 번째 문자의 인덱스를 반환합니다. 행 수준의 메서드에는 지정된 텍스트 행의 문자 수를 반환하는 getLineLength() 및 지정된 줄의 텍스트를 반환하는 getLineText()가 있습니다. Font 클래스는 SWF 파일에 포함된 글꼴을 관리하는 방법을 제공합니다.

텍스트에 대한 보다 저수준의 제어를 위해 flash.text.engine 패키지의 여러 클래스는 Flash 텍스트 엔진을 구성합니다. 이 클래스 집합은 텍스트에 대한 저수준 제어를 제공하며 텍스트 프레임워크 및 구성 요소를 만드는 데 사용하기 위해 설계되었습니다.

2장: ActionScript 시작

프로그래밍 기본 사항

ActionScript는 프로그래밍 언어이기 때문에 먼저 몇 가지 일반적인 컴퓨터 프로그래밍 개념을 이해하면 ActionScript를 익히는 데 도움이 됩니다.

컴퓨터 프로그램으로 수행하는 작업

먼저 컴퓨터 프로그램이란 무엇이며 이를 통해 수행할 수 있는 작업은 무엇인지에 대한 개념을 정립하는 것이 좋습니다. 컴퓨터 프로그램은 다음과 같은 두 가지 측면에서 살펴 볼 수 있습니다.

- 프로그램은 컴퓨터에서 수행할 일련의 명령 또는 단계입니다.
- 각 단계는 궁극적으로 정보 또는 데이터의 일부를 조작하는 것과 관련이 있습니다.

일반적으로 컴퓨터 프로그램은 컴퓨터에 지정하는 단계별 명령 목록이며 이러한 명령은 하나씩 수행됩니다. 각 개별 명령을 **명령문**이라고 합니다. ActionScript에서 작성하는 각 명령문은 세미콜론으로 끝납니다.

사실 프로그램에 지정된 명령에 의해 수행되는 작업은 컴퓨터 메모리에 저장된 데이터의 일부 비트를 조작하는 것이 전부입니다. 간단한 예로 두 숫자를 더하여 결과를 메모리에 저장하도록 컴퓨터에 명령하는 경우를 들 수 있습니다. 보다 복잡한 예로는 화면에 그려진 사각형을 화면의 다른 영역으로 이동하도록 프로그램을 작성하려는 경우가 있습니다. 이 경우 컴퓨터는 사각형의 x 및 y 좌표, 너비와 높이, 색상 등 사각형에 대한 특정 정보를 기억하며, 이러한 정보의 각 비트가 컴퓨터 메모리에 저장됩니다. 사각형을 다른 위치로 이동하는 프로그램에는 "x 좌표를 200으로 변경하고 y 좌표를 150으로 변경하는" 단계들이 포함됩니다. 즉, 이 프로그램은 x 및 y 좌표의 새 값을 지정하게 됩니다. 컴퓨터는 이러한 숫자를 컴퓨터 화면에 표시되는 이미지로 변환하기 위해 메모리에 저장된 데이터를 사용하여 필요한 작업을 배후에서 수행합니다. 그러나 이에 대한 매우 세부적인 지식이 필요하지 않은 경우라면 "화면의 사각형을 이동하는" 과정에서 컴퓨터 메모리의 데이터 비트를 변경하게 된다는 사실을 이해하는 것으로 충분합니다.

변수 및 상수

프로그래밍은 주로 컴퓨터 메모리의 여러 정보를 변경하는 것과 관련됩니다. 따라서 프로그램에서 개별적인 정보를 나타낼 수 있는 방법이 제공되어야 합니다. **변수**는 컴퓨터 메모리에 있는 값을 나타내는 이름입니다. 값을 조작하기 위해 명령문을 작성할 때 값 대신 변수 이름을 사용합니다. 이렇게 하면 프로그램 내에서 변수 이름이 식별될 때마다 컴퓨터에서 메모리를 조회하여 발견된 값을 사용합니다. 예를 들어, 숫자가 저장된 value1과 value2라는 두 개의 변수가 있는 경우 해당 숫자를 추가하기 위해 다음과 같은 명령문을 작성할 수 있습니다.

```
value1 + value2
```

실제로 일련의 단계가 실행되면 컴퓨터에서 각 변수의 값을 확인하고 두 값을 더합니다.

ActionScript 3.0에서 실제로 변수는 다음과 같은 세 부분으로 구성됩니다.

- 변수 이름
- 변수에 저장할 수 있는 데이터의 유형
- 컴퓨터 메모리에 저장된 실제 값

컴퓨터에서 값에 대한 자리 표시자로 이름을 사용하는 방법에 대해 살펴보았습니다. 데이터 유형도 중요합니다. ActionScript에서 변수를 만들 때는 해당 변수가 저장할 데이터의 유형을 지정합니다. 데이터 유형을 지정한 이후 프로그램의 명령은 이 변수에 해당 유형의 데이터만 저장할 수 있습니다. 변수의 값을 조작할 때는 해당 데이터 유형과 관련된 특징이 사용됩니다. ActionScript에서 변수를 만들려면 또는 변수를 **선언**하려면 다음과 같이 var 명령문을 사용합니다.

```
var value1:Number;
```

다음 예제에서는 Number 데이터만 저장할 수 있는 value1이라는 변수를 만들도록 컴퓨터에 지시합니다. "Number"는 ActionScript에 정의된 데이터 유형 중 하나입니다. 또한 다음과 같이 변수에 값을 바로 저장할 수도 있습니다.

```
var value2:Number = 17;
```

Adobe Flash Professional

Flash Professional에서는 또 다른 방법으로 변수를 선언할 수 있습니다. 스테이지에 텍스트 필드, 버튼 심볼 또는 무비 클립 심볼을 배치할 때 속성 관리자에서 인스턴스 이름을 지정할 수 있습니다. 그러면 Flash Professional에서 자동으로 인스턴스 이름과 동일한 이름의 변수를 만듭니다. 이 이름은 ActionScript 코드에서 스테이지 항목을 나타내는 데 사용할 수 있습니다. 예를 들어 스테이지에 동영상 클립 심볼이 있고 이 심볼에 rocketShip이라는 인스턴스 이름을 지정한 경우를 가정해 봅니다. 이 경우 ActionScript 코드에서 rocketShip 변수를 사용할 때마다 실제로는 해당 동영상 클립을 조작하게 됩니다.

변수와 유사한 것으로 **상수**가 있습니다. 상수는 컴퓨터 메모리에 저장되는 지정된 데이터 유형의 값을 나타내는 이름입니다. 차이점은 상수의 경우 ActionScript 응용 프로그램 과정에서 한 번만 값을 할당할 수 있다는 것입니다. 상수의 값이 할당되면 전체 응용 프로그램에서 동일합니다. 상수 선언 구문은 변수 선언 구문과 거의 동일합니다. var 키워드 대신 const 키워드를 사용한다는 것이 유일한 차이점입니다.

```
const SALES_TAX_RATE:Number = 0.07;
```

상수는 전체 프로젝트의 여러 위치에서 사용되며 정상적인 상황에서 변경되지 않는 값을 정의하는 데 유용합니다. 리터럴 값 대신 상수를 사용하면 코드를 보다 쉽게 읽을 수 있습니다. 동일한 코드의 두 가지 버전을 예로 들어 보겠습니다. 한 버전에서는 가격에 SALES_TAX_RATE를 곱하고, 다른 버전에서는 가격에 0.07을 곱한다고 가정합니다. 두 버전 중 SALES_TAX_RATE 상수를 사용하는 버전이 이해하기 쉽습니다. 또한 상수로 정의된 값은 변경할 수 있다는 점을 생각해 보십시오. 프로젝트 전체에서 특정 값을 나타내기 위해 상수를 사용할 경우 한 위치, 즉 상수 선언에서만 값을 변경하면 됩니다. 이와 달리 하드 코딩된 리터럴 값을 사용하면 여러 위치에서 값을 변경해야 합니다.

데이터 유형

ActionScript에는 변수의 데이터 유형으로 사용할 수 있는 여러 데이터 유형이 있습니다. 다음과 같은 데이터 유형을 "간단"하고 "기본적"인 데이터 유형으로 간주할 수 있습니다.

- String: 설명서의 텍스트 또는 이름과 같은 텍스트 값입니다.
- Numeric: ActionScript 3.0에는 숫자 데이터에 사용할 수 있는 세 가지 데이터 유형이 있습니다.
  - Number: 분수이거나 분수가 아닌 값을 비롯한 모든 숫자 값
  - int: 정수(분수가 아닌 모든 숫자)
  - uint: "부호 없는" 정수(음수가 될 수 없는 정수)
- Boolean: 스위치가 켜져 있는지 또는 두 값이 같은지 여부를 지정하는 true 또는 false 값

간단한 데이터 유형은 단일 숫자 또는 텍스트의 단일 시퀀스 같은 단일 정보를 나타냅니다. 그러나 ActionScript에 정의된 대부분의 데이터 유형은 복합 데이터 유형일 수 있습니다. 이러한 데이터 유형은 값 집합을 단일 컨테이너로 나타냅니다. 예를 들어 Date 데이터 유형의 변수는 해당 시점을 의미하는 단일 값을 나타냅니다. 그러나 날짜 값은 단일 값이 아니라 각각 개별 숫자인 일, 월, 년, 시, 분, 초 등의 여러 값으로 표현됩니다. 이러한 특성에도 불구하고 날짜는 일반적으로 단일 값으로 간주되고 있으며 Date 변수를 만들면 날짜를 단일 값으로 처리할 수 있습니다. 그러나 컴퓨터 내부적으로는 이 값을 단일 값이 아니라 여러 값이 결합하여 단일 날짜를 정의하는 그룹으로 간주합니다.

대부분의 내장 데이터 유형 및 프로그래머가 정의한 데이터 유형은 복합 데이터 유형입니다. 널리 사용되는 복합 데이터 유형은 다음과 같습니다.

- MovieClip: 무비 클립 심볼
- TextField: 동적 또는 입력 텍스트 필드
- SimpleButton: 버튼 심볼

• Date: 해당 시점에 대한 정보(날짜 및 시간)

클래스와 객체는 데이터 유형에 대한 동의어로 자주 사용되는 단어입니다. 클래스는 데이터 유형에 대한 정의라고 간단히 설명할 수 있습니다. 클래스는 해당 데이터 유형의 모든 객체를 위한 템플릿과 유사한 것으로, "Example 데이터 유형의 모든 변수는 A, B 및 C라는 특징을 갖습니다."라고 기술하는 것으로 보면 됩니다. 반면 객체는 특정 클래스의 실제 인스턴스입니다. 예를 들어 데이터 유형이 MovieClip인 변수는 MovieClip 객체로 설명될 수 있습니다. 다음과 같이 동일한 변수를 각각 다르게 설명할 수 있습니다.

• myVariable 변수의 데이터 유형은 Number입니다.

• myVariable 변수는 Number 인스턴스입니다.

• myVariable 변수는 Number 객체입니다.

• myVariable 변수는 Number 클래스의 인스턴스입니다.

객체 다루기

ActionScript는 객체 지향 프로그래밍 언어입니다. 객체 지향 프로그래밍은 프로그래밍에 대한 접근 방식 중 하나로서, 프로그램에서 코드를 구성할 때 객체를 사용합니다.

앞에서 "컴퓨터 프로그램"이라는 용어에 대해 컴퓨터가 수행하는 일련의 단계 또는 명령이라고 정의한 바 있습니다. 개념적인 면에서 컴퓨터 프로그램은 명령으로 구성된 하나의 긴 목록이라고 가정할 수 있습니다. 그러나 객체 지향 프로그래밍에서는 프로그램 명령이 여러 객체에 분배됩니다. 코드가 기능별로 그룹화되고 관련된 유형의 기능 또는 관련된 정보가 컨테이너 하나에 그룹화됩니다.

Adobe Flash Professional

Flash Professional에서 심볼을 사용하여 작업해 본 경험이 있으면 객체를 사용하여 작업을 수행할 준비가 되어 있는 것입니다. 사각형 드로잉 같은 동영상 클립 심볼을 정의하고 스테이지에 복사본을 배치했다고 가정해 봅니다. ActionScript에서는 무비 클립 심볼도 객체이며 MovieClip 클래스의 인스턴스입니다.

무비 클립의 다양한 특징을 수정할 수 있습니다. 동영상 클립을 선택하고 속성 관리자에서 x 좌표 또는 폭 등의 값을 변경할 수 있습니다. 또한 알파 또는 투명도를 변경하거나 그림자 필터를 적용하는 등 색을 다양하게 조정할 수 있습니다. [자유 변형 도구]를 사용하여 사각형을 회전하는 것과 같이 다른 Flash Professional 도구를 사용하면 보다 많은 사항을 변경할 수 있습니다. Flash Professional에서 동영상 클립 심볼을 수정할 수 있는 이러한 방법을 ActionScript에서도 모두 사용할 수 있습니다. ActionScript에서는 MovieClip 객체라는 단일 묶음에 함께 포함된 여러 데이터를 변경하여 동영상 클립을 수정할 수 있습니다.

ActionScript 객체 지향 프로그래밍에는 모든 클래스에 포함될 수 있는 다음과 같은 세 가지 특성이 있습니다.

• 속성

• 메서드

• 이벤트

이러한 요소는 프로그램에서 사용하는 데이터를 관리하고 수행할 작업과 순서를 결정하는 데 사용됩니다.

속성

속성은 객체에 함께 묶여 있는 데이터 중 하나를 나타냅니다. 예를 들어 Song 객체에는 artist 속성과 title 속성이 포함될 수 있으며 MovieClip 클래스에는 rotation, x, width 및 alpha와 같은 속성이 포함될 수 있습니다. 속성은 개별 변수처럼 사용할 수 있습니다. 실제로 속성은 특정 객체에 포함된 "자식" 변수라고 생각하면 간단합니다.

속성을 사용하는 ActionScript 코드의 예제는 다음과 같습니다. 다음 코드 행은 square MovieClip을 x 좌표 100 픽셀로 이동합니다.

```
square.x = 100;
```

다음 코드에서는 rotation 속성을 사용하여 triangle MovieClip의 회전과 일치하도록 square MovieClip을 회전합니다.

```
square.rotation = triangle.rotation;
```

다음 코드에서는 수평 배율을 변경하여 square MovieClip의 폭을 1.5배 넓힙니다.

```
square.scaleX = 1.5;
```

객체 이름으로 변수(square, triangle)를 사용하고 그 뒤에 마침표(.)와 속성 이름(x, rotation, scaleX)을 차례로 사용하는 것이 일반적인 구조입니다. 도트 연산자인 마침표는 객체의 자식 요소 중 하나에 액세스한다는 것을 나타내는 데 사용됩니다. 컴퓨터 메모리에 있는 단일 값의 이름으로 "변수 이름-도트-속성 이름"의 전체 구조를 단일 변수와 같이 사용합니다.

메서드

메서드는 객체가 수행할 수 있는 작업입니다. 예를 들어 타임라인의 여러 키프레임과 애니메이션을 사용하여 Flash Professional에서 동영상 클립 심볼을 만든 경우를 가정해 봅니다. 이 동영상 클립을 재생 또는 중지하거나 재생 헤드를 특정 프레임으로 이동하도록 지시할 수 있습니다.

다음 코드는 shortFilm MovieClip의 재생을 시작하도록 지시합니다.

```
shortFilm.play();
```

다음 행은 shortFilm MovieClip의 재생을 중지합니다. 이렇게 하면 비디오를 일시 정지하는 것과 같이 재생 헤드가 제자리에 멈춥니다.

```
shortFilm.stop();
```

다음 코드는 비디오를 되감는 것과 같이 shortFilm MovieClip의 재생 헤드를 Frame 1로 이동하고 재생을 중지합니다.

```
shortFilm.gotoAndStop(1);
```

속성에서와 같이 객체 이름(변수) 뒤에 마침표를 사용한 후 메서드 이름과 괄호를 차례로 입력하여 메서드에 액세스합니다. 괄호는 메서드를 호출하는 것, 즉 객체가 해당 작업을 수행하도록 지시하는 것을 나타내는 방법입니다. 작업을 수행하는 데 필요한 추가 정보를 표시하는 방법으로 괄호 안에 값 또는 변수가 지정되는 경우도 있습니다. 이러한 값을 메서드 매개 변수라고 합니다. 예를 들어 gotoAndStop() 메서드는 이동할 프레임에 대한 정보가 필요하므로 괄호 안에 단일 매개 변수가 있어야 합니다. play() 및 stop()과 같은 기타 메서드는 이름 자체로도 그 기능을 충분히 설명할 수 있으므로 추가 정보가 필요하지 않습니다. 그러나 괄호는 사용해야 합니다.

속성 및 변수와 달리 메서드는 값 자리 표시자로 사용되지 않습니다. 그러나 일부 메서드는 계산을 수행하여 변수처럼 사용할 수 있는 결과를 반환할 수 있습니다. 예를 들어, Number 클래스의 toString() 메서드는 숫자 값을 텍스트 표현으로 변환합니다.

```
var numericData:Number = 9;
var textData:String = numericData.toString();
```

예를 들어 화면의 텍스트 필드에 Number 변수의 값을 표시하려면 toString() 메서드를 사용합니다. TextField 클래스의 text 속성은 String으로 정의되어 있으므로 이 속성에는 텍스트 값만 포함할 수 있습니다. text 속성은 화면에 표시된 실제 텍스트 내용을 나타냅니다. 다음 코드 행은 numericData 변수의 숫자 값을 텍스트로 변환합니다. 그런 다음 이 값이 화면에서 calculatorDisplay TextField 객체에 표시되도록 합니다.

```
calculatorDisplay.text = numericData.toString();
```

이벤트

컴퓨터 프로그램은 컴퓨터에서 단계별로 수행하는 일련의 명령입니다. 일부 간단한 컴퓨터 프로그램에는 프로그램이 끝나는 시점에 컴퓨터에서 수행되는 몇 가지 단계만 포함되어 있습니다. 그러나 ActionScript 프로그램은 사용자 입력 등을 기다리면서 계속해서 실행되도록 설계되었습니다. 이벤트는 컴퓨터에서 언제 어떤 명령을 수행할지 결정하는 메커니즘입니다.

기본적으로 **이벤트**는 ActionScript에서 감지하고 반응할 수 있는 사건입니다. 여러 이벤트가 버튼을 클릭하거나 키보드에서 키를 누르는 것과 같은 사용자 상호 작용과 관련되어 있지만, 이벤트 중에는 이와 다른 유형도 있습니다. 예를 들어, ActionScript를 사용하여 외부 이미지를 로드하는 경우 이미지 로드가 끝나면 알려 주는 이벤트가 있습니다. 개념적인 관점에서 보면 실행 중인 ActionScript 프로그램은 특정 상황이 발생하기를 기다리며 대기하고 있을 뿐입니다. 특정 상황이 발생하면 해당 이벤트에 대해 지정한 ActionScript 코드가 실행됩니다.

기본적 이벤트 처리

이벤트 처리는 특정 이벤트에 대한 응답으로 수행할 특정 작업을 지정하기 위한 방법입니다. 이벤트 처리를 수행하는 ActionScript 코드를 작성할 때 다음 세 가지 중요한 요소를 식별할 수 있어야 합니다.

- 이벤트 소스: 이벤트가 발생할 객체. 예를 들어 클릭한 버튼 또는 이미지를 로드하는 Loader 객체 등이 이벤트 소스입니다. 이벤트 소스를 **이벤트 대상**이라고도 합니다. 해당 객체는 컴퓨터가 이벤트를 찾는 표적, 즉 이벤트가 실제로 발생한 위치이므로 이와 같은 이름으로도 불립니다.
- 이벤트: 발생할 사건 및 응답할 사건. 많은 객체에서 여러 이벤트를 트리거하기 때문에 특정 이벤트를 식별하는 것이 중요합니다.
- 응답: 이벤트가 발생하면 실행할 단계

이벤트를 처리할 ActionScript 코드를 작성할 경우에는 이러한 세 가지 요소가 항상 필요합니다. 코드는 다음과 같은 기본 구조를 따릅니다. 굵게 표시된 요소는 자리 표시자이며 특정 경우에 맞춰 사용자가 채워야 합니다.

```
function eventResponse(eventObject:EventType):void
{
    // Actions performed in response to the event go here.
}

eventSource.addEventListener(EventType.EVENT_NAME, eventResponse);
```

이 코드는 두 가지 작업을 수행합니다. 먼저 함수를 정의하여 이벤트에 대한 응답으로 수행하려는 작업을 지정할 수 있습니다. 그런 다음 소스 객체의 addEventListener() 메서드를 호출합니다. addEventListener()를 호출하면 지정한 이벤트에 대해 해당 함수가 자동으로 "등록"됩니다. 따라서 이벤트가 발생하면 해당 함수의 작업이 수행됩니다. 이러한 부분을 개별적으로 보다 자세히 살펴 보겠습니다.

함수는 단축키 이름 같은 단일 이름으로 작업을 그룹화하여 수행할 수 있는 방법을 제공합니다. 함수는 특정 클래스와 반드시 연관되지는 않는다는 점을 제외하면 메서드와 동일합니다. 실제로 "메서드"라는 용어는 특정 클래스와 연관된 함수로 정의할 수 있습니다. 이벤트 처리를 위해 함수를 만드는 경우 함수의 이름(이 경우 eventResponse)을 선택하고, 매개 변수(이 예제에서는 eventObject)도 하나 지정합니다. 함수 매개 변수를 지정하는 것은 변수를 선언하는 것과 같기 때문에 매개 변수의 데이터 유형도 지정해야 합니다. 이 예제에서 매개 변수의 데이터 유형은 EventType입니다.

수신할 각 이벤트 유형에는 ActionScript 클래스가 연관되어 있습니다. 함수 매개 변수에 지정하는 데이터 유형은 항상 응답하려는 특정 이벤트의 연관된 클래스입니다. 예를 들어 click 이벤트(사용자가 마우스로 항목을 클릭할 때 트리거됨)는 MouseEvent 클래스와 연관되어 있습니다. click 이벤트에 대한 리스너 함수를 작성하려면 MouseEvent 데이터 유형의 매개 변수를 사용하여 리스너 함수를 정의합니다. 마지막으로, 여는 중괄호와 닫는 중괄호({ ... }) 사이에 이벤트가 발생할 때 컴퓨터에서 수행하려는 명령을 입력합니다.

이제 이벤트 처리 함수가 작성되었습니다. 다음 단계에서는 이벤트 발생 시 해당 함수를 호출하도록 이벤트 소스 객체(이벤트가 발생하는 버튼 등의 객체)에게 지시합니다. 해당 객체의 addEventListener() 메서드를 호출하여 이벤트 소스 객체에 함수를 등록합니다. 이벤트를 가지는 모든 객체는 addEventListener() 메서드도 함께 가집니다. addEventListener() 메서드에 다음과 같은 두 개의 매개 변수가 포함됩니다.

- 첫 번째, 응답할 특정 이벤트 이름입니다. 각 이벤트는 특정 클래스와 연결되어 있습니다. 모든 이벤트 클래스는 고유한 이름 같은 특별한 값을 가지며 이러한 값은 각 해당 이벤트에 대해 정의됩니다. 이 값을 첫 번째 매개 변수로 사용합니다.
- 두 번째, 이벤트 응답 함수 이름입니다. 함수 이름이 매개 변수로 전달되면 괄호 없이 작성됩니다.

이벤트 처리 프로세스

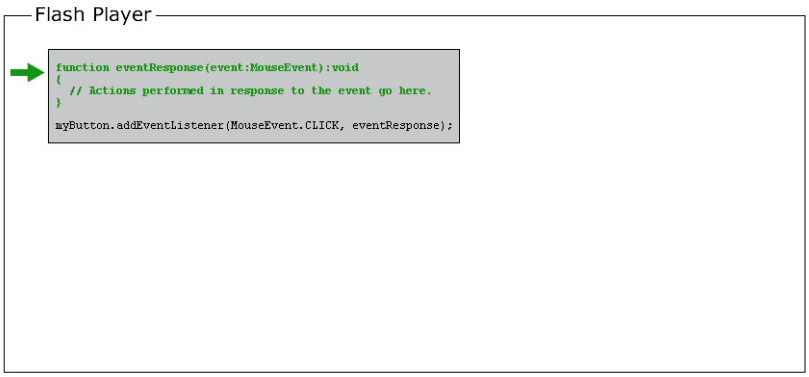
다음은 이벤트 리스너를 만들 때 발생하는 프로세스의 단계별 설명입니다. 이 경우는 myButton이라는 객체를 클릭할 때 호출되는 리스너 함수를 만드는 예제입니다.

프로그래머가 작성하는 실제 코드는 다음과 같습니다.

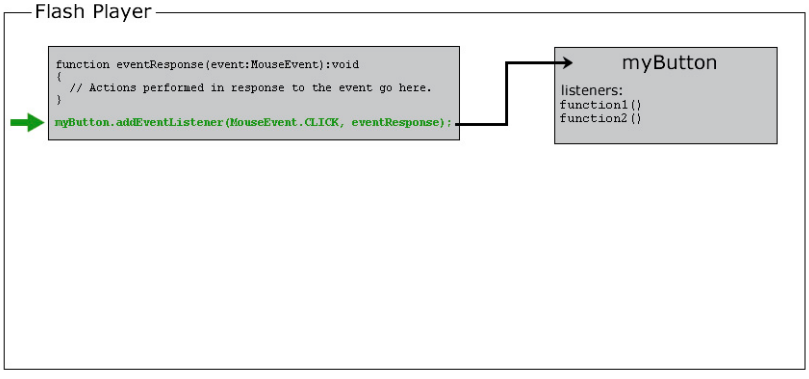
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

myButton.addEventListener(MouseEvent.CLICK, eventResponse);
```

다음은 이 코드가 실행될 때 실제로 작동하는 방식을 나타낸 것입니다.

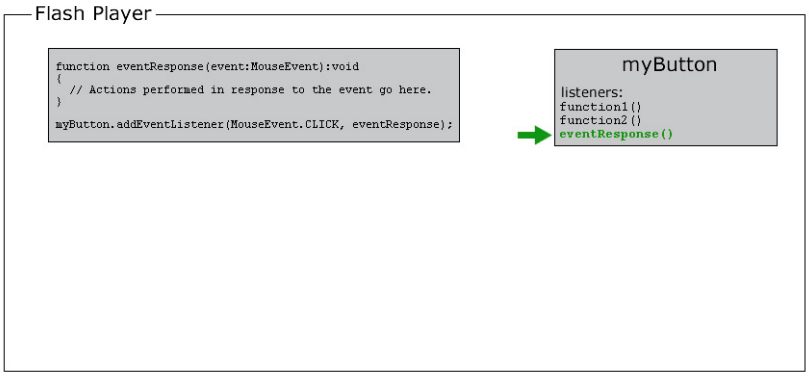
1 SWF 파일이 로드되면 컴퓨터에서는 eventResponse()라는 함수의 존재 사실을 기억해 둡니다.



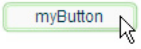
2 그런 다음 컴퓨터에서는 코드, 명확히 말하면 함수에 없는 코드 행을 실행합니다. 이 경우에는 이벤트 소스 객체(myButton)에서 addEventListener() 메서드를 호출하고 eventResponse 함수를 매개 변수로 전달하여 하나의 코드 행만 실행합니다.



내부적으로 myButton에는 각 해당 이벤트를 수신하는 함수 목록이 있습니다. addEventListener() 메서드가 호출되면 myButton은 이벤트 리스너 목록에 eventResponse() 함수를 저장합니다.

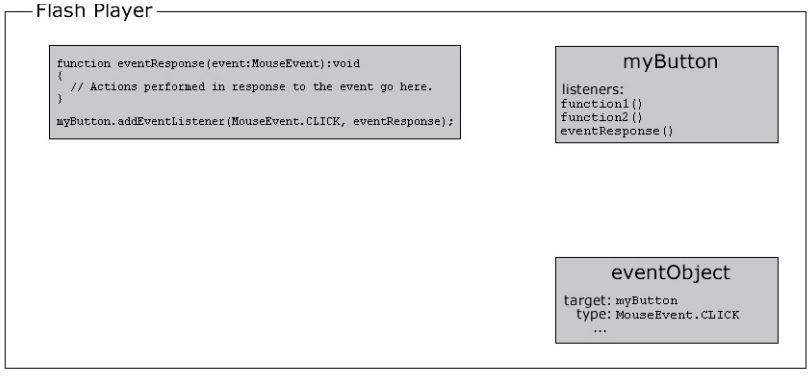


3 어느 시점에서 사용자가 myButton 객체를 클릭하면 click 이벤트(코드에서 MouseEvent.CLICK으로 나타남)가 트리거됩니다.

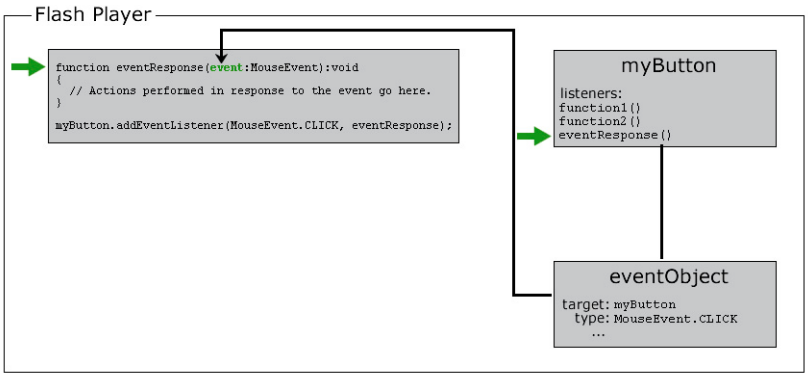


이때 다음과 같은 상황이 발생합니다.

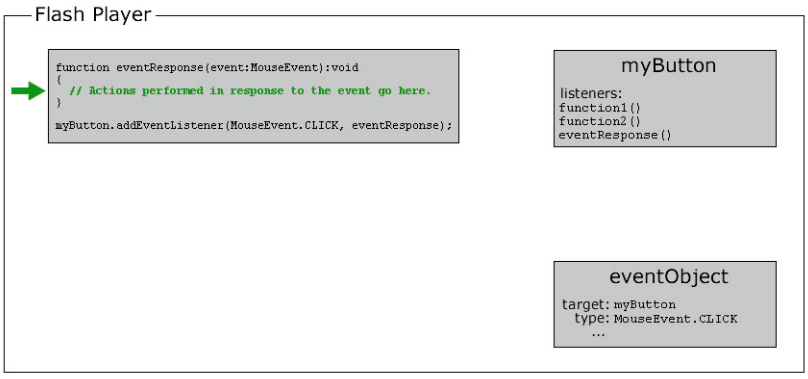
a 해당 이벤트(이 예제에서는 MouseEvent)와 연관된 클래스의 인스턴스인 객체가 만들어집니다. 여러 이벤트에서 이 객체는 Event 클래스의 인스턴스입니다. 마우스 이벤트의 경우 이 객체는 MouseEvent 인스턴스이고, 다른 이벤트의 경우에는 해당 이벤트와 연관된 클래스의 인스턴스입니다. 생성된 이 객체는 **이벤트 객체**라고 하며, 발생한 이벤트에 대한 고유 정보(이벤트 유형, 이벤트 발생 위치 등) 및 적용 가능한 기타 이벤트별 정보가 포함되어 있습니다.



b 그런 다음 컴퓨터에서는 myButton에 의해 저장된 이벤트 리스너 목록을 검토합니다. 이러한 함수를 하나씩 검토하면서 각 함수를 호출하고 이벤트 객체를 매개 변수로 함수에 전달합니다. eventResponse() 함수는 myButton의 리스너 중 하나이므로, 컴퓨터에서는 이 프로세스의 일부로서 eventResponse() 함수를 호출합니다.



c eventResponse() 함수가 호출되면 해당 함수의 코드가 실행되어 특정 작업이 수행됩니다.



이벤트 처리 예제

다음에는 이벤트 처리 코드에 대한 보다 구체적인 예제가 몇 개 나와 있습니다. 이러한 예제를 통해 이벤트 처리 코드 작성 시 사용할 수 있는 공통적인 일부 이벤트 요소 및 가능한 변형에 대해 이해할 수 있습니다.

- 버튼을 클릭하여 현재 무비 클립 재생을 시작합니다. 다음 예제에서 playButton은 버튼의 인스턴스 이름이며 this는 "현재 객체"를 나타내는 특수 이름입니다.

```
this.stop();

function playMovie(event:MouseEvent):void
{
    this.play();
}

playButton.addEventListener(MouseEvent.CLICK, playMovie);
```

- 텍스트 필드에 입력된 내용을 검색합니다. 이 예제에서 entryText는 입력 텍스트 필드이며 outputText는 동적 텍스트 필드입니다.

```
function updateOutput(event:TextEvent):void
{
    var pressedKey:String = event.text;
    outputText.text = "You typed: " + pressedKey;
}

entryText.addEventListener(TextEvent.TEXT_INPUT, updateOutput);
```

- 버튼을 클릭하여 URL로 이동합니다. 이 경우 linkButton은 버튼의 인스턴스 이름입니다.

```
function gotoAdobeSite(event:MouseEvent):void
{
    var adobeURL:URLRequest = new URLRequest("http://www.adobe.com/");
    navigateToURL(adobeURL);
}

linkButton.addEventListener(MouseEvent.CLICK, gotoAdobeSite);
```

객체 인스턴스 만들기

ActionScript에서 객체를 사용하려면 먼저 객체가 있어야 합니다. 객체를 만드는 방법 중 하나는 변수를 선언하는 것입니다. 그러나 변수를 선언하면 컴퓨터 메모리에 빈 공간만 만들어집니다. 따라서 변수에 실제 값을 항상 지정해야 합니다. 즉, 객체를 사용하거나 조작하려면 먼저 객체를 만들고 변수에 해당 객체를 저장합니다. 이처럼 객체를 만드는 과정을 객체 **인스턴스화**라고 합니다. 즉, 특정 클래스의 인스턴스를 만드는 것입니다.

객체 인스턴스를 간단히 만드는 방법은 ActionScript를 전혀 사용하지 않는 것입니다. Flash Professional에서 동영상 클립 심볼, 버튼 심볼 또는 텍스트 필드를 스테이지에 배치하고 인스턴스 이름을 지정합니다. Flash Professional에서는 자동으로 해당 인스턴스 이름의 변수를 선언하고, 객체 인스턴스를 만들고, 해당 객체를 변수에 저장합니다. 이와 유사하게 Flex에서는 Flash Builder 디자인 모드에서 편집기에 구성 요소를 배치하거나 MXML 태그를 코딩하여 MXML로 구성 요소로 만듭니다. 해당 구성 요소에 ID를 지정하면 이 ID는 해당 구성 요소 인스턴스를 포함하는 ActionScript 변수의 이름이 됩니다.

그러나 객체를 시각적으로 만들고 싶지 않을 때도 있고 비시각적 객체의 경우에는 시각적인 방법을 사용할 수 없습니다. 이러한 경우에는 ActionScript만 사용하여 객체 인스턴스를 만들 수 있는 몇 가지 다른 방법을 활용하면 됩니다.

ActionScript 코드 내에 직접 작성되는 값인 **리터럴 표현식**을 사용하여 여러 ActionScript 데이터 유형의 인스턴스를 만들 수 있습니다. 다음에 몇 가지 예제가 나와 있습니다.

- 직접 숫자를 입력하는 리터럴 숫자 값:

```
var someNumber:Number = 17.239;
var someNegativeInteger:int = -53;
var someUint:uint = 22;
```

- 텍스트가 따옴표로 둘러싸인 리터럴 문자열 값:

```
var firstName:String = "George";
var soliloquy:String = "To be or not to be, that is the question...";
```

- true 또는 false 리터럴 값을 사용하는 리터럴 부울 값:

```
var niceWeather:Boolean = true;
var playingOutside:Boolean = false;
```

- 쉼표로 구분된 값 목록을 대괄호로 묶는 리터럴 배열 값:

```
var seasons:Array = ["spring", "summer", "autumn", "winter"];
```

- XML을 직접 입력하는 리터럴 XML 값:

```
var employee:XML = <employee>
        <firstName>Harold</firstName>
        <lastName>Webster</lastName>
    </employee>;
```

또한 ActionScript에서 Array, RegExp, Object 및 Function 데이터 유형의 리터럴 표현식을 정의합니다.

모든 데이터 유형의 인스턴스를 만드는 가장 일반적인 방법은 다음과 같이 클래스 이름과 함께 new 연산자를 사용하는 것입니다.

```
var raceCar:MovieClip = new MovieClip();
var birthday:Date = new Date(2006, 7, 9);
```

new 연산자를 사용하여 객체를 만드는 것을 흔히 "클래스 생성자를 호출"한다고 말합니다. 생성자는 클래스 인스턴스를 만드는 과정 중 호출되는 특수 메서드입니다. 이 방법으로 인스턴스를 만들 경우 클래스 이름 뒤에 괄호를 입력합니다. 이 괄호 안에 매개 변수 값을 지정하는 경우도 있습니다. 이 두 가지 작업은 메서드를 호출할 때도 수행됩니다.

리터럴 표현식을 사용하여 인스턴스를 작성할 수 있는 데이터 유형의 경우에도 new 연산자를 사용하여 객체 인스턴스를 만들 수 있습니다. 예를 들어 다음 두 코드 행은 동일한 작업을 수행합니다.

```
var someNumber:Number = 6.33;
var someNumber:Number = new Number(6.33);
```

객체를 만드는 방법인 new **ClassName**()에 익숙해져야 합니다. ActionScript 데이터 유형 중에는 상응하는 시각적 표현이 존재하지 않는 경우가 많습니다. 따라서 이러한 데이터 유형의 인스턴스는 Flash Professional 스테이지 또는 Flash Builder MXML 편집기의 디자인 모드에서 항목을 배치하는 방법으로는 만들 수 없습니다. ActionScript에서 이러한 데이터 유형의 인스턴스는 new 연산자를 사용하는 방법으로만 만들 수 있습니다.

Adobe Flash Professional

Flash Professional에서는 라이브러리에 정의되어 있지만 스테이지에 배치되지 않은 동영상 클립 심볼의 인스턴스를 만들 때 new 연산자를 사용합니다.

기타 도움말 항목

배열을 사용한 작업

일반 표현식 사용

ActionScript를 사용하여 MovieClip 객체 만들기

일반적 프로그램 요소

ActionScript 프로그램을 만들 때 사용 가능한 추가적인 구성 단위가 몇 가지 있습니다.

연산자

연산자는 계산하는 데 사용되는 특수 기호이며 단어인 경우도 있습니다. 대부분 수학 연산에 사용되며 각 연산 값을 서로 비교하는 경우에도 사용됩니다. 일반적으로 연산자는 하나 이상의 값을 사용하여 하나의 결과로 "산출"합니다. 예를 들면 다음과 같습니다.

- 더하기 연산자(+)는 다음과 같이 두 개의 값을 서로 더하여 단일 숫자로 결과를 산출합니다.

  ```
  var sum:Number = 23 + 32;
  ```

- 곱하기 연산자(*)는 하나의 값을 다른 값으로 곱하여 단일 숫자로 결과를 산출합니다.

  ```
  var energy:Number = mass * speedOfLight * speedOfLight;
  ```

- 항등 연산자(==)는 두 개의 값이 서로 같은지 비교하여 true 또는 false(Boolean) 단일 값으로 결과를 산출합니다.

  ```
  if (dayOfWeek == "Wednesday")
  {
      takeOutTrash();
  }
  ```

위의 예제에서와 같이 항등 연산자와 기타 "비교" 연산자는 특정 명령을 수행할 것인지 여부를 결정하기 위해 대부분의 경우 if 문과 함께 사용됩니다.

설명

ActionScript를 작성할 때 나중을 대비해 메모를 남길 필요가 있는 경우가 많습니다. 예를 들어 특정 코드 행이 작동하는 방식 또는 특정 항목을 선택한 이유에 대해 설명하고 싶을 수 있습니다. **코드 주석**은 코드에서 무시할 수 있는 텍스트를 작성하는 데 사용할 수 있는 도구입니다. ActionScript에는 다음과 같은 두 가지 주석이 있습니다.

- 한 줄 주석: 한 줄 주석은 줄에서 원하는 위치에 두 개의 슬래시를 배치하여 지정합니다. 슬래시 이후부터 해당 줄 끝까지의 모든 내용이 무시됩니다.

```
// This is a comment; it's ignored by the computer.
var age:Number = 10; // Set the age to 10 by default.
```

- 여러 줄 주석: 여러 줄 주석에는 주석 시작 표시자(/*) 뒤에 주석 내용이 포함된 다음 주석 끝 표시자(*/)가 포함됩니다. 주석이 포함된 줄의 수와 상관없이 시작 표시자와 끝 표시자 사이의 모든 내용이 무시됩니다.

```
/*
This is a long description explaining what a particular
function is used for or explaining a section of code.

In any case, the computer ignores these lines.
*/
```

주석의 또 다른 일반적인 용도는 하나 이상의 코드 행을 일시적으로 "해제"하는 것입니다. 예를 들어 작업을 수행하는 다른 방법을 테스트하는 경우에 주석을 사용할 수 있으며, 특정 ActionScript 코드가 예상대로 작동되지 않는 이유를 파악하려고 하는 경우에도 주석을 사용할 수 있습니다.

흐름 제어

프로그램에서 특정 작업을 반복하거나 특정 작업만 수행하거나 특정 조건에 따라 대체 작업을 수행하려는 경우가 자주 있습니다. **흐름 제어**는 수행되는 작업을 제어합니다. ActionScript에서 사용할 수 있는 여러 가지 흐름 제어 요소는 다음과 같습니다.

- 함수: 함수는 단축키와 같은 역할을 합니다. 단일 이름으로 일련의 작업을 그룹화하여 계산을 수행하는 데 사용할 수 있습니다. 특히 함수는 이벤트를 처리하는 데 필수적이지만 일련의 명령을 그룹화하는 일반적인 도구로도 사용됩니다.
- 루프: 루프 구조에서는 특정 횟수 또는 일부 조건이 변경될 때까지 컴퓨터에서 수행할 명령 집합을 지정할 수 있습니다. 주로 루프는 컴퓨터에서 루프를 통해 작업할 때마다 값이 변하는 변수를 사용하여 여러 관련 항목을 조작하는 데 사용됩니다.
- 조건문: 조건문은 특정 상황에서만 수행되는 특정 명령을 지정하는 방법을 제공합니다. 또한 다른 조건에 맞는 대체 명령 집합을 제공하는 데 사용됩니다. 가장 일반적인 조건문의 유형은 if 문입니다. if 문은 괄호 안에 있는 값 또는 표현식을 검사합니다. 값이 true이면 중괄호 안의 코드 행이 실행됩니다. 그렇지 않으면 코드 행이 무시됩니다. 예를 들면 다음과 같습니다.

