3

데이터와 데이터형

'2장. 변수'에서 변수 값에 대해 배우면서 이미 데이터, 즉 우리가 스크립트에서 조작하는 정보에 대해 간단하게 접해 보았다. 이 장에서는 데이터에 대해 더 자세히 알아보고 액션스크립트에서 데이터를 정의하는 법, 데이터의 범주를 나누는 법, 그리고 데이터를 저장하는 법에 대해 살펴보자. 또한 데이터를 만들고 분류하는 방법도 배울 것이다.

데이터 대 정보

매우 넓은 의미에서 보면 '데이터(data)'는 단어에서부터 숫자, 이미지, 동영상, 사운드에 이르기까지 컴퓨터에 저장할 수 있는 모든 것을 뜻한다. 모든 컴퓨터 데이터는 최첨단 제품 광고 같은 데서 볼 수 있듯이 1과 0을 늘어놓은 형태로 저장된다.

```
01010101010101011010101101101010101010100000101010101010110101010
10101010101010101110101010101010101010101010101011111010101010101010101
01010101010101010101010101110101010101010101010101010101010101010101010
```

데이터는 가장 정제되지 않은 상태의(즉 아직 가공되지 않은, 그리고 별 의미가 없는) 정보이다. 하지만 정보는 어떤 의미를 가진 것이다. 예를 들어 8008898969라는 숫자를 생각해 보자. 가공되지 않은 정보는 그다지 의미가 없지만 전화번호로 생각한다면, (800)889-8969가 되어 단순한 데이터에서 벗어나 유용한 정보가 될 수 있다.

이 장에서는 가공되지 않은 컴퓨터 데이터에 의미를 부여하여 사람이 이해할 수 있는 정보로 바꾸는 방법에 대해 알아본다.

데이터형에서 의미 찾기

어떻게 하면 의미를 잃지 않고 정보를 컴퓨터에 가공되지 않은 형태의 데이터로 저장할 수 있을까? 데이터의 범주를 나누고 데이터형을 정의하면 그 의미를 정의하는 데 도움이 된다.

예를 들어 5155534, 5159592, 4593030이라는 세 개의 숫자가 있다고 가정하자(전화 번호, 팩스 번호, 화물 추적 번호 등으로). 이 데이터의 범주를 나누면 데이터의 의미를 보존할 수 있다. 데이터 범주를 나누면 별 의미를 찾을 수 없을 것 같은 일곱 자리 숫자도 의미를 가지게 된다.

프로그래밍 언어에서는 데이터에 대한 기본적인 범주를 제공하는 '데이터형(datatype)'을 사용한다. 예를 들어 거의 모든 프로그래밍 언어에서는 텍스트(문자열) 및 숫자를 저장하고 조작하기 위한 데이터형이 정의되어 있다. 여러 개의 숫자를 구분할 때는 phoneNumber나 faxNumber 같이 쉽게 알아볼 수 있도록 변수 이름을 정하면 된다. 더 복잡한 상황에서는 나중에 배우게 될 '객체(object)'와 '객체 클래스(object class)'를 이용하여 직접 데이터 범주를 만들 수도 있다. 새로운 데이터 범주를 만들기 전에 액션스크립트에서 어떤 데이터형을 제공하는지 알아보자.

액션스크립트의 데이터형

프로그래밍을 하다 보면 제품 이름, 배경색, 밤 하늘에 그릴 별의 개수와 같은 정보를 저장해야 하는 경우가 생긴다. 데이터를 저장할 때는 다음과 같은 액션스크립트 데이터형을 사용한다.