```
if (age < 20)
{
    // show special teenager-targeted content
}
```

if 문과 함께 else 문을 사용하면 조건이 true가 아닌 경우 수행할 다른 명령을 지정할 수 있습니다.

```
if (username == "admin")
{
    // do some administrator-only things, like showing extra options
}
else
{
    // do some non-administrator things
}
```

예제: 애니메이션 포트폴리오 작업(Flash Professional)

이 예제는 여러 ActionScript 조각을 결합하여 하나의 완전한 응용 프로그램을 만드는 방법을 처음으로 살펴보기 위한 것입니다. 애니메이션 포트폴리오 작업은 기존의 선형 애니메이션을 기반으로 적절한 약간의 대화형 요소를 추가할 수 있는 방법을 보여주는 예제입니다. 예를 들어 특정 클라이언트용으로 만든 애니메이션을 온라인 포트폴리오에 통합할 수 있습니다. 애니메이션에 추가할 대화형 비헤이비어에는 보는 사람이 클릭할 수 있는 두 개의 버튼, 즉 애니메이션을 시작하는 버튼 및 별도의 URL(예: 포트폴리오 메뉴 또는 제작자의 홈 페이지)로 이동하는 버튼이 포함됩니다.

이 작업은 다음 주요 부분으로 나눌 수 있습니다.

1 ActionScript 및 대화형 요소 추가를 위한 FLA 파일 준비

2 버튼 만들기 및 추가

3 ActionScript 코드 작성

4 응용 프로그램 테스트.

대화형 요소 추가 준비

대화형 요소를 애니메이션에 추가하기 전에 FLA 파일을 설정하여 새 내용을 추가할 위치를 만들어 놓는 것이 좋습니다. 이 작업에는 버튼을 배치할 수 있는 실제 공간을 스테이지에 만드는 것이 포함됩니다. 또한 서로 다른 항목을 별도로 유지할 수 있는 "공간"을 FLA 파일에 만드는 것도 포함됩니다.

대화형 요소를 추가하기 위해 FLA를 설정하려면:

1 단일 모션 트윈 또는 모양 트윈 같은 단순한 애니메이션을 포함하는 FLA 파일을 만듭니다. 프로젝트에 표시할 애니메이션을 포함하는 FLA 파일이 이미 있으면 해당 파일을 열고 새 이름으로 파일을 저장합니다.

2 두 버튼이 화면에 표시되도록 할 위치를 결정합니다. 한 버튼은 애니메이션을 시작하는 버튼이고 다른 버튼은 제작자 포트폴리오 또는 홈 페이지에 연결하는 버튼입니다. 필요한 경우, 스테이지에서 공간을 없애거나 이 새 내용을 배치할 공간을 추가합니다. 애니메이션에 시작 화면이 없으면 첫 프레임에 시작 화면을 만들 수 있습니다. 이 경우 애니메이션을 이동하여 프레임 2 이후에서 애니메이션이 시작되도록 할 수도 있습니다.

3 타임라인에 있는 다른 레이어 위에 새 레이어를 추가한 다음 이름을 **buttons**로 지정합니다. 이 레이어에서 버튼을 추가하게 됩니다.

4 이 buttons 레이어 위에 새 레이어를 추가한 다음 이름을 **actions**로 지정합니다. 이 레이어에서 ActionScript 코드를 사용자 응용 프로그램에 추가하게 됩니다.

버튼 만들기 및 추가

다음에는 대화형 응용 프로그램의 중심이 될 버튼을 실제로 만들어 배치합니다.

버튼을 만들어 FLA에 추가하려면:

1 드로잉 도구를 사용하여 buttons 레이어에 첫 번째 버튼("재생" 버튼) 모양을 만듭니다. 예를 들어 수평 타원형을 그려 그 위에 텍스트를 표시합니다.

2 [선택 도구]를 사용하여 이 버튼의 그래픽 부분을 모두 선택합니다.

3 주 메뉴에서 [수정] > [심볼로 변환]을 선택합니다.

4 표시되는 대화 상자에서 심볼 유형으로 [버튼]을 선택하고 심볼의 이름을 지정한 후 [확인]을 클릭합니다.

5 이 버튼을 선택한 다음, 속성 관리자에서 버튼의 인스턴스 이름을 **playButton**으로 지정합니다.

6 1-5단계를 반복하여 제작자의 홈 페이지로 연결되는 버튼을 만들고, 이 버튼의 이름을 **homeButton**으로 지정합니다.

코드 작성

이 응용 프로그램의 ActionScript 코드는 모두 동일한 위치에 입력되지만, 세 개의 기능 집합으로 나눌 수 있습니다. 코드가 수행하는 작업은 다음 세 가지입니다.

- SWF 파일 로드 즉시(재생 헤드가 프레임 1에 진입 시) 재생 헤드 중지
- 사용자가 재생 버튼 클릭 시 이벤트를 수신하여 SWF 파일 재생 시작
- 사용자가 제작자 홈 페이지 버튼 클릭 시 이벤트를 수신하여 브라우저를 해당 URL로 보냄

재생 헤드가 프레임 1에 진입 시 재생 헤드를 중지하도록 코드를 작성하려면:

1. actions 레이어의 프레임 1에서 키프레임을 선택합니다.
2. [액션] 패널을 열려면 주 메뉴에서 [윈도우] > [액션]을 선택합니다.
3. [스크립트] 창에 다음 코드를 입력합니다.

```
stop();
```

재생 버튼 클릭 시 애니메이션이 시작되도록 코드를 작성하려면:

1. 앞 단계에서 입력한 코드 맨 끝에 빈 행을 두 개 삽입합니다.
2. 스크립트 아래쪽에 다음 코드를 입력합니다.

```
function startMovie(event:MouseEvent):void
{
    this.play();
}
```

이 코드는 startMovie()라는 함수를 정의합니다. startMovie()가 호출되면 이 함수에 의해 기본 타임라인에서 재생이 시작됩니다.

3. 앞 단계에서 추가된 코드 다음 행에 다음 코드 행을 입력합니다.

```
playButton.addEventListener(MouseEvent.CLICK, startMovie);
```

이 코드 행은 startMovie() 함수를 playButton의 click 이벤트에 대한 리스너로 등록합니다. 즉, playButton이라는 버튼이 클릭될 때마다 startMovie() 함수가 호출되도록 하는 것입니다.

홈 페이지 버튼 클릭 시 브라우저에서 해당 URL로 이동하도록 코드를 작성하려면:

1. 앞 단계에서 입력한 코드 맨 끝에 빈 행을 두 개 삽입합니다.
2. 스크립트 아래쪽에 다음 코드를 입력합니다.

```
function gotoAuthorPage(event:MouseEvent):void
{
    var targetURL:URLRequest = new URLRequest("http://example.com/");
    navigateToURL(targetURL);
}
```

이 코드는 gotoAuthorPage()라는 함수를 정의합니다. 이 함수는 먼저 http://example.com/ URL을 나타내는 URLRequest 인스턴스를 만듭니다. 그런 다음 해당 URL을 navigateToURL() 함수에 전달하여 사용자의 브라우저에서 이 URL이 열리도록 합니다.

3. 앞 단계에서 추가된 코드 다음 행에 다음 코드 행을 입력합니다.

```
homeButton.addEventListener(MouseEvent.CLICK, gotoAuthorPage);
```

이 코드 행은 gotoAuthorPage() 함수를 homeButton의 click 이벤트에 대한 리스너로 등록합니다. 즉, homeButton이라는 버튼이 클릭될 때마다 gotoAuthorPage() 함수가 호출되도록 하는 것입니다.

응용 프로그램 테스트

이제 응용 프로그램이 완벽히 작동됩니다. 실제로 작동되는지 테스트해 보겠습니다.

응용 프로그램을 테스트하려면:

1. 주 메뉴에서 [컨트롤] > [무비 테스트]를 선택합니다. Flash Professional에서는 SWF 파일을 생성하여 이 파일을 Flash Player 윈도우에서 엽니다.
2. 두 버튼이 모두 의도한 대로 작동하는지 테스트합니다.
3. 버튼이 작동하지 않는 경우 다음 사항을 확인합니다.
   - 두 버튼의 인스턴스 이름이 각각 다릅니까?
   - addEventListener() 메서드 호출 시 두 버튼의 인스턴스 이름과 동일한 이름을 사용합니까?
   - addEventListener() 메서드 호출 시 올바른 이벤트 이름이 사용되었습니까?
   - 각 함수에 대해 올바른 매개 변수가 지정되었습니까? 두 메서드 모두 MouseEvent 데이터 유형을 가진 단일 매개 변수가 지정되어야 합니다.

   이러한 모든 실수 및 가능한 대부분의 다른 실수로 인해 오류 메시지가 나타납니다. 오류 메시지는 [동영상 테스트] 명령을 선택하거나 프로젝트 테스트 중에 버튼을 클릭할 때 나타날 수 있습니다. [컴파일러 오류] 패널에서 [동영상 테스트]를 처음 선택할 때 발생하는 컴파일러 오류가 있는지 확인합니다. [출력] 패널에서 내용을 재생하는 동안(예: 버튼을 클릭하는 경우) 발생하는 런타임 오류가 있는지 확인합니다.

ActionScript로 응용 프로그램 만들기

ActionScript를 작성하여 응용 프로그램을 만들려면 사용할 클래스 이름과 구문을 아는 것만으로는 부족합니다. 대부분의 Flash 플랫폼 설명서에서는 구문 및 ActionScript 클래스 사용이라는 두 항목을 다루고 있습니다. 그러나 ActionScript 응용 프로그램을 작성하려면 다음과 같은 정보도 필요합니다.

- ActionScript 작성에 사용 가능한 프로그램
- ActionScript 코드를 구성하는 방법
- 응용 프로그램에 ActionScript 코드를 포함하는 방법
- ActionScript 응용 프로그램을 개발하기 위해 따라야 하는 단계

코드 구성에 대한 옵션

ActionScript 3.0 코드를 사용하여 단순한 그래픽 애니메이션에서 복잡한 클라이언트 서버 트랜잭션 처리 시스템까지의 모든 영역에서 프로그램을 개발할 수 있습니다. 만드는 응용 프로그램의 유형에 따라 프로젝트에 ActionScript를 포함하는 여러 방법 중 하나 이상을 사용합니다.

Flash Professional 타임라인의 프레임에 코드 저장

Flash Professional에서 타임라인의 모든 프레임에 ActionScript 코드를 추가할 수 있습니다. 이 코드는 동영상을 재생하는 동안 재생 헤드가 해당 프레임에 진입할 때 실행됩니다.

프레임에 ActionScript 코드를 추가하면 Flash Professional에서 만들어진 응용 프로그램에 비헤이비어를 간단하게 추가할 수 있습니다. 기본 타임라인의 모든 프레임에 코드를 추가하거나 모든 MovieClip 심볼에 대한 타임라인의 모든 프레임에 코드를 추가할 수 있습니다. 그러나 이와 같은 유연한 코딩은 이후 관리가 쉽지 않습니다. 대규모 응용 프로그램을 만드는 경우 어떤 프레임에 어떤 스크립트가 포함되어 있는지 잊기 쉽습니다. 이러한 복잡한 구조로 인해 시간이 지날수록 응용 프로그램을 유지 관리하는 것이 보다 어려워질 수 있습니다.

많은 개발자들은 타임라인의 첫 번째 프레임 또는 Flash 문서의 특정 레이어에만 코드를 추가하여 Flash Professional에서 ActionScript 코드 구성을 단순화합니다. 코드를 분리하면 Flash FLA 파일에서 코드를 찾고 유지는 작업이 보다 쉬워집니다. 그러나 동일한 코드를 다른 Flash Professional 프로젝트에서 사용하려면 해당 코드를 복사하여 새 파일에 붙여 넣어야 하는 단점이 있습니다.

나중에 ActionScript 코드를 다른 Flash Professional 프로젝트에서 쉽게 사용할 수 있도록 하려면 코드를 외부 ActionScript 파일(확장자가 .as인 텍스트 파일)에 저장합니다.

Flex MXML 파일에 코드 포함

Flash Builder 등의 Flex 개발 환경에서는 ActionScript 코드를 Flex MXML 파일의 <fx:Script> 태그 안에 포함할 수 있습니다. 그러나 이 방법을 사용하면 대형 프로젝트의 복잡성이 더욱 증가되고 다른 Flex 프로젝트에서 동일한 코드를 사용하는 것 또한 더 어려워집니다. 따라서 나중에 ActionScript 코드를 다른 Flex 프로젝트에서 쉽게 사용할 수 있도록 하려면 코드를 외부 ActionScript 파일에 저장하는 것이 좋습니다.

참고: <fx:Script> 태그에 대해 source 매개 변수를 지정할 수 있습니다. source 매개 변수를 사용하면 ActionScript 코드를 <fx:Script> 태그 내에 직접 입력된 것처럼 외부 파일에서 "가져올 수 있습니다". 그러나 사용하는 소스 파일에서 고유한 클래스를 정의할 수 없으므로 활용도는 제한됩니다.

ActionScript 파일에 코드 저장

프로젝트에 중요한 ActionScript 코드가 포함되는 경우 코드를 구성하는 가장 좋은 방법은 별도의 ActionScript 소스 파일(확장명이 .as인 텍스트 파일)에 코드를 구성하는 것입니다. 응용 프로그램에서 파일을 사용할 용도에 따라 둘 중 한 가지 방법으로 ActionScript 파일을 구성할 수 있습니다.

- 구조화되지 않은 ActionScript 코드: 타임라인 스크립트 또는 MXML 파일에 직접 입력된 것처럼 작성된 ActionScript 코드 행(명령문 또는 함수 정의 포함)입니다.

  ActionScript의 include 문 또는 Flex MXML의 <fx:Script> 태그를 사용하면 이러한 방식으로 작성된 ActionScript에 액세스할 수 있습니다. ActionScript include 문은 스크립트에서 특정 위치 및 지정된 범위 내에 외부 ActionScript 파일의 내용을 포함하도록 컴파일러에게 지시합니다. 그 결과는 코드를 직접 입력한 것과 동일합니다. MXML 언어에서는 source 특성이 있는 <fx:Script> 태그를 사용하여 컴파일러가 응용 프로그램의 해당 위치에 로드할 외부 ActionScript를 식별합니다. 예를 들어 다음 태그는 Box.as라는 외부 ActionScript 파일을 로드합니다.

  ```
  <fx:Script source="Box.as" />
  ```

- ActionScript 클래스 정의: 클래스의 메서드 및 속성 정의를 포함하는 ActionScript 클래스의 정의입니다.

  클래스를 정의하면 해당 클래스의 인스턴스를 만들고 해당 속성, 메서드 및 이벤트를 사용하여 클래스의 ActionScript 코드에 액세스할 수 있습니다. 직접 정의한 클래스는 내장 ActionScript 클래스와 동일하게 사용할 수 있으며 다음 두 가지 요건을 갖추어야 합니다.

  - ActionScript 컴파일러에서 해당 이름을 찾을 수 있도록 import 문을 사용하여 클래스의 전체 이름을 지정합니다. 예를 들어 ActionScript에서 MovieClip 클래스를 사용하려면 패키지 및 클래스를 포함하는 전체 이름을 사용하여 클래스를 가져옵니다.

    ```
    import flash.display.MovieClip;
    ```

    또는 MovieClip 클래스가 포함된 패키지를 가져올 수 있습니다. 이는 패키지의 각 클래스에 대해 별도의 import 문을 작성하는 것과 동일합니다.

    ```
    import flash.display.*;
    ```

    코드에서 클래스를 사용하려면 해당 클래스를 가져와야 한다는 이 규칙은 최상위 클래스에는 적용되지 않습니다. 이러한 클래스는 패키지 내에 정의되지 않습니다.

  - 클래스 이름을 명확히 사용하는 코드를 작성합니다. 예를 들어 해당 클래스를 데이터 유형으로 하는 변수를 선언하고 해당 변수에 저장할 이 클래스의 인스턴스를 만듭니다. ActionScript 코드에서 클래스를 사용하여 컴파일러에서 해당 클래스 정의를 로드하도록 지시합니다. 예를 들어 Box라는 외부 클래스가 있을 경우 다음 명령문은 Box 클래스의 인스턴스를 만듭니다.

```
var smallBox:Box = new Box(10,20);
```

컴파일러는 Box 클래스에 대한 참조를 처음 발견하면 Box 클래스 정의를 로드하기 위해 사용 가능한 소스 코드를 검색합니다.

올바른 도구 선택

여러 도구 중 하나 또는 여러 도구를 함께 사용하여 ActionScript 코드를 작성 및 편집할 수 있습니다.

Flash Builder

Adobe Flash Builder는 Flex 프레임워크에 기반하는 프로젝트 또는 ActionScript 코드로만 구성되는 프로젝트를 만드는 데 사용되는 고급 도구입니다. Flash Builder에는 시각적 레이아웃 및 MXML 편집 기능 외에 완벽한 ActionScript 편집기가 포함되어 있으므로 Flex 또는 ActionScript 전용 프로젝트를 만드는 데 사용할 수 있습니다. Flex는 원격 데이터를 사용하여 작업하고 외부 데이터를 사용자 인터페이스 요소에 연결할 수 있는 내장 메커니즘, 유연한 동적 레이아웃 제어 및 미리 만든 사용자 인터페이스 제어 등의 여러 가지 이점이 있습니다. 그러나 이러한 기능을 제공하려면 코드를 추가해야 하기 때문에 Flex를 사용하는 프로젝트에서 SWF 파일의 크기가 Flex를 사용하지 않는 경우에 비해 커질 수 있습니다.

따라서 Flash Builder는 Flex를 사용하여 모든 기능이 제공되는 데이터 기반의 RIA(Rich Internet Application)를 만들거나 단일 도구 내에서 ActionScript 코드와 MXML 코드를 편집하고 응용 프로그램을 시각적으로 배치하려는 경우 사용하는 것이 좋습니다.

ActionScript를 많이 사용하는 프로젝트를 작성하는 대부분의 Flash Professional 사용자는 Flash Professional을 사용하여 시각적 에셋을 만들고 Flash Builder를 ActionScript 코드 편집기로 사용합니다.

Flash Professional

Flash Professional에는 그래픽과 애니메이션을 만드는 기능 외에도 ActionScript 코드로 작업할 수 있는 여러 도구가 포함되어 있습니다. 코드는 FLA 파일 또는 외부 ActionScript 전용 파일의 요소에 연결될 수 있습니다. Flash Professional은 중요한 애니메이션 또는 비디오를 포함하는 프로젝트에 적합하고, 대부분의 그래픽 에셋을 직접 만들려는 경우에 유용합니다. 동일한 응용 프로그램에서 시각적 에셋을 만들고 코드를 작성할 수 있다는 장점도 ActionScript 프로젝트 개발에 Flash Professional을 사용하는 이유 중 하나입니다. Flash Professional에는 미리 만든 사용자 인터페이스 구성 요소도 포함되어 있습니다. 이러한 구성 요소를 사용하여 SWF 파일의 크기를 줄이고 시각적 도구로 프로젝트에 맞게 구성 요소를 스키닝할 수도 있습니다.

Flash Professional에는 다음과 같이 ActionScript 코드 작성을 위한 두 가지 도구가 포함되어 있습니다.

- [액션] 패널: FLA 파일에서 작업할 때 사용할 수 있으며 이 패널을 사용하면 타임라인의 프레임에 첨부된 ActionScript 코드를 작성할 수 있습니다.
- [스크립트] 윈도우: [스크립트] 윈도우는 ActionScript(.as) 코드 파일을 사용하여 작업하는 전용 텍스트 편집기입니다.

타사 ActionScript 편집기

ActionScript(.as) 파일은 간단한 텍스트 파일로 저장되기 때문에 ActionScript 파일을 작성하는 데 일반 텍스트 파일을 편집할 수 있는 모든 프로그램을 사용할 수 있습니다. Adobe ActionScript 제품 외에도 ActionScript 전용 기능이 포함된 여러 타사 텍스트 편집 프로그램이 있습니다. 모든 텍스트 편집기 프로그램을 사용하여 MXML 파일 또는 ActionScript 클래스를 작성할 수 있습니다. 그런 다음 Flex SDK를 사용하여 해당 파일을 토대로 응용 프로그램을 만들 수 있습니다. 프로젝트는 Flex를 사용하는 응용 프로그램이거나 ActionScript 전용 응용 프로그램일 수 있습니다. 일부 개발자는 Flash Builder 또는 타사 ActionScript 편집기를 사용하여 ActionScript 클래스를 작성하고 이와 함께 Flash Professional을 사용하여 그래픽 내용을 만드는 방법을 선택하기도 합니다.

타사 ActionScript 편집기를 선택하는 이유는 다음과 같습니다.

- 별도 프로그램에서 ActionScript 코드를 작성하고 시각적 요소는 Flash Professional에서 디자인하는 것을 선호하는 경우
- 다른 프로그래밍 언어로 응용 프로그램을 작성하거나 HTML 페이지를 만드는 것과 같이 비ActionScript 프로그래밍을 위한 응용 프로그램을 사용하고 있으며 ActionScript 코딩에도 동일한 응용 프로그램을 사용하려는 경우
- Flash Professional 또는 Flex Builder 대신 Flex SDK를 사용하여 ActionScript 전용 또는 Flex 프로젝트를 만들려는 경우

ActionScript 전용 기능을 지원하는 주요 코드 편집기를 몇 가지 소개하면 다음과 같습니다.

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate(ActionScript 및 Flex 번들 포함)

ActionScript 개발 프로세스

ActionScript 프로젝트의 규모에 관계없이 응용 프로그램 설계 및 개발 프로세스를 사용하면 보다 효율적이고 효과적으로 작업을 수행할 수 있습니다. 다음 단계는 ActionScript 3.0을 사용하는 응용 프로그램을 만들기 위한 기본 개발 프로세스를 설명합니다.

1 응용 프로그램을 설계합니다.

응용 프로그램을 만들기 전에 특정 방식으로 해당 응용 프로그램을 기술합니다.

2 ActionScript 3.0 코드를 구성합니다.

Flash Professional, Flash Builder, Dreamweaver 또는 텍스트 편집기를 사용하여 ActionScript 코드를 만들 수 있습니다.

3 코드를 실행할 Flash 또는 Flex 프로젝트를 만듭니다.

Flash Professional에서는 FLA 파일을 만들고, 제작 설정을 지정하고, 사용자 인터페이스 구성 요소를 응용 프로그램에 추가하며, ActionScript 코드를 참조합니다. Flex에서는 응용 프로그램을 정의하고, MXML을 사용하여 사용자 인터페이스 구성 요소를 추가하고, ActionScript 코드를 참조합니다.

4 ActionScript 응용 프로그램을 제작하고 테스트합니다.

응용 프로그램 테스트 과정에는 해당 개발 환경에서 응용 프로그램을 실행하는 작업이 포함되며 응용 프로그램이 의도한 대로 작동하는지 확인해야 합니다.

이러한 단계를 순서대로 수행하지 않아도 되며 다른 작업을 수행하기 전에 수행 중이던 단계를 끝내야 할 필요는 없습니다. 예를 들어 1단계에서 응용 프로그램의 화면을 설계한 다음 3단계에서 그래픽, 버튼 등을 만들 수 있습니다. 그런 다음 2단계에서 ActionScript 코드를 작성하고 4단계에서 테스트합니다. 또는 응용 프로그램의 일부를 설계한 다음 버튼 또는 인터페이스 요소를 한 번에 하나씩 추가하고 각 요소에 대한 ActionScript를 작성하여 만든 응용 프로그램을 테스트할 수 있습니다. 개발 프로세스의 네 단계에 따라 개발 작업을 진행하는 것이 좋습니다. 그러나 실제 개발 환경에서는 앞 단계 또는 뒤 단계로 적절하게 이동하여 작업하는 것이 보다 효과적입니다.

클래스 만들기

프로젝트에 사용할 클래스를 만드는 과정은 어려워 보일 수 있습니다. 그러나 클래스를 만드는 과정에서 보다 어려운 부분은 클래스의 메서드, 속성 및 이벤트를 설계하는 작업입니다.

클래스 설계 전략

객체 지향 설계는 이 분야의 학술적 연구와 실무에 모든 경력을 바친 사람에게도 어렵게 느껴지는 복잡한 주제입니다. 그러나 전문가가 아니더라도 객체 지향 설계를 시작할 수 있는 몇 가지 방법이 제안되어 있습니다.

1 이 클래스의 인스턴스가 응용 프로그램에서 수행할 역할에 대해 생각해 봅니다. 일반적으로 객체는 다음과 같은 세 가지 역할 중 하나를 수행합니다.

- 값 객체: 이러한 객체는 주로 데이터의 컨테이너 역할을 하고, 여러 속성과 적은 수의 메서드(또는 전혀 없음)가 포함되어 있는 경우가 많습니다. 이러한 객체는 일반적으로 명확히 정의된 항목에 대한 코드 표현입니다. 예를 들어 음악 플레이어 응용 프로그램은 실제 노래 한 곡을 나타내는 Song 클래스 및 개념적인 노래 그룹을 나타내는 Playlist 클래스를 포함할 수 있습니다.
- 표시 객체: 실제로 화면에 나타나는 객체입니다. 예를 들어 드롭 다운 목록 또는 상태 판독과 같은 사용자 인터페이스 요소, 비디오 게임의 내용과 같은 그래픽 요소 등이 있습니다.
- 응용 프로그램 구조: 이러한 객체는 응용 프로그램에서 수행하는 처리 또는 논리에서 광범위한 지원 역할을 수행합니다. 예를 들어 생물학 시뮬레이션에서 특정 계산을 수행할 객체를 만들거나 음악 플레이어 응용 프로그램에서 다이얼 컨트롤과 볼륨 판독 간에 값의 동기화를 수행하는 객체를 만들 수 있습니다. 비디오 게임에서 규칙을 관리할 수 있는 객체도 이러한 유형에 해당합니다. 또한 드로잉 응용 프로그램에서 저장된 그림을 로드할 클래스를 만들 수도 있습니다.

2 클래스에 필요한 특정 기능을 결정합니다. 서로 다른 유형의 기능이 클래스의 여러 메서드로 표현되는 경우가 자주 있습니다.

3 클래스를 값 객체로 사용하려는 경우 인스턴스에 포함할 데이터를 결정합니다. 이러한 항목은 속성으로 정의하기에 적합합니다.

4 특정 프로젝트 전용으로 클래스를 설계 중이므로 응용 프로그램에 필요한 기능을 제공하는 것이 가장 중요합니다. 다음과 같은 질문의 답을 스스로 찾아보는 것이 도움이 됩니다.

- 응용 프로그램에서 저장, 추적 및 조작할 정보는 무엇입니까? 이 질문에 대한 답을 찾아보는 것은 필요한 모든 값 객체 및 속성을 식별하는 데 유용합니다.
- 응용 프로그램에서 수행하는 작업에는 무엇이 있습니까? 예를 들어 응용 프로그램을 처음 로드하는 경우, 특정 버튼을 클릭한 경우, 동영상 재생이 중지된 경우 등에 어떤 일이 발생합니까? 이러한 발생 내용은 메서드로 정의하기에 적합합니다. 해당 "작업"에서 개별 값만 변경하는 경우에는 메서드 대신 속성으로 정의할 수도 있습니다.
- 지정된 작업이 있는 경우 해당 작업을 수행하기 위해 필요한 정보는 무엇입니까? 이러한 정보는 메서드의 매개 변수가 됩니다.
- 응용 프로그램에서 작업을 진행할 때 응용 프로그램의 다른 부분에 전달해야 할 클래스의 변경 내용은 무엇입니까? 이러한 변경 내용은 이벤트로 정의하기에 적합합니다.

5 추가하려는 일부 기능은 정의되어 있지 않지만 필요한 객체와 유사한 기존 객체가 있습니까? 그렇다면 하위 클래스를 만드는 것이 좋습니다. **하위 클래스**는 클래스 자체 기능을 모두 정의하기 보다는 기존 클래스의 기능을 기반으로 하는 클래스입니다. 예를 들어 화면에서 시각적 객체로 표시될 클래스를 만들려는 경우 기존 표시 객체의 비헤이비어를 클래스에 대한 기준으로 사용합니다. 이 경우, **기본 클래스**는 MovieClip 또는 Sprite 같은 표시 객체이며 사용자 정의 클래스는 기본 클래스를 확장합니다.

클래스의 코드 작성

클래스 설계 계획이 있는 경우 또는 최소한 클래스에서 저장할 정보 및 클래스에서 수행할 작업을 생각해 둔 경우 클래스를 작성하는 실제 구문은 매우 간단합니다.

ActionScript 클래스를 만드는 최소 단계는 다음과 같습니다.

1 ActionScript 텍스트 편집기 프로그램에서 새 텍스트 문서를 엽니다.

2 class 문을 입력하여 클래스 이름을 정의합니다. class 문을 추가하려면 public class를 입력한 다음 클래스 이름을 입력합니다. 클래스 내용(메서드 및 속성 정의)을 둘러싸는 열기 및 닫기 중괄호를 추가합니다. 예를 들면 다음과 같습니다.

```
public class MyClass
{
}
```

public은 다른 모든 코드에서 클래스에 액세스할 수 있다는 것을 나타냅니다. 다른 예제에 대한 자세한 내용은 액세스 제어 네임스페이스 특성을 참조하십시오.

3 package 문을 입력하여 클래스를 포함하는 패키지의 이름을 지정합니다. 이 구문에서는 package 다음에 전체 패키지 이름이 사용되고 class 문 블록을 둘러싸는 열기 및 닫기 중괄호가 사용됩니다. 예를 들어 이전 단계의 코드를 다음과 같이 변경합니다.

```
package mypackage
{
    public class MyClass
    {
    }
}
```

4 클래스 본문 내에서 var 문을 사용하여 클래스에 각 속성을 정의합니다. 구문은 public 수정자를 추가하여 모든 변수를 선언할 때 사용한 구문과 동일합니다. 예를 들어 클래스 정의의 여는 중괄호와 닫는 중괄호 사이에 다음 줄을 추가하면 textProperty, numericProperty 및 dateProperty 속성이 만들어집니다.

```
public var textProperty:String = "some default value";
public var numericProperty:Number = 17;
public var dateProperty:Date;
```

5 함수를 정의하는 데 사용된 구문과 동일한 구문을 사용하여 클래스에 각 메서드를 정의합니다. 예를 들면 다음과 같습니다.

- myMethod() 메서드를 만들려면 다음을 입력합니다.

  ```
  public function myMethod(param1:String, param2:Number):void
  {
      // do something with parameters
  }
  ```

- 클래스의 인스턴스를 만드는 과정 중에 호출되는 특수 메서드인 생성자를 만들려면 클래스 이름과 정확하게 일치하는 메서드 이름을 만듭니다.

  ```
  public function MyClass()
  {
      // do stuff to set initial values for properties
      // and otherwise set up the object
      textVariable = "Hello there!";
      dateVariable = new Date(2001, 5, 11);
  }
  ```

  클래스에 생성자 메서드가 포함되어 있지 않으면 컴파일러에서 비어 있는 생성자, 즉 매개 변수와 명령문이 없는 생성자를 클래스에 자동으로 만듭니다.

정의할 수 있는 클래스 요소가 몇 가지 더 있으며 이러한 요소는 다른 요소보다 복잡합니다.

- 접근자는 메서드와 속성 사이의 특수한 중간적 요소입니다. 코드를 작성하여 클래스를 정의할 때 메서드와 동일하게 접근자를 작성합니다. 접근자를 작성하면 속성을 정의한 후 단순히 값을 읽고 지정할 수 있었던 것에 비해 여러 가지 작업을 수행할 수 있습니다. 그러나 클래스의 인스턴스를 만들면 접근자는 속성과 같이 처리되고 접근자 이름을 사용하여 값을 읽거나 지정할 수 있습니다.
- ActionScript의 이벤트는 특정 구문을 사용하여 정의하지 않습니다. 대신 EventDispatcher 클래스 기능을 사용하여 클래스에 이벤트를 정의합니다.

기타 도움말 항목

이벤트 처리

예제: 기본 응용 프로그램 만들기

ActionScript 3.0은 Flash Professional 및 Flash Builder 도구 또는 모든 텍스트 편집기를 비롯한 여러 응용 프로그램 개발 환경에 사용할 수 있습니다.

이 예제에서는 Flash Professional 또는 Flash Builder를 사용하여 간단한 ActionScript 3.0 응용 프로그램을 만들고 향상시킬 수 있는 단계에 대해 설명합니다. 작성할 응용 프로그램에서는 Flash Professional 및 Flex에서 외부 ActionScript 3.0 클래스 파일을 사용하는 간단한 패턴을 제공합니다.

ActionScript 응용 프로그램 설계

이 ActionScript 응용 프로그램 예제는 표준 "Hello World" 응용 프로그램이며 이에 대한 설계는 간단합니다.

- 이 응용 프로그램의 이름은 HelloWorld이고,
- "Hello World!"가 포함된 단일 텍스트 필드를 표시합니다.
- 이 응용 프로그램에서는 Greeter라는 객체 지향 클래스 하나만 사용합니다. 이 설계에 의하면 Flash Professional 또는 Flex 프로젝트에서 Greeter 클래스를 사용할 수 있습니다.
- 이 예제에서는 먼저 응용 프로그램의 기본 버전을 만듭니다. 그런 다음 사용자가 사용자 이름을 입력하면 응용 프로그램이 이 이름을 알려진 사용자 목록에서 확인하도록 기능을 추가합니다.

간결한 정의가 완료되었으므로 응용 프로그램 작성을 시작할 수 있습니다.

HelloWorld 프로젝트 및 Greeter 클래스 만들기

Hello World 응용 프로그램의 설계 문장에 따르면 코드는 쉽게 재사용할 수 있어야 합니다. 이 목적에 부합하도록 이 응용 프로그램에서는 Greeter라는 객체 지향 클래스 하나만 사용합니다. 이 클래스는 Flash Builder 또는 Flash Professional에서 만들 응용 프로그램에서 사용합니다.

Flex에서 HelloWorld 프로젝트 및 Greeter 클래스를 만들려면

1. Flash Builder에서 [File] > [New] > [Flex Project]를 선택합니다.
2. 프로젝트 이름으로 HelloWorld를 입력합니다. 응용 프로그램 유형이 "Web (runs in Adobe Flash Player)"으로 설정되어 있는지 확인하고 [Finish]를 클릭합니다.

   Flash Builder에서는 사용자가 지정한 프로젝트를 만들고 Package Explorer에 해당 프로젝트를 표시합니다. 기본적으로 HelloWorld.mxml 파일이 프로젝트에 이미 포함되어 있으며, 편집기에서 해당 파일이 열립니다.
3. 이제 Flash Builder에서 사용자 정의 ActionScript 클래스 파일을 만들기 위해 [File] > [New] > [ActionScript Class]를 선택합니다.
4. [New ActionScript Class] 대화 상자의 [Name] 필드에 클래스 이름으로 **Greeter**를 입력한 다음 [Finish]를 클릭합니다.

   새 ActionScript 편집 윈도우가 나타납니다.

   계속해서 Greeter 클래스에 코드 추가를 진행합니다.

Flash Professional에서 Greeter 클래스를 만들려면

1. Flash Professional에서 [파일] > [새로 만들기]를 선택합니다.
2. [새 문서] 대화 상자에서 [ActionScript 파일]을 선택하고 [확인]을 클릭합니다.

   새 ActionScript 편집 윈도우가 나타납니다.
3. [파일] > [저장]을 선택합니다. 응용 프로그램을 넣을 폴더를 선택하고 ActionScript 파일의 이름을 **Greeter.as**로 변경한 다음 [확인]을 클릭합니다.

계속해서 Greeter 클래스에 코드 추가를 진행합니다.

Greeter 클래스에 코드 추가

Greeter 클래스는 HelloWorld 응용 프로그램에서 사용할 객체인 Greeter를 정의합니다.

Greeter 클래스에 코드를 추가하려면:

1 새 파일에 다음 코드를 입력합니다(코드 중 일부가 추가되었을 수 있음).

```
package
{
    public class Greeter
    {
        public function sayHello():String
        {
            var greeting:String;
            greeting = "Hello World!";
            return greeting;
        }
    }
}
```

Greeter 클래스에는 "Hello World!"라고 인사하는 문자열을 반환하는 단일 sayHello() 메서드가 포함되어 있습니다.

2 [파일] > [저장]을 선택하여 이 ActionScript 파일을 저장합니다.

이제 응용 프로그램에서 Greeter 클래스를 사용할 수 있습니다.

ActionScript 코드를 사용하는 응용 프로그램 만들기

앞에서 만든 Greeter 클래스는 소프트웨어 기능의 독립된 집합을 정의하지만 완전한 응용 프로그램을 나타내는 것은 아닙니다. 이 클래스를 사용하려면 Flash Professional 문서 또는 Flex 프로젝트를 만들어야 합니다.

코드에서는 Greeter 클래스의 인스턴스가 필요합니다. 이 설명서에 Greeter 클래스를 응용 프로그램에서 사용하는 방법이 설명되어 있습니다.

Flash Professional을 사용하여 ActionScript 응용 프로그램을 만들려면

1 [파일] > [새로 만들기]를 선택합니다.

2 [새 문서] 대화 상자에서 [Flash 파일(ActionScript 3.0)]을 선택하고 [확인]을 클릭합니다.

새 문서 윈도우가 나타납니다.

3 [파일] > [저장]을 선택합니다. Greeter.as 클래스 파일이 들어 있는 동일한 폴더를 선택하여 Flash 문서를 **HelloWorld.fla**로 지정한 다음 [확인]을 클릭합니다.

4 Flash Professional 도구 팔레트에서 [텍스트 도구]를 선택합니다. 스테이지에서 드래그하여 300 픽셀(폭) x 100 픽셀(높이) 정도 크기의 새 텍스트 필드를 정의합니다.

5 스테이지의 텍스트 필드를 선택한 상태로, [속성] 패널에서 텍스트 유형을 "동적 텍스트"로 설정하고 텍스트 필드의 인스턴스 이름으로 **mainText**를 입력합니다.

6 기본 타임라인의 첫 번째 프레임을 클릭합니다. [윈도우] > [액션]을 선택하여 [액션] 패널을 엽니다.

7 [액션] 패널에 다음 스크립트를 입력합니다.

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello();
```

8 파일을 저장합니다.

계속해서 ActionScript 응용 프로그램 제작 및 테스트를 진행합니다.

Flash Builder를 사용하여 ActionScript 응용 프로그램을 만들려면

1 HelloWorld.mxml 파일을 열고 다음 목록과 일치하도록 코드를 추가합니다.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400"/>

</s:Application>
```

이 Flex 프로젝트에는 다음과 같은 네 가지 MXML 태그가 포함되어 있습니다.

- 응용 프로그램 컨테이너를 정의하는 <s:Application> 태그
- Application 태그의 레이아웃 스타일(세로 레이아웃)을 정의하는 <s:layout> 태그
- 일부 ActionScript 코드가 포함되는 <fx:Script> 태그
- 사용자에게 텍스트 메시지를 표시할 필드를 정의하는 <s:TextArea> 태그

<fx:Script> 태그의 코드는 응용 프로그램 로드 시 호출되는 initApp() 메서드를 정의합니다. initApp() 메서드는 mainTxt TextArea의 텍스트 값을 "Hello World!"로 설정합니다. 이 문자열은 방금 작성한 사용자 정의 클래스 Greeter의 sayHello() 메서드에서 반환된 문자열입니다.

2 [File] > [Save]를 선택하여 응용 프로그램을 저장합니다.

계속해서 ActionScript 응용 프로그램 제작 및 테스트를 진행합니다.

ActionScript 응용 프로그램 제작 및 테스트

Software 개발은 반복되는 과정입니다. 코드를 작성하고 컴파일한 다음 해당 코드가 오류 없이 컴파일될 때까지 코드를 편집합니다. 컴파일된 응용 프로그램을 실행하고 테스트하여 의도했던 설계가 실현되었는지 확인합니다. 원하는 설계가 실현되지 않은 경우 실현될 때까지 코드를 다시 편집합니다. Flash Professional 및 Flash Builder 개발 환경에서는 응용 프로그램을 제작, 테스트 및 디버깅하는 여러 가지 방법을 제공합니다.

이 설명서에는 각 환경에서 HelloWorld 응용 프로그램을 테스트하기 위한 기본 단계가 설명되어 있습니다.

Flash Professional을 사용하여 ActionScript 응용 프로그램을 제작 및 테스트하려면

1 응용 프로그램을 제작한 후 컴파일 오류가 있는지 확인합니다. Flash Professional에서 [컨트롤] > [동영상 테스트]를 선택하여 ActionScript 코드를 컴파일하고 HelloWorld 응용 프로그램을 실행합니다.

2 응용 프로그램 테스트 시 [출력] 윈도우에 오류나 경고가 나타나면 HelloWorld.fla 파일이나 HelloWorld.as 파일에서 오류를 수정합니다. 그런 다음 응용 프로그램을 다시 테스트해 봅니다.

3 컴파일 오류가 없는 경우에는 Hello World 응용 프로그램이 표시된 Flash Player 윈도우가 나타납니다.

단순하지만 완전한 ActionScript 3.0 사용 객체 지향 응용 프로그램을 만들었습니다. 계속해서 HelloWorld 응용 프로그램 개선을 진행합니다.

Flash Builder를 사용하여 ActionScript 응용 프로그램을 제작 및 테스트하려면

1 [Run] > [Run HelloWorld]를 선택합니다.

2 HelloWorld 응용 프로그램이 시작됩니다.

- 응용 프로그램 테스트 시 [Output] 윈도우에 오류나 경고가 나타나면 HelloWorld.mxml 파일이나 Greeter.as 파일에서 오류를 수정합니다. 그런 다음 응용 프로그램을 다시 테스트해 봅니다.
- 컴파일 오류가 없는 경우에는 Hello World 응용 프로그램이 표시된 브라우저 윈도우가 열리고 "Hello World!" 텍스트가 나타납니다.

단순하지만 완전한 ActionScript 3.0 사용 객체 지향 응용 프로그램을 만들었습니다. 계속해서 HelloWorld 응용 프로그램 개선을 진행합니다.

HelloWorld 응용 프로그램 개선

보다 재미있는 응용 프로그램을 만들기 위해 응용 프로그램에서 사용자에게 이름을 입력할 것을 요청하고 사용자가 입력한 이름이 유효한지 미리 정의된 이름 목록을 통해 확인하도록 합니다.

먼저 Greeter 클래스를 업데이트하고 새 기능을 추가합니다. 그런 다음 응용 프로그램을 업데이트하여 새 기능을 사용합니다.

Greeter.as 파일을 업데이트하려면:

1 Greeter.as 파일을 엽니다.

2 파일 내용을 다음과 같이 변경합니다. 새로 추가된 행과 변경된 행은 굵은체로 표시됩니다.

```
package
{
    public class Greeter
    {
        /**
         * Defines the names that receive a proper greeting.
         */
        public static var validNames:Array = ["Sammy", "Frank", "Dean"];

        /**
         * Builds a greeting string using the given name.
         */
        public function sayHello(userName:String = ""):String
        {
            var greeting:String;
            if (userName == "")
            {
                greeting = "Hello. Please type your user name, and then press "
                           + "the Enter key.";
            }
            else if (validName(userName))
            {
                greeting = "Hello, " + userName + ".";
            }
            else
            {
```

```
                greeting = "Sorry " + userName + ", you are not on the list.";
            }
            return greeting;
        }

        /**
         * Checks whether a name is in the validNames list.
         */
        public static function validName(inputName:String = ""):Boolean
        {
            if (validNames.indexOf(inputName) > -1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

Greeter 클래스에 여러 가지 새로운 기능이 포함되었습니다.

- validNames 배열에 유효한 사용자 이름이 나열됩니다. 이 배열은 Greeter 클래스가 로드되면 세 개의 이름이 있는 목록으로 초기화됩니다.
- 이제 sayHello() 메서드에서 사용자 이름을 받아들이며 일부 조건에 따라 인사말을 변경합니다. userName이 비어 있는 문자열("")인 경우 사용자에게 이름을 묻는 메시지가 나타나도록 greeting 속성이 설정됩니다. 사용자 이름이 유효한 경우 인사말은 "Hello, **userName**"입니다. 마지막으로 위의 두 조건이 모두 충족되지 않은 경우 greeting 변수가 "Sorry **userName**, you are not on the list"로 설정됩니다.
- validNames 배열에 inputName이 있으면 validName() 메서드가 true를 반환하고 그렇지 않으면 false를 반환합니다. validNames.indexOf(inputName) 명령문은 inputName 문자열에 대해 validNames 배열에 있는 각 문자열을 검사합니다. Array.indexOf() 메서드는 배열에 있는 객체의 첫 번째 인스턴스에 대한 인덱스 위치를 반환합니다. 배열에 객체가 없는 경우에는 값 -1을 반환합니다.

다음으로 이 ActionScript 클래스를 참조하는 응용 프로그램 파일을 편집합니다.

Flash Professional을 사용하여 응용 프로그램을 수정하려면

1. HelloWorld.fla 파일을 엽니다.
2. 비어 있는 문자열("")이 Greeter 클래스의 sayHello() 메서드에 전달되도록 프레임 1의 스크립트를 다음과 같이 수정합니다.

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");
```

3. 도구 팔레트에서 [텍스트 도구]를 선택한 다음 스테이지에 새 텍스트 필드를 두 개 만듭니다. 두 텍스트 필드를 기존의 mainText 텍스트 필드 바로 아래에 나란히 배치합니다.
4. 레이블인 첫 번째 새 텍스트 필드에 텍스트 **User Name:**을 입력합니다.
5. 나머지 새 텍스트 필드를 선택한 다음 속성 관리자에서 텍스트 필드 유형으로 [입력 텍스트]를 선택합니다. [행 유형]으로 [한 행]을 선택합니다. 인스턴스 이름으로 **textIn**을 입력합니다.
6. 기본 타임라인의 첫 번째 프레임을 클릭합니다.
7. [액션] 패널에서 기존 스크립트의 맨 끝에 다음 행을 추가합니다.

```
mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

이 새 코드는 다음 기능을 추가합니다.

- 처음 두 행은 두 텍스트 필드의 경계를 정의하는 역할만 합니다.
- textIn 필드와 같은 입력 텍스트 필드에는 전달 가능한 이벤트 집합이 포함되어 있습니다. addEventListener() 메서드를 사용하여 특정 유형의 이벤트 발생 시 실행되는 함수를 정의할 수 있습니다. 이 경우의 이벤트는 키보드의 키를 누르는 동작입니다.
- keyPressed() 사용자 정의 함수는 누른 키가 Enter 키인지 여부를 확인합니다. Enter 키인 경우 myGreeter 객체의 sayHello() 메서드를 호출하고 textIn 텍스트 필드의 텍스트를 매개 변수로 전달합니다. 이 메서드는 전달된 값을 기반으로 하여 인사말 문자열을 반환합니다. 그러면 반환된 문자열은 mainText 텍스트 필드의 text 속성에 할당됩니다.

프레임 1에 대한 전체 스크립트는 다음과 같습니다.

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");

mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

8 파일을 저장합니다.

9 [컨트롤] > [무비 테스트]를 선택하여 응용 프로그램을 실행합니다.

응용 프로그램을 실행하면 사용자 이름을 입력하라는 메시지가 나타납니다. 사용자 이름이 유효하면(예: Sammy, Frank, Dean 등) 응용 프로그램에서는 "hello"라는 확인 메시지를 표시합니다.

Flash Builder를 사용하여 응용 프로그램을 수정하려면

1 HelloWorld.mxml 파일을 엽니다.

2 그런 다음 <mx:TextArea> 태그를 수정하여 TextArea 구성 요소가 표시 전용임을 사용자에게 나타냅니다. 배경색을 밝은 회색으로 변경하고 editable 특성을 false로 설정합니다.

```
<s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false" />
```

3 이제 <s:TextArea> 닫는 태그 바로 뒤에 다음 행을 추가합니다. 이 행에서는 사용자가 사용자 이름 값을 입력할 수 있는 TextInput 구성 요소를 만듭니다.

```
    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>
```

enter 특성은 사용자가 userNameTxt 필드에서 Enter 키를 누를 경우 발생하는 내용을 정의합니다. 이 예제에서 코드는 해당 필드의 텍스트를 Greeter.sayHello() 메서드로 전달합니다. 이에 따라 mainTxt 필드의 인사말이 변경됩니다.

HelloWorld.mxml 파일의 내용은 이제 다음과 같습니다.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false"/>

    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>

</s:Application>
```

4 편집한 HelloWorld.mxml 파일을 저장합니다. [Run] > [Run HelloWorld]를 선택하여 응용 프로그램을 실행합니다.

응용 프로그램을 실행하면 사용자 이름을 입력하라는 메시지가 나타납니다. 사용자 이름이 유효하면(예: Sammy, Frank, Dean 등) 응용 프로그램에서는 "Hello, **userName**"이라는 확인 메시지를 표시합니다.

3장: ActionScript 언어 및 구문

ActionScript 3.0에는 기본 ActionScript 언어와 Adobe Flash 플랫폼 API(Application Programming Interface)가 포함되어 있습니다. 기본 언어는 ActionScript에서 언어의 구문과 최상위 데이터 유형을 정의하는 부분입니다. ActionScript 3.0은 Adobe Flash Player 및 Adobe AIR 같은 Adobe Flash 플랫폼 런타임에 대해 프로그래밍 방식으로 액세스할 수 있도록 해줍니다.

언어 개요

객체는 ActionScript 3.0 언어의 핵심 개념이며 기본적인 구성 단위입니다. 객체는 선언된 모든 변수, 작성된 모든 함수 및 만들어진 모든 클래스 인스턴스입니다. ActionScript 3.0 프로그램은 작업을 수행하고 이벤트에 응답하고 서로 통신하는 객체의 그룹으로 간주할 수 있습니다.

Java 또는 C++의 OOP(객체 지향 프로그래밍)에 익숙한 프로그래머는 객체를 두 가지 멤버가 포함된 모듈로 생각할 수 있습니다. 즉, 데이터는 멤버 변수나 속성에 저장되고 비헤이비어는 메서드를 통해 액세스할 수 있는 것으로 생각할 수 있습니다. ActionScript 3.0에서는 이와 비슷하지만 약간 다른 방식으로 객체를 정의합니다. ActionScript 3.0에서 객체는 단지 속성 모음일 뿐입니다. 이러한 속성은 데이터뿐만 아니라 함수 또는 다른 객체도 저장할 수 있는 컨테이너입니다. 이 방식으로 객체에 연결되는 함수를 메서드라고 합니다.

Java 또는 C++에 대한 지식이 있는 프로그래머에게는 ActionScript 3.0의 정의가 조금 이상하게 보일 수도 있지만 실제로 ActionScript 3.0 클래스를 사용하여 객체 유형을 정의하는 것은 Java 또는 C++로 클래스를 정의하는 방식과 매우 유사합니다. ActionScript 객체 모델과 기타 고급 항목을 설명하는 경우에는 두 객체 정의를 구분하는 것이 중요하지만 대부분의 경우 **속성**이라는 용어는 메서드가 아닌 클래스 멤버 변수를 의미합니다. 예를 들어 ActionScript 3.0 Reference for the Adobe Flash Platform에서 **속성**이라는 용어는 변수 또는 getter-setter 속성을 의미하는 데 사용됩니다. **메서드**라는 용어는 클래스의 일부인 함수를 의미합니다.

Java 또는 C++와 달리 ActionScript에서는 클래스가 추상적인 엔터티만은 아니라는 미묘한 차이점이 있습니다. ActionScript 클래스는 클래스의 속성과 메서드를 저장하는 **클래스 객체**로 구체화됩니다. 이로 인해 클래스 또는 패키지의 최상위에서 명령문이나 실행 코드를 포함하는 등 Java 및 C++ 프로그래머에게는 생소할 수 있는 기법을 사용하는 것이 가능합니다.

ActionScript 클래스와 Java 또는 C++ 클래스 간의 또 다른 차이점은 모든 ActionScript 클래스에는 **프로토타입 객체**가 있다는 점입니다. 이전 버전의 ActionScript에서는 여러 프로토타입 객체를 **프로토타입 체인**에 함께 연결하여 전체 클래스 상속 계층 구조의 기초로 제공했습니다. 그러나 ActionScript 3.0의 상속 시스템에서는 프로토타입 객체가 작은 역할만 담당합니다. 비록 역할이 축소되기는 했지만 클래스의 모든 인스턴스에서 속성과 해당 값을 공유하려는 경우 이전처럼 프로토타입 객체를 정적 속성 및 메서드 대신 사용할 수 있습니다.

이전에는 고급 ActionScript 프로그래머가 특수 내장 언어 요소를 사용하여 프로토타입 체인을 직접 조작할 수 있었습니다. 이제 클래스 기반 프로그래밍 인터페이스의 구현이 더욱 향상되어 __proto__ 및 __resolve 등 많은 특수 언어 요소가 더 이상 언어에 포함되지 않습니다. 또한 내부 상속 메커니즘을 최적화한 결과 성능은 크게 향상되었지만 상속 메커니즘에 직접 액세스할 수 없게 되었습니다.

객체 및 클래스

ActionScript 3.0에서 모든 객체는 클래스로 정의됩니다. 클래스는 객체 유형의 청사진 또는 템플릿으로 간주할 수 있습니다. 클래스 정의에 변수, 상수 및 메서드를 포함할 수 있습니다. 변수 및 상수에는 데이터 값이 저장되며, 메서드는 클래스에 연결된 비헤이비어를 캡슐화하는 함수입니다. 속성에 저장된 값은 **프리미티브 값** 또는 다른 객체일 수 있습니다. 프리미티브 값에는 Number, String 및 부울 값이 있습니다.

ActionScript에는 여러 내장 클래스가 기본 언어의 일부로 포함되어 있습니다. Number, Boolean 및 String과 같은 일부 내장 클래스는 ActionScript에서 사용할 수 있는 프리미티브 값을 나타냅니다. 그 밖에 Array, Math 및 XML과 같은 클래스는 보다 복잡한 객체를 정의합니다.

내장되어 있거나 사용자 정의된 모든 클래스는 Object 클래스에서 파생됩니다. 이전의 ActionScript에 익숙한 프로그래머의 경우, 모든 클래스가 여전히 Object 클래스에서 파생되지만 Object 데이터 유형이 더 이상 기본 데이터 유형이 아니라는 점에 유의해야 합니다. ActionScript 2.0에서는 유형 약어가 생략된 변수를 Object 유형으로 간주하므로 다음과 같은 두 개의 코드 행이 동일했습니다.

```
var someObj:Object;
var someObj;
```

그러나 ActionScript 3.0에서는 유형이 지정되지 않은 변수의 개념을 소개하며 이는 다음과 같은 두 가지 방법으로 지정할 수 있습니다.

```
var someObj:*;
var someObj;
```

유형이 지정되지 않은 변수는 Object 유형의 변수와 다릅니다. 주요 차이점은 유형이 지정되지 않은 변수에는 undefined 특수 값을 저장할 수 있는 반면 Object 유형의 변수에는 이 값을 저장할 수 없다는 것입니다.

class 키워드를 사용하여 클래스를 직접 정의할 수 있습니다. 클래스 속성은 세 가지 방법으로 선언할 수 있습니다. 상수는 const 키워드를 사용하여 정의하고, 변수는 var 키워드를 사용하여 정의하며, getter 및 setter 속성은 메서드 선언에서 get 및 set 특성을 사용하여 정의할 수 있습니다. 메서드는 function 키워드를 사용하여 선언할 수 있습니다.

new 연산자를 사용하여 클래스의 인스턴스를 만듭니다. 다음 예제에서는 myBirthday라는 Date 클래스의 인스턴스를 만듭니다.

```
var myBirthday:Date = new Date();
```

패키지 및 네임스페이스

패키지와 네임스페이스는 서로 관련된 개념입니다. 패키지를 사용하면 클래스 정의를 묶음 처리하여 쉽게 코드를 공유하고 이름이 충돌하는 것을 최소화할 수 있습니다. 네임스페이스를 사용하면 속성 및 메서드 이름과 같은 식별자의 가시성을 제어할 수 있습니다. 네임스페이스는 패키지 내부 또는 외부에 있는 코드에 적용할 수 있습니다. 패키지를 사용하여 클래스 파일을 구성할 수 있으며, 네임스페이스를 사용하여 개별 속성 및 메서드의 가시성을 관리할 수 있습니다.

패키지

ActionScript 3.0의 패키지는 네임스페이스로 구현되지만 네임스페이스와 동일한 의미로 사용되지는 않습니다. 패키지를 선언하면 특수한 유형의 네임스페이스가 자동으로 만들어지며 이 네임스페이스는 컴파일 타임에 확인할 수 있습니다. 명시적으로 만든 네임스페이스는 컴파일 타임에 확인되지 않을 수도 있습니다.

다음 예제에서는 package 지시문을 사용하여 하나의 클래스가 포함된 단순한 패키지를 만듭니다.

```
package samples
{
    public class SampleCode
    {
        public var sampleGreeting:String;
        public function sampleFunction()
        {
            trace(sampleGreeting + " from sampleFunction()");
        }
    }
}
```

이 예제에서 클래스 이름은 SampleCode입니다. 이 클래스는 samples 패키지 내에 있으므로 클래스 이름이 컴파일 타임에 컴파일러에 의해 samples.SampleCode로 자동으로 정규화됩니다. 또한 컴파일러에서 모든 속성 또는 메서드의 이름도 정규화하므로 sampleGreeting 및 sampleFunction()은 각각 samples.SampleCode.sampleGreeting 및 samples.SampleCode.sampleFunction()이 됩니다.

대부분의 개발자 특히, Java 프로그래밍 경력이 있는 개발자는 패키지의 최상위에 클래스만 배치하려고 할 수 있습니다. 그러나 ActionScript 3.0에서는 패키지의 최상위에 클래스뿐만 아니라 변수, 함수 및 명령문도 배치할 수 있습니다. 이 기능의 고급 용도 중 하나는 패키지의 최상위에 네임스페이스를 정의하여 해당 패키지의 모든 클래스에서 사용할 수 있다는 것입니다. 그러나 패키지의 최상위에서는 두 개의 액세스 지정자 즉, public 및 internal만 사용할 수 있습니다. 중첩된 클래스를 전용(private)으로 선언할 수 있는 Java와 달리 ActionScript 3.0에서는 중첩된 클래스 또는 전용 클래스를 지원하지 않습니다.

ActionScript 3.0 패키지는 여러 면에서 Java 프로그래밍 언어로 구현된 패키지와 유사합니다. 이전 예제에서 확인할 수 있듯이 Java의 경우와 마찬가지로 도트 연산자(.)를 사용하여 정규화된 패키지 참조를 표시합니다. 패키지를 사용하면 코드를 직관적인 계층 구조로 구성하여 다른 프로그래머에게 제공할 수 있습니다. 이와 같이 다른 사람과 공유할 패키지를 만들고 다른 사람이 만든 패키지를 코드에 사용할 수 있으므로 코드 공유가 촉진됩니다.

패키지를 사용하면 사용할 식별자 이름을 고유하게 만들어 다른 식별자 이름과 충돌하지 않도록 할 수도 있습니다. 실제로 일부 사용자는 이 기능이 패키지의 주요 장점이라고 말합니다. 예를 들어 서로의 코드를 공유하려는 두 명의 프로그래머가 SampleCode라는 클래스를 각각 만들 수 있습니다. 패키지를 사용하지 않으면 이름이 충돌하며 이 문제는 하나의 클래스 이름을 변경해야 해결할 수 있습니다. 그러나 패키지를 사용하면 두 클래스 중 하나를 패키지에 배치하거나 둘 모두를 이름이 다른 두 개의 패키지에 각각 배치하여 이름 충돌을 피할 수 있습니다.

패키지 이름에 도트를 사용하여 중첩된 패키지를 만들 수도 있습니다. 이렇게 하면 패키지의 계층적 구조를 만들 수 있습니다. ActionScript 3.0에서 제공하는 flash.display 패키지는 이에 대한 좋은 예입니다. flash.display 패키지는 flash 패키지 내에 중첩됩니다.

ActionScript 3.0은 대부분 flash 패키지 아래에 구성됩니다. 예를 들어, flash.display 패키지에 표시 목록 API가 포함되어 있으며, flash.events 패키지에는 새 이벤트 모델이 포함되어 있습니다.

패키지 만들기

ActionScript 3.0에서는 패키지, 클래스 및 소스 파일을 매우 유연하게 구성할 수 있습니다. 이전 버전의 ActionScript에서는 소스 파일당 하나의 클래스만 허용되었으며 소스 파일 이름이 클래스 이름과 일치해야 했습니다. ActionScript 3.0에서는 하나의 소스 파일에 여러 클래스를 포함할 수 있지만 해당 파일의 외부에 있는 코드에는 각 파일의 클래스를 하나만 사용할 수 있습니다. 즉, 패키지 선언 내에는 각 파일의 클래스를 하나만 선언할 수 있습니다. 추가적인 클래스는 패키지 정의 외부에 선언하여 소스 파일 외부에 있는 코드에서 추가적인 클래스를 참조할 수 없도록 합니다. 패키지 정의 내에 선언된 클래스 이름은 소스 파일 이름과 일치해야 합니다.

ActionScript 3.0에서 패키지를 보다 융통성 있게 선언할 수 있습니다. 이전 버전의 ActionScript에서 패키지는 단순히 소스 파일이 있는 디렉토리를 나타냈으며 사용자는 package 문을 사용하여 패키지를 선언하지 않고 클래스 선언에 정규화된 클래스 이름의 일부로 패키지 이름을 포함했습니다. ActionScript 3.0에서도 패키지는 디렉토리를 나타내지만 패키지에 클래스 이외의 요소를 포함할 수 있습니다. ActionScript 3.0에서는 package 문을 사용하여 패키지를 선언하므로 패키지의 최상위에서 변수, 함수 및 네임스페이스를 선언할 수도 있습니다. 또한 패키지의 최상위에 실행 가능한 명령문을 포함할 수 있습니다. 패키지의 최상위에서 변수, 함수 또는 네임스페이스를 선언하는 경우 해당 수준에서 public 및 internal 특성만 사용할 수 있으며 선언 대상이 클래스, 변수, 함수 또는 네임스페이스인지에 관계없이 파일당 하나의 패키지 수준 선언에서만 public 특성을 사용할 수 있습니다.

패키지는 코드를 체계화하고 이름 충돌을 방지하는 데 유용합니다. 패키지 개념은 클래스 상속 개념과 관련이 없으므로 이 둘을 혼동해서는 안됩니다. 동일한 패키지에 있는 두 개의 클래스는 공통된 네임스페이스에 있지만 그 외에 다른 방식으로 상호 관련되어야 하는 것은 아닙니다. 마찬가지로 중첩된 패키지와 상위 패키지 간에 아무런 의미적 관련성이 없을 수 있습니다.

패키지 가져오기

패키지 내의 클래스를 사용하려면 패키지 또는 특정 클래스 중 하나를 가져와야 합니다. 이는 클래스 가져오기를 생략해도 되었던 ActionScript 2.0과 다른 점입니다.

예를 들어 앞에서 제공된 SampleCode 클래스 예제에 대해 살펴 보십시오. samples라는 패키지에 있는 SampleCode 클래스를 사용하려면 다음 import 문 중 하나를 사용해야 합니다.

```
import samples.*;
```

또는

```
import samples.SampleCode;
```

일반적으로 import 문은 최대한 구체적이어야 합니다. samples 패키지에서 SampleCode 클래스만 사용하려는 경우 SampleCode 클래스가 속해 있는 전체 패키지가 아닌 SampleCode 클래스만 가져와야 합니다. 전체 패키지를 가져오면 예기치 못한 이름 충돌이 발생할 수 있습니다.

패키지 또는 클래스를 정의하는 소스 코드도 **클래스 경로** 내에 배치해야 합니다. 클래스 경로는 컴파일러가 가져온 패키지 및 클래스를 검색할 위치를 결정하는 로컬 디렉토리 경로의 사용자 정의 목록입니다. 클래스 경로는 **빌드 경로** 또는 **소스 경로**라고도 합니다.

클래스 또는 패키지를 올바르게 가져오면 정규화된 클래스 이름(samples.SampleCode) 또는 클래스 자체 이름(SampleCode) 중 하나를 사용할 수 있습니다.

정규화된 이름은 동일하게 이름이 지정된 클래스, 메서드 또는 속성으로 인해 코드가 모호해지는 경우 유용하지만 모든 식별자에 사용하면 관리하기 어려울 수 있습니다. 예를 들어, SampleCode 클래스 인스턴스를 인스턴스화할 때 정규화된 이름을 사용하면 코드가 길어집니다.

```
var mySample:samples.SampleCode = new samples.SampleCode();
```

패키지가 중첩될수록 코드의 가독성은 떨어집니다. 모호한 식별자가 문제가 되지 않는 경우에는 간단한 식별자를 사용하여 더 읽기 쉬운 코드를 만들 수 있습니다. 예를 들어, 클래스 식별자만 사용하는 경우 다음과 같이 SampleCode 클래스의 새 인스턴스를 인스턴스화하는 것이 훨씬 간단합니다.

```
var mySample:SampleCode = new SampleCode();
```

먼저 적절한 패키지 또는 클래스를 가져오지 않고 식별자 이름을 사용하려고 하면 컴파일러에서 클래스 정의를 찾을 수 없게 됩니다. 반면에 패키지 또는 클래스를 가져온 경우 가져온 이름과 충돌하는 이름을 정의하려고 하면 오류가 발생합니다.

패키지를 만든 경우 해당 패키지의 모든 멤버에 대한 기본 액세스 지정자는 internal입니다. 이는 기본적으로 패키지 멤버를 해당 패키지의 다른 멤버만 참조할 수 있다는 것을 의미합니다. 패키지 외부의 코드에서 클래스를 사용할 수 있도록 하려면 해당 클래스를 public으로 선언해야 합니다. 예를 들어, 다음 패키지에 SampleCode 및 CodeFormatter 클래스가 포함되어 있습니다.

```
// SampleCode.as file
package samples
{
    public class SampleCode {}
}

// CodeFormatter.as file
package samples
{
    class CodeFormatter {}
}
```

SampleCode 클래스는 public 클래스로 선언되어 있으므로 패키지 외부에서 참조할 수 있습니다. 그러나 CodeFormatter 클래스는 samples 패키지 내에서만 참조할 수 있습니다. samples 패키지 외부에서 CodeFormatter 클래스에 액세스하려고 하면 다음 예제와 같이 오류가 발생합니다.

```
import samples.SampleCode;
import samples.CodeFormatter;
var mySample:SampleCode = new SampleCode(); // okay, public class
var myFormatter:CodeFormatter = new CodeFormatter(); // error
```

패키지 외부에서 두 클래스를 모두 사용할 수 있도록 하려면 두 클래스를 모두 public으로 선언해야 합니다. 패키지 선언에는 public 특성을 적용할 수 없습니다.

정규화된 이름은 패키지를 사용할 때 발생할 수 있는 이름 충돌 문제를 해결하는 데 유용합니다. 동일한 식별자를 사용하여 클래스를 정의하는 두 개의 패키지를 가져오는 경우 이름 충돌 문제가 발생할 수 있습니다. 예를 들어 마찬가지로 SampleCode라는 클래스가 포함되어 있는 다음 패키지를 살펴보십시오.

```
package langref.samples
{
    public class SampleCode {}
}
```

두 클래스를 모두 가져오면 SampleCode 클래스를 사용할 때 다음과 같이 이름이 충돌하게 됩니다.

```
import samples.SampleCode;
import langref.samples.SampleCode;
var mySample:SampleCode = new SampleCode(); // name conflict
```

컴파일러는 어떤 SampleCode 클래스를 사용해야 하는지 알 수 없습니다. 이 충돌 문제를 해결하려면 다음과 같이 각 클래스의 정규화된 이름을 사용해야 합니다.

```
var sample1:samples.SampleCode = new samples.SampleCode();
var sample2:langref.samples.SampleCode = new langref.samples.SampleCode();
```

참고: C++ 사용 경력이 있는 프로그래머가 import 문과 #include를 혼동하는 경우가 많습니다. C++ 컴파일러의 경우 파일을 한 번에 하나씩만 처리하고 헤더 파일이 명시적으로 포함되어 있지 않으면 다른 파일에서 클래스 정의를 찾지 않으므로 C++에서 #include 지시문은 필수적인 요소입니다. ActionScript 3.0에는 include 지시문이 있지만 클래스 및 패키지를 가져오기 위한 것이 아닙니다. ActionScript 3.0에서 클래스 또는 패키지를 가져오려면 import 문을 사용하고 패키지가 포함된 소스 파일을 클래스 경로에 배치해야 합니다.

네임스페이스

네임스페이스를 사용하면 사용자가 만든 속성 및 메서드의 가시성을 제어할 수 있습니다. public, private, protected 및 internal 액세스 제어 지정자를 내장 네임스페이스로 간주할 수 있습니다. 이러한 미리 정의된 액세스 제어 지정자가 사용자 요구에 적합하지 않은 경우 직접 네임스페이스를 만들 수 있습니다.

ActionScript 구현의 구문 및 세부 사항이 XML의 구문 및 세부 사항과 약간 다르지만 XML 네임스페이스에 익숙한 사용자에게는 이 설명서의 내용이 낯설게 느껴지지 않을 것입니다. 네임스페이스의 개념은 매우 간단하지만 이전에 네임스페이스를 사용하여 작업한 경험이 없는 경우 이를 구현하려면 특정 용어를 익혀야 합니다.

네임스페이스의 작동을 이해하려면 속성 또는 메서드 이름에 식별자 및 네임스페이스가 항상 포함된다는 점을 이해하는 것이 좋습니다. 식별자는 일반적으로 이름으로 간주되는 것을 말합니다. 예를 들어 다음 클래스 정의에서 식별자는 sampleGreeting과 sampleFunction()입니다.

```
class SampleCode
{
    var sampleGreeting:String;
    function sampleFunction () {
        trace(sampleGreeting + " from sampleFunction()");
    }
}
```

클래스 등을 정의할 때 네임스페이스 특성을 먼저 지정하지 않으면 해당 정의에 사용된 이름이 기본 internal 네임스페이스로 정규화되므로 동일한 패키지에 있는 호출자만 해당 이름을 참조할 수 있습니다. 컴파일러가 엄격 모드로 설정되어 있는 경우 네임스페이스 특성이 없는 모든 식별자에 internal 네임스페이스가 적용된다는 경고가 표시됩니다. 식별자를 모든 범위에서 사용할 수 있도록 하려면 식별자 이름 앞에 public 특성을 명시해야 합니다. 이전 예제 코드에서 sampleGreeting 및 sampleFunction()에는 모두 internal 네임스페이스 값이 적용됩니다.

네임스페이스를 사용할 때 수행해야 할 세 가지 기본 단계는 다음과 같습니다. 첫 번째, namespace 키워드를 사용하여 네임스페이스를 정의해야 합니다. 예를 들어 다음 코드는 version1 네임스페이스를 정의합니다.

```
namespace version1;
```

두 번째, 속성 또는 메서드 선언에서 액세스 제어 지정자 대신 네임스페이스를 사용하여 네임스페이스를 적용합니다. 다음 예제에서는 myFunction() 함수를 version1 네임스페이스에 배치합니다.

```
version1 function myFunction() {}
```

세 번째, 네임스페이스가 적용되면 use 지시문을 사용하거나 식별자 이름을 네임스페이스로 정규화하여 참조할 수 있습니다. 다음 예제에서는 use 지시문을 통해 myFunction() 함수를 참조합니다.

```
use namespace version1;
myFunction();
```

다음 예제에서 볼 수 있는 것과 같이 정규화된 이름을 사용하여 myFunction() 함수를 참조할 수도 있습니다.

```
version1::myFunction();
```

네임스페이스 정의

네임스페이스에는 URI(Uniform Resource Identifier)라는 하나의 값이 포함되며, 이를 **네임스페이스 이**름이라고도 합니다. URI를 사용하여 네임스페이스 정의를 고유하게 만들 수 있습니다.

두 가지 방법 중 하나로 네임스페이스 정의를 선언하여 네임스페이스를 만듭니다. XML 네임스페이스를 정의하듯이 명시적 URI가 있는 네임스페이스를 정의할 수 있으나 URI를 생략하는 것도 가능합니다. 다음 예제에서는 URI를 사용하여 네임스페이스를 정의하는 방법을 보여 줍니다.

```
namespace flash_proxy = "http://www.adobe.com/flash/proxy";
```

URI는 해당 네임스페이스에 대한 고유한 식별 문자열 역할을 합니다. URI를 생략하면 다음 예제에서와 같이 컴파일러에서 URI 대신 고유한 내부 식별 문자열을 만듭니다. 사용자는 이 내부 식별 문자열에 액세스하지 못합니다.

```
namespace flash_proxy;
```

URI 사용 여부와 관계없이 한 번 정의한 네임스페이스는 동일한 범위에서 다시 정의할 수 없습니다. 이전에 정의한 네임스페이스를 동일한 범위에서 정의하려고 하면 컴파일러 오류가 발생합니다.

패키지 또는 클래스 내에 네임스페이스가 정의된 경우 적절한 액세스 제어 지정자를 사용하지 않으면 해당 패키지 또는 클래스 외부의 코드에서 네임스페이스를 참조할 수 없습니다. 예를 들어 다음 코드에서는 flash.utils 패키지 내에 정의된 flash_proxy 네임스페이스를 보여 줍니다. 다음 예제에서는 액세스 제어 지정자가 없으므로 flash.utils 패키지 내의 코드에서만 flash_proxy 네임스페이스를 참조할 수 있고 패키지 외부의 모든 코드에서는 참조할 수 없습니다.

```
package flash.utils
{
    namespace flash_proxy;
}
```

다음 코드에서는 패키지 외부의 코드에서 flash_proxy 네임스페이스를 참조할 수 있도록 public 특성을 사용합니다.

```
package flash.utils
{
    public namespace flash_proxy;
}
```

네임스페이스 적용

네임스페이스를 적용한다는 것은 정의를 네임스페이스에 배치하는 것을 의미합니다. 네임스페이스에 배치할 수 있는 정의에는 함수, 변수 및 상수가 있으며 사용자 정의 네임스페이스에 클래스를 배치할 수는 없습니다.

예를 들어 public 액세스 제어 네임스페이스를 사용하여 선언된 함수를 살펴보십시오. 함수 정의에서 public 특성을 사용하여 함수를 공용 네임스페이스에 배치하면 모든 코드에서 해당 함수를 사용할 수 있습니다. 네임스페이스를 정의하면 public 특성을 사용할 때와 동일한 방법으로 정의된 네임스페이스를 사용할 수 있으며, 사용자 정의 네임스페이스를 참조할 수 있는 코드에서 해당 정의를 사용할 수 있습니다. 예를 들어 example1 네임스페이스를 정의하면 다음 예제에서 볼 수 있는 것과 같이 example1을 특성으로 사용하여 myFunction() 메서드를 추가할 수 있습니다.

```
namespace example1;
class someClass
{
    example1 myFunction() {}
}
```

example1 네임스페이스를 특성으로 사용하여 myFunction() 메서드를 선언하면 이 메서드가 example1 네임스페이스에 속하게 됩니다.

네임스페이스를 적용할 때에는 다음 사항에 주의해야 합니다.

- 각 선언에는 네임스페이스를 하나만 적용할 수 있습니다.
- 한 번에 둘 이상의 정의에 네임스페이스 특성을 적용할 수 없습니다. 즉, 서로 다른 10개의 함수에 네임스페이스를 적용하려면 10개의 각 함수 정의에 네임스페이스를 특성으로 추가해야 합니다.
- 또한 네임스페이스와 액세스 제어 지정자는 상호 배타적이므로 네임스페이스를 적용하는 경우 액세스 제어 지정자를 지정할 수 없습니다. 즉, 네임스페이스를 적용한 함수 또는 속성을 public, private, protected 또는 internal로 선언할 수 없습니다.

네임스페이스 참조

public, private, protected, internal 등의 액세스 제어 네임스페이스 중 하나로 선언된 메서드 또는 속성을 사용할 때 네임스페이스를 명시적으로 참조할 필요는 없습니다. 이러한 특수 네임스페이스에 대한 액세스는 컨텍스트에 의해 제어되기 때문입니다. 예를 들어 정의를 private 네임스페이스에 배치하면 자동으로 동일한 클래스 내의 코드에서 사용할 수 있게 됩니다. 그러나 사용자가 정의한 네임스페이스에는 이와 같은 컨텍스트 민감도가 없습니다. 사용자 정의 네임스페이스에 배치한 메서드 또는 속성을 사용하려면 네임스페이스를 참조해야 합니다.

use namespace 지시문을 사용하여 네임스페이스를 참조하거나, 이름 한정 기호(::)를 사용하여 네임스페이스로 이름을 정규화할 수 있습니다. use namespace 지시문을 사용하여 네임스페이스를 참조하면 네임스페이스가 "열려" 정규화되지 않은 모든 식별자에 해당 네임스페이스를 적용할 수 있습니다. 예를 들어 example1 네임스페이스를 정의한 경우 use namespace example1을 사용하여 해당 네임스페이스에 있는 이름에 액세스할 수 있습니다.

```
use namespace example1;
myFunction();
```

한 번에 둘 이상의 네임스페이스를 열 수 있습니다. use namespace를 사용하여 네임스페이스를 열면 네임스페이스가 열린 코드 블록 전체에서 열린 상태로 남아 있게 됩니다. 열린 네임스페이스를 명시적으로 닫을 수는 없습니다.

그러나 둘 이상의 네임스페이스가 열려 있으면 이름이 충돌할 가능성이 커집니다. 네임스페이스를 열지 않으려면 use namespace 지시문을 사용하는 대신 네임스페이스 및 이름 한정 기호로 메서드 또는 속성 이름을 정규화하면 됩니다. 예를 들어 다음 코드에서는 example1 네임스페이스를 사용하여 myFunction()이라는 이름을 정규화하는 방법을 보여 줍니다.

```
example1::myFunction();
```

네임스페이스 사용

ActionScript 3.0의 일부인 flash.utils.Proxy 클래스에서 이름 충돌을 방지하는 데 사용되는 네임스페이스에 대한 실제 예를 찾아볼 수 있습니다. ActionScript 2.0의 Object.__resolve 속성을 대체하는 Proxy 클래스를 사용하면 정의되지 않은 속성 또는 메서드에 대한 참조를 차단하여 오류가 발생하는 것을 막을 수 있습니다. 이름이 충돌하지 않도록 하려면 Proxy 클래스의 메서드를 모두 flash_proxy 네임스페이스에 배치합니다.

flash_proxy 네임스페이스가 사용되는 방법을 보다 잘 이해하려면 Proxy 클래스를 사용하는 방법을 알아야 합니다. Proxy 클래스의 기능은 이 클래스에서 상속된 클래스에서만 사용할 수 있습니다. 즉, 객체에 Proxy 클래스의 메서드를 사용하려면 객체의 클래스 정의에서 Proxy 클래스를 확장해야 합니다. 예를 들어 정의되지 않은 메서드 호출을 차단하려면 Proxy 클래스를 확장한 다음 Proxy 클래스의 callProperty() 메서드를 재정의합니다.

네임스페이스 구현 과정은 일반적으로 네임스페이스 정의, 적용 및 참조의 3단계로 구성된다는 것을 기억하고 있을 것입니다. 그러나 Proxy 클래스의 어떤 메서드도 명시적으로 호출되지 않기 때문에 flash_proxy 네임스페이스는 정의 및 적용되기는 하지만 참조되지는 않습니다. ActionScript 3.0이 flash_proxy 네임스페이스를 정의하고 이를 Proxy 클래스에 적용합니다. 사용자가 작성하는 코드에서는 Proxy 클래스를 확장하는 클래스에 flash_proxy 네임스페이스를 적용하기만 하면 됩니다.

다음과 유사한 방식으로 flash_proxy 네임스페이스가 flash.utils 패키지에 정의되어 있습니다.

```
package flash.utils
{
    public namespace flash_proxy;
}
```

이 네임스페이스는 Proxy 클래스에서 발췌한 다음 예제에서 볼 수 있는 것과 같이 Proxy 클래스의 메서드에 적용됩니다.

```
public class Proxy
{
    flash_proxy function callProperty(name:*, ... rest):*
    flash_proxy function deleteProperty(name:*):Boolean
    ...
}
```

다음 코드에서 볼 수 있는 것과 같이 먼저 Proxy 클래스와 flash_proxy 네임스페이스를 모두 가져와야 합니다. 그런 다음 Proxy 클래스를 확장하도록 클래스를 선언해야 하며, 엄격 모드에서 컴파일하는 경우에는 dynamic 특성도 추가해야 합니다. callProperty() 메서드를 재정의하는 경우 flash_proxy 네임스페이스를 사용해야 합니다.

```
package
{
    import flash.utils.Proxy;
    import flash.utils.flash_proxy;

    dynamic class MyProxy extends Proxy
    {
        flash_proxy override function callProperty(name:*, ...rest):*
        {
            trace("method call intercepted: " + name);
        }
    }
}
```

MyProxy 클래스의 인스턴스를 만들고 다음 예제에서 호출된 testing() 메서드처럼 정의되지 않은 메서드를 호출하는 경우, Proxy 객체에서 메서드 호출을 차단하고 재정의된 callProperty() 메서드 내의 명령문(이 경우 간단한 trace() 문)을 실행합니다.

```
var mySample:MyProxy = new MyProxy();
mySample.testing(); // method call intercepted: testing
```

flash_proxy 네임스페이스 내에 Proxy 클래스의 메서드를 배치하면 두 가지 장점이 있습니다. 첫 번째, 별도의 네임스페이스를 사용하면 Proxy 클래스를 확장하는 모든 클래스의 공용 인터페이스가 단순화됩니다. Proxy 클래스에서 재정의할 수 있는 메서드는 약 12개이며 모두 직접 호출되도록 설계되지 않았습니다. 공용 네임스페이스에 메서드를 모두 배치하면 혼동될 수 있습니다. 두 번째, Proxy의 하위 클래스에 Proxy 클래스의 메서드와 이름이 일치하는 인스턴스 메서드가 포함되어 있는 경우 flash_proxy 네임스페이스를 사용하면 이름 충돌이 방지됩니다. 예를 들어 사용자가 정의한 메서드 중 하나에 callProperty()라는 이름을 지정할 수 있습니다. 다음 코드에서는 사용자가 정의한 callProperty() 메서드가 다른 네임스페이스에 있으므로 오류가 발생하지 않습니다.

```
dynamic class MyProxy extends Proxy
{
    public function callProperty() {}
    flash_proxy override function callProperty(name:*, ...rest):*
    {
        trace("method call intercepted: " + name);
    }
}
```

네임스페이스는 public, private, internal 및 protected와 같은 네 가지 액세스 제어 지정자로는 불가능한 방식으로 메서드 또는 속성에 대한 액세스를 제공하려는 경우에도 유용합니다. 예를 들어, 몇 가지 유틸리티 메서드가 여러 패키지에 분산되어 있는 경우가 있을 수 있습니다. 이러한 메서드를 모든 패키지에서 사용할 수 있도록 하되 공용으로 지정하지 않으려고 합니다. 이렇게 하려면 네임스페이스를 만들어 사용자 고유의 특수 액세스 제어 지정자로 사용하면 됩니다.

다음 예제에서는 사용자 정의 네임스페이스를 사용하여 다른 패키지에 있는 두 개의 함수를 그룹화합니다. 동일한 네임스페이스로 함수를 그룹화하면 클래스 또는 패키지에서 use namespace 문을 한 번만 사용하여 두 함수를 참조하도록 할 수 있습니다.

이 예제에서는 네 개의 파일을 사용하여 이러한 기법을 설명합니다. 네 개의 파일 모두 클래스 경로에 있어야 합니다. 첫 번째 파일인 myInternal.as는 myInternal 네임스페이스를 정의하는 데 사용됩니다. 파일이 example 패키지에 있으므로 해당 파일을 example 폴더에 배치해야 합니다. 네임스페이스가 public으로 표시되어 있으므로 다른 패키지로 가져올 수 있습니다.

```
// myInternal.as in folder example
package example
{
    public namespace myInternal = "http://www.adobe.com/2006/actionscript/examples";
}
```

두 번째와 세 번째 파일인 Utility.as 및 Helper.as에서는 다른 패키지에서도 사용할 수 있어야 하는 메서드가 포함된 클래스를 정의합니다. Utility 클래스가 example.alpha 패키지에 있으므로 example 폴더의 하위 폴더인 alpha 폴더에 파일을 배치해야 합니다. Helper 클래스가 example.beta 패키지에 있으므로 example 폴더의 하위 폴더인 beta 폴더에 파일을 배치해야 합니다. 이러한 example.alpha 및 example.beta 패키지를 사용하려면 네임스페이스를 가져와야 합니다.