- 액션스크립트에서는 "hi there," 같은 텍스트를 저장하기 위한 '문자열(string)' 데이터형을 지원한다. 문자열은 문자(알파벳, 숫자 및 문장부호)를 한 줄로 늘어놓은 것을 의미한다.
- 351이나 7.5와 같은 숫자를 저장하기 위한 '숫자(number)' 데이터형이 제공된다. 숫자는 어떤 대상의 개수를 세거나 수학적인 계산을 하는 데 쓰인다.
- 논리 계산을 위한 '부울(Boolean)' 데이터형이 있다. 부울 데이터를 이용하면 어떤 조건이나 비교 결과 상태를 표현하거나 저장할 수 있다. 부울 데이터는 true와 false의 두 가지 값만을 가질 수 있다.
- 데이터가 없음을 나타내기 위해 null과 undefined라는 두 가지 특별한 데이터 값을 지원한다. 이 두 값은 null과 undefined라는 두 데이터형에서 가질 수 있는 유일한 값이라고 생각할 수 있다.
- 여러 데이터 목록을 지원하기 위해 '배열(array)' 데이터형이 제공된다.
- 마지막으로 임의의 내장 데이터 또는 사용자 정의 데이터를 저장할 수 있는 강력한 '객체(object)' 데이터형을 지원한다.

액션스크립트에서 저장하는 모든 데이터는 위에 있는 범주 중 하나로 분류할 수 있다. '4장. 원시 데이터형'에서 각 데이터형에 대해 배우기 전에 데이터를 사용하는 것과 관련된 일반적인 문제를 짚고 넘어가도록 하자.

데이터 생성 및 범주 나누기

액션스크립트에서 새로운 데이터를 만드는 방법에는 두 가지가 있는데, 두 방법 모두 표현식(스크립트에서 데이터를 나타내는 구문)을 이용해야 한다.

리터럴 표현식(또는 줄여서 리터럴)은 그 자체로 데이터가 되는 일련의 문자, 숫자, 문장기호이다. 데이터 리터럴은 프로그램 소스 코드에서 데이터를 있는 그대로 표현한 것이다. 이는 데이터를 저장하는 그릇에 해당하는 변수와 대조를 이룬다. 각 데이터형에는 리터럴을 만들기 위한 규칙이 정해져 있다. 리터럴의 예를 들자면 다음과 같다.

```
"loading...please wait"  // 문자열 리터럴
1.51                     // 숫자 리터럴
["jane", "jonathan"]     // 배열 리터럴
```

무비 클립은 리터럴로 표현할 수 없고 대신 인스턴스 이름으로 참조한다는 점에 유의하자.

또한 '복합 표현식(complex expression)'을 이용하여 프로그램을 통해 데이터를 만들 수도 있다. 복합 표현식에서는 데이터를 있는 그대로 나타내는 것이 아니라 계산해서 값을 구해야 하는 코드 구문으로 표현한다. 계산 결과가 표현하려는 데이터가 된다. 아래 예로 나와 있는 복합 표현식을 계산하면 각각 하나씩 데이터가 나온다.

```
1999 + 1       // 2000이라는 데이터가 된다.
"hi " + "ma!"  // "hi ma!"라는 데이터가 된다.
firstName      // firstName 변수의 값이 된다.
_currentframe  // 플레이헤드가 현재 위치해 있는 플레이헤드의 프레임 번호
new Date()     // 현재 날짜와 시각이 저장된 새로운 Date 객체
```

1999나 1과 같은 개별 리터럴 표현식도 1999 + 1과 같이 더 큰 복합 표현식의 일부로 쓰일 수 있다는 점을 기억해 두자.

데이터를 만들기 위해 리터럴 표현식을 쓰거나 복합 표현식을 쓰거나 상관없이 나중에 사용하기 위해서는 데이터를 저장해야 한다. 앞에서 구한 "hi" + "ma!" 같은 표현식은 변수 같은 것에 저장하지 않으면 나중에 다시 불러서 쓸 수가 없다.

```
// 이 데이터는 저장되지 않기 때문에 만들어지자마자 사라져 버린다.
"hi " + "ma";

// 이 데이터는 변수에 저장되므로 나중에
// welcomeMessage라는 변수를 통해 사용할 수 있다.
var welcomeMessage = "hi " + "ma";
```

데이터를 적절한 형으로 어떻게 나눌까? 즉 주어진 데이터가 숫자, 문자열, 배열 또는 다른 데이터형이라는 것을 어떻게 알 수 있을까? 대부분의 경우에 우리가 직접 데이터를 분류할 필요는 없다. 액션스크립트에서 내부 규칙에 따라 자동으로 각 데이터의 데이터형을 대입하거나 추정해내기 때문이다.