```
// Utility.as in the example/alpha folder
package example.alpha
{
    import example.myInternal;

    public class Utility
    {
        private static var _taskCounter:int = 0;

        public static function someTask()
        {
            _taskCounter++;
        }

        myInternal static function get taskCounter():int
        {
            return _taskCounter;
        }
    }
}

// Helper.as in the example/beta folder
package example.beta
{
    import example.myInternal;

    public class Helper
    {
        private static var _timeStamp:Date;

        public static function someTask()
        {
            _timeStamp = new Date();
        }

        myInternal static function get lastCalled():Date
        {
            return _timeStamp;
        }
    }
}
```

네 번째 파일인 NamespaceUseCase.as에는 기본 응용 프로그램 클래스가 정의되어 있으며 example 폴더와 상위 폴더가 같은 폴더에 있어야 합니다. Flash Professional에서 이 클래스는 FLA의 문서 클래스로 사용됩니다. NamespaceUseCase 클래스는 myInternal 네임스페이스를 가져와서 이를 사용하여 다른 패키지에 있는 두 개의 정적 메서드를 호출하기도 합니다. 예제에서는 코드를 단순화하기 위해서만 정적 메서드를 사용합니다. 정적 메서드와 인스턴스 메서드를 모두 myInternal 네임스페이스에 배치할 수 있습니다.

```
// NamespaceUseCase.as
package
{
    import flash.display.MovieClip;
    import example.myInternal; // import namespace
    import example.alpha.Utility;// import Utility class
    import example.beta.Helper;// import Helper class

    public class NamespaceUseCase extends MovieClip
    {
        public function NamespaceUseCase()
        {
            use namespace myInternal;

            Utility.someTask();
            Utility.someTask();
            trace(Utility.taskCounter); // 2

            Helper.someTask();
            trace(Helper.lastCalled); // [time someTask() was last called]
        }
    }
}
```

변수

변수를 사용하면 프로그램에서 사용하는 값을 저장할 수 있습니다. 변수를 선언하려면 변수 이름과 함께 var 문을 사용해야 합니다. ActionScript 3.0에서는 항상 var 문을 사용해야 합니다. 예를 들어 다음 ActionScript 행에서는 i라는 이름의 변수를 선언합니다.

```
var i;
```

변수를 선언할 때 var 문을 생략하면 엄격 모드에서는 컴파일러 오류가 발생하고, 표준 모드에서는 런타임 오류가 발생합니다. 예를 들어 i 변수가 이미 정의되어 있지 않으면 다음 코드 행에서 오류가 발생합니다.

```
i; // error if i was not previously defined
```

변수를 데이터 유형과 연결하려면 변수를 선언할 때 연결해야 합니다. 변수 유형을 지정하지 않고도 변수를 선언할 수는 있지만 이 경우 엄격 모드에서는 컴파일러 경고가 발생합니다. 변수 이름 뒤에 콜론(:)과 변수 유형을 차례로 추가하여 변수 유형을 지정합니다. 예를 들어 다음 코드에서는 int 유형의 i 변수를 선언합니다.

```
var i:int;
```

대입 연산자(=)를 사용하여 변수에 값을 지정할 수 있습니다. 예를 들어 다음 코드에서는 i 변수를 선언하고 해당 값을 20으로 지정합니다.

```
var i:int;
i = 20;
```

다음 예제와 같이 변수를 선언하는 동시에 변수에 값을 지정하는 것이 보다 간편할 수 있습니다.

```
var i:int = 20;
```

변수를 선언할 때 변수에 값을 지정하는 방법은 정수 및 문자열과 같은 프리미티브 값을 지정하는 경우뿐만 아니라 배열을 만들거나 클래스의 인스턴스를 인스턴스화하는 경우에도 일반적으로 사용됩니다. 다음 예제에서는 하나의 코드 행으로 배열을 선언하고 값을 지정하는 방법을 보여 줍니다.

```
var numArray:Array = ["zero", "one", "two"];
```

new 연산자를 사용하여 클래스의 인스턴스를 만들 수 있습니다. 다음 예제에서는 CustomClass 인스턴스를 만들고 customItem 변수에 새로 만든 클래스 인스턴스에 대한 참조를 지정합니다.

```
var customItem:CustomClass = new CustomClass();
```

선언할 변수가 둘 이상이면 변수를 구분하는 쉼표 연산자(,)를 사용하여 하나의 코드 행에서 모든 변수를 선언할 수 있습니다. 예를 들어, 다음 코드에서는 하나의 코드 행에 세 개의 변수를 선언합니다.

```
var a:int, b:int, c:int;
```

동일한 코드 행의 각 변수에 값을 지정할 수도 있습니다. 예를 들어 다음 코드에서는 a, b, c 등 세 개의 변수를 선언하고 각각 값을 지정합니다.

```
var a:int = 10, b:int = 20, c:int = 30;
```

쉼표 연산자를 사용하여 변수 선언을 명령문 하나로 그룹화할 수는 있지만 코드에 대한 가독성이 떨어질 수 있습니다.

변수 범위 이해

변수의 범위는 코드에서 어휘 참조로 변수에 액세스할 수 있는 영역입니다. 전역 변수는 코드의 모든 영역에 정의되는 반면 로컬 변수는 코드의 한 부분에만 정의됩니다. ActionScript 3.0에서 변수는 변수가 선언되는 함수 또는 클래스의 범위가 항상 지정됩니다. 전역 변수는 모든 함수 또는 클래스 정의 외부에 정의된 변수입니다. 예를 들어 다음 코드에서는 모든 함수 외부에 strGlobal 변수를 선언하여 전역 변수로 만듭니다. 이 예제에서는 함수 정의 내부와 외부에서 모두 사용할 수 있는 전역 변수를 보여 줍니다.

```
var strGlobal:String = "Global";
function scopeTest()
{
    trace(strGlobal); // Global
}
scopeTest();
trace(strGlobal); // Global
```

함수 정의 내부에 변수를 선언하여 로컬 변수를 선언합니다. 함수 정의는 로컬 변수를 정의할 수 있는 가장 작은 코드 영역입니다. 함수 내에 선언된 로컬 변수는 해당 함수 안에서만 존재합니다. 예를 들어 localScope() 함수 내에 str2 변수를 선언하는 경우 해당 변수는 함수 외부에서 사용할 수 없습니다.

```
function localScope()
{
    var strLocal:String = "local";
}
localScope();
trace(strLocal); // error because strLocal is not defined globally
```

로컬 변수에 사용하는 변수 이름이 이미 전역 변수로 선언되어 있는 경우 해당 로컬 변수가 범위 안에 있는 동안에는 로컬 정의에 의해 전역 정의가 가려집니다. 이러한 경우에도 전역 변수는 함수 외부에서 계속 존재합니다. 예를 들어 다음 코드에서는 str1이라는 전역 문자열 변수를 만든 다음 scopeTest() 함수 내에 동일한 이름의 로컬 변수를 만듭니다. 함수 내의 trace 문은 변수의 로컬 값을 출력하지만 함수 외부의 trace 문은 변수의 전역 값을 출력합니다.

```
var str1:String = "Global";
function scopeTest ()
{
    var str1:String = "Local";
    trace(str1); // Local
}
scopeTest();
trace(str1); // Global
```

C++ 및 Java의 변수와는 달리 ActionScript 변수에는 블록 수준 범위가 없습니다. 코드 블록은 여는 중괄호({)와 닫는 중괄호(}) 사이의 모든 명령문 그룹입니다. C++ 및 Java 등 일부 프로그래밍 언어에서는 코드 블록 내에 선언된 변수를 해당 코드 블록 밖에서 사용할 수 없습니다. 이 범위 제한을 블록 수준 범위라고 하며 ActionScript에는 이러한 제한이 없습니다. 코드 블록 내에 변수를 선언하면 해당 코드 블록뿐만 아니라 코드 블록이 속한 함수의 다른 모든 부분에서 해당 변수를 사용할 수 있습니다. 예를 들어, 다음 함수에는 여러 블록 범위에 정의된 변수가 포함되어 있습니다. 이러한 변수는 모두 함수 전체에서 사용할 수 있습니다.

```
function blockTest (testArray:Array)
{
    var numElements:int = testArray.length;
    if (numElements > 0)
    {
        var elemStr:String = "Element #";
        for (var i:int = 0; i < numElements; i++)
        {
            var valueStr:String = i + ": " + testArray[i];
            trace(elemStr + valueStr);
        }
        trace(elemStr, valueStr, i); // all still defined
    }
    trace(elemStr, valueStr, i); // all defined if numElements > 0
}

blockTest(["Earth", "Moon", "Sun"]);
```

블록 수준 범위가 없다는 것은 함수 정의가 끝나기 전에 변수를 선언하기만 하면 해당 변수가 선언되기 이전에도 변수를 읽거나 변수에 쓸 수 있다는 것을 의미합니다. 이는 컴파일러에서 모든 변수 선언을 함수 맨 위로 이동하는 **호이스팅**이라는 기술에 의해 가능합니다. 예를 들어 다음 코드에서는 num 변수가 선언되기 전에 첫 번째 trace() 함수를 num 변수에 대해 호출하지만 정상적으로 컴파일됩니다.

```
trace(num); // NaN
var num:Number = 10;
trace(num); // 10
```

그러나 컴파일러에서 대입문을 호이스팅하지는 않습니다. 따라서 num에 대한 첫 번째 trace() 호출의 결과가 Number 데이터 유형의 변수에 대한 기본값인 NaN(숫자 아님)이 되는 것입니다. 이는 다음 예제와 같이 변수가 선언되기 전에도 변수에 값을 지정할 수 있다는 것을 의미합니다.

```
num = 5;
trace(num); // 5
var num:Number = 10;
trace(num); // 10
```

기본값

기본값은 변수에 값을 설정하기 전에 해당 변수가 이미 보유하고 있는 값입니다. 변수에 처음으로 값을 설정하는 것은 즉, 변수를 **초기화**하는 것입니다. 그러나 변수를 선언하고 값을 설정하지 않으면 변수는 **초기화되지 않은** 것입니다. 초기화되지 않은 변수의 값은 데이터 유형에 따라 달라집니다. 다음 표에는 변수의 기본값이 데이터 유형별로 설명되어 있습니다.

데이터 유형	기본값
부울	false
int	0
Number	NaN
Object	null
String	null

데이터 유형	기본값
uint	0
선언되지 않음(유형 약어 *에 해당)	undefined
사용자 정의 클래스를 포함한 기타 모든 클래스	null

Number 유형의 변수인 경우 기본값은 NaN(숫자 아님)이며, 이는 숫자를 나타내지 않는 값을 의미하는 특수 값으로 IEEE-754 표준에 정의되어 있습니다.

변수를 선언하고 해당 데이터 유형을 선언하지 않은 경우 기본 데이터 유형인 *가 적용되며 이는 변수의 유형이 지정되지 않았다는 것을 의미합니다. 유형이 지정되지 않은 변수에 값을 설정하여 초기화하지 않으면 기본값이 undefined가 됩니다.

Boolean, Number, int 및 uint 이외의 데이터 유형인 경우 초기화되지 않은 모든 변수의 기본값은 null입니다. 이는 ActionScript 3.0에 의해 정의된 모든 클래스 및 사용자가 만든 모든 사용자 정의 클래스에 적용됩니다.

null 값은 Boolean, Number, int 또는 uint 유형의 변수에 사용할 수 없습니다. 이러한 변수에 null 값을 지정하려고 하면 이 값이 해당 데이터 유형에 대한 기본값으로 변환됩니다. Object 유형의 변수에는 null값을 지정할 수 있습니다. Object 유형의 변수에 undefined 값을 지정하려고 하면 이 값이 null로 변환됩니다.

Number 유형의 변수인지 확인할 때는 isNaN()이라는 특수한 최상위 함수를 사용합니다. 이 함수는 변수가 숫자가 아니면 부울 값 true를 반환하고, 그렇지 않으면 false를 반환합니다.

데이터 유형

데이터 유형은 값의 집합을 정의합니다. 예를 들어 Boolean 데이터 유형은 정확히 두 가지 값, true와 false의 집합입니다. ActionScript 3.0에서는 Boolean 데이터 유형 외에도 String, Number 및 Array 등 보다 일반적으로 사용되는 여러 데이터 유형을 정의합니다. 클래스 또는 인터페이스를 사용하여 직접 데이터 유형을 정의하고 사용자 정의 값 집합을 정의할 수 있습니다. 프리미티브 값 또는 복합 값인지에 관계없이 ActionScript 3.0에서 모든 값은 객체입니다.

프리미티브 값은 Boolean, int, Number, String 및 uint 데이터 유형 중 하나에 속한 값입니다. 일반적으로 프리미티브 값을 사용하여 작업하면 복합 값을 사용하여 작업하는 것보다 속도가 빠릅니다. 이는 ActionScript에서 메모리와 속도를 최적화할 수 있는 특수 방식으로 프리미티브 값을 저장하기 때문입니다.

참고: 기술적인 세부 사항에 관심이 있는 독자를 위해 설명하자면 ActionScript 내부적으로는 프리미티브 값을 변경되지 않는 객체로 저장합니다. 프리미티브 값을 변경되지 않는 객체로 저장한다는 것은 참조에 의한 전달이 실제로는 값에 의한 전달과 동일함을 의미합니다. 일반적으로 값 자체의 크기보다 참조의 크기가 매우 작으므로 참조에 의한 전달을 통해 메모리 사용량이 줄어들고 실행 속도가 빨라집니다.

복합 값은 프리미티브 값이 아닌 값입니다. 복합 값 집합을 정의하는 데이터 유형에는 Array, Date, Error, Function, RegExp, XML, XMLList 등이 있습니다.

많은 프로그래밍 언어에서 프리미티브 값과 래퍼 객체를 구분합니다. 예를 들어, Java에는 int 프리미티브 값과 이 값을 래핑하는 java.lang.Integer 클래스가 있습니다. Java 프리미티브 값은 객체가 아니지만 Java 프리미티브 값의 래퍼는 객체입니다. 이러한 차이점으로 인해 일부 연산에는 프리미티브 값이 유용하고 나머지 연산에는 래퍼 객체가 보다 적합합니다. ActionScript 3.0에서 프리미티브 값과 해당 래퍼 객체는 실제로 구분할 수 없습니다. 프리미티브 값을 포함한 모든 값이 객체입니다. 런타임에서는 이러한 프리미티브 유형을 객체처럼 동작하지만 객체를 만드는 작업과 관련된 일반적인 오버헤드를 발생시키지 않는 특별한 경우로 간주합니다. 이는 다음 두 코드 행이 동일하다는 것을 의미합니다.

```
var someInt:int = 3;
var someInt:int = new int(3);
```

위에 나열된 프리미티브 및 복합 데이터 유형은 모두 ActionScript 3.0 기본 클래스에 의해 정의됩니다. 기본 클래스를 사용하면 new 연산자 대신 리터럴 값을 사용하여 객체를 만들 수 있습니다. 예를 들어 다음과 같이 리터럴 값 또는 Array 클래스 생성자를 사용하여 배열을 만들 수 있습니다.

```
var someArray:Array = [1, 2, 3]; // literal value
var someArray:Array = new Array(1,2,3); // Array constructor
```

유형 검사

유형 검사는 컴파일 타임이나 런타임에 수행할 수 있습니다. C++ 및 Java와 같이 정적으로 유형이 지정되는 언어는 컴파일 타임에 유형 검사를 수행합니다. Smalltalk 및 Python과 같이 동적으로 유형이 지정되는 언어는 런타임에 유형 검사를 처리합니다. 동적으로 유형이 지정되는 언어인 ActionScript 3.0의 경우 런타임에 유형 검사를 수행하지만 **엄격 모드**라는 특수 컴파일러 모드에서의 컴파일 타임 유형 검사도 지원합니다. 엄격 모드에서는 컴파일 타임과 런타임에 모두 유형 검사가 수행되지만, 표준 모드에서는 런타임에만 유형 검사가 수행됩니다.

동적으로 유형이 지정되는 언어를 사용하면 코드를 구성할 때 상당한 융통성을 누릴 수 있지만 런타임에 유형 오류가 출력되는 것을 감수해야 합니다. 정적으로 유형이 지정되는 언어를 사용하면 컴파일 타임에 유형 오류를 확인할 수 있지만 컴파일러에게 유형 정보를 제공해야 하는 부담을 지게 됩니다.

컴파일 타임 유형 검사

프로젝트 규모가 커지면 일반적으로 데이터 유형 유연성보다 유형 오류를 가능한 빨리 포착하는 것이 중요해지므로 대형 프로젝트에서는 컴파일 타임 유형 검사가 자주 사용됩니다. 이러한 이유 때문에 Flash Professional 및 Flash Builder의 ActionScript 컴파일러가 엄격 모드에서 실행되도록 기본 설정되어 있습니다.

Adobe Flash Builder

[Project Properties] 대화 상자의 ActionScript 컴파일러 설정을 통해 Flash Builder에서 엄격 모드를 비활성화할 수 있습니다.

컴파일 타임 유형 검사를 제공하려면 컴파일러에서 코드의 변수 또는 표현식에 대한 데이터 유형 정보를 알고 있어야 합니다. 변수의 데이터 유형을 명시적으로 선언하려면 변수 이름에 콜론 연산자(:)를 추가하고 그 뒤에 데이터 유형을 접미어로 추가합니다. 데이터 유형을 매개 변수와 연결하려면 콜론 연산자와 데이터 유형을 차례로 사용합니다. 예를 들어 다음 코드에서는 xParam 매개 변수에 데이터 유형 정보를 추가하고 명시적 데이터 유형을 사용하여 myParam 변수를 선언합니다.

```
function runtimeTest(xParam:String)
{
    trace(xParam);
}
var myParam:String = "hello";
runtimeTest(myParam);
```

엄격 모드의 ActionScript 컴파일러에서는 컴파일러 오류에 따라 유형 불일치를 보고합니다. 예를 들어 다음 코드에서는 Object 유형의 xParam 함수 매개 변수를 선언하지만 이후 해당 매개 변수에 String 및 Number 유형 값을 지정하려고 합니다. 이로 인해 엄격 모드에서 컴파일러 오류가 발생합니다.

```
function dynamicTest(xParam:Object)
{
    if (xParam is String)
    {
        var myStr:String = xParam; // compiler error in strict mode
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam; // compiler error in strict mode
        trace("Number: " + myNum);
    }
}
```

그러나 엄격 모드에서도 대입문의 우변을 유형이 지정되지 않은 상태로 두면 컴파일 타임 유형 검사에서 해당 대입문을 제외할 수 있습니다. 유형 약어를 생략하거나 특수 별표(*) 유형 약어를 사용하여 유형이 지정되지 않은 변수 또는 표현식을 표시할 수 있습니다. 예를 들어 이전 예제를 수정하여 xParam 매개 변수의 유형 약어를 생략하면 엄격 모드에서 코드가 컴파일됩니다.

```
function dynamicTest(xParam)
{
    if (xParam is String)
    {
        var myStr:String = xParam;
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam;
        trace("Number: " + myNum);
    }
}
dynamicTest(100)
dynamicTest("one hundred");
```

런타임 유형 검사

ActionScript 3.0에서는 엄격 모드 또는 표준 모드 중 어느 모드에서 컴파일했는지와 관계없이 런타임 유형 검사가 수행됩니다. 배열 인수가 필요한 함수에 값 3이 전달되는 경우를 생각해 봅시다. 값 3이 Array 데이터 유형과 호환되지 않으므로 엄격 모드의 컴파일러에서 오류가 발생합니다. 엄격 모드를 사용하지 않고 표준 모드에서 실행하는 경우 컴파일러에서는 유형 불일치에 대한 오류가 발생하지 않지만 런타임 유형 검사를 수행하면 런타임 오류가 발생합니다.

다음 예제에서는 필요한 Array 인수 대신 값 3이 전달된 typeTest() 함수를 보여 줍니다. 값 3이 매개 변수의 선언된 데이터 유형(Array)의 멤버가 아니기 때문에 표준 모드에서 런타임 오류가 발생합니다.

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum:Number = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0 standard mode
```

엄격 모드에서 작업하는 경우에도 런타임 유형 오류가 발생할 수 있습니다. 엄격 모드에서 유형이 지정되지 않은 변수를 사용하면 컴파일 타임 유형 검사에서 해당 변수가 제외되므로 런타임에 유형 오류가 발생할 수도 있습니다. 유형이 지정되지 않은 변수를 사용하는 경우 유형 검사는 취소되지 않고 런타임까지 연기됩니다. 예를 들어 이전 예제의 myNum 변수에 선언된 데이터 유형이 없는 경우 컴파일러에서는 유형 불일치를 감지할 수 없지만 코드에서는 런타임 오류가 발생합니다. 이는 Flash Player에서 대입문의 결과 값인 3으로 설정된 myNum의 런타임 값을 Array 데이터 유형으로 설정된 xParam의 유형과 비교하기 때문입니다.

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0
```

또한 런타임 유형 검사의 경우 컴파일 타임 검사에 비해 상속의 사용을 덜 엄격하게 검사합니다. 표준 모드에서 유형 검사를 런타임으로 연기하면 하위 클래스를 **업캐스팅**한 경우에도 하위 클래스의 속성을 참조할 수 있습니다. 기본 클래스를 사용하여 클래스 인스턴스의 유형을 선언하고, 하위 클래스를 사용하여 클래스 인스턴스를 인스턴스화하면 업캐스팅이 일어납니다. 예를 들어 확장할 수 있는 ClassBase 클래스를 만들 수 있습니다. final 특성이 있는 클래스는 확장할 수 없습니다.

```
class ClassBase
{
}
```

이어서 다음과 같이 someString 속성 하나가 있는 ClassExtender라는 ClassBase의 하위 클래스를 만들 수 있습니다.

```
class ClassExtender extends ClassBase
{
    var someString:String;
}
```

두 클래스를 사용하면 ClassBase 데이터 유형으로 선언되고 ClassExtender 생성자로 인스턴스화되는 클래스 인스턴스를 만들 수 있습니다. 기본 클래스에는 하위 클래스에 없는 속성 또는 메서드가 포함되어 있지 않기 때문에 업캐스팅은 안전한 작업으로 간주됩니다.

```
var myClass:ClassBase = new ClassExtender();
```

그러나 하위 클래스에는 기본 클래스에 없는 속성 또는 메서드가 포함되어 있습니다. 예를 들어 ClassExtender 클래스에는 ClassBase 클래스에 없는 someString 속성이 포함되어 있습니다. ActionScript 3.0 표준 모드에서 다음 예제와 같이 컴파일 타임 오류를 생성하지 않고 myClass 인스턴스를 사용하여 이 속성을 참조할 수 있습니다.

```
var myClass:ClassBase = new ClassExtender();
myClass.someString = "hello";
// no error in ActionScript 3.0 standard mode
```

is 연산자

is 연산자를 사용하면 변수 또는 표현식이 지정된 데이터 유형의 멤버인지 여부를 테스트할 수 있습니다. 이전 버전의 ActionScript에서는 instanceof 연산자가 이 기능을 제공했지만, ActionScript 3.0에서는 데이터 유형 멤버 여부를 테스트하는 데 instanceof 연산자를 사용할 수 없습니다. x instanceof y 표현식으로는 x의 프로토타입 체인에서 y가 존재하는지만 확인할 수 있고 ActionScript 3.0에서는 프로토타입 체인을 통해 완전한 상속 계층 구조에 액세스할 수 없으므로 유형 검사를 수동으로 수행하려면 instanceof 연산자 대신 is 연산자를 사용해야 합니다.

is 연산자는 정확한 상속 계층 구조를 검사하며, 객체가 특정 클래스의 인스턴스인지 여부뿐 아니라 객체가 특정 인터페이스를 구현하는 클래스의 인스턴스인지 여부를 확인하는 데 사용할 수 있습니다. 다음 예제에서는 mySprite라는 Sprite 클래스의 인스턴스를 만들고 is 연산자를 사용하여 mySprite가 Sprite 및 DisplayObject 클래스의 인스턴스인지 여부와 IEventDispatcher 인터페이스를 구현하는지 여부를 테스트합니다.

```
var mySprite:Sprite = new Sprite();
trace(mySprite is Sprite); // true
trace(mySprite is DisplayObject);// true
trace(mySprite is IEventDispatcher); // true
```

is 연산자는 상속 계층 구조를 검사하고 mySprite가 Sprite 및 DisplayObject 클래스와 호환되는지 보고합니다. Sprite 클래스는 DisplayObject 클래스의 하위 클래스입니다. 또한 is 연산자는 mySprite가 IEventDispatcher 인터페이스를 구현하는 클래스에서 상속되는지 여부를 확인합니다. Sprite 클래스가 IEventDispatcher 인터페이스를 구현하는 EventDispatcher 클래스에서 상속되므로 is 연산자는 mySprite가 동일한 인터페이스를 구현하고 있음을 정확하게 보고합니다.

다음 예제에서는 이전 예제와 동일한 테스트를 보여 주지만 is 연산자 대신 instanceof를 사용합니다. instanceof 연산자는 mySprite가 Sprite 또는 DisplayObject의 인스턴스인지를 정확하게 식별하지만 mySprite에서 IEventDispatcher 인터페이스를 구현하는지 여부를 테스트하기 위해 사용하는 경우에는 false를 반환합니다.

```
trace(mySprite instanceof Sprite); // true
trace(mySprite instanceof DisplayObject);// true
trace(mySprite instanceof IEventDispatcher); // false
```

as 연산자

as 연산자 역시 표현식이 지정된 데이터 유형의 멤버인지 여부를 확인하는 데 사용할 수 있습니다. is 연산자와 달리 as 연산자는 부울 값을 반환하지 않습니다. as 연산자는 true 대신 표현식의 값을 반환하고 false 대신 null을 반환합니다. 다음 예제에서는 Sprite 인스턴스가 DisplayObject, IEventDispatcher 및 Number 데이터 유형의 멤버인지 여부를 확인할 때 is 연산자 대신 as 연산자를 사용하는 경우에 대한 결과를 보여 줍니다.

```
var mySprite:Sprite = new Sprite();
trace(mySprite as Sprite); // [object Sprite]
trace(mySprite as DisplayObject); // [object Sprite]
trace(mySprite as IEventDispatcher); // [object Sprite]
trace(mySprite as Number);           // null
```

as 연산자를 사용하는 경우 오른쪽 피연산자는 데이터 유형이어야 합니다. 오른쪽 피연산자로 데이터 유형이 아닌 표현식을 사용하려고 하면 오류가 발생합니다.

동적 클래스

동적 클래스는 속성 및 메서드를 추가하거나 변경하여 런타임에 변경할 수 있는 객체를 정의합니다. String 클래스와 같이 동적이지 않은 클래스는 **봉인된** 클래스입니다. 런타임에 속성 또는 메서드를 봉인된 클래스에 추가할 수 없습니다.

클래스를 선언할 때 dynamic 특성을 사용하여 동적 클래스를 만듭니다. 예를 들어 다음 코드에서는 Protean이라는 동적 클래스를 만듭니다.

```
dynamic class Protean
{
    private var privateGreeting:String = "hi";
    public var publicGreeting:String = "hello";
    function Protean()
    {
        trace("Protean instance created");
    }
}
```

나중에 Protean 클래스의 인스턴스를 인스턴스화하는 경우 클래스 정의 외부에 있는 인스턴스에 속성 또는 메서드를 추가할 수 있습니다. 예를 들어 다음 코드에서는 Protean 클래스의 인스턴스를 만들고 aString 속성과 aNumber 속성을 해당 인스턴스에 추가합니다.

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
trace(myProtean.aString, myProtean.aNumber); // testing 3
```

동적 클래스의 인스턴스에 추가한 속성은 런타임 엔터티이므로 모든 유형 검사는 런타임에 수행됩니다. 이러한 방법으로 추가한 속성에 유형 약어를 추가할 수 없습니다.

함수를 정의하고 해당 함수를 myProtean 인스턴스의 속성에 연결하여 myProtean 인스턴스에 메서드를 추가할 수도 있습니다. 다음 코드에서는 trace 문을 traceProtean() 메서드로 이동합니다.

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
myProtean.traceProtean = function ()
{
    trace(this.aString, this.aNumber);
};
myProtean.traceProtean(); // testing 3
```

그러나 이렇게 만든 메서드에서는 Protean 클래스의 모든 전용 속성 또는 메서드에 액세스하지 못합니다. 또한 Protean 클래스의 공용 속성 또는 메서드에 대한 참조를 this 키워드 또는 클래스 이름으로 정규화해야 합니다. 다음 예제에서는 Protean 클래스의 전용 및 공용 변수에 액세스하려고 하는 traceProtean() 메서드를 보여 줍니다.

```
myProtean.traceProtean = function ()
{
    trace(myProtean.privateGreeting); // undefined
    trace(myProtean.publicGreeting); // hello
};
myProtean.traceProtean();
```

데이터 유형 설명

프리미티브 데이터 유형에는 Boolean, int, Null, Number, String, uint 및 void 등이 포함됩니다. 또한 ActionScript 기본 클래스에서 Object, Array, Date, Error, Function, RegExp, XML 및 XMLList 등의 복합 데이터 유형을 정의합니다.

Boolean 데이터 유형

Boolean 데이터 유형에는 두 가지 값, 즉true와 false가 포함됩니다. 그 외 다른 값은 Boolean 유형의 변수에 사용할 수 없습니다. 선언되었지만 초기화되지 않은 부울 변수의 기본값은 false입니다.

int 데이터 유형

int 데이터 유형은 내부적으로 32비트의 정수로 저장되며

-2,147,483,648 (-2^{31})부터 2,147,483,647($2^{31} - 1$)까지의 정수 집합(-2,147,483,648 및 2,147,483,647 포함)으로 구성됩니다. 이전 버전의 ActionScript에서는 정수와 부동 소수점 숫자에 모두 사용되는 Number 데이터 유형만 제공했습니다. ActionScript 3.0에서는 이제 32비트의 부호가 있거나 없는 정수에 대한 저수준의 시스템 유형에 액세스할 수 있습니다. 변수에서 부동 소수점 숫자가 필요하지 않는 경우 Number 데이터 유형 대신 int 데이터 유형을 사용하는 것이 보다 빠르고 효율적입니다.

최소 및 최대 int 값의 범위를 벗어나는 정수 값인 경우 Number 데이터 유형을 사용하여 양수와 음수 각각 9,007,199,254,740,992(53비트 정수 값) 사이의 값을 처리할 수 있습니다. int 데이터 유형의 변수에 대한 기본값은 0입니다.

Null 데이터 유형

Null 데이터 유형에는 null 값만 포함됩니다. 이는 Object 클래스를 포함하여 복합 데이터 유형을 정의하는 모든 클래스와 String 데이터 유형의 기본값입니다. Boolean, Number, int 및 uint 같은 기타 프리미티브 데이터 유형에는 null 값이 포함되어 있지 않습니다. Boolean, Number, int 또는 uint 유형의 변수에 null을 지정하려고 하면 null 값은 런타임에 적절한 기본값으로 변환됩니다. 이 데이터 유형은 유형 약어로 사용할 수 없습니다.

Number 데이터 유형

ActionScript 3.0에서 Number 데이터 유형은 정수, 부호 없는 정수, 부동 소수점 숫자 등을 나타낼 수 있습니다. 그러나 성능을 최대화하려면 32비트 int 및 uint 유형이 저장할 수 있는 값보다 큰 정수 값 또는 부동 소수점 숫자에만 Number 데이터 유형을 사용해야 합니다. 부동 소수점 숫자를 저장하려면 숫자 안에 소수점을 넣습니다. 소수점을 생략하면 숫자가 정수로 저장됩니다.

Number 데이터 유형에서는 이진 부동 소수점 산술에 대한 IEEE 표준(IEEE-754)에 지정된 64비트 배정밀도 형식을 사용합니다. 이 표준에는 64비트를 사용하여 부동 소수점 숫자를 저장하는 방법이 규정되어 있습니다. 1비트는 숫자가 양수 또는 음수인지 지정하는 데 사용됩니다. 11비트는 2를 밑수로 하는 지수를 저장하는 데 사용됩니다. 남은 52비트는 지수에 따라 거듭제곱되는 숫자인 유효 숫자(또는 가수)를 저장하는 데 사용됩니다.

Number 데이터 유형에서는 지수를 저장하는 데 일부 비트를 사용하므로 모든 비트가 유효 숫자를 저장하는 데 사용되는 경우에 비해 매우 큰 부동 소수점 숫자를 저장할 수 있습니다. 예를 들어, Number 데이터 유형에서 유효 숫자를 저장하는 데 64비트를 모두 사용하는 경우, 저장할 수 있는 가장 큰 숫자는 $2^{65} - 1$입니다. 11비트를 사용하여 지수를 저장하면 Number 데이터 유형에서 유효 숫자를 2^{1023}으로 거듭제곱할 수 있습니다.

Number 유형에서 나타낼 수 있는 최소값 및 최대값은 Number.MAX_VALUE 및 Number.MIN_VALUE라는 Number 클래스의 정적 속성에 저장됩니다.

```
Number.MAX_VALUE == 1.79769313486231e+308
Number.MIN_VALUE == 4.940656458412467e-324
```

Number 데이터 유형의 경우 숫자의 범위는 방대하지만 정밀도는 떨어집니다. Number 데이터 유형에서는 유효 숫자를 저장하는 데 52비트를 사용하므로 정확하게 나타내기 위해 52비트 이상이 필요한 숫자(예: 분수 1/3)는 근삿값만 표현할 수 있습니다. 응용 프로그램에서 절대 정밀도의 십진수가 필요한 경우 이진 부동 소수점 산술과 대립되는 십진 부동 소수점 산술을 구현하는 소프트웨어를 사용해야 합니다.

Number 데이터 유형을 사용하여 정수 값을 저장하면 유효 숫자의 52비트만 사용됩니다. Number 데이터 유형에서는 해당 52비트 및 특수 은폐 비트를 사용하여 -9,007,199,254,740,992(-2^{53})부터 9,007,199,254,740,992(2^{53})까지의 정수를 나타냅니다.

NaN 값은 Number 유형의 변수에 대한 기본값으로 사용될 뿐만 아니라 숫자를 반환해야 하지만 이를 반환하지 않는 모든 연산의 결과로도 사용됩니다. 예를 들어 음수의 제곱근을 계산하려고 하면 결과는 NaN이 됩니다. 기타 특수 Number 값에 양의 무한대와 음의 무한대가 포함됩니다.

참고: 제수가 0인 경우 0으로 나눈 결과도 NaN입니다. 피제수가 양수인 경우 0으로 나누면 infinity가 되고, 피제수가 음수인 경우 0으로 나누면 -infinity가 됩니다.

String 데이터 유형

String 데이터 유형은 16비트 문자열을 나타냅니다. 문자열은 내부적으로 UTF-16 포맷을 사용하여 유니코드 문자로 저장합니다. 문자열은 Java 프로그래밍 언어와 마찬가지로 변경할 수 없는 값입니다. String 값에 대한 작업은 해당 문자열의 새 인스턴스를 반환합니다. String 데이터 유형으로 선언되는 변수의 기본값은 null입니다. null 값은 빈 문자열("")과는 다릅니다. null 값은 변수에 저장된 값이 없음을 의미하지만 빈 문자열은 변수에 저장된 값이 문자를 포함하지 않는 문자열임을 의미합니다.

uint 데이터 유형

uint 데이터 유형은 내부적으로 32비트의 부호 없는 정수로 저장되며 0부터 4,294,967,295(2^{32} - 1)까지의 정수 집합(0 및 4,294,967,295 포함)을 포함합니다. 음이 아닌 정수를 호출하는 특수한 경우에는 uint 데이터 유형을 사용합니다. 예를 들어 int 데이터 유형에는 색상 값 처리에 적합하지 않은 내부 부호 비트가 포함되어 있기 때문에 픽셀의 색상 값을 나타내려면 uint 데이터 유형을 사용해야 합니다. 최대 uint 값보다 큰 정수 값의 경우에는 53비트 정수 값을 처리할 수 있는 Number 데이터 유형을 사용합니다. uint 데이터 유형의 변수에 대한 기본값은 0입니다.

Void 데이터 유형

void 데이터 유형에는 undefined 값만 포함됩니다. 이전 버전의 ActionScript에서는 Object 클래스의 인스턴스에 대한 기본값이 undefined였지만, ActionScript 3.0에서는 Object 인스턴스에 대한 기본값이 null입니다. Object 클래스의 인스턴스에 undefined 값을 지정하려고 하면 이 값이 null로 변환됩니다. undefined 값은 유형이 지정되지 않은 변수에만 지정할 수 있습니다. 유형이 지정되지 않은 변수는 유형 약어가 없거나 유형 약어에 별표(*) 기호를 사용하는 변수입니다. void는 반환 유형 약어로만 사용할 수 있습니다.

Object 데이터 유형

Object 데이터 유형은 Object 클래스에 의해 정의됩니다. Object 클래스는 ActionScript에서 모든 클래스 정의에 대한 기본 클래스 역할을 합니다. ActionScript 3.0 버전의 Object 데이터 유형은 세 가지 면에서 이전 버전의 Object 데이터 유형과 다릅니다. 첫 번째, Object 데이터 유형은 더 이상 유형 약어가 없는 변수에 지정되는 기본 데이터 유형이 아닙니다. 두 번째, Object 데이터 유형에는 Object 인스턴스의 기본값이었던 undefined 값이 더 이상 포함되지 않습니다. 세 번째, ActionScript 3.0에서는 Object 클래스의 인스턴스에 대한 기본값이 null입니다.

이전 버전의 ActionScript에서 유형 약어가 없는 변수는 자동으로 Object 데이터 유형으로 지정되었습니다. ActionScript 3.0에는 유형이 지정되지 않은 변수라는 개념이 도입되면서 이전 버전과는 다르게 처리됩니다. 이제 유형 약어가 없는 변수는 유형이 지정되지 않은 변수로 간주됩니다. 변수의 유형을 지정하지 않으려는 자신의 의도를 코드를 읽는 사람에게 분명히 밝히려면 유형 약어로 별표(*) 기호를 사용할 수 있습니다. 이는 유형 약어를 생략하는 것과 같습니다. 다음 예제에서는 유형이 지정되지 않은 변수 x를 선언하는 동일한 두 명령문을 보여 줍니다.

```
var x
var x:*
```

유형이 지정되지 않은 변수에만 undefined 값을 저장할 수 있습니다. 데이터 유형이 있는 변수에 undefined 값을 지정하려고 하면 런타임에서 undefined 값을 해당 데이터 유형의 기본값으로 변환합니다. Object 데이터 유형의 인스턴스인 경우 기본값은 null입니다. 따라서 Object 인스턴스에 undefined를 지정하려고 하면 이 값은 null로 변환됩니다.

유형 변환

어떤 값이 다른 데이터 유형의 값으로 변형되면 유형 변환이 발생한다고 합니다. 유형 변환은 **암시적** 또는 **명시적**으로 수행될 수 있습니다. 암시적 변환은 **강제 형 변환**이라고도 하며 런타임에 수행되는 경우도 있습니다. 예를 들어 Boolean 데이터 유형에 값 2를 지정하면 변수에 값을 지정하기 전에 값 2가 부울 값인 true로 변환됩니다. 컴파일러가 특정 데이터 유형의 변수를 다른 데이터 유형의 변수로 처리하도록 코드에서 지정하면 **형 변환**이라는 명시적 변환이 발생합니다. 프리미티브 값이 포함되어 있으면 형 변환에 의해 실제로 한 데이터 유형에서 다른 데이터 유형으로 값이 변환됩니다. 객체를 다른 유형으로 변환하려면 객체 이름을 괄호로 묶고 그 앞에 새 유형의 이름을 붙여야 합니다. 예를 들어, 다음 코드는 부울 값을 사용하며 이 값을 정수로 변환합니다.

```
var myBoolean:Boolean = true;
var myINT:int = int(myBoolean);
trace(myINT); // 1
```

암시적 변환

런타임에 암시적 변환이 발생하는 몇 가지 경우는 다음과 같습니다.

- 대입문인 경우
- 값이 함수 인수로 전달되는 경우
- 함수에서 값이 반환되는 경우
- 더하기(+) 연산자와 같은 특정 연산자를 사용하는 표현식의 경우

  사용자 정의 유형의 경우 변환할 값이 대상 클래스 또는 대상 클래스에서 파생된 클래스의 인스턴스이면 암시적 변환이 이루어집니다. 암시적 변환에 실패하면 오류가 발생합니다. 예를 들어, 다음 코드에는 성공적인 암시적 변환과 실패한 암시적 변환이 포함되어 있습니다.

  ```
  class A {}
  class B extends A {}

  var objA:A = new A();
  var objB:B = new B();
  var arr:Array = new Array();

  objA = objB; // Conversion succeeds.
  objB = arr; // Conversion fails.
  ```

  프리미티브 유형의 경우 명시적 변환 함수에서 호출하는 내부 변환 알고리즘과 동일한 알고리즘을 호출하여 암시적 변환을 처리합니다.

명시적 변환

엄격 모드에서 컴파일할 때 유형 불일치로 인해 컴파일 타임 오류가 발생하지 않도록 하려면 명시적 변환 또는 형 변환을 사용하는 것이 도움이 됩니다. 강제 형 변환이 런타임에 값을 올바르게 변환한다는 것을 알고 있는 경우를 가정해 봅시다. 예를 들어, 양식에서 전달된 데이터를 처리하는 경우 특정 문자열 값을 숫자 값으로 변환하기 위해 강제 형 변환을 사용할 수 있습니다. 다음 코드는 표준 모드에서 올바르게 실행되지만 컴파일 타임에 오류가 발생합니다.

```
var quantityField:String = "3";
var quantity:int = quantityField; // compile time error in strict mode
```

계속해서 엄격 모드를 사용하면서 문자열이 정수로 변환되도록 하려면 다음과 같이 명시적 변환을 사용할 수 있습니다.

```
var quantityField:String = "3";
var quantity:int = int(quantityField); // Explicit conversion succeeds.
```

int, uint 및 Number로 형 변환

모든 데이터 유형을 int, uint 및 Number 등의 숫자 유형 중 하나로 변환할 수 있습니다. 어떤 이유로 숫자를 변환할 수 없는 경우 int 및 uint 데이터 유형에 기본값으로 0이 지정되고 Number 데이터 유형에는 기본값으로 NaN이 지정됩니다. 부울 값을 숫자로 변환하면 true는 값 1이 되며 false는 값 0이 됩니다.