리터럴 데이터형의 결정법

인터프리터에서는 아래에 있는 코드의 주석에 설명해 놓은 것과 같이 문장 구조를 검사함으로써 리터럴 데이터형을 추정해낸다.

```
"animal"      // 큰따옴표로부터 "animal"이 문자열이라는 것을 알 수 있다.
1.35          // 정수와 소수점만 있는 경우에는 숫자이다.
true          // true라는 키워드로부터 부울형임을 알 수 있다.
null          // null이라는 키워드로부터 null 형임을 알 수 있다.
undefined     // undefined라는 키워드로부터 undefined 형임을 알 수 있다.

["hello", 2, true] // 대괄호로 둘러싸고 각 값을 쉼표로 분리해 놓은
                   // 것으로부터 배열이라는 것을 알 수 있다.

{x: 234, y: 456}   // 중괄호로 둘러싸고 이름과 값을 쌍으로 분리하고
                   // 각 쌍은 쉼표로 구분한 것으로 보아 객체임을 알 수 있다.
```

위에서 볼 수 있듯이 데이터 리터럴을 다룰 때는 올바른 표기법을 사용하는 것이 매우 중요하다. 잘못된 표기법을 이용하면 인터프리터에서 데이터 내용물을 잘못 해석하거나 오류가 발생할 수 있다.

```
animal   // 큰따옴표가 없기 때문에 문자열이 아닌 변수로 인식한다.
"1.35"   // 숫자를 큰따옴표로 둘러싸면 숫자가 아닌 문자열로 인식한다.
1. 35    // 3 앞에 공백이 있으므로 오류가 발생한다.
"animal  // 오른쪽에 큰따옴표가 없기 때문에 오류가 발생한다.
```

복합 표현식 데이터형의 결정법

인터프리터에서는 표현식 값을 계산하여 데이터형을 결정한다. 아래 예를 생각해 보자.

```
pointerX = _xmouse;
```

_xmouse에는 마우스 포인터 위치가 숫자로 저장되어 있으므로, _xmouse라는 표현식 형은 언제나 숫자이다. 따라서 pointerX도 숫자가 된다.

대개 인터프리터에서 자동으로 결정된 데이터형은 사용자가 원하는 데이터형과 일치한다. 하지만 모호한 경우도 있기 때문에 사용자가 인터프리터에서 표현식의 데이터형을 결정할 때 사용하는 규칙을 이해하고 있어야 한다(〔예제 2-2〕 및 〔예제 2-3〕 참조). 다음과 같은 표현식을 살펴보자.

```
"1" + 2;
```

왼쪽의 피연산자는 문자열("1")이지만 오른쪽의 피연산자는 숫자(2)이다. + 연산자는 숫자(덧셈)와 문자열(문자열 합치기)에 모두 작용하는 연산자이다. 그렇다면 "1" + 2라는 표현식의 결과가 3이 되어야 할까, "12"가 되어야 할까? 이처럼 모호한 상황에 대처하기 위해 인터프리터에서는 하나의 고정된 규칙을 따르게 된다. 더하기 연산자(+)는 언제나 숫자보다는 문자열을 선호한다는 것이다. 따라서 "1" + 2 라는 표현식은 숫자 3이 아닌, 문자열 "12"가 된다. 이러한 규칙도 약간 임의적이긴 하지만 코드를 해석하는 데 하나의 일관된 방법을 정할 수 있기 때문에, 이러한 규칙이 있는 것이 좋다. 이러한 규칙은 덧셈 연산자의 일반적인 용법을 염두에 두고 만들어졌다. 두 피연산자 중 하나가 문자열이라면 아래 예에서 볼 수 있듯이 그 피연산자를 그냥 더하는 것보다는 문자열을 합치는 경우가 더 많기 때문이다.

```
trace ("The value of x is: " + x);
```

서로 종류가 다른 데이터형을 가진 데이터를 합치거나 미리 예상하고 있던 데이터형과 맞지 않는 데이터를 사용하는 경우에는 문제가 생길 소지가 있다. 이런 상황 때문에 인터프리터는 임의의, 하지만 예측할 수 있는 규칙에 따라 자동으로 데이터형을 변환할 수밖에 없다. 자동 변환이 일어나는 상황과 한 데이터형에서 다른 데이터형으로 데이터를 변환할 때 예상되는 결과에 대해 살펴보자.

데이터형 변환

앞 절에서 다룬 예제를 조금 더 자세히 살펴보자. 그 예제에서 각 데이터("1"과 2)는 서로 다른 데이터형을 가진다. 첫째 데이터는 문자열이고 둘째 데이터는 숫자이다. 인터프리터에서는 두 값을 합쳐서 "12"라는 문자열을 만든다. 이 때 인터프리터에서 우선 숫자 2를 문자열 "2"로 변환해야 한다는 점을 주목하자. 자동 변환이 일어나기 전에는 "2"라는 문자열을 "1"이라는 문자열과 합칠 수 없다.

데이터형 변환은 단순히 데이터형을 바꾸는 것에 불과하다. 모든 데이터형 변환이 자동으로 이루어지는 것은 아니다. 액션스크립트에서 자동으로 데이터형을 변환하는 규칙과 다른 방법으로 데이터형을 바꾸고 싶다면 데이터형을 수동으로 바꿔야 한다.

자동 형 변환

어떤 값을 원래 데이터형에 알맞지 않은 부분에서 사용하면 인터프리터에서는 변환을 시도한다. 즉 인터프리터에서 A라는 데이터형을 가진 데이터를 예상하고 있는데 B라는 데이터형을 가진 데이터를 제공한다면, 인터프리터에서는 B형 데이터를 A형 데이터로 변환하려고 시도하게 된다. 예를 들어 아래 코드에서는 "Flash"라는 문자열을 뺄셈 연산자의 오른쪽 피연산자로 사용하였다. 뺄셈 연산자에서는 숫자만을 사용할 수 있기 때문에, 인터프리터에서는 "Flash"라는 문자열을 숫자로 변환하려고 한다.

```
999 - "Flash";
```

물론 "Flash"라는 문자열은 어떤 숫자로도 변환될 수 없으므로 특별한 숫자 데이터 값인 NaN(Not a Number, 숫자가 아니라는 뜻)으로 변환된다. NaN은 숫자 데이터형에서 사용할 수 있는 값으로, 주로 숫자로 표현할 수 없는 경우를 표현하기 위해 만들어졌다. "Flash"가 NaN으로 변환되면 인터프리터 입장에서는 위 표현식이 아래와 같이 바뀌게 된다(물론 우리 눈에는 보이지 않는다).

```
999 - NaN;
```

이제 두 피연산자가 모두 숫자형이므로 연산을 처리할 수 있다. 999 - NaN을 계산하면 NaN이 나오므로 결국 위에 있는 표현식 값은 NaN이 된다.

NaN이라는 값으로 나오는 표현식은 거의 쓸모가 없다. 사실 대부분의 경우 변환을 하고 나면 뭔가 의미 있는 결과가 나온다. 예를 들어 문자열에 숫자만 들어 있다면 그 문자열은 숫자로 변환된다.

```
999 - "9";  // 숫자 999 빼기 문자열 "9"
```

위 표현식은 내부적으로 다음과 같이 변환된다.

```
999 - 9;    // 숫자 999 빼기 숫자 9
```

따라서 표현식을 계산한 결과는 990이 된다. 자동 변환은 덧셈 연산자, 비교 연산자, 그리고 조건문이나 순환문에서 주로 쓰인다. 자동 변환과 관련된 표현식 결과를 정확하게 알 수 있으려면 다음과 같은 세 가지 질문에 답할 수 있어야 한다. (1) 주어진 상황에서 어떤 데이터형이 예상될까? (2) 그 상황에서 예기치 못한 데이터형이 들어가면 어떻게 될까? (3) 변환이 일어날 때 그 결과는 어떻게 될까?