```
var myBoolean:Boolean = true;
var myUINT:uint = uint(myBoolean);
var myINT:int = int(myBoolean);
var myNum:Number = Number(myBoolean);
trace(myUINT, myINT, myNum); // 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // 0 0 0
```

숫자만 포함된 문자열 값을 숫자 유형 중 하나로 변환할 수 있습니다. 음수 형태의 문자열 또는 16진수 값(예: 0x1A)을 나타내는 문자열도 숫자 유형으로 변환할 수 있습니다. 변환 과정에서 문자열 값에 있는 선행 또는 후행 공백 문자는 무시됩니다. Number() 를 사용하여 부동 소수점 숫자 형태의 문자열을 변환할 수도 있습니다. 소수점이 포함된 문자열을 uint() 및 int()에 전달하면 소수점과 그 뒤의 문자를 잘라내고 정수를 반환합니다. 예를 들어 다음 문자열 값을 숫자로 변환할 수 있습니다.

```
trace(uint("5")); // 5
trace(uint("-5")); // 4294967291. It wraps around from MAX_VALUE
trace(uint(" 27 ")); // 27
trace(uint("3.7")); // 3
trace(int("3.7")); // 3
trace(int("0x1A")); // 26
trace(Number("3.7")); // 3.7
```

숫자가 아닌 문자가 포함된 문자열 값을 int() 또는 uint()로 변환하면 0이 반환되고 Number()로 변환하면 NaN이 반환됩니다 변환 과정에서 선행 또는 후행 공백 문자는 무시되지만 문자열에 두 개의 숫자를 구분하는 공백이 있는 경우 0 또는 NaN을 반환합니다.

```
trace(uint("5a")); // 0
trace(uint("ten")); // 0
trace(uint("17 63")); // 0
```

ActionScript 3.0에서 Number() 함수는 더 이상 8진수 또는 밑수가 8인 숫자를 지원하지 않습니다. ActionScript 2.0에서 Number() 함수에 0으로 시작되는 문자열을 제공하면 숫자가 8진수로 해석된 후 10진수로 변환됩니다. ActionScript 3.0에서 Number() 함수는 문자열 맨 앞에 있는 0을 무시하기 때문에 이전 버전에서와 같이 수행되지 않습니다. 예를 들어 다음 코드를 서로 다른 버전의 ActionScript를 사용하여 컴파일하면 각기 다른 결과가 출력됩니다.

```
trace(Number("044"));
// ActionScript 3.0 44
// ActionScript 2.0 36
```

특정 숫자 유형의 값이 다른 숫자 유형의 변수에 지정된 경우에는 변환할 필요가 없습니다. 엄격 모드에서도 숫자 유형이 다른 숫자 유형으로 암시적으로 변환됩니다. 이는 경우에 따라 유형 범위를 벗어나면 예기치 못한 값이 발생할 수 있다는 것을 의미합니다. 경우에 따라 예기치 못한 값이 생성될 수 있지만 다음 예제는 엄격 모드에서 모두 컴파일됩니다.

```
var myUInt:uint = -3; // Assign int/Number value to uint variable
trace(myUInt); // 4294967293

var myNum:Number = sampleUINT; // Assign int/uint value to Number variable
trace(myNum) // 4294967293

var myInt:int = uint.MAX_VALUE + 1; // Assign Number value to uint variable
trace(myInt); // 0

myInt = int.MAX_VALUE + 1; // Assign uint/Number value to int variable
trace(myInt); // -2147483648
```

다음 표에는 다른 데이터 유형에서 Number, int 또는 uint 데이터 유형으로의 변환 결과가 요약되어 있습니다.

데이터 유형 또는 값	Number, int 또는 uint로 변환한 결과
부울	값이 true이면 1로 변환되고 그렇지 않으면 0으로 변환됩니다.
DATE	Date 객체의 내부 표현 즉, 표준시 1970년 1월 1일 자정 이후 경과된 밀리초로 변환됩니다.
null	0
Object	null인 인스턴스를 Number로 변환하면 NaN이 반환되고 그 외의 경우에는 0이 반환됩니다.
String	문자열을 숫자로 변환할 수 있는 경우 숫자가 반환되고, 그렇지 않은 경우 Number로 변환하면 NaN이 반환되고 int 또는 uint로 변환하면 0이 반환됩니다.
undefined	Number로 변환하면 NaN이 반환되고 int 또는 uint로 변환하면 0이 반환됩니다.

Boolean으로 변환

숫자 데이터 유형(uint, int 및 Number)에서 Boolean으로 변환하면, 숫자 값이 0인 경우 false가 반환되고 그렇지 않으면 true가 반환됩니다. Number 데이터 유형의 경우 NaN 값 역시 false가 됩니다. 다음 예제에서는 숫자 -1, 0 및 1을 변환한 결과를 보여 줍니다.

```
var myNum:Number;
for (myNum = -1; myNum<2; myNum++)
{
    trace("Boolean(" + myNum +") is " + Boolean(myNum));
}
```

이 예제의 출력에서 숫자 0만 false 값을 반환하는 것을 확인할 수 있습니다.

```
Boolean(-1) is true
Boolean(0) is false
Boolean(1) is true
```

문자열 값에서 Boolean으로 변환하는 경우 문자열이 null이거나 빈 문자열("")이면 false를 반환합니다. 그렇지 않으면 true를 반환합니다.

```
var str1:String; // Uninitialized string is null.
trace(Boolean(str1)); // false

var str2:String = ""; // empty string
trace(Boolean(str2)); // false

var str3:String = " "; // white space only
trace(Boolean(str3)); // true
```

Object 클래스의 인스턴스에서 Boolean으로 변환하는 경우 인스턴스가 null이면 false를 반환하고 그렇지 않으면 true를 반환합니다.

```
var myObj:Object; // Uninitialized object is null.
trace(Boolean(myObj)); // false

myObj = new Object(); // instantiate
trace(Boolean(myObj)); // true
```

엄격 모드에서 모든 데이터 유형의 값을 Boolean으로 변환하지 않고 부울 변수에 지정할 수 있다는 점에서 부울 변수는 특수하게 처리되는 변수입니다. 엄격 모드에서도 모든 데이터 유형에서 Boolean 데이터 유형으로 암시적으로 강제 형 변환합니다. 즉, 대부분의 다른 모든 데이터 유형과 달리 엄격 모드 오류를 방지하기 위해 Boolean으로 변환할 필요가 없습니다. 다음 예제는 엄격 모드에서 모두 컴파일되고 런타임에 예상대로 작동됩니다.

```
var myObj:Object = new Object(); // instantiate
var bool:Boolean = myObj;
trace(bool); // true
bool = "random string";
trace(bool); // true
bool = new Array();
trace(bool); // true
bool = NaN;
trace(bool); // false
```

다음 표에는 다른 데이터 유형에서 Boolean 데이터 유형으로 변환한 결과가 요약되어 있습니다.

데이터 유형 또는 값	Boolean로 변환한 결과
String	값이 null 또는 빈 문자열("")이면 false를 반환하고 그렇지 않으면 true를 반환합니다.
null	false
Number, int 또는 uint	값이 NaN 또는 0이면 false를 반환하고 그렇지 않으면 true를 반환합니다.
Object	인스턴스가 null이면 false를 반환하고 그렇지 않으면 true를 반환합니다.

String으로 변환

모든 숫자 데이터 유형에서 String 데이터 유형으로 변환하면 숫자의 문자열 표현을 반환합니다. 부울 값에서 String 데이터 유형으로 변환하면 값이 true인 경우 "true" 문자열을 반환하고 값이 false인 경우 "false" 문자열을 반환합니다.

Object 클래스의 인스턴스에서 String 데이터 유형으로 변환하면 인스턴스가 null인 경우 "null" 문자열을 반환합니다. 그렇지 않은 경우, Object 클래스에서 String 유형으로 변환하면 "[object Object]" 문자열을 반환합니다.

Array 클래스의 인스턴스에서 String으로 변환하면 모든 배열 요소가 쉼표로 구분된 목록으로 구성된 문자열을 반환합니다. 예를 들어, 다음과 같이 String 데이터 유형으로 변환하면 세 가지 배열 요소가 모두 포함되어 있는 하나의 문자열을 반환합니다.

```
var myArray:Array = ["primary", "secondary", "tertiary"];
trace(String(myArray)); // primary,secondary,tertiary
```

Date 클래스의 인스턴스에서 String으로 변환하면 인스턴스에 포함된 날짜의 문자열 표현을 반환합니다. 예를 들어, 다음 예제에서는 Date 클래스 인스턴스의 문자열 표현(태평양 일광 절약 시간으로 결과 표시)을 반환합니다.

```
var myDate:Date = new Date(2005,6,1);
trace(String(myDate)); // Fri Jul 1 00:00:00 GMT-0700 2005
```

다음 표에는 다른 데이터 유형에서 String 데이터 유형으로 변환한 결과가 요약되어 있습니다.

데이터 유형 또는 값	문자열로 변환한 결과
배열	모든 배열 요소로 구성된 문자열을 반환합니다.
부울	"true" 또는 "false"
DATE	Date 객체의 문자열 표현을 반환합니다.
null	"null"
Number, int 또는 uint	숫자의 문자열 표현을 반환합니다.
Object	인스턴스가 null이면 "null"을 반환하고 그렇지 않으면 "[object Object]"를 반환합니다.

구문

언어의 구문에는 실행 코드를 작성할 때 따라야 하는 규칙 집합이 정의되어 있습니다.

대/소문자 구분

ActionScript 3.0에서는 대/소문자를 구분합니다. 대/소문자만 다르고 이름이 같은 식별자는 서로 다른 식별자로 간주합니다. 예를 들어 다음 코드에서는 두 개의 서로 다른 변수를 만듭니다.

```
var num1:int;
var Num1:int;
```

도트 구문

도트 연산자(.)는 객체의 속성 및 메서드에 액세스하는 방법을 제공합니다. 도트 구문을 사용하면 도트 연산자 및 속성 또는 메서드 이름 앞에 인스턴스 이름을 사용하여 클래스 속성 또는 메서드를 참조할 수 있습니다. 예를 들어 다음과 같은 클래스 정의를 생각해 봅시다.

```
class DotExample
{
    public var prop1:String;
    public function method1():void {}
}
```

도트 구문을 사용하면 다음 코드에서 만든 인스턴스 이름을 사용하여 prop1 속성 및 method1() 메서드에 액세스할 수 있습니다.

```
var myDotEx:DotExample = new DotExample();
myDotEx.prop1 = "hello";
myDotEx.method1();
```

패키지를 정의할 때 도트 구문을 사용할 수 있습니다. 도트 연산자를 사용하여 중첩된 패키지를 참조합니다. 예를 들어, 플래시 패키지 내에 중첩된 이벤트 패키지에 EventDispatcher 클래스가 있습니다. 다음 표현식을 사용하여 이벤트 패키지를 참조할 수 있습니다.

```
flash.events
```

이 표현식을 사용하여 EventDispatcher 클래스를 참조할 수도 있습니다.

```
flash.events.EventDispatcher
```

슬래시 구문

ActionScript 3.0에서는 슬래시 구문을 지원하지 않습니다. 슬래시 구문은 무비 클립의 경로 또는 변수를 나타내기 위해 이전 버전의 ActionScript에서 사용되었습니다.

리터럴

리터럴은 코드에 직접 나타나는 값으로, 다음 예는 모두 리터럴입니다.

```
17
"hello"
-3
9.4
null
undefined
true
false
```

리터럴을 묶어 복합 리터럴을 만들 수도 있습니다. 배열 리터럴은 대괄호([])로 묶이며 쉼표를 사용하여 배열 요소를 구분합니다.

배열 리터럴은 배열을 초기화할 때 사용할 수 있습니다. 다음 예제는 배열 리터럴을 사용하여 초기화된 두 개의 배열을 보여 줍니다. new 문을 사용하고 복합 리터럴을 Array 클래스 생성자에 매개 변수로 전달할 수 있지만 Object, Array, String, Number, int, uint, XML, XMLList, Boolean 등의 ActionScript 기본 클래스의 인스턴스를 인스턴스화할 때 직접 리터럴 값을 지정할 수도 있습니다.

```
// Use new statement.
var myStrings:Array = new Array(["alpha", "beta", "gamma"]);
var myNums:Array = new Array([1,2,3,5,8]);

// Assign literal directly.
var myStrings:Array = ["alpha", "beta", "gamma"];
var myNums:Array = [1,2,3,5,8];
```

리터럴을 사용하여 일반 객체를 초기화할 수도 있습니다. 일반 객체는 Object 클래스의 인스턴스입니다. 객체 리터럴은 중괄호({})로 묶이며 쉼표를 사용하여 객체 속성을 구분합니다. 각 속성은 콜론(:)으로 선언되며 이는 속성 이름과 속성 값을 구분합니다.

new 문을 사용하여 일반 객체를 만들고 객체 리터럴을 Object 클래스 생성자에 매개 변수로 전달할 수도 있고, 선언할 인스턴스에 직접 객체 리터럴을 지정할 수 있습니다. 다음 예제에서는 새 일반 객체를 만든 다음 세 가지 속성(propA, propB 및 propC)을 사용하여 객체를 초기화하는 두 가지 방법을 보여 줍니다. 여기서 속성 값은 각각 1, 2, 3으로 설정됩니다.

```
// Use new statement and add properties.
var myObject:Object = new Object();
myObject.propA = 1;
myObject.propB = 2;
myObject.propC = 3;

// Assign literal directly.
var myObject:Object = {propA:1, propB:2, propC:3};
```

기타 도움말 항목

문자열을 사용한 작업

일반 표현식 사용

XML 변수 초기화

세미콜론

세미콜론(;)을 사용하여 명령문을 종결할 수 있습니다. 또는 세미콜론을 생략할 수도 있으며 이 경우 컴파일러에서 각 코드 행을 단일 명령문으로 간주합니다. 많은 프로그래머가 세미콜론을 사용하여 명령문을 끝내는 것에 익숙하기 때문에 명령문을 종결할 때 세미콜론을 사용하면 보다 쉽게 코드를 읽을 수 있습니다.

세미콜론을 사용하여 명령문을 종결할 경우 한 행에 여러 개의 명령문을 배치할 수 있지만 그럴 경우 코드를 읽기가 어려워질 수 있습니다.

괄호

ActionScript 3.0에서는 괄호(())를 세 가지 방식으로 사용할 수 있습니다. 첫 번째, 괄호를 사용하여 표현식에서 연산 순서를 변경할 수 있습니다. 괄호 안에 그룹화되어 있는 연산이 항상 가장 먼저 실행됩니다. 예를 들어 다음 코드에서는 괄호를 사용하여 연산 순서를 변경합니다.

```
trace(2 + 3 * 4); // 14
trace((2 + 3) * 4); // 20
```

두 번째, 쉼표 연산자(,)가 포함된 괄호를 사용하여 다음 예제와 같이 일련의 표현식을 평가하고 최종 표현식의 결과를 반환합니다.

```
var a:int = 2;
var b:int = 3;
trace((a++, b++, a+b)); // 7
```

세 번째, 괄호를 사용하여 다음 예제와 같이 함수 또는 메서드에 하나 이상의 매개 변수를 전달할 수 있습니다. 다음 예제에서는 String 값을 trace() 함수에 전달합니다.

```
trace("hello"); // hello
```

설명

ActionScript 3.0 코드에 두 가지 유형의 주석 즉, 한 줄 주석 및 여러 줄 주석을 사용할 수 있습니다. 이러한 주석 처리 메커니즘은 C++ 및 Java에서의 주석 처리 메커니즘과 유사합니다. 컴파일러는 주석으로 표시된 텍스트를 무시합니다.

한 줄 주석은 두 개의 슬래시(//)로 시작하고 줄이 끝날 때까지 계속됩니다. 예를 들어 다음 코드에 한 줄 주석이 포함되어 있습니다.

```
var someNumber:Number = 3; // a single line comment
```

여러 줄 주석은 슬래시와 별표(/*)로 시작하고 별표와 슬래시(*/)로 끝납니다.

```
/* This is multiline comment that can span
more than one line of code. */
```

키워드 및 예약어

예약어는 ActionScript에서 사용하도록 예약된 단어이기 때문에 코드에서 식별자로 사용할 수 없는 단어입니다. 예약어에는 컴파일러에 의해 프로그램 네임스페이스에서 제거된 **사전식 키워드**가 포함됩니다. 사전식 키워드를 식별자로 사용하면 컴파일러에서 오류를 보고합니다. 다음 표에 ActionScript 3.0의 사전식 키워드가 나와 있습니다.

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	for	function
if	implements	import	in
instanceof	인터페이스	internal	is
native	new	null	package
private	protected	public	return
super	switch	this	throw
~	true	try	typeof
use	var	void	while
with			

식별자로 사용할 수 있지만 특정 컨텍스트에서 특별한 의미를 나타내는 **구문 키워드**라는 작은 키워드 집합이 있습니다. 다음 표에 ActionScript 3.0의 구문 키워드가 나와 있습니다.

each	get	set	namespace
include	동적	final	native
override	정적		

나중에 사용할 수 있는 예약어라고도 하는 식별자가 몇 가지 있습니다. 이러한 식별자 중 일부는 ActionScript 3.0을 통합하는 소프트웨어에서 키워드로 간주될 수 있지만 ActionScript 3.0에서는 예약어가 아닙니다. 코드에 이러한 여러 식별자를 사용할 수 있지만 후속 언어 버전에서 이들이 키워드로 표시될 수 있으므로 사용하지 않는 것이 좋습니다.

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic
long	prototype	short	synchronized
throws	~	transient	type
virtual	volatile		

상수

ActionScript 3.0에서는 상수를 만드는 데 사용할 수 있는 const 문을 지원합니다. 상수는 변경할 수 없는 고정된 값을 갖는 속성입니다. 상수에는 값을 한 번만 지정할 수 있으며 상수의 선언과 아주 근접한 위치에서 값을 지정해야 합니다. 예를 들어, 상수를 클래스의 멤버로 선언하는 경우 선언의 일부로서 해당 상수에 값을 지정하거나 클래스 생성자 내에서 값을 지정할 수 있습니다.

다음 코드에서는 두 개의 상수를 선언합니다. 첫 번째 상수 MINIMUM은 선언문의 일부로 값이 지정됩니다. 두 번째 상수 MAXIMUM은 생성자에서 값이 지정됩니다. 엄격 모드에서는 초기화 시에만 상수 값을 지정할 수 있으므로 이 예제는 표준 모드에서만 컴파일됩니다.

```
class A
{
    public const MINIMUM:int = 0;
    public const MAXIMUM:int;

    public function A()
    {
        MAXIMUM = 10;
    }
}

var a:A = new A();
trace(a.MINIMUM); // 0
trace(a.MAXIMUM); // 10
```

다른 방법으로 초기 값을 상수에 지정하려고 하면 오류가 발생합니다. 예를 들어 클래스 외부에서 MAXIMUM의 초기 값을 설정하려고 하면 런타임 오류가 발생합니다.

```
class A
{
public const MINIMUM:int = 0;
public const MAXIMUM:int;
}

var a:A = new A();
a["MAXIMUM"] = 10; // run-time error
```

ActionScript 3.0에는 사용자를 위해 다양한 상수가 정의되어 있습니다. 규칙에 따라 ActionScript의 상수에는 밑줄(_)로 구분된 단어와 함께 모든 대문자를 사용할 수 있습니다. 예를 들어, MouseEvent 클래스 정의에서는 이 이름 지정 규칙을 사용하여 상수를 정의하며, 각 상수는 마우스 입력과 연관된 이벤트를 나타냅니다.

```
package flash.events
{
    public class MouseEvent extends Event
    {
    public static const CLICK:String = "click";
    public static const DOUBLE_CLICK:String = "doubleClick";
    public static const MOUSE_DOWN:String = "mouseDown";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...
    }
}
```

연산자

연산자는 하나 이상의 피연산자를 사용하여 값을 반환하는 특수 함수입니다. **피연산자**는 연산자가 입력값으로 사용하는 값으로, 일반적으로 리터럴, 변수 또는 표현식입니다. 예를 들어, 다음 코드에서 더하기(+) 및 곱하기(*) 연산자는 세 개의 리터럴 피연산자(2, 3, 4)와 함께 사용되어 결과 값을 반환합니다. 그런 다음 이 값은 대입(=) 연산자가 사용하여 반환된 값 14를 변수 sumNumber에 지정합니다.

```
var sumNumber:uint = 2 + 3 * 4; // uint = 14
```

연산자에는 단항, 이항 또는 삼항 연산자가 있습니다. **단항** 연산자는 한 개의 피연산자를 사용합니다. 예를 들어 증가(++) 연산자는 한 개의 피연산자만 사용하므로 단항 연산자입니다. **이항** 연산자는 두 개의 피연산자를 사용합니다. 예를 들어, 나누기(/) 연산자는 두 개의 피연산자를 사용합니다. **삼항** 연산자는 세 개의 피연산자를 사용합니다. 예를 들어 조건(?:) 연산자는 세 개의 피연산자를 사용합니다.

일부 연산자는 **오버로드**할 수 있습니다. 이는 연산자에 전달된 피연산자의 유형 또는 개수에 따라 연산자의 동작이 달라진다는 것을 의미합니다. 더하기(+) 연산자는 피연산자의 데이터 유형에 따라 다르게 동작하는 오버로드된 연산자의 예입니다. 피연산자가 모두 숫자인 경우 더하기 연산자는 값의 합계를 반환합니다. 피연산자가 모두 문자열이면 더하기 연산자는 두 피연산자를 연결한 문자열을 반환합니다. 다음 예제 코드에서는 연산자가 피연산자에 따라 다르게 동작하는 방식을 보여 줍니다.

```
trace(5 + 5); // 10
trace("5" + "5"); // 55
```

연산자는 제공된 피연산자의 수에 따라 다르게 동작할 수도 있습니다. 빼기(-) 연산자는 단항 연산자이면서 이항 연산자입니다. 피연산자가 한 개만 제공되면 빼기 연산자는 피연산자를 음수로 만들어 결과를 반환합니다. 피연산자가 두 개 제공되면 빼기 연산자는 피연산자 간의 차이를 반환합니다. 다음 예제에서는 먼저 단항 연산자로 사용된 다음 이항 연산자로 사용된 빼기 연산자를 보여 줍니다.

```
trace(-3); // -3
trace(7 - 2); // 5
```

연산자 우선 순위와 연산 순서

연산자 우선 순위와 연산 순서에 따라 연산자 처리 순서가 결정됩니다. 산술에 익숙한 독자에게는 컴파일러에서 더하기(+) 연산자보다 곱하기(*) 연산자를 먼저 처리하는 것이 당연해 보이겠지만 컴파일러에게는 어떤 연산자를 먼저 처리할지에 대한 명시적인 지시 사항이 필요합니다. 이러한 지시 사항을 전체적으로 **연산자 우선 순위**라고 합니다. ActionScript에 정의된 기본 연산자 우선 순위는 괄호(()) 연산자를 사용하여 변경할 수 있습니다. 예를 들어 다음 코드는 컴파일러에서 곱하기 연산자보다 더하기 연산자를 먼저 처리하도록 이전 예제의 기본 우선 순위를 변경합니다.

```
var sumNumber:uint = (2 + 3) * 4; // uint == 20
```

우선 순위가 동일한 둘 이상의 연산자가 동일한 표현식에 나타나는 경우가 발생할 수도 있습니다. 이 경우 컴파일러는 **연산 순서** 규칙을 사용하여 먼저 처리할 연산자를 결정합니다. 대입 연산자를 제외한 모든 이진 연산자는 **왼쪽 연관**이므로 왼쪽에 있는 연산자가 오른쪽에 있는 연산자보다 먼저 처리됩니다. 대입 연산자와 조건(?:) 연산자는 **오른쪽 연관**이므로 오른쪽에 있는 연산자가 왼쪽에 있는 연산자보다 먼저 처리됩니다.

예를 들어, 보다 작음(<) 연산자와 보다 큼(>) 연산자의 경우 우선 순위가 동일합니다. 두 연산자는 왼쪽 연관이므로 동일한 표현식에 사용될 경우 왼쪽에 있는 연산자가 먼저 처리됩니다. 즉, 다음 두 명령문의 결과는 동일합니다.

```
trace(3 > 2 < 1); // false
trace((3 > 2) < 1); // false
```

보다 큼 연산자가 먼저 처리되며 피연산자 3이 피연산자 2보다 크기 때문에 true를 반환합니다. 그런 다음 true 값이 피연산자 1과 함께 보다 작음 연산자로 전달됩니다. 다음 코드는 이 중간 상태를 나타냅니다.

```
trace((true) < 1);
```

보다 작음 연산자는 true 값을 숫자 값 1로 변환하고 두 번째 피연산자 1과 비교한 후 값 1이 1보다 작지 않으므로 false 값을 반환합니다.

```
trace(1 < 1); // false
```

괄호 연산자를 사용하면 기본적으로 적용되는 왼쪽 연산 순서를 변경할 수 있습니다. 연산자와 피연산자를 괄호로 묶어 컴파일러에서 보다 작음 연산자를 먼저 처리하도록 지시할 수 있습니다. 다음 예제에서는 괄호 연산자를 사용하여 이전 예제에서와 동일한 숫자로 다른 결과를 생성합니다.

```
trace(3 > (2 < 1)); // true
```

보다 작음 연산자가 먼저 처리되며 피연산자 2가 피연산자 1보다 크기 때문에 false 값을 반환합니다. 그런 다음 false 값이 피연산자 3과 함께 보다 큼 연산자로 전달됩니다. 다음 코드는 이 중간 상태를 나타냅니다.

```
trace(3 > (false));
```

보다 큼 연산자는 false 값을 숫자 값 0으로 변환하고 다른 피연산자 3과 비교한 후 값 3이 0보다 크므로 true 값을 반환합니다.

```
trace(3 > 0); // true
```

다음 표에는 ActionScript 3.0의 연산자가 우선 순위의 내림차순으로 나열되어 있습니다. 표의 각 행에는 우선 순위가 같은 연산자가 포함되어 있습니다. 이 표에서 각 행의 연산자는 아래 행에 있는 연산자보다 우선 순위가 높습니다.

그룹	**연산자**
기본	[] {x:y} () f(x) new x.y x[y] <></> @ :: ..
후위	x++ x--
단항	++x --x + - ~ ! delete typeof void
곱셈	* / %
추가	+ -
비트 시프트	<< >> >>>
비교	< > <= >= as in instanceof is
항등	== != === !==
비트 AND	&
비트 XOR	^
비트 OR	|
논리 AND	&&
논리 OR	||
조건	?:
대입	= *= /= %= += -= <<= >>= >>>= &= ^= |=
쉼표	,

기본 연산자

기본 연산자에는 Array 및 Object 리터럴을 만들고, 표현식을 그룹화하고, 함수를 호출하고, 클래스 인스턴스를 인스턴스화하며, 속성에 액세스하는 데 사용되는 연산자가 포함됩니다.

다음 표에 나열된 기본 연산자의 우선 순위는 모두 동일합니다. E4X 사양의 일부인 연산자는 E4X 표기법에 따라 표시합니다.

연산자	수행하는 연산
[]	배열 초기화
{x:y}	객체 초기화
()	표현식 그룹화
f(x)	함수 호출
new	생성자 호출
x.y x[y]	속성에 액세스
<></>	XMLList 객체 초기화(E4X)
@	특성에 액세스(E4X)
::	이름 정규화(E4X)
..	자손 XML 요소에 액세스(E4X)

후위 연산자

후위 연산자는 하나의 피연산자를 사용하며 값을 증가시키거나 감소시킵니다. 이 연산자는 단항 연산자이나, 높은 우선 순위와 특별한 동작 때문에 다른 단항 연산자와는 별개로 분류됩니다. 후위 연산자를 큰 표현식의 일부분으로 사용할 경우 후위 연산자가 처리되기 전에 표현식 값이 반환됩니다. 예를 들어, 다음 코드는 값이 증가되기 전에 xNum++ 표현식의 값이 처리되는 방식을 보여 줍니다.

```
var xNum:Number = 0;
trace(xNum++); // 0
trace(xNum); // 1
```

다음 표에 나열된 후위 연산자의 우선 순위는 모두 동일합니다.

연산자	수행하는 연산
++	증가(후위)
--	감소(후위)

단항 연산자

단항 연산자는 하나의 피연산자를 사용합니다. 이 그룹의 증가(++) 및 감소(--) 연산자는 **전위연산자**로 표현식에서 피연산자 앞에 나타납니다. 전위 연산자의 경우 전체 표현식 값이 반환되기 전에 증가 또는 감소 연산이 완료된다는 점에서 후위 연산자와 다릅니다. 예를 들어 다음 코드에서는 값이 증가한 후에 ++xNum 표현식의 값이 처리되는 방식을 보여 줍니다.

```
var xNum:Number = 0;
trace(++xNum); // 1
trace(xNum); // 1
```

다음 표에 나열된 단항 연산자의 우선 순위는 모두 동일합니다.

연산자	수행하는 연산
++	증가(전위)
--	감소(전위)
+	단항 덧셈
-	단항 뺄셈(부정)
!	논리 NOT
~	비트 NOT
delete	속성 삭제
typeof	유형 정보 반환
void	정의되지 않은 값 반환

곱셈 연산자

곱셈 연산자는 피연산자 2개를 사용하여 곱하기, 나누기, 모듈러스 등을 계산합니다.

다음 표에 나열된 곱셈 연산자의 우선 순위는 모두 동일합니다.

연산자	수행하는 연산
*	곱하기
/	나누기
%	모듈러스

덧셈 연산자

덧셈 연산자는 피연산자 2개를 사용하여 더하기 또는 빼기를 계산합니다. 다음 표에 나열된 덧셈 연산자의 우선 순위는 모두 동일합니다.

연산자	수행하는 연산
+	더하기
-	빼기

비트 시프트 연산자

비트 시프트 연산자는 피연산자 2개를 사용하여 처음 피연산자의 비트를 두 번째 연산자에서 지정한 곳까지 확장합니다. 다음 표에 나열된 비트 시프트 연산자의 우선 순위는 모두 동일합니다.

연산자	수행하는 연산
<<	비트 왼쪽 시프트
>>	비트 오른쪽 시프트
>>>	(비트 부호 없는 오른쪽 시프트)

비교 연산자

비교 연산자는 피연산자 2개를 사용하여 값을 비교하며 부울 값을 반환합니다. 다음 표에 나열된 비교 연산자의 우선 순위는 모두 동일합니다.

연산자	수행하는 연산
<	보다 작음
>	보다 큼
<=	보다 작거나 같음
>=	보다 크거나 같음
as	데이터 유형 확인
in	객체 속성
instanceof	프로토타입 체인 확인
is	데이터 유형 확인

항등 연산자

항등 연산자는 피연산자 2개를 사용하여 값을 비교하고 부울 값을 반환합니다. 다음 표에 나열된 항등 연산자의 우선 순위는 모두 동일합니다.

연산자	수행하는 연산
==	항등
!=	비항등
===	완전 항등
!==	완전 비항등

비트 논리 연산자

비트 논리 연산자는 피연산자 2개를 사용하여 비트 수준 논리 연산을 수행합니다. 비트 논리 연산자 간에는 우선 순위가 다릅니다. 다음 표에는 비트 논리 연산자가 우선 순위의 내림차순으로 나열되어 있습니다.

연산자	수행하는 연산
&	비트 AND
^	비트 XOR
|	비트 OR

논리 연산자

논리 연산자는 피연산자 2개를 사용하여 부울 결과를 반환합니다. 논리 연산자 간에는 우선 순위가 다릅니다. 다음 표에는 논리 연산자가 우선 순위의 내림차순으로 나열되어 있습니다.

연산자	수행하는 연산
&&	논리 AND
||	논리 OR

조건 연산자

조건 연산자는 삼항 연산자로서 피연산자 3개를 사용합니다. 조건 연산자는 if..else 조건문을 적용하는 간단한 방법입니다.

연산자	수행하는 연산
?:	조건

대입 연산자

대입 연산자는 두 피연산자를 사용하여 한 피연산자의 값에 따라 다른 피연산자에 값을 대입합니다. 다음 표에 나열된 대입 연산자의 우선 순위는 모두 동일합니다.

연산자	수행하는 연산
=	대입
*=	곱하기 대입
/=	나누기 대입
%=	모듈러스 대입
+=	더하기 대입
-=	빼기 대입
<<=	비트 왼쪽 시프트 대입
>>=	비트 오른쪽 시프트 대입
>>>=	비트 부호 없는 오른쪽 시프트 대입
&=	비트 AND 대입
^=	비트 XOR 대입
|=	비트 OR 대입

조건문

ActionScript 3.0에서는 프로그램 흐름을 제어하는 데 사용할 수 있는 세 가지 기본 조건문을 제공합니다.

if..else

if..else 조건문을 사용하면 조건을 테스트한 다음 해당 조건이 존재하면 코드 블록을 실행하고 해당 조건이 존재하지 않으면 다른 코드 블록을 실행할 수 있습니다. 예를 들어 다음 코드에서는 x 값이 20을 초과하는지 테스트한 다음 20을 초과하면 trace() 함수를 호출하고 20을 초과하지 않으면 다른 trace() 함수를 호출합니다.

```
if (x > 20)
{
    trace("x is > 20");
}
else
{
    trace("x is <= 20");
}
```

다른 코드 블록을 실행하지 않으려면 else 문 없이 if 문을 사용할 수 있습니다.

if..else if

if..else if 조건문을 사용하여 둘 이상의 조건을 테스트할 수 있습니다. 다음 코드는 x 값이 20을 초과하는지 여부를 테스트할 뿐 아니라 x 값이 음수인지 여부도 테스트합니다.

```
if (x > 20)
{
    trace("x is > 20");
}
else if (x < 0)
{
    trace("x is negative");
}
```

if 또는 else 문 다음에 단 하나의 명령문이 오는 경우 명령문을 중괄호로 묶지 않아도 됩니다. 예를 들어 다음 코드에서는 중괄호를 사용하지 않습니다.

```
if (x > 0)
    trace("x is positive");
else if (x < 0)
    trace("x is negative");
else
    trace("x is 0");
```

그러나 명령문이 나중에 중괄호가 없는 조건문에 추가될 경우 예기치 못한 비헤이비어가 발생할 수 있으므로 항상 중괄호를 사용하는 것이 좋습니다. 예를 들어, 다음 코드에서는 조건이 true로 평가되는지에 관계없이 positiveNums의 값은 1씩 증가합니다.

```
var x:int;
var positiveNums:int = 0;

if (x > 0)
    trace("x is positive");
    positiveNums++;

trace(positiveNums); // 1
```

switch

switch 문은 동일한 조건 표현식에 따라 실행 경로가 여러 가지로 달라지는 경우에 유용합니다. 일련의 if..else if 문과 유사한 기능을 제공하지만 보다 쉽게 읽을 수 있습니다. switch 문은 조건의 부울 값을 테스트하는 대신 표현식을 평가하고 그 결과를 사용하여 실행할 코드 블록을 결정합니다. 코드 블록은 case 문으로 시작하고 break 문으로 끝납니다. 예를 들어, 다음 switch 문은 Date.getDay() 메서드에서 반환된 일수에 따라 요일을 표시합니다.

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
    case 0:
        trace("Sunday");
        break;
    case 1:
        trace("Monday");
        break;
    case 2:
        trace("Tuesday");
        break;
    case 3:
        trace("Wednesday");
        break;
    case 4:
        trace("Thursday");
        break;
    case 5:
        trace("Friday");
        break;
    case 6:
        trace("Saturday");
        break;
    default:
        trace("Out of range");
        break;
}
```

루프

반복문을 사용하면 일련의 값 또는 변수를 사용하여 특정 코드 블록을 반복해서 수행할 수 있습니다. 코드 블록은 항상 중괄호({})로 묶는 것이 좋습니다. 코드 블록에 명령문이 하나만 포함되어 있으면 중괄호를 생략해도 되지만, 조건문에서와 같이 나중에 추가된 명령문을 코드 블록에서 실수로 제외할 수 있기 때문에 중괄호를 사용하는 것이 좋습니다. 코드 블록에 포함할 명령문을 나중에 추가하고 필수 중괄호는 추가하지 않은 경우 명령문이 루프의 일부로 실행되지 않습니다.

for

for 루프를 사용하면 특정 값 범위에서 변수를 반복할 수 있습니다. for 문에 초기 값으로 설정되는 변수, 루핑 종료 시점을 결정하는 조건문, 각 루프의 변수 값을 변경하는 표현식과 같은 세 개의 표현식을 제공해야 합니다. 예를 들어, 다음 루프는 5번 반복됩니다. i 변수 값은 0에서 시작하여 4에서 끝나며 각 행에 0부터 4까지의 숫자로 출력됩니다.

```
var i:int;
for (i = 0; i < 5; i++)
{
    trace(i);
}
```

for..in

for..in 루프는 객체의 속성 또는 배열 요소를 반복합니다. 예를 들어, for..in 루프를 사용하여 일반 객체의 속성을 반복할 수 있습니다. 객체 속성은 특정한 순서로 보존되지 않으므로 임의의 순서로 나타납니다.

```
var myObj:Object = {x:20, y:30};
for (var i:String in myObj)
{
    trace(i + ": " + myObj[i]);
}
// output:
// x: 20
// y: 30
```

배열 요소를 반복할 수도 있습니다.

```
var myArray:Array = ["one", "two", "three"];
for (var i:String in myArray)
{
    trace(myArray[i]);
}
// output:
// one
// two
// three
```

객체가 봉인 클래스(내장 클래스 및 사용자 정의 클래스 포함)의 인스턴스인 경우 객체의 속성을 반복할 수 없습니다. 속성을 반복할 수 있는 클래스는 동적 클래스뿐입니다. 동적 클래스의 인스턴스라도 동적으로 추가된 속성만 반복할 수 있습니다.

for each..in

for each..in 루프는 컬렉션의 항목을 반복합니다. XML 또는 XMLList 객체의 태그, 객체 속성의 값, 배열의 요소 등이 컬렉션의 항목이 될 수 있습니다. 예를 들어 다음 코드와 같이 for each..in 루프를 사용하여 일반 객체의 속성을 반복할 수 있지만, for..in 루프와 달리 for each..in 루프의 반복 변수에는 속성 이름 대신 속성 값이 포함됩니다.

```
var myObj:Object = {x:20, y:30};
for each (var num in myObj)
{
    trace(num);
}
// output:
// 20
// 30
```

다음 예제와 같이 XML 또는 XMLList 객체를 반복할 수 있습니다.

```
var myXML:XML = <users>
    <fname>Jane</fname>
    <fname>Susan</fname>
    <fname>John</fname>
</users>;

for each (var item in myXML.fname)
{
    trace(item);
}
/* output
Jane
Susan
John
*/
```

다음 예제와 같이 배열 요소도 반복할 수 있습니다.

```
var myArray:Array = ["one", "two", "three"];
for each (var item in myArray)
{
    trace(item);
}
// output:
// one
// two
// three
```

객체가 봉인 클래스의 인스턴스인 경우 객체의 속성을 반복할 수 없습니다. 동적 클래스의 인스턴스인 경우에도 고정된 모든 속성 즉, 클래스 정의의 일부로 정의된 속성은 반복할 수 없습니다.

while

while 루프는 조건이 true일 경우 반복되는 if 문과 비슷합니다. 예를 들어, 다음 코드에서는 for 루프의 예제와 동일한 결과를 생성합니다.

```
var i:int = 0;
while (i < 5)
{
    trace(i);
    i++;
}
```

for 루프 대신 while 루프를 사용할 때의 한 가지 단점은 while 루프를 사용할 때 무한 루프가 작성될 가능성이 높다는 점입니다. for 루프 예제 코드는 카운터 변수를 증가시키는 표현식을 생략하면 컴파일되지 않지만, while 루프 예제는 해당 단계를 생략해도 컴파일됩니다. 루프에 i를 증가시키는 표현식이 없으면 무한 루프가 됩니다.

do..while

do..while 루프는 코드 블록이 한 번 이상 실행되는 while 루프입니다. 이는 코드 블록이 실행된 후 조건이 확인되기 때문입니다. 다음 코드는 조건을 충족하지 않아도 출력을 생성하는 간단한 do..while 루프 예제를 보여 줍니다.

```
var i:int = 5;
do
{
    trace(i);
    i++;
} while (i < 5);
// output: 5
```

함수

함수는 특정 작업을 수행하는 코드 블록이며 프로그램에서 다시 사용할 수 있습니다. ActionScript 3.0에는 메서드와 함수 클로저라는 두 가지 함수가 있습니다. 함수가 정의된 컨텍스트에 따라 함수를 메서드 또는 함수 클로저라고 부릅니다. 함수를 클래스 정의의 일부로 정의하거나 객체의 인스턴스에 추가한 경우 이 함수를 메서드라고 합니다. 이와 다른 방식으로 함수를 정의한 경우 이 함수를 함수 클로저라고 합니다.

ActionScript에서 함수는 항상 매우 중요한 역할을 수행해 왔습니다. 예를 들어, ActionScript 1.0에서는 class 키워드가 없어 생성자 함수에 의해 "클래스"가 정의되었습니다. 언어에 class 키워드가 추가되었지만 언어에서 제공하는 모든 기능을 활용하려면 함수를 확실하게 이해하는 것이 여전히 중요합니다. ActionScript 함수가 C++ 또는 Java와 같은 언어에서 제공되는 함수와 유사하게 동작할 것이라고 예상하는 프로그래머에게는 ActionScript 함수를 이해하는 것이 어려운 과제일 수 있습니다. 숙련된 프로그래머에게는 기본 함수 정의 및 호출이 어렵지 않을 수 있지만 ActionScript 함수의 일부 고급 기능에 대해서는 설명이 필요합니다.

기본 함수 개념

함수 호출

함수 식별자 뒤에 괄호 연산자(())를 사용하여 함수를 호출합니다. 함수에 전달하려는 모든 함수 매개 변수를 괄호 연산자를 사용하여 묶습니다. 예를 들어 trace() 함수는 ActionScript 3.0에서 최상위 함수 중 하나입니다.

```
trace("Use trace to help debug your script");
```

매개 변수 없이 함수를 호출하려면 비어 있는 괄호를 사용해야 합니다. 예를 들어, 매개 변수를 사용하지 않는 Math.random() 메서드를 사용하여 임의의 숫자를 생성합니다.

```
var randomNum:Number = Math.random();
```

함수 정의

ActionScript 3.0에서는 함수 명령문을 사용하거나 함수 표현식을 사용하는 두 가지 방법으로 함수를 정의합니다. 정적 또는 동적인 프로그래밍 스타일 중 선호하는 스타일에 따라 방법을 선택합니다. 정적 또는 엄격 모드의 프로그래밍을 선호하는 경우 함수 명령문을 사용하여 함수를 정의합니다. 특정한 요구에 함수 표현식이 적합하면 함수 표현식을 사용하여 함수를 정의합니다. 함수 표현식은 주로 동적 또는 표준 모드의 프로그래밍에 사용됩니다.