첫째와 둘째 질문에 답하려면 이 책의 다른 부분에서 적절한 내용을 찾아봐야 한다(예를 들어 조건문에서 예상되는 데이터형을 알아보려면 '7장. 조건문'을 참조하면 된다).

아래 나와 있는 세 개의 표에서는 자동 변환 규칙을 설명하고 있는데, 이 규칙을 제대로 알면 셋째 질문에 대해 답할 수 있다. 〔표 3-1〕에는 각 데이터형을 숫자로 변환했을 때의 결과가 나와 있다.

〔표 3-1〕 숫자로 변환하기

원본 데이터	변환 결과
undefined	0
null	0
부울형	원본 값이 true이면 1, false이면 0
숫자 문자열	문자열이 10진수 숫자와 공백, 지수, 소수점, 덧셈 기호, 뺄셈 기호만으로 이루어질 때는 같은 값을 가지는 숫자로 변환됨(예: "-1.485e2")
기타 문자열	비어 있는 문자열, 숫자가 아닌 문자열, 또는 "x", "0x", "FF" 등으로 시작하는 문자열은 모두 NaN으로 변환됨
"Infinity"	Infinity
"-Infinity"	-Infinity
"NaN"	NaN
배열	NaN
객체	객체의 valueOf( ) 메소드의 리턴 값
무비 클립	NaN

〔표 3-2〕에는 각 데이터형을 문자열로 변환할 때의 결과가 나와 있다.

〔표 3-2〕 문자열로 변환하기

원본 데이터	변환 결과
undefined	“ ”(비어 있는 문자열)
null	“null”
부울형	원본 값이 true이면 “true”, false이면 “false”
NaN	“NaN”
0	“0”
Infinity	“Infinity”
-Infinity	“-Infinity”
다른 숫자값	그 숫자에 해당하는 문자열(예: 944.345는 “944.345”로 변환됨)
배열	각 원소를 쉼표로 구분한 리스트
객체	객체의 toString( )을 호출한 결과. 기본적으로 객체의 toString( ) 메소드는 “[object Object]”를 리턴한다. toString( ) 메소드를 직접 만들어서 더 의미 있는 결과를 리턴하도록 만들 수도 있다(예: Date 객체의 toString( )에서는 “Sun May 14 11:38:10 EDT 2000”과 같은 결과를 리턴한다).
무비 클립	플레이어의 문서 레벨에서 시작하는 무비 클립 인스턴스의 절대 경로(예: “_level0.ball”)

〔표 3-3〕은 각 데이터형을 부울형으로 변환할 때의 결과이다.

〔표 3-3〕 부울형으로 변환하기

원본 데이터	변환 결과
undefined	false
null	false
NaN	false
0	false
Infinity	true
-Infinity	true
다른 숫자값	true

원본 데이터	변환 결과
비어있지 않은 문자열	문자열을 0이 아닌 숫자로 변환할 수 있으면 true, 그렇지 않으면 false. ECMA-262에서는 모든 비어있지 않은 문자열은 true로 변환된다(플래시 5에서는 플래시 4와의 호환성을 위해 이러한 규칙을 따르지 않는다).
비어있는 문자열(“”)	false
배열	true
객체	true
무비 클립	true

수동 형 변환

자동 형 변환 결과가 프로그래머가 원하는 결과와 다르다면 수동으로 데이터형을 바꿀 수 있다. 실제 작업을 할 때는 앞에 있는 표에 나온 규칙이 여전히 적용된다는 점을 염두에 두어야 한다.

toString() 메소드를 이용하여 문자열로 변환하기

아래 예에서 볼 수 있듯이 어떤 데이터라도 toString( ) 메소드를 호출하여 문자열로 변경할 수 있다.

```
x.toString();       // 변수 x의 문자열 값을 구한다.
(523).toString();   // "523"이 리턴된다. 이 때 "."이 소수점으로 인식되지
                    // 않도록 523을 괄호로 둘러싼다.
```