함수 명령문

함수 명령문은 엄격 모드에서 함수를 정의할 때 주로 이용하는 방법입니다. 함수 명령문은 function 키워드로 시작하며 그 뒤에 다음이 추가됩니다.

- 함수 이름
- 괄호 안에 쉼표로 구분된 목록의 매개 변수
- 함수 본문, 즉 함수를 호출하면 실행되는 중괄호로 묶인 ActionScript 코드

예를 들어 다음 코드에서는 매개 변수가 정의된 함수를 만든 다음 매개 변수 값으로 "hello" 문자열을 사용하여 함수를 호출합니다.

```
function traceParameter(aParam:String)
{
    trace(aParam);
}

traceParameter("hello"); // hello
```

함수 표현식

함수를 선언하는 두 번째 방법은 함수 리터럴 또는 익명 함수라고도 하는 함수 표현식이 포함된 대입문을 사용하는 것입니다. 이는 이전 버전의 ActionScript에서 많이 사용했던 방식으로 함수 명령문을 사용하는 방식에 비해 코드가 길어집니다.

함수 표현식이 포함된 대입문은 var 키워드로 시작하며 그 뒤에 다음이 추가됩니다.

- 함수 이름
- 콜론 연산자(:)
- 데이터 유형을 나타내는 Function 클래스
- 대입 연산자(=)
- function 키워드
- 괄호 안에 쉼표로 구분된 목록의 매개 변수
- 함수 본문, 즉 함수를 호출하면 실행되는 중괄호로 묶인 ActionScript 코드

  예를 들어, 다음 코드는 함수 표현식을 사용하여 traceParameter 함수를 선언합니다.

```
var traceParameter:Function = function (aParam:String)
{
    trace(aParam);
};
traceParameter("hello"); // hello
```

함수 명령문에서와는 달리 함수 이름을 지정하지 않습니다. 함수 표현식과 함수 명령문 간의 또 다른 중요한 차이점은 함수 표현식은 명령문이 아닌 표현식이라는 점입니다. 함수 명령문과 달리 함수 표현식은 독립적으로 사용할 수 없습니다. 함수 표현식은 일반적으로 대입문과 같은 명령문의 일부로만 사용할 수 있습니다. 다음 예제에서는 배열 요소에 지정된 함수 표현식을 보여 줍니다.

```
var traceArray:Array = new Array();
traceArray[0] = function (aParam:String)
{
    trace(aParam);
};
traceArray[0]("hello");
```

명령문과 표현식 간에 선택

일반적으로 특정 상황에서 표현식이 필요하지 않은 경우 함수 명령문을 사용합니다. 함수 명령문이 상대적으로 간단하며 엄격 모드와 표준 모드에서 함수 표현식보다 일관성 있게 사용할 수 있습니다.

함수 명령문은 함수 표현식이 포함된 대입문보다 쉽게 읽을 수 있습니다. 함수 명령문은 코드를 더욱 간결하게 만들어 var 및 function 키워드를 모두 사용해야 하는 함수 표현식보다 혼란을 덜 초래합니다.

함수 명령문을 사용하여 선언된 메서드를 호출할 때 엄격 모드와 표준 모드 모두에서 도트 구문을 사용할 수 있다는 점에서 함수 명령문은 두 컴파일러 모드 간의 일관성이 함수 표현식보다 낫습니다. 함수 표현식으로 선언된 메서드의 경우 도트 구문을 사용하여 호출하지 못할 수도 있습니다. 예를 들어, 다음 코드에서는 함수 표현식으로 선언된 methodExpression() 메서드와 함수 명령문으로 선언된 methodStatement() 메서드가 있는 Example 클래스를 정의합니다. 엄격 모드에서는 methodExpression() 메서드를 호출하기 위해 도트 구문을 사용할 수 없습니다.

```
class Example
{
var methodExpression = function() {}
function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // error in strict mode; okay in standard mode
myEx.methodStatement(); // okay in strict and standard modes
```

함수 표현식은 런타임 또는 동적인 비헤이비어에 중점을 두는 프로그래밍에 사용하는 것이 보다 적합합니다. 엄격 모드를 사용하려고 하지만 함수 표현식으로 선언된 메서드를 호출해야 하는 경우 다음 두 가지 방법 중 하나를 사용할 수 있습니다. 첫 번째, 도트(.) 연산자 대신 대괄호([])를 사용하여 메서드를 호출할 수 있습니다. 엄격 모드와 표준 모드 모두에서 다음 메서드 호출이 성공했습니다.

```
myExample["methodLiteral"]();
```

두 번째, 전체 클래스를 동적 클래스로 선언할 수 있습니다. 이렇게 하면 도트 연산자를 사용하여 메서드를 호출할 수 있지만 해당 클래스의 모든 인스턴스에 대한 엄격 모드 기능의 일부가 제대로 작동하지 않을 수 있습니다. 예를 들어, 동적 클래스의 인스턴스의 정의되지 않은 속성에 액세스하려고 하는 경우 컴파일러에서 오류를 생성하지 않습니다.

함수 표현식을 사용하는 것이 유용한 경우도 있습니다. 함수 표현식은 한 번만 사용하고 폐기할 함수를 정의하는 경우에 일반적으로 사용됩니다. 이보다 사용 빈도는 낮지만 프로토타입 속성에 함수를 연결하는 경우에도 사용됩니다. 자세한 내용은 프로토타입 객체을 참조하십시오.

사용할 방법을 선택할 때는 함수 명령문과 함수 표현식 간의 두 가지 미세한 차이점을 고려해야 합니다. 첫 번째 차이점은 메모리 관리 및 가비지 컬렉션과 관련해서 함수 표현식은 독립된 객체가 아니라는 점입니다. 즉, 배열 요소 또는 객체 속성과 같은 다른 객체에 함수 표현식을 지정할 때는 코드에서 해당 함수 표현식에 대한 참조만 만듭니다. 함수 표현식이 연결된 배열 또는 객체가 범위를 벗어나거나 더 이상 사용할 수 없는 경우 함수 표현식에 더 이상 액세스할 수 없습니다. 배열 또는 객체를 삭제하면 함수 표현식에서 사용하는 메모리가 가비지 컬렉션의 대상이 되므로 해당 메모리를 다른 용도로 이용하거나 재사용할 수 있습니다.

다음 예제에서는 함수 표현식의 경우 표현식이 지정된 속성이 삭제되면 함수를 더 이상 사용할 수 없다는 것을 보여 줍니다. Test 클래스는 동적이므로 함수 표현식이 저장된 functionExp 속성을 추가할 수 있습니다. 도트 연산자를 사용하여 functionExp() 함수를 호출할 수 있지만 functionExp 속성이 삭제되면 더 이상 함수에 액세스할 수 없습니다.

```
dynamic class Test {}
var myTest:Test = new Test();

// function expression
myTest.functionExp = function () { trace("Function expression") };
myTest.functionExp(); // Function expression
delete myTest.functionExp;
myTest.functionExp(); // error
```

반면에 먼저 함수 명령문으로 함수를 정의하면 이 함수는 독립된 객체로 존재하며 함수가 연결된 속성을 삭제한 후에도 계속해서 사용할 수 있습니다. delete 연산자는 객체의 속성에 대해서만 작동하므로 stateFunc() 함수 자체를 삭제하는 호출도 작동하지 않습니다.

```
dynamic class Test {}
var myTest:Test = new Test();

// function statement
function stateFunc() { trace("Function statement") }
myTest.statement = stateFunc;
myTest.statement(); // Function statement
delete myTest.statement;
delete stateFunc; // no effect
stateFunc();// Function statement
myTest.statement(); // error
```

함수 명령문과 함수 표현식 간의 두 번째 차이점은 함수 명령문은 함수 명령문 앞에 나오는 명령문을 포함하여 함수 명령문이 정의된 범위 전체에서 사용된다는 점입니다. 반면에 함수 표현식은 후속 명령문에 대해서만 정의됩니다. 예를 들어 다음 코드에서는 scopeTest() 함수를 정의하기 전에 해당 함수를 성공적으로 호출합니다.

```
statementTest(); // statementTest

function statementTest():void
{
    trace("statementTest");
}
```

함수 표현식은 정의하기 전에 사용할 수 없기 때문에 다음 코드에서 런타임 오류가 발생합니다.

```
expressionTest(); // run-time error

var expressionTest:Function = function ()
{
    trace("expressionTest");
}
```

함수에서 값 반환

함수에서 값을 반환하려면 return 문 다음에 반환할 표현식 또는 리터럴 값을 지정합니다. 예를 들어, 다음 코드는 매개 변수를 나타내는 표현식을 반환합니다.

```
function doubleNum(baseNum:int):int
{
    return (baseNum * 2);
}
```

return 문은 함수를 종료하므로 다음과 같이 return 문 아래의 명령문이 모두 실행되지 않습니다.

```
function doubleNum(baseNum:int):int {
    return (baseNum * 2);
    trace("after return"); // This trace statement will not be executed.
}
```

반환 유형을 지정하기로 결정한 경우 엄격 모드에서는 적절한 유형의 값을 반환해야 합니다. 예를 들어 다음 코드에서는 올바른 값을 반환하지 않기 때문에 엄격 모드에서 오류가 발생합니다.

```
function doubleNum(baseNum:int):int
{
    trace("after return");
}
```

중첩 함수

함수를 중첩할 수 있으므로 다른 함수 내에 함수를 선언할 수 있습니다. 함수에 대한 참조가 외부 코드로 전달되지 않으면 중첩 함수는 부모 함수 내에서만 사용할 수 있습니다. 예를 들어, 다음 코드는 getNameAndVersion() 함수 내에 두 개의 중첩 함수를 선언합니다.

```
function getNameAndVersion():String
{
    function getVersion():String
    {
        return "10";
    }
    function getProductName():String
    {
        return "Flash Player";
    }
    return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Flash Player 10
```

중첩 함수가 외부 코드로 전달되면 해당 함수는 함수 클로저로 전달되므로 함수가 정의될 때 범위에 있는 모든 정의가 함수에 그대로 유지됩니다. 자세한 내용은 함수 범위을 참조하십시오.

함수 매개 변수

ActionScript 3.0의 함수 매개 변수에는 이 언어를 처음 접하는 프로그래머에게는 신기할 수 있는 몇 가지 기능이 있습니다. 대부분의 프로그래머는 값 또는 참조에 의한 매개 변수 전달이라는 개념에 익숙하지만 arguments 객체 및 ... (rest) 매개 변수에는 익숙하지 않을 수 있습니다.

값 또는 참조에 의한 인수 전달

여러 프로그래밍 언어에서 값에 의한 인수 전달과 참조에 의한 인수 전달 간의 차이점은 코드를 설계하는 방법에 영향을 미칠 수 있기 때문에 잘 이해해야 합니다.

값에 의한 전달은 인수 값이 함수 내에서 사용할 로컬 변수에 복사되는 것을 의미합니다. 참조에 의한 전달은 실제 값 대신 인수에 대한 참조만 전달되는 것을 의미합니다. 이 경우 실제 인수가 복사되지는 않습니다. 대신 인수로 전달된 변수에 대한 참조가 만들어져 함수 내에서 사용할 로컬 변수에 지정됩니다. 로컬 변수를 함수 외부의 변수에 대한 참조로 사용하여 원본 변수의 값을 변경할 수 있습니다.

ActionScript 3.0에서는 모든 값이 객체로 저장되므로 모든 인수가 참조를 기준으로 전달됩니다. 그러나 Boolean, Number, int, uint 및 String 등이 포함된 프리미티브 데이터 유형에 속한 객체에는 값을 기준으로 전달된 것과 같이 동작하도록 하는 특수 연산자가 있습니다. 예를 들어 다음 코드에서는 xParam 및 yParam이라고 하는 int 유형의 두 매개 변수를 정의하는 passPrimitives() 함수를 만듭니다. 이러한 매개 변수는 passPrimitives() 함수 본문 내에 선언된 로컬 변수와 유사합니다. xValue 및 yValue 인수를 사용하여 함수를 호출하면 xParam 및 yParam 매개 변수는 xValue 및 yValue로 표시되는 int 객체에 대한 참조로 초기화됩니다. 인수가 프리미티브 값이기 때문에 값을 기준으로 전달된 것처럼 동작합니다. 처음에는 xParam 및 yParam에 xValue 및 yValue 객체에 대한 참조만 포함되지만 함수 본문 내에서 xParam 및 yParam의 값을 변경하면 메모리에 새 값의 복사본이 생성됩니다.

```
function passPrimitives(xParam:int, yParam:int):void
{
    xParam++;
    yParam++;
    trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);// 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);// 10 15
```

passPrimitives() 함수 내에서는 xParam 및 yParam의 값이 증가되지만 마지막 trace 문에서와 같이 xValue 및 yValue의 값에는 영향을 주지 않습니다. 함수 내에 있는 xValue 및 yValue는 함수 외부에 있는 동일한 이름의 변수와 별개로 존재하는 메모리의 새 위치를 가리키기 때문에 매개 변수가 xValue 및 yValue 변수와 이름이 동일한 경우에도 영향을 미치지 않습니다.

프리미티브 데이터 유형에 속하지 않은 다른 모든 객체는 항상 참조로 전달되므로 원본 변수의 값을 변경할 수 있습니다. 예를 들어, 다음 코드에서는 x와 y라는 두 가지 속성이 있는 objVar 객체를 만듭니다. 이 객체는 passByRef() 함수에 인수로 전달됩니다. 이 객체는 프리미티브 유형이 아니기 때문에 참조를 기준으로 전달될 뿐만 아니라 참조를 유지합니다. 즉, 함수 내에 있는 매개 변수가 변경되면 함수 외부의 객체 속성에도 적용됩니다.

```
function passByRef(objParam:Object):void
{
    objParam.x++;
    objParam.y++;
    trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar); // 11 16
trace(objVar.x, objVar.y); // 11 16
```

objParam 매개 변수가 objVar 전역 변수와 동일한 객체를 참조합니다. 예제의 trace 문에서 볼 수 있듯이 objParam 객체의 x 및 y 속성을 변경하면 objVar 객체에 반영됩니다.

매개 변수 기본값

ActionScript 3.0에서는 함수의 **매개 변수 기본값**을 선언할 수 있습니다. 매개 변수 기본값이 있는 함수를 호출할 때 기본값이 있는 매개 변수를 생략하면 해당 매개 변수에 대해 함수 정의에 지정된 값이 사용됩니다. 기본값이 있는 모든 매개 변수는 매개 변수 목록의 끝에 배치해야 합니다. 기본값으로 지정된 값은 컴파일 타임 상수여야 합니다. 매개 변수에 대해 기본값을 사용하여 매개 변수를 효과적으로 **선택적 매개 변수**로 만들 수 있습니다. 기본값이 없는 매개 변수는 **필수 매개 변수**로 간주됩니다.

예를 들어, 다음 코드는 세 개의 매개 변수가 있는 함수를 만들며 이 중 두 매개 변수에는 기본값이 있습니다. 하나의 매개 변수만 사용하여 함수를 호출하면 나머지 매개 변수에는 기본값이 사용됩니다.

```
function defaultValues(x:int, y:int = 3, z:int = 5):void
{
    trace(x, y, z);
}
defaultValues(1); // 1 3 5
```

arguments 객체

함수에 인수가 전달되면 arguments 객체를 사용하여 함수에 전달된 매개 변수에 대한 정보에 액세스할 수 있습니다. arguments 객체의 중요한 특징은 다음과 같습니다.

- arguments 객체는 함수에 전달된 매개 변수를 모두 포함하는 배열입니다.
- arguments.length 속성은 함수에 전달된 매개 변수의 수를 보고합니다.
- arguments.callee 속성은 함수 자체에 대한 참조를 제공하며 이는 함수 표현식에 대한 재귀 호출에 유용합니다.

참고: arguments 객체는 이름이 arguments인 매개 변수가 있거나 사용자가 ... (rest) 매개 변수를 사용하는 경우 사용할 수 없습니다.

함수 본문에서 arguments 객체를 참조하는 경우 ActionScript 3.0에서는 함수 정의에 정의된 매개 변수보다 많은 매개 변수를 사용하여 함수를 호출할 수 있지만, 매개 변수의 수가 필수 매개 변수(경우에 따라 모든 선택적 매개 변수도 포함)의 수와 일치하지 않으면 엄격 모드에서 컴파일러 오류가 발생합니다. arguments 객체를 배열로 사용하면 해당 매개 변수가 함수 정의에 정의되어 있는지 여부에 관계없이 함수에 전달된 모든 매개 변수에 액세스할 수 있습니다. 다음은 arguments.length 속성과 함께 arguments 배열을 사용하여 traceArgArray() 함수에 전달된 매개 변수를 모두 추적하는 예제입니다. 이 예제는 표준 모드에서만 컴파일됩니다.

```
function traceArgArray(x:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

arguments.callee 속성은 주로 재귀를 만들 때 익명 함수에 사용됩니다. 이 속성을 사용하여 코드에 융통성을 추가할 수 있습니다. 함수 이름 대신 arguments.callee를 사용하면 개발 주기 과정에서 재귀 함수 이름이 변경된 경우 함수 본문에서 재귀 호출을 변경하는 것에 신경 쓰지 않아도 됩니다. arguments.callee 속성은 다음 함수 표현식에서 재귀를 활성화하는 데 사용됩니다.

```
var factorial:Function = function (x:uint)
{
    if(x == 0)
    {
        return 1;
    }
    else
    {
        return (x * arguments.callee(x - 1));
    }
}

trace(factorial(5)); // 120
```

함수 선언에서 ...(rest) 매개 변수를 사용하면 arguments 객체는 사용할 수 없습니다. 대신 선언한 매개 변수 이름을 사용하여 매개 변수에 액세스해야 합니다.

또한 매개 변수 이름으로 "arguments" 문자열을 사용하면 arguments 객체가 가려지므로 이렇게 사용하지 않는 것이 좋습니다. 예를 들어, traceArgArray() 함수가 다시 작성되어 arguments 매개 변수가 추가되는 경우 함수 본문에 있는 arguments에 대한 참조를 사용하면 arguments 객체가 아닌 매개 변수를 참조하게 됩니다. 다음 코드는 출력을 생성하지 않습니다.

```
function traceArgArray(x:int, arguments:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// no output
```

이전 버전의 ActionScript에 있는 arguments 객체에는 현재 함수를 호출한 함수에 대한 참조인 caller 속성도 포함되어 있습니다. ActionScript 3.0에는 caller 속성이 없지만 호출 함수에 대한 참조가 필요한 경우 호출 함수 자체에 대한 참조인 추가 매개 변수가 전달되도록 호출 함수를 변경할 수 있습니다.

... (rest) 매개 변수

ActionScript 3.0에는 ... (rest) 매개 변수라는 새 매개 변수 선언이 도입되었습니다. 이 매개 변수를 사용하면 쉼표로 구분된 인수를 무제한으로 받아들이는 배열 매개 변수를 지정할 수 있습니다. 매개 변수는 예약어가 아닌 어떤 이름이나 가질 수 있습니다. 이 매개 변수 선언은 마지막으로 지정된 매개 변수여야 합니다. 이 매개 변수를 사용하면 arguments 객체를 사용할 수 없습니다. ... (rest) 매개 변수는 arguments 배열 및 arguments.length 속성과 동일한 기능을 제공하지만 arguments.callee와 유사한 기능은 제공하지 않습니다. ... (rest) 매개 변수를 사용하기 전에 arguments.callee를 사용할 필요가 없는지 확인해야 합니다.

다음은 traceArgArray() 함수를 다시 작성하기 위해 arguments 객체 대신 ... (rest) 매개 변수를 사용하는 예제입니다.

```
function traceArgArray(... args):void
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

... (rest) 매개 변수가 마지막으로 나열되기만 하면 다른 매개 변수와 함께 사용할 수도 있습니다. 다음은 traceArgArray() 함수를 수정하여 int 유형의 x를 첫 번째 매개 변수로 사용하고 ... (rest) 매개 변수를 두 번째 매개 변수로 사용하는 예제입니다. 첫 번째 매개 변수가 더 이상 ... (rest) 매개 변수에서 만든 배열의 일부가 아니기 때문에 첫 번째 값은 출력에서 제외됩니다.

```
function traceArgArray(x: int, ... args)
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 2
// 3
```

객체와 같은 함수

ActionScript 3.0에서 함수는 객체입니다. 함수를 만드는 것은 객체를 만드는 것을 의미합니다. 이 객체는 다른 함수에 매개 변수로 전달될 수 있을 뿐만 아니라 다른 함수에 연결된 속성 및 메서드를 가질 수도 있습니다.

다른 함수에 인수로 전달된 함수는 값이 아닌 참조를 기준으로 전달됩니다. 함수를 인수로 전달하는 경우 식별자만 사용하며 메서드를 호출하는 데 사용하는 괄호 연산자는 사용하지 않습니다. 예를 들어, 다음 코드는 clickListener() 함수를 addEventListener() 메서드에 인수로 전달합니다.

```
addEventListener(MouseEvent.CLICK, clickListener);
```

ActionScript를 처음 접하는 프로그래머에게는 이상하게 보일 수 있지만 함수는 다른 모든 객체와 같이 속성과 메서드를 가질 수 있습니다. 실제로 모든 함수에는 해당 함수에 대해 정의된 매개 변수의 수를 저장하는 length라는 읽기 전용 속성이 있습니다. 이 속성은 함수에 전달된 인수의 수를 보고하는 arguments.length 속성과는 다릅니다. ActionScript에서는 함수에 전달된 인수의 수가 해당 함수에 대해 정의된 매개 변수의 수를 초과할 수 있음을 상기하십시오. 다음 예제에서 두 속성 간의 차이점을 보여 줍니다. 이 예제의 경우 엄격 모드에서는 전달된 인수의 수와 정의된 매개 변수의 수가 정확하게 일치해야 하므로 표준 모드에서만 컴파일됩니다.

```
// Compiles only in standard mode
function traceLength(x:uint, y:uint):void
{
    trace("arguments received: " + arguments.length);
    trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* output:
arguments received: 4
arguments expected: 2 */
```

표준 모드에서 함수 본문 외부에 함수 속성을 정의하는 방법으로 함수 속성을 직접 정의할 수 있습니다. 함수 속성은 함수와 연관된 변수의 상태를 저장할 수 있는 준 정적 속성 역할을 수행할 수 있습니다. 예를 들어, 특정 함수가 호출된 횟수를 추적할 수 있습니다. 사용자의 특정 명령 사용 횟수를 추적하는 기능이 있는 게임을 작성하는 경우 정적 클래스 속성을 사용할 수도 있지만 함수 호출 횟수를 추적하는 기능이 유용할 수 있습니다. 다음은 함수 선언 외부에서 함수 속성을 만들고 함수가 호출될 때마다 속성을 증가시키는 예제입니다. 엄격 모드에서는 함수에 동적 속성을 추가할 수 없기 때문에 이 예제는 표준 모드에서만 컴파일됩니다.

```
// Compiles only in standard mode
var someFunction:Function = function ():void
{
    someFunction.counter++;
}

someFunction.counter = 0;

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

함수 범위

함수 범위는 함수를 호출할 수 있는 프로그램의 위치뿐만 아니라 함수가 액세스할 수 있는 정의를 결정합니다. 변수 식별자에 적용되는 동일한 범위 규칙이 함수 식별자에 적용됩니다. 전역 범위에 선언된 함수는 코드 전체에 사용할 수 있습니다. 예를 들어 ActionScript 3.0에는 코드의 어느 위치에서든 사용할 수 있는 isNaN() 및 parseInt() 등의 전역 함수가 포함되어 있습니다. 다른 함수 내에 선언된 중첩 함수는 선언된 함수 내의 어느 곳에서나 사용할 수 있습니다.

범위 체인

함수가 실행되면 여러 객체와 속성이 만들어집니다. 첫 번째, 함수 본문에 선언된 함수나 로컬 변수 및 매개 변수를 저장하는 **activation 객체**라는 특수 객체가 만들어집니다. activation 객체는 내부 메커니즘이므로 직접 액세스할 수 없습니다. 두 번째, 런타임에서 식별자 선언을 찾을 때 확인하는 정렬된 객체 목록이 포함되어 있는 **범위 체인**이 만들어집니다. 실행되는 모든 함수의 내부 속성에 범위 체인이 저장됩니다. 중첩 함수의 범위 체인은 중첩 함수의 activation 객체에서 시작하여 부모 함수의 activation 객체로 이어집니다. 전역 객체에 도달할 때까지 이러한 방식으로 체인이 계속됩니다. ActionScript 프로그램이 시작되고 모든 전역 변수 및 함수가 포함되면 전역 객체가 만들어집니다.

함수 클로저

함수 클로저는 함수의 스냅샷 및 **사전적 환경**이 포함된 객체입니다. 함수의 사전적 환경에는 함수 범위 체인에 있는 모든 변수, 속성, 메서드 및 객체가 해당 값과 함께 포함됩니다. 함수 클로저는 객체 또는 클래스와는 별도로 함수가 실행될 때 만들어집니다. 자신이 정의된 범위를 유지하는 함수 클로저로 인해 함수가 인수 또는 반환 값으로 다른 범위로 전달될 때 흥미로운 결과가 나타납니다.

예를 들어 다음 코드는 두 개의 함수를 만듭니다. foo()는 사각형의 면적을 계산하는 rectArea()라는 중첩된 함수를 반환하고, bar()는 foo()를 호출하고 반환된 함수 클로저를 myProduct 변수에 저장합니다. bar() 함수에서 로컬 변수 x를 값 2로 정의하지만 함수 클로저 myProduct()를 호출하면 함수 클로저에서는 foo() 함수에 정의된 변수 x (값 40)가 그대로 유지됩니다. 따라서 bar() 함수에서 8 대신 160을 반환합니다.

```
function foo():Function
{
    var x:int = 40;
    function rectArea(y:int):int // function closure defined
    {
        return x * y
    }
    return rectArea;
}
function bar():void
{
    var x:int = 2;
    var y:int = 4;
    var myProduct:Function = foo();
    trace(myProduct(4)); // function closure called
}
bar(); // 160
```

메서드 역시 자신이 만들어진 사전적 환경에 대한 정보를 유지한다는 점에서 함수 클로저와 유사하게 동작합니다. 이 특성은 인스턴스에서 메서드를 추출하여 바인딩된 메서드가 만들어질 때 가장 잘 나타납니다. 함수 클로저와 바인딩된 메서드 간의 주요 차이점은 바인딩된 메서드에 있는 this 키워드 값은 항상 처음에 연결된 인스턴스를 참조하는 반면 함수 클로저에 있는 this 키워드 값은 변경할 수 있다는 것입니다.

4장: ActionScript의 객체 지향 프로그래밍

객체 지향 프로그래밍 소개

OOP(객체 지향 프로그래밍)는 프로그램 내의 코드를 객체로 그룹화하여 코드를 구성하는 방법입니다. 이 정의에서 객체라는 용어는 정보(데이터 값)와 기능이 포함된 개별 요소를 의미합니다. 객체 지향 접근을 사용하여 프로그램을 구성할 경우 특정 정보를 해당 정보와 관련된 공통 기능 또는 동작과 함께 그룹화할 수 있습니다. 예를 들어 앨범 제목, 트랙 제목 또는 아티스트 이름 등의 음악 정보를 "재생 목록에 트랙 추가" 또는 "해당 아티스트의 모든 노래 재생" 같은 기능과 함께 그룹화할 수 있습니다. 이러한 정보와 기능은 단일 항목인 객체(예: "Album" 또는 "MusicTrack")로 결합됩니다. 값과 함수를 하나로 묶을 경우 여러 가지 장점이 있습니다. 주요 장점 중 하나는 여러 변수 대신 하나의 변수만 사용하면 된다는 것입니다. 또한 관련된 기능이 단일 그룹으로 유지된다는 장점도 있습니다. 정보와 기능을 결합하면 실제와 보다 근접한 구조를 갖는 프로그램을 개발할 수 있다는 장점 또한 빼놓을 수 없습니다.

클래스

클래스는 객체를 추상적으로 나타낸 것입니다. 클래스에는 객체에 담을 수 있는 데이터 유형 및 객체에 적용할 수 있는 비헤이비어에 대한 정보가 저장됩니다. 상호 작용하는 객체의 수가 적은 소규모 스크립트를 작성할 때는 이러한 추상화의 유용성이 잘 드러나지 않습니다. 그러나 프로그램의 범위가 커질수록 관리해야 하는 객체 수가 증가하기 마련입니다. 이러한 경우 클래스를 사용하면 객체가 만들어지는 방식 및 객체가 상호 작용하는 방식을 보다 효율적으로 제어할 수 있습니다.

ActionScript 1.0부터 ActionScript 프로그래머는 Function 객체를 사용하여 클래스와 유사한 구문을 만들 수 있었습니다. ActionScript 2.0에서는 class 및 extends 등의 키워드를 통해 클래스가 공식적으로 지원되었습니다. ActionScript 3.0에서는 ActionScript 2.0에 도입된 키워드가 계속 지원될 뿐만 아니라 새로운 기능이 추가되었습니다. 예를 들어 protected 및 internal 특성이 도입되어 ActionScript 3.0에서 액세스 제어가 향상되었습니다. 또한 final 및 override 키워드를 통해 상속을 보다 세밀히 제어할 수 있습니다.

ActionScript에서는 Java, C++ 또는 C# 등의 프로그래밍 언어로 클래스를 만든 경험이 있는 개발자에게 익숙한 환경을 제공합니다. class, extends 및 public 등 다른 프로그래밍 언어에서와 동일한 키워드 및 특성 이름이 ActionScript에서 많이 사용됩니다.

참고: Adobe ActionScript 설명서에서 속성이라는 용어는 변수, 상수 및 메서드를 비롯하여 객체 또는 클래스의 멤버를 나타냅니다. 또한 클래스와 정적이라는 용어가 서로 구별되지 않고 사용되는 경우가 많지만 여기서는 이러한 용어가 서로 구별됩니다. 예를 들어 "클래스 속성"이라는 용어는 정적 멤버뿐만 아니라 클래스의 모든 멤버를 나타내는 데 사용됩니다.

클래스 정의

ActionScript 3.0 클래스 정의에는 ActionScript 2.0 클래스 정의와 비슷한 구문이 사용됩니다. 올바른 클래스 정의 구문에는 class 키워드 뒤에 클래스 이름이 나옵니다. 클래스 이름 뒤에는 중괄호({})로 묶인 클래스 본문이 나옵니다. 예를 들어 다음 코드에서는 visible이라는 변수 하나가 들어 있는 Shape라는 클래스가 만들어집니다.

```
public class Shape
{
    var visible:Boolean = true;
}
```

중요한 구문 변경 내용 중 하나는 패키지 안에 있는 클래스의 정의와 관련된 것입니다. ActionScript 2.0에서는 클래스가 패키지 안에 있는 경우 클래스 선언에 패키지 이름이 포함되어야 합니다. ActionScript 3.0에서는 package 문이 도입되어, 패키지 이름이 클래스 선언이 아닌 패키지 선언에 포함되어야 합니다. 예를 들어 다음 클래스 선언에서는 flash.display 패키지에 속한 BitmapData 클래스가 ActionScript 2.0과 ActionScript 3.0에서 정의된 방식을 볼 수 있습니다.

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
    public class BitmapData {}
}
```

클래스 특성

ActionScript 3.0에서는 다음 네 가지 특성 중 하나를 사용하여 클래스 정의를 수정할 수 있습니다.

특성	정의
dynamic	런타임에 속성을 인스턴스에 추가할 수 있게 합니다.
final	다른 클래스에서 확장할 수 없습니다.
internal(기본값)	현재 패키지 내에서 참조할 수 있습니다.
public	모든 위치에서 참조할 수 있습니다.

internal의 경우를 제외하고 이러한 각 특성에 연결된 비헤이비어를 사용하려면 특성을 명시적으로 포함합니다. 예를 들어 클래스를 정의할 때 dynamic 특성을 포함하지 않으면 런타임에 속성을 클래스 인스턴스에 추가할 수 없습니다. 특성을 명시적으로 지정하려면 다음 코드와 같이 클래스 정의 시작 부분에 특성을 배치합니다.

```
dynamic class Shape {}
```

abstract라는 특성은 이 목록에 포함되어 있지 않습니다. ActionScript 3.0에서 추상 클래스는 지원되지 않습니다. private 및 protected라는 특성도 목록에 포함되어 있지 않습니다. 이러한 특성은 클래스 정의 내에서만 의미가 있으며 클래스 자체에는 적용할 수 없습니다. 패키지 외부에서 클래스를 공용으로 참조할 수 없게 하려면 클래스를 패키지 내에 배치하고 클래스에 internal 특성을 지정합니다. 또는 internal과 public 특성을 모두 생략하여 컴파일러에서 자동으로 internal 특성을 추가하도록 합니다. 클래스가 정의된 소스 파일 외부에서 해당 클래스를 참조할 수 없도록 클래스를 정의할 수도 있습니다. 이렇게 하려면 클래스를 소스 파일 맨 아래에서 패키지 정의의 닫는 중괄호 밑에 배치합니다.

클래스 본문

클래스 본문은 중괄호로 묶이며 클래스의 변수, 상수 및 메서드를 정의합니다. 다음 예제에서는 ActionScript 3.0의 Accessibility 클래스 선언을 보여 줍니다.

```
public final class Accessibility
{
    public static function get active():Boolean;
    public static function updateProperties():void;
}
```

클래스 본문 내에 네임스페이스를 정의할 수도 있습니다. 다음 예제에서는 클래스 본문 내에 네임스페이스를 정의하여 해당 클래스에서 메서드의 특성으로 사용하는 방법을 보여 줍니다.

```
public class SampleClass
{
    public namespace sampleNamespace;
    sampleNamespace function doSomething():void;
}
```

ActionScript 3.0에서는 클래스 본문 내에 정의뿐만 아니라 명령문도 포함시킬 수 있습니다. 클래스 본문 내에 있지만 메서드 정의 외부에 있는 명령문은 정확히 한 번만 실행됩니다. 이 명령문의 실행은 클래스 정의가 처음 발견되고 관련 클래스 객체가 만들어질 때 이루어집니다. 다음 예제에는 외부 함수인 hello()에 대한 호출과 클래스가 정의될 때 확인 메시지를 출력하는 trace 문이 포함되어 있습니다.

```
function hello():String
{
    trace("hola");
}
class SampleClass
{
    hello();
    trace("class created");
}
// output when class is created
hola
class created
```

ActionScript 3.0에서는 같은 클래스 본문 내에서 이름이 같은 정적 속성과 인스턴스 속성을 정의할 수 있습니다. 예를 들어 다음 코드에서는 message라는 정적 변수 및 이름이 같은 인스턴스 변수를 선언합니다.

```
class StaticTest
{
    static var message:String = "static variable";
    var message:String = "instance variable";
}
// In your script
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // output: static variable
trace(myST.message); // output: instance variable
```

클래스 속성 특성

ActionScript 객체 모델에서 속성이라는 용어는 변수, 상수 및 메서드 등 클래스의 멤버가 될 수 있는 모든 항목을 나타냅니다. 그러나 Adobe ActionScript 3.0 Reference for the Adobe Flash Platform에서 이 용어는 보다 좁은 의미로 사용됩니다. 이 컨텍스트에서 속성이라는 용어는 변수이거나 getter 또는 setter 메서드에 의해 정의되는 클래스 멤버만 나타냅니다. ActionScript 3.0에는 클래스의 모든 속성과 함께 사용할 수 있는 특성 집합이 있습니다. 다음 표에 이러한 특성들이 나열되어 있습니다.

특성	정의
internal(기본값)	같은 패키지 내에서 참조할 수 있습니다.
private	같은 클래스에서 참조할 수 있습니다.
protected	같은 클래스 및 파생 클래스에서 참조할 수 있습니다.
public	모든 위치에서 참조할 수 있습니다.
static	속성이 클래스의 인스턴스가 아니라 클래스에 속하도록 지정합니다.
UserDefinedNamespace	사용자 정의 네임스페이스 이름입니다.

액세스 제어 네임스페이스 특성

ActionScript 3.0에서는 클래스 내에 정의된 속성에 대한 액세스를 제어하는 네 가지 특수 특성인 public, private, protected 및 internal이 제공됩니다.

public 특성을 사용하면 스크립트의 모든 위치에서 속성을 참조할 수 있습니다. 예를 들어 해당 패키지 외부의 코드에서 메서드를 사용할 수 있게 하려면 메서드를 public 특성으로 선언해야 합니다. 이는 var, const 또는 function 등 속성을 정의하는 데 사용된 키워드에 관계없이 모든 속성에서 마찬가지입니다.

private 특성을 사용하면 해당 속성이 정의된 클래스 내에서만 속성에 액세스할 수 있습니다. 이 private 특성의 비헤이비어는 ActionScript 2.0에서 하위 클래스가 수퍼 클래스의 전용 속성에 액세스하도록 허용했던 비헤이비어와는 다릅니다. 또한 런타임 액세스 비헤이비어에도 중요한 차이점이 있습니다. ActionScript 2.0에서는 private 키워드를 사용하면 컴파일 타임에만 액세스가 금지되었고 런타임에는 이러한 제한을 쉽게 피할 수 있었습니다. ActionScript 3.0에서는 더 이상 그렇지 않습니다. private으로 표시된 속성은 컴파일 타임과 런타임 시 모두 사용할 수 없습니다.

예를 들어 다음 코드에서는 전용 변수 하나가 있는 PrivateExample이라는 간단한 클래스를 만든 다음 클래스 외부에서 전용 변수에 액세스합니다.

```
class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // ActionScript 2.0 allows access, but in ActionScript 3.0, this is a run-time
error.
```

ActionScript 3.0의 경우 myExample.privVar과 같이 도트 연산자를 사용하여 전용 속성에 액세스하려고 하면 엄격 모드를 사용하는 경우 컴파일 타임 오류가 발생합니다. 그렇지 않은 경우에는 myExample["privVar"]과 같이 속성 액세스 연산자를 사용한 경우처럼 런타임에 오류가 보고됩니다.

다음 표에서는 동적이 아닌 봉인 클래스에 속한 전용 속성에 액세스한 결과를 보여 줍니다.

	엄격 모드	**표준 모드**
도트 연산자(.)	컴파일 타임 오류	런타임 오류
대괄호 연산자([])	런타임 오류	런타임 오류

dynamic 특성으로 선언된 클래스의 경우에는 전용 변수에 액세스해도 런타임 오류가 발생하지 않습니다. 대신 값을 참조할 수 없으므로 undefined 값이 반환됩니다. 그러나 엄격 모드에서 도트 연산자를 사용하면 컴파일 타임 오류가 발생합니다. 다음 예제는 이전 예제와 동일하지만 PrivateExample 클래스가 동적 클래스로 선언됩니다.

```
dynamic class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // output: undefined
```

동적 클래스를 사용하면 클래스 외부 코드에서 전용 속성에 액세스할 때 일반적으로 오류가 발생하는 대신 undefined 값이 반환됩니다. 다음 표를 보면 엄격 모드에서 도트 연산자를 사용하여 전용 속성에 액세스할 때만 오류가 발생함을 알 수 있습니다.

	엄격 모드	**표준 모드**
도트 연산자(.)	컴파일 타임 오류	undefined
대괄호 연산자([])	undefined	undefined

ActionScript 3.0에서 새로 도입된 protected 특성을 사용하면 자체 클래스나 하위 클래스에서 속성에 액세스할 수 있습니다. 즉, protected 속성은 자체 클래스 내에서 또는 상속 계층 구조에서 해당 클래스 아래쪽에 있는 모든 클래스에서 사용할 수 있습니다. 하위 클래스가 수퍼 클래스와 같은 패키지에 있는지 또는 다른 패키지에 있는지에 관계없이 하위 클래스에서 protected 속성을 사용할 수 있습니다.

ActionScript 2.0에 익숙한 사용자에게 이 기능은 ActionScript 2.0의 private 특성과 비슷하고, ActionScript 3.0의 protected 특성은 Java의 protected 특성과 비슷합니다. 그러나 Java 버전의 경우 같은 패키지 내에 있는 호출자에도 액세스할 수 있다는 차이점이 있습니다. protected 특성은 하위 클래스에 필요한 변수 또는 메서드를 상속 체인 외부에 있는 코드로부터 숨기려는 경우에 유용합니다.

ActionScript 3.0에서 새로 도입된 internal 특성을 사용하면 자체 패키지 내에서 속성을 호출할 수 있습니다. 이는 패키지 안에 있는 코드의 기본 특성이며 다음 특성이 지정되지 않은 모든 속성에 적용됩니다.

- public
- private
- protected
- 사용자 정의 네임스페이스

internal 특성은 Java의 기본 액세스 제어와 비슷하지만 Java의 경우에는 이 액세스 수준에 명시적으로 지정된 이름이 없으며 이 액세스 수준을 사용하려면 다른 액세스 수식어를 모두 생략해야 합니다. ActionScript 3.0에서 internal 특성을 사용하면 자체 패키지 내에서만 속성을 호출하도록 하려는 의도를 명시적으로 나타낼 수 있습니다.

static 특성

var, const 또는 function 키워드로 선언된 속성에 사용할 수 있는 static 특성을 통해 속성을 클래스의 인스턴스가 아닌 클래스 자체에 연결할 수 있습니다. 클래스 외부의 코드에서 정적 속성을 호출하려면 인스턴스 이름 대신 클래스 이름을 사용해야 합니다.

정적 속성은 하위 클래스로 상속되지는 않지만 하위 클래스의 범위 체인에 포함됩니다. 즉, 하위 클래스의 본문 내에서는 정적 변수 또는 메서드가 정의된 클래스를 참조하지 않고도 해당 변수 또는 메서드를 사용할 수 있습니다.

사용자 정의 네임스페이스 특성

미리 정의된 액세스 제어 특성을 사용하는 대신 사용자 정의 네임스페이스를 만들어 특성으로 사용할 수 있습니다. 네임스페이스 특성은 정의마다 하나만 사용할 수 있으며 액세스 제어 특성(public, private, protected, internal)과는 함께 사용할 수 없습니다.

변수

변수는 var 또는 const 키워드로 선언할 수 있습니다. var 키워드로 선언된 변수의 값은 스크립트 실행 중 여러 차례 변경될 수 있습니다. const 키워드로 선언된 변수는 **상수**라고 하며, 값을 한 번만 지정할 수 있습니다. 초기화된 상수에 새 값을 지정하려고 하면 오류가 발생합니다.

정적 변수

정적 변수는 static 키워드를 var 또는 const 문 중 하나와 조합하여 선언합니다. 정적 변수는 클래스의 인스턴스가 아닌 클래스 자체에 연결되며, 객체의 전체 클래스에 적용되는 정보를 저장 및 공유하는 데 유용합니다. 예를 들어 클래스가 인스턴스화된 횟수를 집계하거나 클래스 인스턴스의 최대 개수를 저장하려는 경우에 정적 변수가 적합합니다.