숫자에 대해 toString( ) 메소드를 호출할 때는 문자열로 변환할 때 수의 진법(2진법, 8진법, 16진법 등)을 의미하는 숫자를 인자로 정해줄 수 있다. 이렇게 하면 16진수, 10진수, 8진수와 같은 진법을 마음대로 변환할 수 있다.

```
var myColor = 255;
var hexColor = myColor.toString(16);  // hexColor를 "ff"로 변환한다.
```

String() 함수를 이용하여 문자열로 변환하기

String() 함수를 이용하면 toString() 메소드와 같은 결과가 나오지만 문법이 약간 다르다.

```
String(x);    // x를 문자열로 변환한다.
String(523);  // 523을 "523"이라는 문자열로 변환한다.
```

String() 함수 같은 이름을 가진 내장 클래스 생성자를 혼동하지 않도록 주의하자. 이 두 가지에 대한 설명은 모두 '3부. 레퍼런스'에 자세히 나와 있다.

비어있는 문자열과 결합하여 문자열로 변경하기

덧셈 연산자(+)를 사용할 때 피연산자 중에 문자열이 있으면 자동으로 문자열로 변환되므로, 임의의 데이터에 ""를 더하면 데이터가 문자열로 변환된다.

```
x + "";      // x를 문자열로 변환한다.
523 + "";    // 523을 문자열 "523"으로 변환한다.
```

Number() 함수를 이용하여 숫자로 변환하기

String() 함수를 이용하면 데이터가 문자열로 바뀌는 것처럼, Number() 함수를 이용하면 인자가 숫자형으로 변환된다. 실수형으로 변환하는 것이 불가능하거나 논리적으로 알맞지 않을 때는 〔표 3-1〕에 나온 것과 같은 특별한 숫자 값을 리턴한다.

```
Number(age);     // age의 값을 숫자로 변환한 값을 리턴한다.
Number("29");    // 숫자 29가 리턴된다.
Number("sara");  // NaN이 리턴된다.
```

Number() 함수와 같은 이름을 가진 내장 클래스 생성자를 혼동하지 않도록 주의하자. 이 두 가지에 대한 설명도 3부에 나와 있다.

사용자가 스크린에서 입력한 텍스트 필드는 모두 문자열형이므로 수학 계산을 할 때는 그 문자열을 숫자로 변환해야 한다. 예를 들어 price1과 price2라는 텍스트 필드의 합을 구하려면 다음과 같이 해야 한다.

```
totalCost = Number(price1) + Number(price2);
```

이렇게 하지 않으면 price1과 price2를 숫자로 더하는 것이 아니라 문자열로 합치기 때문이다. 텍스트 필드에 대해서는 '18장. 온스크린 텍스트 필드'를 참조하기 바란다.

0을 빼서 숫자로 변환하기

어떤 데이터를 숫자로 바꿀 때는 그 데이터에서 0을 빼는 트릭을 이용할 수도 있다. 이 경우에도 [표 3-1]에서 설명한 변환 규칙이 적용된다.

```
"953" - 0      // 953
"molly" - 0    // NaN
x - 0          // x의 값을 숫자로 변환한 값
```

parseInt()와 parseFloat() 함수를 이용하여 숫자로 변환하기

parseInt()와 parseFloat() 함수는 숫자와 문자를 포함하고 있는 문자열을 숫자로 변환한다. parseInt() 함수는 주어진 문자열의 첫째 문자가 숫자일 때 그 문자열에 들어 있는 첫째 정수를 추출한다. 그렇지 않은 경우에는 NaN을 리턴한다. parseInt()로 추출한 숫자는 문자열의 첫째 숫자(공백 제외)로 시작하여 처음으로 숫자가 아닌 문자가 나오는 위치 또는 처음으로 소수점이 나오는 위치 바로 앞에 있는 숫자로 끝난다.