다음 예제에서는 totalCount 변수를 만들어 클래스 인스턴스화 횟수를 추적하고 MAX_NUM 상수를 만들어 인스턴스화 최대 횟수를 저장합니다. totalCount 및 MAX_NUM 변수는 특정 인스턴스가 아닌 전체 클래스에 적용되는 값을 포함하므로 정적 변수입니다.

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

StaticVars 클래스의 외부에 있는 코드나 하위 클래스에 있는 코드에서는 클래스 자체를 통해서만 totalCount 및 MAX_NUM 속성을 참조할 수 있습니다. 예를 들어 다음 코드를 사용해야 합니다.

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

클래스 인스턴스를 통해서는 정적 변수에 액세스할 수 없으므로 다음 코드에서는 오류가 반환됩니다.

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

static과 const 키워드를 모두 사용하여 선언한 변수는 StaticVars 클래스에서 MAX_NUM의 경우와 같이 상수를 선언함과 동시에 초기화해야 합니다. 생성자나 인스턴스 메서드 내에서는 MAX_NUM에 값을 지정할 수 없습니다. 다음 코드에서는 정적 상수를 제대로 초기화하지 않았으므로 오류가 발생합니다.

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

인스턴스 변수

인스턴스 변수에는 static 키워드 없이 var 및 const 키워드로 선언된 속성이 포함됩니다. 인스턴스 변수는 전체 클래스가 아닌 클래스 인스턴스에 연결되며 인스턴스에 고유한 값을 저장하는 데 유용합니다. 예를 들어 Array 클래스에는 Array 클래스의 특정 인스턴스에 들어 있는 배열 요소의 수가 저장되는 length라는 인스턴스 속성이 있습니다.

인스턴스 변수는 var 또는 const 등 선언된 방식에 관계없이 하위 클래스에서 재정의할 수 없습니다. 그러나 getter 및 setter 메서드를 재정의하면 변수를 재정의하는 것과 비슷한 결과를 달성할 수 있습니다.

메서드

메서드는 클래스 정의에 포함되는 함수입니다. 클래스의 인스턴스를 만들면 해당 인스턴스에 메서드가 바인딩됩니다. 클래스 외부에 정의된 함수와 달리 메서드는 자신이 연결된 인스턴스와 분리하여 사용될 수 없습니다.

메서드는 function 키워드를 사용하여 정의됩니다. 모든 클래스 속성과 마찬가지로 private, protected, public, internal, static 등의 클래스 속성 특성이나 사용자 정의 네임스페이스를 메서드에 적용할 수 있습니다. 함수 명령문은 다음과 같이 사용할 수 있습니다.

```
public function sampleFunction():String {}
```

다음과 같이 함수 표현식이 지정된 변수를 사용할 수도 있습니다.

```
public var sampleFunction:Function = function () {}
```

대부분의 경우에는 다음과 같은 이유로 인해 함수 표현식 대신 함수 명령문을 사용합니다.

- 함수 명령문은 보다 간결하고 쉽게 읽을 수 있습니다.
- 함수 명령문을 통해 override 및 final 키워드를 사용할 수 있습니다.
- 함수 명령문을 사용하면 식별자(함수 이름)와 메서드 본문 내의 코드가 보다 견고하게 결합됩니다. 대입문을 사용하면 변수 값을 변경할 수 있으므로 변수와 해당 함수 표현식 간의 연결은 언제든지 끊어질 수 있습니다. 변수를 var 대신 const로 선언하면 이 문제를 해결할 수는 있지만 코드를 읽기 어려워지고 override 및 final 키워드를 사용할 수 없게 되므로 이는 좋은 방법이 아닙니다.

프로토타입 객체에 함수를 연결하려는 경우에는 함수 표현식을 사용해야 합니다.

생성자 메서드

생성자라고도 하는 생성자 메서드는 해당 메서드가 정의된 클래스와 이름이 같은 함수입니다. 생성자 메서드에 포함된 코드는 new 키워드를 사용하여 클래스의 인스턴스를 만들 때마다 실행됩니다. 예를 들어 다음 코드에서는 status라는 속성 하나가 들어 있는 Example이라는 간단한 클래스를 정의합니다. status 변수의 초기값은 생성자 함수 안에서 설정됩니다.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

생성자 메서드는 항상 공용이지만 public 특성은 생략할 수 있습니다. 생성자에서는 private, protected 또는 internal 등의 다른 액세스 제어 지정자를 사용할 수 없습니다. 또한 생성자 메서드에는 사용자 정의 네임스페이스를 사용할 수 없습니다.

생성자에서는 super() 문을 사용하여 바로 위 수퍼 클래스의 생성자를 명시적으로 호출할 수 있습니다. 수퍼 클래스 생성자를 명시적으로 호출하지 않으면 컴파일러에서 생성자 본문의 첫 명령문 앞에 자동으로 호출을 삽입합니다. super 접두어로 수퍼 클래스를 참조하여 수퍼 클래스의 메서드를 호출할 수도 있습니다. 같은 생성자 본문에서 super()와 super를 함께 사용하려는 경우 super()를 먼저 호출해야 합니다. 그렇지 않으면 super 참조가 정상적으로 작동하지 않습니다. 또한 throw 또는 return 문을 사용하기 전에 super() 생성자를 호출해야 합니다.

다음 예제를 통해 super() 생성자를 호출하기 전에 super 참조를 사용한 결과를 확인해 볼 수 있습니다. 새 클래스인 ExampleEx 에서는 Example 클래스를 확장합니다. ExampleEx 생성자에서 super()를 호출하기 전에 수퍼 클래스에 정의된 status 변수에 액세스합니다. 이 경우 super() 생성자가 실행되기 전에는 status 변수를 사용할 수 없으므로 ExampleEx 생성자 내의 trace() 문에서 null 값을 출력합니다.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

생성자 내에서도 return 문을 사용할 수 있지만 값을 반환할 수는 없습니다. 즉, return 문에 표현식이나 값을 연결하면 안 됩니다. 따라서 생성자 메서드는 값을 반환할 수 없으며 반환 유형을 지정할 수 없습니다.

클래스에 생성자 메서드를 정의하지 않으면 컴파일러에서 자동으로 빈 생성자를 만듭니다. 클래스에서 다른 클래스를 확장하는 경우에는 컴파일러에서 자동으로 생성되는 생성자에 super() 호출이 포함됩니다.

정적 메서드

클래스 메서드라고도 하는 정적 메서드는 static 키워드로 선언되는 메서드입니다. 정적 메서드는 클래스 인스턴스가 아닌 클래스 자체에 연결되며, 개별 인스턴스가 아닌 전체적인 상태에 영향을 주는 기능을 캡슐화하는 데 유용합니다. 정적 메서드는 전체 클래스에 연결되므로 클래스의 인스턴스가 아닌 클래스를 통해서만 액세스할 수 있습니다.

정적 메서드는 클래스 인스턴스의 상태에 영향을 주는 데 국한되지 않는 기능을 캡슐화하는 데 유용합니다. 즉, 클래스 인스턴스의 값에 직접 영향을 주지 않는 기능을 제공하는 메서드는 정적 메서드로 선언해야 합니다. 예를 들어 Date 클래스에는 문자열을 받아 숫자로 변환하는 parse()라는 정적 메서드가 있습니다. 이 메서드는 클래스의 개별 인스턴스에 영향을 주지 않으므로 정적입니다. 대신 parse() 메서드는 날짜 값을 나타내는 문자열을 받아 파싱하여 Date 객체의 내부 표현과 호환되는 형식으로 숫자를 반환합니다. 이 메서드는 Date 클래스의 인스턴스에 직접 적용되지 않으므로 인스턴스 메서드가 아닙니다.

정적 parse() 메서드는 Date 클래스에 속하는 getMonth() 등의 인스턴스 메서드와 대조됩니다. getMonth() 메서드는 Date 인스턴스의 월을 나타내는 특정 구성 요소를 검색하여 인스턴스의 값에 직접 작용하므로 인스턴스 메서드입니다.

정적 메서드는 개별 인스턴스에 바인딩되지 않으므로 정적 메서드 본문 내에서는 this 또는 super 키워드를 사용할 수 없습니다. this 참조와 super 참조는 인스턴스 메서드의 컨텍스트 내에서만 의미가 있습니다.

몇 가지 다른 클래스 기반 프로그래밍 언어와 달리 ActionScript 3.0에서는 정적 메서드가 상속되지 않습니다.

인스턴스 메서드

인스턴스 메서드는 static 키워드 없이 선언된 메서드입니다. 인스턴스 메서드는 전체 클래스가 아닌 클래스 인스턴스에 연결되며, 클래스의 개별 인스턴스에 영향을 주는 기능을 구현하는 데 유용합니다. 예를 들어 Array 클래스에는 Array 인스턴스에 직접 작용하는 sort()라는 인스턴스 메서드가 있습니다.

인스턴스 메서드 본문 내에서는 정적 변수와 인스턴스 변수가 모두 범위에 포함되므로 간단한 식별자를 사용하여 같은 클래스에 정의된 해당 변수를 참조할 수 있습니다. 예를 들어 다음 CustomArray 클래스에서는 Array 클래스를 확장합니다. CustomArray 클래스에는 클래스 인스턴스의 총 개수를 추적하는 arrayCountTotal이라는 정적 변수, 인스턴스가 작성된 순서를 추적하는 arrayNumber라는 인스턴스 변수 및 이러한 변수의 값을 반환하는 getPosition()이라는 인스턴스 메서드가 정의됩니다.

```
public class CustomArray extends Array
{
    public static var arrayCountTotal:int = 0;
    public var arrayNumber:int;

    public function CustomArray()
    {
        arrayNumber = ++arrayCountTotal;
    }

    public function getArrayPosition():String
    {
         return ("Array " + arrayNumber + " of " + arrayCountTotal);
    }
}
```

클래스 외부의 코드에서는 CustomArray.arrayCountTotal과 같이 클래스 객체를 통해 정적 변수 arrayCountTotal을 액세스해야 하지만, getPosition() 메서드 본문 내에 있는 코드에서는 정적 변수 arrayCountTotal을 직접 참조할 수 있습니다. 이는 수퍼 클래스에 있는 정적 변수의 경우에도 마찬가지입니다. ActionScript 3.0에서는 정적 속성이 상속되지 않지만 수퍼 클래스의 정적 속성은 범위에 포함됩니다. 예를 들어 Array 클래스에는 DESCENDING이라는 상수를 비롯한 몇 가지 정적 변수가 있습니다. Array의 하위 클래스에 있는 코드에서는 간단한 식별자를 사용하여 정적 상수 DESCENDING에 액세스할 수 있습니다.

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

인스턴스 메서드의 본문 내에서 this 참조의 값은 해당 메서드가 연결된 인스턴스에 대한 참조입니다. 다음 코드에서는 this 참조가 해당 메서드가 포함된 인스턴스를 가리킴을 보여 줍니다.

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

인스턴스 메서드의 상속은 override 및 final 키워드를 통해 제어할 수 있습니다. 상속된 메서드를 다시 정의하려면 override 특성을 사용하고, 하위 클래스에서 메서드를 재정의할 수 없게 하려면 final 특성을 사용합니다.

get 및 set 접근자 메서드

getter 및 **setter**라고도 하는 get 및 set 접근자 함수를 통해 정보 은폐 및 캡슐화라는 프로그래밍 원칙을 준수하면서 클래스에 사용하기 쉬운 프로그래밍 인터페이스를 제공할 수 있습니다. get 및 set 함수를 사용하면 클래스 속성을 클래스 전용으로 유지하면서 클래스 사용자가 클래스 메서드를 호출하는 대신 클래스 변수에 액세스하는 것처럼 이러한 속성에 액세스하도록 허용할 수 있습니다.

이러한 방법을 사용하면 getPropertyName() 및 setPropertyName() 등 이름이 복잡한 기존 접근자 함수를 사용하지 않아도 된다는 장점이 있습니다. 또한 읽기 및 쓰기 액세스가 모두 허용되는 속성마다 두 개의 공용 함수를 만들지 않아도 됩니다.

GetSet이라는 다음 예제 클래스에는 privateProperty라는 전용 변수에 대한 액세스를 제공하는 publicAccess()라는 get 및 set 접근자 함수가 있습니다.

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

다음과 같이 privateProperty 속성에 직접 액세스하려고 하면 오류가 발생합니다.

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

GetSet 클래스 사용자는 속성처럼 보이는 publicAccess라는 항목을 대신 사용하게 되지만, 실제로 이는 privateProperty라는 전용 속성에 작용하는 get 및 set 접근자 함수 쌍입니다. 다음 예제에서는 GetSet 클래스를 인스턴스화한 다음 publicAccess라는 공용 접근자를 사용하여 privateProperty의 값을 설정합니다.

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

getter 및 setter 함수를 사용하여 수퍼 클래스에서 상속된 속성을 재정의할 수도 있습니다. 일반적인 클래스 멤버 변수를 사용할 때는 이러한 작업이 불가능합니다. var 키워드로 선언된 클래스 멤버 변수는 하위 클래스에서 재정의할 수 없습니다. 그러나 getter 및 setter 함수를 사용하여 만든 속성에는 이러한 제한이 없습니다. 수퍼 클래스에서 상속된 getter 및 setter 함수에 override 특성을 사용할 수 있습니다.

바인딩된 메서드

메서드 클로저라고도 하는 바인딩된 메서드는 해당 인스턴스에서 추출된 메서드입니다. 바인딩된 메서드의 예로는 함수에 인수로 전달된 메서드 또는 함수에서 값으로 반환된 메서드가 있습니다. ActionScript 3.0에 새로 도입된 바인딩된 메서드는 인스턴스에서 추출되어도 의미상의 환경이 유지된다는 점에서 함수 클로저와 비슷합니다. 그러나 바인딩된 메서드에서는 this 참조가 해당 메서드를 구현하는 인스턴스에 연결된 상태로 유지된다는 점에서 함수 클로저와 다릅니다. 즉, 바인딩된 메서드의 this 참조는 항상 메서드를 구현한 원래 객체를 가리킵니다. 함수 클로저의 경우에는 this 참조가 일반적이므로 호출 시점에서 함수에 연결된 객체를 가리킵니다.

this 키워드를 사용하는 경우 바인딩된 메서드를 잘 이해해야 합니다. this 키워드는 메서드의 부모 객체를 참조합니다. 대부분의 ActionScript 프로그래머는 this 키워드가 항상 메서드 정의가 들어 있는 객체 또는 클래스를 나타낸다고 예상합니다. 그러나 메서드 바인딩을 사용하지 않으면 그렇지 않을 수도 있습니다. 예를 들어 이전 버전의 ActionScript에서는 this 참조가 메서드를 구현한 인스턴스를 참조하지 않을 수도 있었습니다. ActionScript 2.0의 경우 인스턴스에서 메서드를 추출하면 this 참조가 원래 인

스턴스에 바인딩되지 않을 뿐만 아니라 인스턴스 클래스의 멤버 변수 및 메서드를 사용할 수 없습니다. ActionScript 3.0에서는 메서드를 매개 변수로 전달할 때 바인딩된 메서드가 자동으로 만들어지므로 이러한 문제가 없습니다. 바인딩된 메서드를 사용하면 this 키워드가 항상 메서드가 정의된 객체 또는 클래스를 참조하게 됩니다.

다음 코드에서는 ThisTest라는 클래스를 정의합니다. 이 클래스에는 바인딩된 메서드를 정의하는 foo()라는 메서드 및 바인딩된 메서드를 반환하는 bar()라는 메서드가 있습니다. 클래스 외부에 있는 코드에서는 ThisTest 클래스의 인스턴스를 만들고 bar() 메서드를 호출한 다음 반환 값을 myFunc라는 변수에 저장합니다.

```
class ThisTest
{
    private var num:Number = 3;
    function foo():void // bound method defined
    {
        trace("foo's this: " + this);
        trace("num: " + num);
    }
    function bar():Function
    {
        return foo; // bound method returned
    }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
output: num: 3 */
```

코드의 마지막 두 행을 보면 this 참조는 전역 객체를 가리키지만 바인딩된 메서드인 foo()의 this 참조는 계속 ThisTest 클래스를 가리키는 것을 알 수 있습니다. 또한 myFunc 변수에 저장된 바인딩된 메서드에서는 ThisTest 클래스의 멤버 변수에 계속 액세스할 수 있습니다. 이 코드를 ActionScript 2.0에서 실행하면 this 참조가 서로 일치하게 되고 num 변수는 undefined가 됩니다.

바인딩된 메서드를 사용하는 것이 특히 유용한 부분은 이벤트 핸들러입니다. addEventListener() 메서드에는 함수 또는 메서드를 인수로 전달해야 하기 때문입니다.

클래스와 열거형

열거형은 소규모 값 집합을 캡슐화하기 위해 만드는 사용자 정의 데이터 유형입니다. ActionScript 3.0에서는 C++의 enum 키워드나 Java의 Enumeration 인터페이스와 같은 구체적인 열거형 기능이 지원되지 않습니다. 그러나 클래스와 정적 상수를 사용하여 열거형을 만들 수 있습니다. 예를 들어 ActionScript 3.0의 PrintJob 클래스에서는 다음 코드와 같이 PrintJobOrientation이라는 열거형을 사용하여 값 "landscape"와 "portrait"를 저장합니다.

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

열거형 클래스는 확장할 필요가 없으므로 관습적으로 final 특성으로 선언됩니다. 이러한 클래스는 정적 멤버만 포함하므로 클래스의 인스턴스를 만들 수 없습니다. 대신 다음 인용 코드와 같이 클래스 객체를 직접 사용하여 열거형 값에 액세스합니다.

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

ActionScript 3.0의 모든 열거형 클래스에는 String, int 또는 uint 유형의 변수만 들어 있습니다. 리터럴 문자열이나 숫자 값 대신 열거형을 사용하면 철자를 잘못 입력한 오류를 쉽게 찾을 수 있다는 장점이 있습니다. 열거형의 이름을 잘못 입력하면 ActionScript 컴파일러에서 오류가 발생합니다. 리터럴 값을 사용하면 단어의 철자를 잘못 입력하거나 숫자를 잘못 입력해도 컴파일러에서 메시지가 표시되지 않습니다. 이전 예제의 경우 다음 인용 코드와 같이 열거형 상수 이름을 잘못 입력하면 컴파일러에서 오류가 발생합니다.

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

그러나 다음과 같이 문자열 리터럴 값을 잘못 입력한 경우에는 컴파일러에서 오류가 발생하지 않습니다.

```
    if (pj.orientation == "portrai") // no compiler error
```

열거형을 만드는 두 번째 방법에서도 열거형에 사용할 정적 속성이 있는 별도의 클래스를 만듭니다. 그러나 두 번째 방법은 각 정적 속성에 문자열 또는 정수 값 대신 클래스 인스턴스가 포함된다는 점이 다릅니다. 예를 들어 다음 코드에서는 요일에 대한 열거형 클래스를 만듭니다.

```
public final class Day
{
    public static const MONDAY:Day = new Day();
    public static const TUESDAY:Day = new Day();
    public static const WEDNESDAY:Day = new Day();
    public static const THURSDAY:Day = new Day();
    public static const FRIDAY:Day = new Day();
    public static const SATURDAY:Day = new Day();
    public static const SUNDAY:Day = new Day();
}
```

ActionScript 3.0에는 이 기술이 사용되지 않지만, 이 기술을 통해 제공되는 향상된 유형 확인을 선호하는 많은 개발자는 이 기술을 사용하고 있습니다. 예를 들어 열거형 값을 반환하는 메서드에서는 반환 값을 열거형 데이터 유형으로 제한할 수 있습니다. 다음 코드에서는 요일을 반환하는 함수뿐만 아니라 열거형을 유형 약어로 사용하는 함수 호출도 보여 줍니다.

```
function getDay():Day
{
    var date:Date = new Date();
    var retDay:Day;
    switch (date.day)
    {
        case 0:
            retDay = Day.MONDAY;
            break;
        case 1:
            retDay = Day.TUESDAY;
            break;
        case 2:
            retDay = Day.WEDNESDAY;
            break;
        case 3:
            retDay = Day.THURSDAY;
            break;
        case 4:
            retDay = Day.FRIDAY;
            break;
        case 5:
            retDay = Day.SATURDAY;
            break;
        case 6:
            retDay = Day.SUNDAY;
            break;
    }
    return retDay;
}

var dayOfWeek:Day = getDay();
```

Day 클래스를 개선하여 각 요일을 정수에 연결하고 문자열로 나타낸 요일을 반환하는 toString() 메서드를 제공할 수도 있습니다.

포함된 에셋 클래스

ActionScript 3.0에서는 포함된 에셋 클래스라는 특수 클래스를 사용하여 포함된 에셋을 나타냅니다. 포함된 에셋은 컴파일 타임에 SWF 파일에 포함된 사운드, 이미지 또는 글꼴 등의 에셋입니다. 에셋을 동적으로 로드하지 않고 에셋을 포함하면 런타임에 항상 에셋을 사용할 수 있다는 장점이 있지만 SWF 파일 크기가 증가하는 단점이 있습니다.

Flash Professional에서 포함된 에셋 클래스 사용

에셋을 포함하려면 먼저 FLA 파일의 라이브러리에 에셋을 배치합니다. 그런 다음 에셋의 링크 속성을 사용하여 포함된 에셋 클래스에 이름을 지정합니다. 이 이름을 가진 클래스가 클래스 경로에 없는 경우 사용자가 지정한 이름의 에셋이 자동으로 생성됩니다. 그런 다음 포함된 에셋 클래스의 인스턴스를 만든 후, 이 클래스에 의해 정의되거나 상속된 속성 및 메서드를 사용합니다. 예를 들어 다음과 같은 코드를 사용하면 포함된 에셋 클래스 PianoMusic에 연결된 포함된 사운드를 재생할 수 있습니다.

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

또는 다음에 설명된 대로 [Embed] 메타데이터 태그를 사용하여 Flash Professional 프로젝트에 에셋을 포함할 수 있습니다. 코드에서 [Embed] 메타데이터 태그가 사용된 경우 Flash Professional에서는 Flash Professional 컴파일러 대신 Flex 컴파일러를 사용하여 프로젝트를 컴파일합니다.

Flex 컴파일러를 사용하여 포함된 에셋 클래스 사용

Flex 컴파일러를 사용하여 코드를 컴파일할 경우 ActionScript 코드에 에셋을 포함하려면 [Embed] 메타데이터 태그를 사용합니다. 기본 소스 폴더나 프로젝트의 빌드 경로에 있는 다른 폴더에 에셋을 배치합니다. Flex 컴파일러가 Embed 메타데이터 태그를 발견하면 포함된 에셋 클래스가 만들어집니다. [Embed] 메타데이터 태그 바로 뒤에 선언하는 Class 데이터 유형의 변수를 통해 클래스에 액세스할 수 있습니다. 예를 들어 다음 코드는 sound1.mp3라는 사운드를 포함하고 soundCls 변수를 사용하여 해당 사운드와 연관된 포함 에셋 클래스에 대한 참조를 저장합니다. 그런 다음 이 예제에서는 포함된 에셋 클래스의 인스턴스를 만들고 해당 인스턴스에서 play() 메서드를 호출합니다.

```
package
{
    import flash.display.Sprite;
    import flash.media.SoundChannel;
    import mx.core.SoundAsset;

    public class SoundAssetExample extends Sprite
    {
        [Embed(source="sound1.mp3")]
        public var soundCls:Class;

        public function SoundAssetExample()
        {
            var mySound:SoundAsset = new soundCls() as SoundAsset;
            var sndChannel:SoundChannel = mySound.play();
        }
    }
}
```

Adobe Flash Builder

Flash Builder ActionScript 프로젝트에 [Embed] 메타데이터 태그를 사용하려면 Flex 프레임워크에서 필요한 모든 클래스를 가져와야 합니다. 예를 들어 사운드를 포함하려면 mx.core.SoundAsset 클래스를 가져옵니다. Flex 프레임워크를 사용하려면 ActionScript 빌드 경로에 framework.swc 파일을 포함합니다. 이렇게 하면 SWF 파일의 크기가 증가합니다.

Adobe Flex

또는 Flex에서 MXML 태그 정의에 @Embed() 지시문을 사용하여 에셋을 포함할 수 있습니다.

인터페이스

인터페이스는 관련되지 않은 여러 객체가 서로 통신할 수 있게 하는 메서드 선언을 모아 놓은 것입니다. 예를 들어 ActionScript 3.0에는 클래스에서 이벤트 객체를 처리하는 데 사용할 수 있는 메서드 선언이 들어 있는 IEventDispatcher 인터페이스가 정의되어 있습니다. 여러 객체는 IEventDispatcher 인터페이스를 통해 표준화된 방식으로 이벤트 객체를 서로 전달할 수 있습니다. 다음 코드에서는 IEventDispatcher 인터페이스의 정의를 보여 줍니다.

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
            useCapture:Boolean=false, priority:int=0,
            useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
            useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

인터페이스는 메서드의 인터페이스와 해당 구현이 서로 구분된다는 점을 기반으로 합니다. 메서드의 인터페이스에는 해당 메서드를 호출하는 데 필요한 메서드 이름, 모든 매개 변수 및 반환 유형 등의 모든 정보가 포함됩니다. 메서드 구현에는 인터페이스 정보뿐만 아니라 메서드의 비헤이비어를 수행하는 실행 가능한 명령문도 포함됩니다. 인터페이스 정의에는 메서드 인터페이스만 들어 있으며, 해당 인터페이스를 구현하는 모든 클래스에서는 메서드 구현을 정의해야 합니다.

ActionScript 3.0의 EventDispatcher 클래스에서는 모든 IEventDispatcher 인터페이스 메서드를 정의하고 각 메서드에 본문을 추가하여 IEventDispatcher 인터페이스를 구현합니다. 다음은 EventDispatcher 클래스 정의에서 인용한 코드입니다.

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

IEventDispatcher 인터페이스는 EventDispatcher 인스턴스에서 이벤트 객체를 처리하고 마찬가지로 IEventDispatcher 인터페이스를 구현하는 다른 객체로 이벤트 객체를 전달하는 데 사용되는 프로토콜 역할을 합니다.

또한 인터페이스는 클래스와 마찬가지로 데이터 유형을 정의합니다. 따라서 클래스와 마찬가지로 인터페이스를 유형 약어로 사용할 수 있습니다. 인터페이스는 데이터 유형이므로 데이터 유형이 필요한 is 및 as 등의 연산자와 함께 사용할 수 있습니다. 그러나 클래스와 달리 인터페이스는 인스턴스화할 수 없습니다. 이러한 차이점으로 인해 인터페이스를 추상적인 데이터 유형으로, 클래스를 구체적인 데이터 유형으로 간주하는 프로그래머가 많습니다.

인터페이스 정의

인터페이스 정의의 구조는 클래스 정의의 구조와 비슷하지만 인터페이스에는 본문이 없는 메서드만 포함될 수 있습니다. 인터페이스에는 변수 또는 상수가 포함될 수 없지만 getter 및 setter는 포함될 수 있습니다. 인터페이스를 정의하려면 interface 키워드를 사용합니다. 예를 들어 다음 IExternalizable 인터페이스는 ActionScript 3.0의 flash.utils 패키지에 속합니다. IExternalizable 인터페이스에서는 객체를 직렬화하는 프로토콜을 정의합니다. 직렬화란 객체를 장치에 저장하거나 네트워크를 통해 전송하는 데 적합한 형식으로 변환하는 것입니다.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

IExternalizable 인터페이스는 public 액세스 제어 수정자와 함께 선언되어 있습니다. 인터페이스 정의에는 public 및 internal 액세스 제어 지정자만 사용할 수 있습니다. 인터페이스 정의 내부의 메서드 선언에는 액세스 제어 지정자를 사용할 수 없습니다.

ActionScript 3.0에서는 인터페이스 이름을 대문자 I로 시작하는 규칙을 따르고 있지만, 모든 유효한 식별자를 인터페이스 이름으로 사용할 수 있습니다. 인터페이스 정의는 패키지의 최상위 수준에 배치되는 경우가 많습니다. 클래스 정의 내부나 다른 인터페이스 정의 내부에는 인터페이스 정의를 배치할 수 없습니다.

인터페이스는 하나 이상의 다른 인터페이스를 확장할 수 있습니다. 예를 들어 다음 IExample 인터페이스에서는 IExternalizable 인터페이스를 확장합니다.

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

IExample 인터페이스를 구현하는 모든 클래스에는 extra() 메서드뿐만 아니라 IExternalizable 인터페이스에서 상속된 writeExternal() 및 readExternal() 메서드에 대한 구현도 포함되어야 합니다.

클래스에서 인터페이스 구현

클래스는 인터페이스를 구현할 수 있는 유일한 ActionScript 3.0 언어 요소입니다. 하나 이상의 인터페이스를 구현하려면 클래스 선언에 implements 키워드를 사용합니다. 다음 예제에서는 IAlpha 및 IBeta라는 두 인터페이스를 정의하고, 두 인터페이스를 모두 구현하는 Alpha라는 클래스를 정의합니다.

```
interface IAlpha
{
    function foo(str:String):String;
}

interface IBeta
{
    function bar():void;
}

class Alpha implements IAlpha, IBeta
{
    public function foo(param:String):String {}
    public function bar():void {}
}
```

인터페이스를 구현하는 클래스에서 구현된 메서드는 다음 조건에 맞아야 합니다.

- public 액세스 제어 식별자를 사용해야 합니다.
- 인터페이스 메서드와 같은 이름을 사용해야 합니다.
- 매개 변수 개수가 같아야 하고 각 매개 변수의 데이터 유형이 인터페이스 메서드 매개 변수의 데이터 유형과 일치해야 합니다.
- 같은 반환 유형을 사용해야 합니다.

  ```
  public function foo(param:String):String {}
  ```

그러나 구현하는 메서드의 매개 변수 이름은 융통성 있게 지정할 수 있습니다. 구현된 메서드에서 매개 변수 개수 및 각 매개 변수의 데이터 유형은 인터페이스 메서드와 일치해야 하지만 매개 변수 이름은 일치하지 않아도 됩니다. 예를 들어 이전 예제에서 Alpha.foo() 메서드의 매개 변수 이름은 param입니다.

그러나 IAlpha.foo() 인터페이스 메서드에서는 매개 변수 이름이 str입니다.

```
function foo(str:String):String;
```

매개 변수 기본값도 유연하게 지정할 수 있습니다. 인터페이스 정의에는 매개 변수 기본값이 지정된 함수 선언이 포함될 수 있습니다. 이러한 함수 선언을 구현하는 메서드에는 인터페이스 정의에 지정된 값과 데이터 유형이 같은 매개 변수 기본값이 있어야 하지만 실제 값은 일치하지 않아도 됩니다. 예를 들어 다음 코드에서는 매개 변수 기본값이 3인 메서드가 들어 있는 인터페이스를 정의합니다.

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

다음 클래스 정의에서는 IGamma 인터페이스를 구현하지만 다른 매개 변수 기본값을 사용합니다.

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

이러한 유연성을 발휘할 수 있는 이유는, 인터페이스 구현에 대한 규칙이 데이터 유형의 호환성을 보장하기 위해 설계되었으며 이를 위해 매개 변수 이름과 매개 변수 기본값이 항상 일치할 필요는 없기 때문입니다.

상속

상속은 코드 재사용의 한 형태로서, 프로그래머는 이를 통해 기존 클래스를 기반으로 새 클래스를 개발할 수 있습니다. 기존 클래스는 일반적으로 **기본 클래스** 또는 **수퍼 클래스**라고 하며 새 클래스는 **하위 클래스**라고 합니다. 상속의 가장 중요한 장점은 기존 코드를 그대로 두면서 기본 클래스의 코드를 재사용할 수 있다는 점입니다. 또한 상속을 사용해도 다른 클래스와 기본 클래스가 상호 작용하는 방식은 변경할 필요가 없습니다. 완벽하게 테스트되었거나 이미 사용 중인 기존 클래스를 수정하는 대신 상속을 사용하면 해당 클래스를 통합된 모듈로 취급하면서 속성 또는 메서드를 추가하여 확장할 수 있습니다. 따라서 클래스가 다른 클래스에서 상속됨을 나타내려면 extends 키워드를 사용합니다.

또한 상속을 통해 코드에서 **다형성**의 장점을 활용할 수 있습니다. 다형성이란 다양한 데이터 유형에 이름이 같은 메서드를 적용하여 서로 다른 결과를 얻을 수 있는 기능입니다. 간단한 예제로, Shape라는 기본 클래스와 Circle 및 Square라는 하위 클래스 두 개를 가정해 봅니다. Shape 클래스에는 도형의 넓이를 반환하는 area()라는 메서드가 정의됩니다. 다형성을 구현하면 Circle 유형 객체와 Square 유형 객체를 대상으로 area() 메서드를 호출하여 자동으로 올바른 계산 결과를 얻을 수 있습니다. 상속을 사용하면 하위 클래스에서 기본 클래스의 메서드를 상속하여 다시 정의(**재정의**)할 수 있으므로 다형성을 구현할 수 있습니다. 다음 예제에서는 Circle 및 Square 클래스에 의해 area() 메서드가 다시 정의됩니다.

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

각 클래스는 데이터 유형을 정의하므로 상속을 사용하면 기본 클래스와 이를 확장하는 클래스 사이에 특수한 관계가 성립됩니다. 하위 클래스는 항상 기본 클래스의 모든 속성을 가지게 되므로 하위 클래스의 인스턴스를 항상 기본 클래스의 인스턴스 대신 사용할 수 있습니다. 예를 들어 메서드에 Shape 유형 매개 변수가 정의된 경우 Circle은 Shape를 확장하므로 다음과 같이 Circle 유형 인수를 전달할 수 있습니다.

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

인스턴스 속성과 상속

function, var 또는 const 등 정의에 사용된 키워드에 관계없이 인스턴스 속성은 기본 클래스에서 private 특성으로 선언되어 있지 않으면 모든 하위 클래스에 상속됩니다. 예를 들어 ActionScript 3.0의 Event 클래스에는 모든 이벤트 객체에 공통되는 속성을 상속하는 여러 하위 클래스가 있습니다.

일부 이벤트 유형의 경우에는 Event 클래스에 이벤트를 정의하는 데 필요한 모든 속성이 들어 있습니다. 이러한 이벤트 유형에는 Event 클래스에 정의된 인스턴스 속성 이외에 추가 속성이 필요하지 않습니다. 이러한 이벤트의 예로는 데이터가 성공적으로 로드될 때 발생하는 complete 이벤트 및 네트워크 연결이 설정될 때 발생하는 connect 이벤트가 있습니다.

다음은 Event 클래스에서 인용한 코드로서 하위 클래스로 상속되는 속성 및 메서드 중 일부를 보여 줍니다. 이러한 속성은 상속되므로 모든 하위 클래스의 인스턴스에서 속성에 액세스할 수 있습니다.

```
public class Event
{
    public function get type():String;
    public function get bubbles():Boolean;
    ...

    public function stopPropagation():void {}
    public function stopImmediatePropagation():void {}
    public function preventDefault():void {}
    public function isDefaultPrevented():Boolean {}
    ...
}
```

Event 클래스에 없는 고유 속성이 필요한 이벤트 유형도 있습니다. 이러한 이벤트는 Event 클래스에 정의된 속성에 새 속성을 추가할 수 있도록 Event 클래스의 하위 클래스를 통해 정의됩니다. 이러한 하위 클래스의 예로는 마우스 이동이나 마우스 클릭에 관련된 mouseMove 및 click 등의 이벤트에 고유한 속성을 추가하는 MouseEvent 클래스가 있습니다. 다음은 MouseEvent 클래스에서 인용한 코드로서 기본 클래스에는 없고 하위 클래스에만 있는 속성의 정의를 보여 줍니다.

```
public class MouseEvent extends Event
{
    public static const CLICK:String= "click";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...

    public function get stageX():Number {}
    public function get stageY():Number {}
    ...
}
```

액세스 제어 지정자와 상속

public 키워드로 선언된 속성은 코드의 모든 위치에서 참조할 수 있습니다. 즉, public 키워드는 private, protected 및 internal 키워드와 달리 속성 상속에 제한이 없습니다.

private 키워드로 선언된 속성은 해당 속성이 정의된 클래스 내에서만 참조할 수 있으므로 하위 클래스로 상속되지 않습니다. 이 비헤이비어는 private 키워드가 ActionScript 3.0의 protected 키워드와 비슷하게 작동했던 이전 버전의 ActionScript와 다릅니다.

protected 키워드는 속성이 정의된 클래스뿐만 아니라 모든 하위 클래스에서도 속성을 참조할 수 있음을 나타냅니다. Java 프로그래밍 언어의 protected 키워드와 달리 ActionScript 3.0에서는 protected 키워드를 사용하는 경우 같은 패키지의 다른 모든 클래스에서 해당 속성을 참조할 수 없습니다. ActionScript 3.0에서는 하위 클래스에서만 protected 키워드로 선언된 속성에 액세스할 수 있습니다. 또한 하위 클래스가 기본 클래스와 같은 패키지에 있는지 또는 다른 패키지에 있는지에 관계없이 하위 클래스에서 protected 속성을 참조할 수 있습니다.

속성이 정의된 패키지 내에서만 속성을 참조할 수 있게 하려면 internal 키워드를 사용하거나 액세스 제어 지정자를 사용하지 않습니다. internal 액세스 제어 지정자는 키워드가 지정되지 않은 경우에 적용되는 기본 액세스 제어 지정자입니다. internal로 표시된 속성은 같은 패키지 안에 있는 하위 클래스에만 상속됩니다.

다음 예제를 통해 각 액세스 제어 지정자가 패키지 경계를 벗어나는 상속에 주는 영향을 확인할 수 있습니다. 다음 코드에서는 AccessControl이라는 기본 응용 프로그램 클래스 및 Base와 Extender라는 기타 클래스 두 개를 정의합니다. Base 클래스는 foo라는 패키지에 있고, Base 클래스의 하위 클래스인 Extender 클래스는 bar라는 패키지에 있습니다. AccessControl 클래스에서는 Extender 클래스만 가져와서 Extender의 인스턴스를 만들고, 이 인스턴스에서는 Base 클래스에 정의된 str이라는 변수에 액세스합니다. str 변수는 public으로 선언되어 있으므로 이 코드는 다음과 같이 컴파일 및 실행됩니다.

```
// Base.as in a folder named foo
package foo
{
    public class Base
    {
        public var str:String = "hello"; // change public on this line
    }
}

// Extender.as in a folder named bar
package bar
{
    import foo.Base;
    public class Extender extends Base
    {
        public function getString():String {
            return str;
        }
    }
}

// main application class in file named AccessControl.as
package
{
    import flash.display.MovieClip;
    import bar.Extender;
    public class AccessControl extends MovieClip
    {
        public function AccessControl()
        {
            var myExt:Extender = new Extender();
            trace(myExt.str);// error if str is not public
            trace(myExt.getString()); // error if str is private or internal
        }
    }
}
```

다른 액세스 제어 지정자가 이전 예제의 컴파일 및 실행에 주는 영향을 확인하려면 AccessControl 클래스에서 다음 행을 삭제하거나 주석 처리한 후 str 변수의 액세스 제어 지정자를 private, protected 또는 internal로 변경합니다.

```
trace(myExt.str);// error if str is not public
```

변수는 재정의할 수 없음

var 또는 const 키워드로 선언된 속성은 상속되지만 재정의할 수는 없습니다. 속성을 재정의한다는 것은 하위 클래스에서 속성을 다시 정의함을 의미합니다. 재정의할 수 있는 속성 유형은 function 키워드로 선언된 속성인 get 및 set 접근자뿐입니다. 인스턴스 변수는 재정의할 수 없지만 인스턴스 변수에 대한 getter 및 setter 메서드를 만들고 해당 메서드를 재정의하면 비슷한 기능을 달성할 수 있습니다.

메서드 재정의

메서드를 재정의한다는 것은 상속된 메서드의 비헤이비어를 다시 정의함을 의미합니다. 정적 메서드는 상속되지 않으며 재정의할 수 없습니다. 그러나 인스턴스 메서드는 하위 클래스로 상속되며 다음 두 가지 조건에 맞는 경우 재정의할 수 있습니다.

- 인스턴스 메서드가 기본 클래스에서 final 키워드로 선언되어 있지 않아야 합니다. 인스턴스 메서드에 final 키워드가 사용되면 하위 클래스에서 메서드를 재정의할 수 없습니다.
- 인스턴스 메서드가 기본 클래스에서 private 액세스 제어 지정자로 선언되어 있지 않아야 합니다. 메서드가 기본 클래스에서 private으로 표시된 경우에는 하위 클래스에서 기본 클래스 메서드를 참조할 수 없으므로 이름이 같은 메서드를 정의할 때 override 키워드를 사용할 필요가 없습니다.

  이러한 조건에 맞는 인스턴스 메서드를 재정의하려면 하위 클래스의 메서드 정의에서 override 키워드를 사용해야 하고 메서드의 수퍼 클래스 버전과 다음과 같은 점에서 일치해야 합니다.

- 재정의 메서드의 액세스 제어 수준이 기본 클래스 메서드와 일치해야 합니다. internal로 표시된 메서드의 액세스 제어 수준은 액세스 제어 지정자가 없는 메서드와 같습니다.
- 재정의 메서드의 매개 변수 개수가 기본 클래스 메서드와 일치해야 합니다.
- 재정의 메서드 매개 변수의 데이터 유형 약어가 기본 클래스 메서드의 매개 변수와 일치해야 합니다.
- 재정의 메서드의 반환 유형이 기본 클래스 메서드와 일치해야 합니다.

그러나 재정의 메서드의 매개 변수 이름은 기본 클래스의 매개 변수 이름과 일치하지 않아도 되며, 매개 변수 개수와 각 매개 변수의 데이터 유형만 일치하면 됩니다.

super 문

프로그래머는 메서드를 재정의할 때 기존 비헤이비어를 완전히 대체하기보다는 해당 수퍼 클래스 메서드의 비헤이비어에 기능을 추가하려는 경우가 많습니다. 이렇게 하려면 하위 클래스의 메서드에서 자신의 수퍼 클래스 버전을 호출할 수 있는 메커니즘이 필요합니다. super 문은 바로 위 수퍼 클래스를 참조하므로 이러한 메커니즘을 지원합니다. 다음 예제에서는 thanks()라는 메서드가 포함된 Base라는 클래스를 정의하고, Base 클래스의 하위 클래스이며 thanks() 메서드를 재정의하는 Extender 클래스를 정의합니다. Extender.thanks() 메서드에서는 super 문을 사용하여 Base.thanks()를 호출합니다.

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

getter 및 setter 재정의

수퍼 클래스에 정의된 변수는 재정의할 수 없지만 getter 및 setter는 재정의할 수 있습니다. 예를 들어 다음 코드에서는 ActionScript 3.0의 MovieClip 클래스에 정의되어 있는 currentLabel이라는 getter를 재정의합니다.

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

OverrideExample 클래스 생성자에 있는 trace() 문에서는 Override: null이 산출되며, 이는 currentLabel 속성이 예제에서 재정의되었음을 보여 줍니다.

상속되지 않는 정적 속성

정적 속성은 하위 클래스로 상속되지 않습니다. 즉, 하위 클래스의 인스턴스를 통해서는 정적 속성에 액세스할 수 없습니다. 정적 속성은 해당 속성이 정의된 클래스 객체를 통해서만 액세스할 수 있습니다. 예를 들어 다음 코드에서는 Base라는 기본 클래스와 Base를 확장하는 Extender라는 하위 클래스를 정의합니다. Base 클래스에는 test라는 정적 변수가 정의됩니다. 다음 인용 코드는 엄격 모드의 경우 컴파일되지 않으며 표준 모드의 경우 런타임 오류가 발생합니다.

```
package {
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// error
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base { }
```

정적 변수 test에 액세스하려면 다음 코드와 같이 클래스 객체를 사용해야 합니다.

```
Base.test;
```

그러나 정적 속성과 이름이 같은 인스턴스 속성을 정의할 수는 있습니다. 이러한 인스턴스 속성은 동일한 클래스에 정적 속성과 함께 정의하거나 하위 클래스에 정의할 수 있습니다. 예를 들어 이전 예제의 Base 클래스에 test라는 인스턴스 속성이 있을 수 있습니다. 다음 코드에서는 인스턴스 속성이 Extender 클래스로 상속되므로 코드가 정상적으로 컴파일 및 실행됩니다. test 인스턴스 변수의 정의를 잘라내어 Extender 클래스에 붙여 넣어도 코드가 정상적으로 컴파일 및 실행됩니다.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

정적 속성과 범위 체인

정적 속성은 상속되지 않지만, 해당 속성이 정의된 클래스 및 해당 클래스에 대한 모든 하위 클래스의 범위 체인에 포함됩니다. 따라서 정적 속성은 자신이 정의된 클래스 및 모든 하위 클래스의 범위 내에 있습니다. 즉, 정적 속성이 정의된 클래스 본문 및 해당 클래스의 모든 하위 클래스 내에서 정적 속성에 직접 액세스할 수 있습니다.

다음 예제에서는 이전 예제에서 정의한 클래스를 수정하여 Base 클래스에 정의된 정적 변수 test가 Extender 클래스의 범위 내에 있음을 보여 줍니다. 즉, Extender 클래스에서는 test가 정의된 클래스의 이름을 접두어로 사용하지 않고도 정적 변수 test에 액세스할 수 있습니다.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base
{
    public function Extender()
    {
        trace(test); // output: static
    }

}
```

같은 클래스나 하위 클래스에 정적 속성과 이름이 같은 인스턴스 속성이 정의된 경우 범위 체인에서 인스턴스 속성의 우선 순위가 더 높습니다. 따라서 인스턴스 속성이 정적 속성을 가리게 되어 정적 속성 값 대신 인스턴스 속성 값이 사용됩니다. 예를 들어 다음 코드에서는 Extender 클래스에 test라는 인스턴스 변수가 정의되면 trace() 문에 정적 변수 값 대신 인스턴스 변수 값이 사용됨을 보여 줍니다.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base
{
    public static var test:String = "static";
}

class Extender extends Base
{
    public var test:String = "instance";
    public function Extender()
    {
        trace(test); // output: instance
    }

}
```

고급 항목

ActionScript OOP 지원 이력

ActionScript 3.0은 이전 버전의 ActionScript를 기초로 제작되었으므로 ActionScript 객체 모델의 발전 과정을 이해하면 도움이 됩니다. ActionScript는 초기 버전의 Flash Professional을 위한 간단한 스크립트 메커니즘으로 처음 개발되었습니다. 이후 프로그래머들은 ActionScript로 점차 복잡한 응용 프로그램을 작성하기 시작했습니다. 이러한 프로그래머의 수요에 대응하기 위해 이후 릴리스에서는 복잡한 응용 프로그램을 만드는 데 필요한 언어 기능이 계속 추가되었습니다.

ActionScript 1.0

ActionScript 1.0은 Flash Player 6 및 이전 버전에서 사용된 언어 버전입니다. 이러한 초기 개발 단계에서도 ActionScript 객체 모델은 기초적인 데이터 유형으로서 객체라는 개념에 기반을 두었습니다. ActionScript 객체는 속성 그룹이 있는 복합 데이터 유형입니다. 객체 모델에서 속성이라는 용어에는 객체에 연결된 변수, 함수 또는 메서드 등의 모든 항목이 포함됩니다.

이러한 1세대 ActionScript에서는 class 키워드를 통한 클래스 정의가 지원되지 않지만 프로토타입 객체라는 특수한 객체를 사용하여 클래스를 정의할 수 있습니다. Java 및 C++ 등의 클래스 기반 언어와 같이 class 키워드를 사용하여 구체적인 객체로 인스턴스화하는 추상적인 클래스 정의를 만드는 대신, ActionScript 1.0과 같은 프로토타입 기반 언어에서는 기존 객체를 모델(프로토타입)로 사용하여 다른 객체를 만듭니다. 클래스 기반 언어의 객체는 자신의 템플릿 역할을 하는 클래스를 가리킬 수 있지만, 프로토타입 기반 언어의 객체는 자신의 템플릿 역할을 하는 프로토타입이라는 다른 객체를 대신 가리킵니다.

ActionScript 1.0에서 클래스를 만들려면 해당 클래스의 생성자 함수를 정의해야 합니다. ActionScript에서는 함수가 추상적인 정의가 아닌 실제 객체입니다. 작성된 생성자 함수는 해당 클래스 인스턴스의 프로토타입 객체 역할을 합니다. 다음 코드에서는 Shape라는 클래스를 만들고 기본적으로 true로 설정되는 visible이라는 속성 하나를 정의합니다.

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

이 생성자 함수는 다음과 같이 new 연산자로 인스턴스화할 수 있는 Shape 클래스를 정의합니다.

```
myShape = new Shape();
```

Shape() 생성자 함수는 Shape 클래스 인스턴스의 프로토타입 역할을 할 뿐만 아니라 Shape의 하위 클래스(Shape 클래스를 확장하는 다른 클래스)의 프로토타입 역할도 합니다.

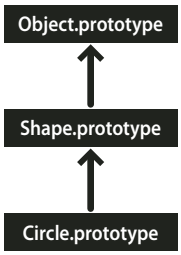
Shape 클래스의 하위 클래스를 만들려면 두 단계를 거쳐야 합니다. 우선 다음과 같이 클래스의 생성자 함수를 정의하여 클래스를 만듭니다.

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

다음으로 new 연산자를 사용하여 Shape 클래스가 Circle 클래스의 프로토타입임을 선언합니다. 기본적으로 작성되는 모든 클래스에는 Object 클래스가 프로토타입으로 사용됩니다. 즉, Circle.prototype에는 현재 일반 객체(Object 클래스의 인스턴스)가 들어 있습니다. Circle의 프로토타입을 Object 대신 Shape로 지정하려면 다음 코드를 사용하여 Circle.prototype에 일반 객체 대신 Shape 객체가 포함되도록 값을 변경합니다.

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

Shape 클래스와 Circle 클래스는 이제 상속 관계로 연결되었으며 이러한 관계를 일반적으로 **프로토타입 체인**이라고 합니다. 다음 그림은 프로토타입 체인 관계를 보여 줍니다.



모든 프로토타입 체인 끝에 있는 기본 클래스는 Object 클래스입니다. Object 클래스에는 Object.prototype이라는 정적 속성이 들어 있으며 이 속성은 ActionScript 1.0에서 만드는 모든 객체의 기본 프로토타입 객체를 가리킵니다. 예제 프로토타입 체인에서 다음에 있는 객체는 Shape 객체입니다. 이는 Shape.prototype 속성을 명시적으로 설정하지 않았으므로 일반 객체(Object 클래스의 인스턴스)가 계속 들어 있기 때문입니다. 이 체인의 마지막 링크는 해당 프로토타입인 Shape 클래스에 연결되어 있는 Circle 클래스입니다. Circle.prototype 속성에는 Shape 객체가 들어 있습니다.

다음 예제와 같이 Circle 클래스의 인스턴스를 만들면 Circle 클래스의 프로토타입 체인이 인스턴스로 상속됩니다.

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

앞의 예제에서 Shape 클래스의 멤버로 visible이라는 속성을 포함했습니다. 이 예제에서는 visible 속성이 myCircle 객체에 포함되어 있지 않고 Shape 객체의 멤버일 뿐이지만, 다음 코드 행을 실행하면 true가 산출됩니다.

```
trace(myCircle.visible); // output: true
```

런타임은 프로토타입 체인을 확인하여 myCircle 객체에 visible 속성이 상속됨을 인식할 수 있습니다. 이 코드를 실행하면 런타임은 우선 myCircle 객체의 속성 중에서 이름이 visible인 속성을 검색하지만 이러한 속성을 찾을 수 없습니다. 다음으로 Circle.prototype 객체에서 속성을 검색하지만 visible이라는 속성을 여전히 찾을 수 없습니다. 런타임은 프로토타입 체인을 거슬러 올라가면서 Shape.prototype 객체에 정의된 visible 속성을 찾게 되고 이 속성의 값을 출력합니다.

이 설명서에는 이해를 돕기 위해 프로토타입 체인과 관련된 자세한 세부 내용이 상당 부분 생략되어 있습니다. 대신 ActionScript 3.0 객체 모델을 이해하는 데 필요한 수준의 정보만을 제공합니다.

ActionScript2.0

ActionScript 2.0에는 class, extends, public 및 private 등의 키워드가 새로 도입되었으며, 이를 통해 Java 및 C++ 등의 클래스 기반 언어를 사용하는 작업자에게 익숙한 방식으로 클래스를 정의할 수 있습니다. 단, ActionScript 1.0과 ActionScript 2.0 사이에서 기본 상속 메커니즘은 변경되지 않았습니다. ActionScript 2.0에는 클래스를 정의하는 구문이 새로 추가되었을 뿐입니다. 프로토타입 체인은 두 가지 언어 버전에서 동일한 방식으로 작동합니다.

다음 인용 코드와 같이 ActionScript 2.0에 새로 도입된 구문을 통해 많은 프로그래머에게 친숙한 방식으로 클래스를 정의할 수 있습니다.

```
// base class
class Shape
{
var visible:Boolean = true;
}
```

ActionScript 2.0에는 컴파일 타임 유형 확인에 사용되는 유형 약어도 도입되었습니다. 이를 통해 이전 예제의 visible 속성에 부울 값만 포함되도록 선언할 수 있습니다. 또한 새로운 extends 키워드를 통해 하위 클래스를 간편하게 만들 수 있습니다. 다음 예제에서는 ActionScript 1.0의 경우 두 단계가 필요한 작업을 extends 키워드를 사용하여 한 단계로 수행합니다.

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

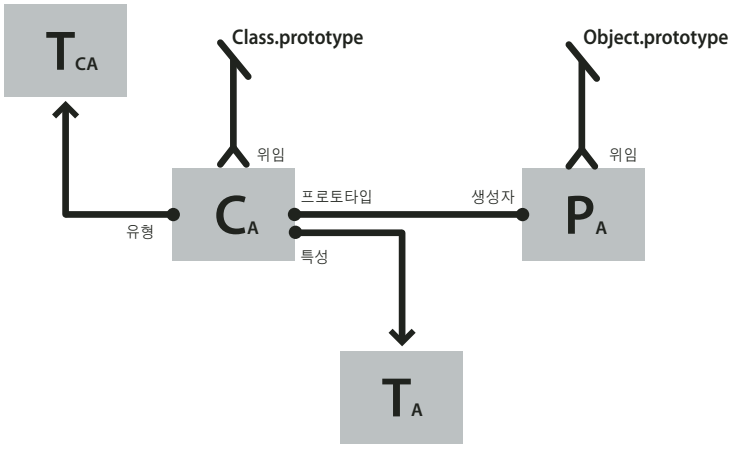
생성자는 이제 클래스 정의의 일부로 선언되며 클래스 속성인 id 및 radius도 명시적으로 선언해야 합니다.

ActionScript 2.0에는 인터페이스를 정의하는 기능이 추가되었으며, 이를 통해 객체 간 통신을 위해 정식으로 정의된 프로토콜을 사용하여 객체 지향 프로그램을 개선할 수 있습니다.

ActionScript 3.0 클래스 객체

일반적으로 Java 및 C++와 관련된 객체 지향 프로그래밍 패러다임에서는 클래스를 사용하여 객체 유형을 정의합니다. 또한 이러한 패러다임을 채택한 프로그래밍 언어에서는 클래스를 사용하여 클래스가 정의하는 데이터 유형의 인스턴스를 만듭니다. ActionScript에서도 이러한 두 가지 용도로 클래스를 사용하지만, ActionScript는 프로토타입 기반 언어로 시작되었으므로 독특한 특성이 있습니다. ActionScript에서는 각 클래스 정의마다 비헤이비어와 상태를 공유할 수 있는 특수 클래스 객체를 만듭니다. 그러나 대부분의 ActionScript 프로그래머는 이 특징으로 인해 코딩에 실질적인 영향을 받지는 않습니다. ActionScript 3.0은 이러한 특수 클래스 객체를 사용하거나 이해하지 않고도 복잡한 객체 지향 ActionScript 응용 프로그램을 만들 수 있도록 설계되었습니다.

다음 그림에는 class A {} 문으로 정의된 A라는 간단한 클래스를 나타내는 클래스 객체의 구조가 표현되어 있습니다.



그림의 각 사각형은 객체를 나타냅니다. 그림의 각 객체에는 A라는 아래 첨자가 있으며 이 문자는 클래스 A에 속함을 나타냅니다. 클래스 객체(CA)에는 중요한 몇 가지 다른 객체를 가리키는 참조가 들어 있습니다. 인스턴스 traits 객체(TA)에는 클래스 정의 내에 정의된 인스턴스 속성이 저장됩니다. 클래스 traits 객체(TCA)는 클래스의 내부 유형을 나타내며 이 객체에는 클래스에 정의된 정적 속성이 저장됩니다. 여기에서 아래 첨자 C는 "클래스"를 나타냅니다. 프로토타입 객체(PA)는 항상 constructor 속성을 통해 자신이 원래 연결된 클래스 객체를 참조합니다.

traits 객체

ActionScript 3.0에 새로 도입된 traits 객체는 성능을 고려하여 구현되었습니다. 이전 버전의 ActionScript에서는 이름을 조회할 때 Flash Player에서 프로토타입 체인을 확인했으므로 시간이 오래 걸렸습니다. ActionScript 3.0에서는 상속된 속성이 수퍼클래스에서 하위 클래스의 traits 객체로 복사되므로 이름을 훨씬 빠르고 효율적으로 조회할 수 있습니다.

traits 객체는 프로그래머 코드에서 직접 액세스할 수 없지만 성능 및 메모리 사용을 개선해 줍니다. traits 객체는 AVM2에 클래스의 레이아웃과 내용에 대한 자세한 정보를 제공합니다. AVM2에서는 이러한 정보를 바탕으로 시간이 걸리는 이름 조회를 수행하지 않고 직접 속성에 액세스하거나 메서드를 호출하는 기계어 명령을 생성할 수 있으므로 실행 시간이 크게 줄어듭니다.

traits 객체가 도입됨에 따라 객체가 점유하는 메모리가 이전 버전의 ActionScript보다 크게 줄어듭니다. 예를 들어 클래스가 동적으로 선언되지 않아 봉인된 경우에는 클래스의 인스턴스에 동적으로 추가된 속성을 위한 해시 테이블이 필요하지 않으므로, 인스턴스에는 traits 객체에 대한 포인터 및 클래스에 정의된 고정 속성을 위한 슬롯 몇 개만 있으면 됩니다. 따라서 ActionScript 2.0에서는 100바이트의 메모리가 필요한 객체가 ActionScript 3.0에서는 20바이트만 필요할 수도 있습니다.

참고: traits 객체는 내부적으로 구현되는 세부 사항이며, 이후 버전의 ActionScript에서는 변경되거나 없어질 수도 있습니다.

프로토타입 객체

모든 ActionScript 클래스 객체에는 클래스의 프로토타입 객체를 참조하는 prototype이라는 속성이 있습니다. 프로토타입 객체는 ActionScript가 프로토타입 기반 언어로 시작되었기 때문에 남아 있는 항목입니다. 자세한 내용은 ActionScript OOP 지원 이력을 참조하십시오.

prototype 속성은 읽기 전용이므로 다른 객체를 가리키도록 수정할 수 없습니다. 이는 ActionScript 이전 버전의 클래스 prototype 속성과 다른 점입니다. 이전 버전에서는 다른 클래스를 가리키도록 프로토타입을 다시 지정할 수 있었습니다. prototype 속성은 읽기 전용이지만, 이 속성이 참조하는 프로토타입 객체는 그렇지 않습니다. 즉, 프로토타입 객체에 새 속성을 추가할 수 있습니다. 프로토타입 객체에 추가된 속성은 클래스의 모든 인스턴스 사이에 공유됩니다.

이전 버전의 ActionScript에서 유일한 상속 메커니즘인 프로토타입 체인은 ActionScript 3.0에서 보조적인 역할만 합니다. 기본 상속 메커니즘인 고정 속성 상속은 traits 객체에 의해 내부적으로 처리됩니다. 고정된 속성은 클래스 정의의 일부로 정의된 변수 또는 메서드입니다. 고정 속성 상속은 class, extends 및 override 등의 키워드와 관련된 상속 메커니즘이므로 클래스 상속이라고도 합니다.

프로토타입 체인은 고정 속성 상속 대신 사용할 수 있는 더욱 동적인 상속 메커니즘을 제공합니다. 클래스를 정의할 때뿐만 아니라 런타임에도 클래스 객체의 prototype 속성을 통해 클래스의 프로토타입 객체에 속성을 추가할 수 있습니다. 그러나 컴파일러가 엄격 모드로 설정되어 있으면 클래스를 dynamic 키워드로 선언하지 않은 경우에는 프로토타입 객체에 추가된 속성에 액세스하지 못할 수도 있습니다.

Object 클래스는 프로토타입 객체에 연결된 여러 속성을 가진 클래스의 전형입니다. Object 클래스의 toString() 및 valueOf() 메서드는 실제로는 Object 클래스의 프로토타입 객체의 속성에 지정된 함수입니다. 다음 예제에서는 이러한 메서드가 어떻게 선언될 수 있는지를 이론적 관점에서 보여 줍니다. 그러나 실제 구현은 세부 구현 사항에 따라 다릅니다.

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

앞에서 언급했듯이 클래스 정의 외부에 있는 클래스의 프로토타입 객체에 속성을 연결할 수 있습니다. 예를 들어 toString() 메서드를 다음과 같이 Object 클래스 정의 외부에 정의할 수도 있습니다.

```
Object.prototype.toString = function()
{
    // statements
};
```

그러나 고정 속성 상속과는 달리 프로토타입 상속에서는 하위 클래스에서 메서드를 재정의하는 경우 override 키워드가 필요하지 않습니다. 예를 들어 Object 클래스의 하위 클래스에서 valueOf() 메서드를 재정의하려는 경우 세 가지 옵션 중 하나를 선택할 수 있습니다. 첫 번째 옵션은 클래스 정의 내에 있는 하위 클래스의 프로토타입 객체에 valueOf() 메서드를 정의하는 것입니다. 다음 코드에서는 Object의 하위 클래스 Foo를 만들고 클래스 정의의 일부로서 Foo의 프로토타입 객체에 valueOf() 메서드를 재정의합니다. 모든 클래스는 Object에서 상속되므로 extends 키워드는 사용하지 않아도 됩니다.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

두 번째 옵션은 다음 코드에 표시된 대로 클래스 정의 외부에 있는 Foo의 프로토타입 객체에 valueOf() 메서드를 정의하는 것입니다.

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

세 번째 옵션은 valueOf()라는 이름의 고정 속성을 Foo 클래스의 일부로서 정의하는 것입니다. 이 기법은 고정 속성 상속과 프로토타입 상속을 혼합한다는 점에서 다른 두 옵션과 차별화됩니다. Foo의 하위 클래스에서 valueOf() 메서드를 재정의할 때는 항상 override 키워드를 사용해야 합니다. 다음 코드에서는 Foo에 고정 속성으로 정의된 valueOf() 메서드를 보여 줍니다.

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

AS3 네임스페이스

고정 속성 상속과 프로토타입 상속이라는 별개의 상속 메커니즘이 공존함으로 인해 기본 클래스의 속성 및 메서드와 관련하여 흥미로운 호환성 문제가 발생합니다. ActionScript의 기반이 되는 ECMAScript 언어 사양과의 호환성을 위해서는 프로토타입 상속을 사용해야 합니다. 즉, 기본 클래스의 속성 및 메서드가 해당 클래스의 프로토타입 객체에 정의되어 있어야 합니다. 반면에 ActionScript 3.0과의 호환성을 위해서는 고정 속성 상속을 사용해야 합니다. 즉, 기본 클래스의 속성 및 메서드가 const, var 및 function 키워드를 통해 해당 클래스 정의에 정의되어 있어야 합니다. 프로토타입 버전 대신 고정 속성을 사용하면 런타임 성능이 크게 향상되는 장점도 있습니다.

ActionScript 3.0에서는 기본 클래스에 대해 프로토타입 상속 및 고정 속성 상속을 모두 사용하여 이 문제를 해결합니다. 각 기본 클래스에는 속성 및 메서드 집합이 두 개씩 있습니다. 한 집합은 ECMAScript 사양과의 호환성을 위해 프로토타입 객체에 정의되어 있고 다른 집합은 ActionScript 3.0과의 호환성을 위해 고정 속성 및 AS3 네임스페이스를 사용하여 정의되어 있습니다.

AS3 네임스페이스는 두 개의 속성 및 메서드 집합 중 하나를 선택할 때 사용할 수 있는 편리한 메커니즘을 제공합니다. AS3 네임스페이스를 사용하지 않으면 기본 클래스의 인스턴스에서는 기본 클래스의 프로토타입 객체에 정의된 속성 및 메서드를 상속합니다. AS3 네임스페이스를 사용하면 항상 프로토타입 속성 대신 고정 속성이 선택되므로 기본 클래스의 인스턴스에서는 AS3 버전을 상속합니다. 즉, 고정 속성을 사용할 수 있는 경우에는 언제나 동일한 이름의 프로토타입 속성 대신 고정 속성이 사용됩니다.

속성 또는 메서드를 선택하여 AS3 네임스페이스로 정규화하면 해당 속성 또는 메서드의 AS3 네임스페이스 버전을 사용할 수 있습니다. 예를 들어 다음 코드에서는 Array.pop() 메서드의 AS3 버전이 사용됩니다.

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

또는 use namespace 지시문을 사용하여 코드 블록 내의 모든 정의에 대한 AS3 네임스페이스를 열 수 있습니다. 예를 들어 다음 코드에서는 use namespace 지시문을 사용하여 pop() 및 push() 메서드에 대한 AS3 네임스페이스를 엽니다.

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0에서는 각 속성 집합에 대한 컴파일러 옵션을 제공하여 전체 프로그램에 AS3 네임스페이스를 적용할 수 있도록 합니다. -as3 컴파일러 옵션은 AS3 네임스페이스를 나타내며 -es 컴파일러 옵션은 프로토타입 상속 옵션을 나타냅니다. 여기서 es는 ECMAScript를 의미합니다. 전체 프로그램에 대해 AS3 네임스페이스를 열려면 -as3 컴파일러 옵션을 true로 설정하고 -es 컴파일러 옵션을 false로 설정합니다. 프로토타입 버전을 사용하려면 컴파일러 옵션의 값을 반대로 설정하면 됩니다. Flash Builder 및 Adobe Flash Professional에서 기본 컴파일러 설정은 -as3 = true 및 -es = false입니다.

기본 클래스를 확장하고 메서드를 재정의하려면 재정의된 메서드를 선언하는 방법에 AS3 네임스페이스가 어떻게 영향을 미칠 수 있는지 이해해야 합니다. AS3 네임스페이스를 사용하고 있는 경우에 기본 클래스의 메서드를 재정의하려면 override 특성과 함께 AS3 네임스페이스를 사용해야 합니다. AS3 네임스페이스를 사용하지 않는 상태에서 기본 클래스 메서드를 하위 클래스에서 재정의하려면 AS3 네임스페이스나 override 키워드를 사용해서는 안됩니다.

예제: GeometricShapes

GeometricShapes 샘플 응용 프로그램에서는 ActionScript 3.0을 사용하여 얼마나 많은 객체 지향 개념과 기능을 적용할 수 있는지 보여 줍니다.

- 클래스 정의
- 클래스 확장
- 다형성 및 override 키워드
- 정의, 확장 및 인터페이스 구현

이 예제에는 클래스 인스턴스를 생성하는 "팩토리 메서드"가 나와 있으며 반환값을 인터페이스의 인스턴스로 선언하고 반환된 객체를 일반적인 방식으로 사용하는 방법도 보여 줍니다.

이 샘플에 대한 응용 프로그램 파일을 가져오려면 www.adobe.com/go/learn_programmingAS3samples_flash_kr를 참조하십시오. GeometricShapes 응용 프로그램 파일은 Samples/GeometricShapes 폴더에서 찾을 수 있습니다. 이 응용 프로그램은 다음과 같은 파일로 구성됩니다.

파일	설명
GeometricShapes.mxml 또는 GeometricShapes.fla	Flash(FLA) 또는 Flex(MXML) 형식의 기본 응용 프로그램 파일입니다.
com/example/programmingas3/geometricshapes/IGeometricShape.as	모든 기하학적 모양 클래스에서 구현될 메서드를 정의하는 기본 인터페이스입니다.
com/example/programmingas3/geometricshapes/IPolygon.as	변이 여러 개인 GeometricShapes 응용 프로그램 클래스에서 구현될 메서드를 정의하는 인터페이스입니다.
com/example/programmingas3/geometricshapes/RegularPolygon.as	길이가 같은 여러 변이 중심을 기준으로 대칭을 이루는 기하학적 형태 유형입니다.
com/example/programmingas3/geometricshapes/Circle.as	원을 정의하는 기하학적 모양 유형입니다.

파일	설명
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	모든 변의 길이가 같은 삼각형을 정의하는 RegularPolygon의 하위 클래스입니다.
com/example/programmingas3/geometricshapes/Square.as	네 변의 길이가 모두 같은 사각형을 정의하는 RegularPolygon의 하위 클래스입니다.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	지정된 모양 유형과 크기로 모양을 만드는 factory 메서드가 포함된 클래스입니다.

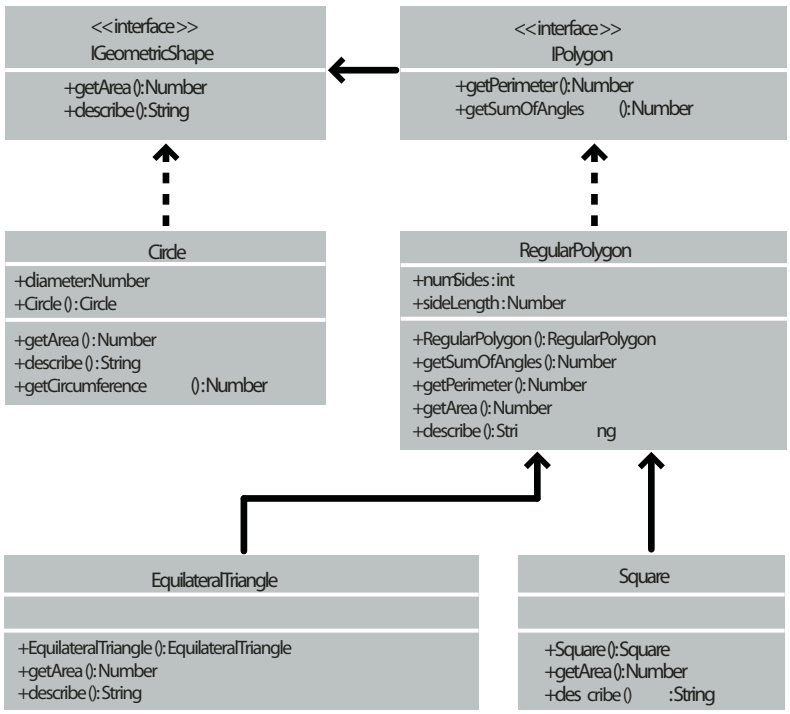
GeometricShapes 클래스 정의

GeometricShapes 응용 프로그램에서는 사용자가 기하학적 형태의 유형 및 크기를 지정하도록 합니다. 그런 다음 사용자에게 형태에 대해 설명하고 면적 및 둘레를 알려 줍니다.

이 응용 프로그램의 사용자 인터페이스는 간단하게 구성되어 있습니다. 형태의 유형을 선택하고 크기를 설정하고 설명을 표시하는 데 사용하는 컨트롤 몇 개로 구성됩니다. 이 응용 프로그램에서 가장 흥미로운 부분은 표면에 드러나지 않는 클래스 및 인터페이스의 구조에 있습니다.

이 응용 프로그램에서는 기하학적 형태를 다루지만 그래픽으로 표시하지는 않습니다.

이 예제에서 기하학적 형태를 정의하는 클래스 및 인터페이스는 UML(Unified Modeling Language)을 사용하여 다음 다이어그램과 같이 나타낼 수 있습니다.



GeometricShapes 예제 클래스

인터페이스를 사용하여 공통 비헤이비어 정의

이 GeometricShapes 응용 프로그램에서는 세 가지 유형의 형태, 즉 원, 사각형 및 등변 삼각형을 다룹니다. GeometricShapes의 클래스 구조는 매우 작은 인터페이스인 IGeometricShape에서 시작됩니다. 이 인터페이스에는 세 가지 유형의 형태 모두에 공통적인 메서드가 나열되어 있습니다.

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

이 인터페이스에는 두 개의 메서드가 정의되어 있습니다. getArea() 메서드는 형태의 면적을 계산하여 반환하며 describe() 메서드는 문자열을 결합하여 형태의 속성을 설명하는 텍스트를 만듭니다.

각 모양의 둘레도 확인하고 싶은 경우를 가정해 봅니다. 원의 둘레는 원주라고 하며 고유한 계산 방법이 있으므로 비헤이비어가 삼각형 또는 사각형과 달라지게 됩니다. 그러나 삼각형, 사각형 및 기타 다각형 간에는 충분한 유사성이 있으므로 이들을 위한 새로운 클래스 인터페이스, 즉 IPolygon을 정의하는 것은 의미가 있습니다. 또한 IPolygon 인터페이스는 여기에 나와 있듯이 상당히 간단하게 정의됩니다.

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

이 인터페이스에는 모든 다각형에 공통적인 두 개의 메서드가 정의되어 있습니다. getPerimeter() 메서드는 모든 변의 길이를 합산하여 측정하며 getSumOfAngles() 메서드는 내부 각도를 모두 더합니다.

IPolygon 인터페이스는 IGeometricShape 인터페이스를 확장하므로 IPolygon 인터페이스를 구현하는 모든 클래스에서는 IGeometricShape 인터페이스와 IPolygon 인터페이스에 두 개씩 정의되어 있는 네 개의 메서드를 모두 선언해야 합니다.

형태 클래스 정의

형태 유형별로 공통적인 메서드가 발견되면 각각의 형태 클래스를 정의할 수 있습니다. 구현해야 하는 메서드의 수를 기준으로 할 때 가장 간단한 형태는 다음에 나와 있듯이 Circle 클래스입니다.

```
package com.example.programmingas3.geometricshapes
{
    public class Circle implements IGeometricShape
    {
        public var diameter:Number;

        public function Circle(diam:Number = 100):void
        {
            this.diameter = diam;
        }

        public function getArea():Number
        {
            // The formula is Pi * radius * radius.
            var radius:Number = diameter / 2;
            return Math.PI * radius * radius;
        }

        public function getCircumference():Number
        {
            // The formula is Pi * diameter.
            return Math.PI * diameter;
        }

        public function describe():String
        {
            var desc:String = "This shape is a Circle.\n";
            desc += "Its diameter is " + diameter + " pixels.\n";
            desc += "Its area is " + getArea() + ".\n";
            desc += "Its circumference is " + getCircumference() + ".\n";
            return desc;
        }
    }
}
```

Circle 클래스는 IGeometricShape 인터페이스를 구현하므로 getArea() 메서드 및 describe() 메서드에 대한 코드가 작성되어야 합니다. 또한 이 클래스에는 getCircumference()라는 고유한 메서드가 정의되어 있습니다. Circle 클래스에는 다른 다각형 클래스에서는 찾을 수 없는 diameter라는 속성도 선언되어 있습니다.

다른 두 유형의 형태, 즉 정사각형과 등변 삼각형은 몇 가지 추가적인 공통점이 있습니다. 예를 들어 두 형태 모두 변의 길이가 동일하며 일반적인 공식을 사용하여 둘레 및 내부 각도의 합을 계산할 수 있습니다. 사실 이러한 일반적인 공식은 앞으로 정의할 다른 모든 일반 다각형에도 적용됩니다.

RegularPolygon 클래스는 이후에 Square 클래스 및 EquilateralTriangle 클래스의 수퍼 클래스가 됩니다. 수퍼 클래스를 사용하면 한 곳에 공통적인 메서드를 정의할 수 있으므로 이러한 메서드를 각 하위 클래스에서 별도로 정의하지 않아도 됩니다. RegularPolygon 클래스의 코드가 다음에 나와 있습니다.

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + " degrees.\n";
            return desc;
        }
    }
}
```

RegularPolygon 클래스에서는 먼저 모든 일반 다각형에 공통적인 두 가지 속성, 즉 sideLength 속성(각 변의 길이) 및 numSides 속성(변의 수)을 선언합니다.

RegularPolygon 클래스에서는 IPolygon 인터페이스를 구현하며 이 인터페이스의 메서드 네 개를 모두 선언합니다. 이러한 메서드 중 두 개, 즉 getPerimeter() 및 getSumOfAngles() 메서드는 일반적인 공식을 사용하여 구현합니다.

getArea() 메서드에 사용되는 공식은 형태에 따라 다르므로 이 메서드의 기본 클래스 버전에는 하위 클래스 메서드에 상속될 수 있는 공통적인 논리를 포함할 수 없습니다. 대신 간단히 기본값 0을 반환하여 면적이 계산되지 않았음을 나타냅니다. 각 형태의 면적을 올바르게 계산하려면 RegularPolygon 클래스의 하위 클래스에서 getArea() 메서드를 재정의해야 합니다.

EquilateralTriangle 클래스의 다음 코드에서는 getArea() 메서드를 재정의하는 방법을 보여 줍니다.

```
package com.example.programmingas3.geometricshapes
{
    public class EquilateralTriangle extends RegularPolygon
    {
        public function EquilateralTriangle(len:Number = 100):void
        {
            super(len, 3);
        }

        public override function getArea():Number
        {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
        }

        public override function describe():String
        {
                /* starts with the name of the shape, then delegates the rest
                of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
        desc += super.describe();
        return desc;
        }
    }
}
```

override 키워드는 EquilateralTriangle.getArea() 메서드로 RegularPolygon 수퍼 클래스의 getArea() 메서드를 재정의하려는 의도를 나타냅니다. EquilateralTriangle.getArea() 메서드가 호출되면 위 코드의 공식을 사용하여 면적을 계산하며 RegularPolygon.getArea() 메서드의 코드는 결코 실행되지 않습니다.

한편 EquilateralTriangle 클래스에 getPerimeter() 메서드의 자체 버전은 정의되어 있지 않습니다. EquilateralTriangle.getPerimeter() 메서드가 호출되면 이 호출은 상속 체인을 따라 위로 전달되어 RegularPolygon 클래스의 getPerimeter() 메서드의 코드가 실행됩니다.

EquilateralTriangle() 생성자에서는 super() 문을 사용하여 수퍼 클래스의 RegularPolygon() 생성자를 명시적으로 호출합니다. 두 생성자의 매개 변수 집합이 동일한 경우에는 EquilateralTriangle() 생성자를 완전히 생략할 수 있으며 이렇게 하면 RegularPolygon() 생성자가 대신 실행됩니다. 그러나 RegularPolygon() 생성자에서는 numSides라는 추가적인 매개 변수가 필요합니다. 따라서 EquilateralTriangle() 생성자에서는 super(len, 3)을 호출하여 len 입력 매개 변수와 값 3을 전달함으로써 삼각형의 변이 3개임을 알립니다.

describe() 메서드에서도 super() 문을 사용하지만 다른 방법이 적용됩니다. 이 방법에서는 super() 문을 사용하여 describe() 메서드의 RegularPolygon 수퍼 클래스 버전을 호출합니다. EquilateralTriangle.describe() 메서드에서는 먼저 desc 문자열 변수를 형태 유형에 대한 명령문으로 설정합니다. 그런 다음 super.describe()를 호출하여 RegularPolygon.describe() 메서드의 결과를 구하고 이 결과를 desc 문자열에 추가합니다.

Square 클래스는 여기서 자세히 설명되지는 않지만 생성자와 getArea() 및 describe() 메서드의 자체 구현을 제공한다는 점에서 EquilateralTriangle 클래스와 유사합니다.

다형성 및 팩토리 메서드

인터페이스 및 상속을 잘 활용하는 클래스의 집합은 여러 가지 흥미로운 방식으로 사용될 수 있습니다. 예를 들어 지금까지 설명한 모든 형태 클래스는 IGeometricShape 인터페이스를 직접 구현하거나 이 인터페이스를 구현한 수퍼 클래스를 확장합니다. 따라서 변수를 IGeometricShape의 인스턴스로 정의하면 describe() 메서드를 호출하기 위해 해당 인스턴스가 실제로 Circle 클래스의 인스턴스인지 아니면 Square 클래스의 인스턴스인지 알아야 할 필요가 없습니다.

다음 코드에서는 이에 대한 예제를 보여 줍니다.

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

변수가 IGeometricShape 인터페이스의 인스턴스로 정의되어 있지만 기반 클래스는 Circle이므로 myShape.describe()를 호출하면 Circle.describe()가 실행됩니다.

이 예제에서는 다형성의 원리가 적용됨을 보여 줍니다. 정확히 동일한 메서드를 호출해도 메서드가 호출된 객체의 클래스에 따라 다른 코드가 실행되는 것을 확인할 수 있습니다.

GeometricShapes 응용 프로그램에서는 팩토리 메서드로 알려진 디자인 패턴의 단순화된 버전을 사용하여 이러한 종류의 인터페이스 기반 다형성을 적용합니다. **팩토리 메서드**라는 용어는 기본 데이터 유형이나 내용이 컨텍스트에 따라 달라질 수 있는 객체를 반환하는 함수를 의미합니다.

여기에 나와 있는 GeometricShapeFactory 클래스에는 createShape()라는 이름의 팩토리 메서드가 정의되어 있습니다.

```
package com.example.programmingas3.geometricshapes
{
    public class GeometricShapeFactory
    {
        public static var currentShape:IGeometricShape;

        public static function createShape(shapeName:String,
                                           len:Number):IGeometricShape
        {
            switch (shapeName)
            {
                case "Triangle":
                    return new EquilateralTriangle(len);

                case "Square":
                    return new Square(len);

                case "Circle":
                    return new Circle(len);
            }
            return null;
        }

        public static function describeShape(shapeType:String, shapeSize:Number):String
        {
            GeometricShapeFactory.currentShape =
                GeometricShapeFactory.createShape(shapeType, shapeSize);
            return GeometricShapeFactory.currentShape.describe();
        }
    }
}
```

createShape() 팩토리 메서드에서는 형태 하위 클래스의 생성자에서 생성되는 인스턴스의 세부 사항을 정의하도록 하는 한편 새 객체를 반환할 때는 응용 프로그램에서 더 일반적인 방식으로 이 객체를 처리할 수 있도록 IGeometricShape 인스턴스로 반환합니다.

이전 예제의 describeShape() 메서드는 응용 프로그램에서 팩토리 메서드를 사용하여 더 구체적인 객체에 대한 일반적인 참조를 얻을 수 있는 방법을 보여 줍니다. 응용 프로그램에서는 새로 생성된 Circle 객체에 대한 설명을 다음과 같이 확인할 수 있습니다.

```
GeometricShapeFactory.describeShape("Circle", 100);
```

그런 다음 describeShape() 메서드에서는 동일한 매개 변수를 사용하여 createShape() 팩토리 메서드를 호출하고 반환된 새 Circle 객체를 currentShape라는 정적 변수에 저장합니다. 이 변수의 유형은 IGeometricShape 객체입니다. 이어서 currentShape 객체에 대해 describe() 메서드를 호출하면 자동으로 Circle.describe() 메서드가 실행되고 이 메서드에서는 원에 대한 상세한 설명을 반환합니다.

샘플 응용 프로그램 개선

응용 프로그램을 개선하거나 변경하면 인터페이스 및 상속의 진정한 효과가 분명하게 드러납니다.

이 샘플 응용 프로그램에 새 형태로 5각형을 추가하려는 경우를 가정해 봅니다. RegularPolygon 클래스를 확장하고 getArea() 및 describe() 메서드의 자체 버전을 정의하는 Pentagon 클래스를 만듭니다. 그런 다음 응용 프로그램의 사용자 인터페이스에 있는 콤보 상자에 Pentagon 옵션을 새로 추가합니다. 더 이상의 작업은 필요하지 않습니다. Pentagon 클래스에서는 상속을 통해 RegularPolygon 클래스의 getPerimeter() 메서드 및 getSumOfAngles() 메서드의 기능을 자동적으로 사용합니다. Pentagon 클래스는 IGeometricShape 인터페이스를 구현한 클래스에서 상속되었으므로 Pentagon 인스턴스를 IGeometricShape 인터페이스로도 취급할 수 있습니다. 따라서 새 유형의 모양을 추가하기 위해 GeometricShapeFactory 클래스에 있는 메서드의 메서드 서명을 변경할 필요는 없으며 GeometricShapeFactory 클래스를 사용하는 코드를 변경할 필요도 없습니다.

Pentagon 클래스를 Geometric Shapes 예제에 추가하는 연습을 해 보십시오. 이 연습을 통해 응용 프로그램에 새 기능을 추가할 때의 작업 로드가 인터페이스와 상속으로 인해 얼마나 줄어들 수 있는지 확인하십시오.