parseInt()를 사용한 예는 다음과 같다.

```
parseInt("1a")                // 1이 추출된다.
parseInt("1.3a"               // 1이 추출된다.
parseInt("     1a")           // 1이 추출된다.
parseInt("I am 14 years old") // NaN이 리턴된다.
                              // (첫 번째 문자가 숫자가 아니다)
parseInt("14 years old")      // 14가 추출된다.
```

parseFloat() 함수는 문자열의 첫째 문자가 숫자일 때 문자열에 나타난 첫째 부동소수점 수를 추출한다(부동소수점 수는 -10.5나 345.678과 같이 소수점이 포함된 양수 또는 음수를 의미한다). parseInt()와 마찬가지로 parseFloat() 함수도 문자열의 첫째 문자(공백 제외)가 숫자로서 의미가 없는 문자일 경우에는 NaN을 리턴한다. parseFloat()에 의해 추출된 숫자는 문자열에 있는 첫째 문자(공백 제외)로 시작하

여 처음으로 숫자가 아닌 문자(+, -, 0-9, 소수점을 제외한 문자)가 나오는 위치 바로 앞에 있는 숫자로 끝난다.

parseFloat()을 사용한 예는 다음과 같다.

```
parseFloat("1.3a");             // 1.3이 추출된다.
parseFloat("2.75 years old")    // 2.75가 추출된다.
parseFloat("1nce upon a time")  // 1이 추출된다.
parseFloat("I'm 3.5 feet tall") // NaN이 리턴된다.
```

parseInt()와 parseFloat()에 대한 자세한 정보는 3부를 참조하기 바란다.

부울형으로 변환하기

데이터를 부울형으로 변환할 때는 Boolean()이라는 전역 함수를 사용하면 된다. 이 함수의 사용법은 String()이나 Number()와 비슷하다. 예를 들면 다음과 같다.

```
Boolean(5);  // true
Boolean(x);  // x를 부울형으로 변환한 값
```

전역 함수 Boolean()과 같은 이름을 가지는 내장 클래스 생성자를 혼동하지 않도록 주의하자. 이 두 가지에 대한 설명도 마찬가지로 3부에 나와 있다.

변환 지속시간

대입하는 과정을 제외하면 변수, 배열의 원소, 객체 속성 등의 형 변환은 모두 일시적이다. 아래와 같은 선언문에서는 임시 변환이 일어날 뿐이다.

```
var x = "10";     // x는 문자열이다.
y = x - 5;        // y의 값은 5가 된다. x의 값은 숫자로 변환된다.
trace(typeof x);  // "string"이 출력된다. x는 일시적으로 변환되었을
                  // 뿐이기 때문이다.
```

대입하면 아래에서 볼 수 있듯이 영구적으로 변환된다.

```
x = "10";         // x는 문자열이다.
x = x - 5;        // x가 숫자로 변환된다.
trace(typeof x);  // "number"가 출력된다. 대입 과정에서 변환되었기 때문에
                  // 변환 결과가 영구적이다.
```

플래시 4에서 플래시 5로 데이터형 변환

플래시 4에서는 문자열 연산자와 숫자 연산자가 완전히 다르다. 숫자에만 사용할 수 있는 연산자와 문자열에만 사용할 수 있는 연산자가 분리되어 있다. 예를 들어 플래시 4에서 문자열 합치기 연산자는 &이고 숫자를 더하는 연산자는 +이다. 마찬가지로 문자열 비교 연산은 eq와 ne 연산자를 이용하여 처리했지만, 숫자열 비교는 =와 <>를 이용하여 처리한다.

〔표 3-4〕 플래시 4와 플래시 5의 연산자 비교

연산	플래시 4 연산자	플래시 5 연산자
문자열 병합	&	+ 또는 add
문자열 비교(같음)	eq	==
문자열 비교(같지 않음)	ne	!=
문자열 비교(순서 비교)	ge, gt, le, lt	>=, >, <=, <
숫자 덧셈	+	+
숫자 비교(같음)	=	==
숫자 비교(같지 않음)	<>	!=
숫자 비교(대소 비교)	>=, >, <=, <	>=, >, <=, <

하지만 플래시 5에는 문자열과 숫자에서 동시에 사용할 수 있는 연산자도 있다. 예를 들어 + 연산자를 문자열에 사용하면 피연산자를 합쳐서 새로운 문자열을 만든다. 하지만 숫자에 사용하면 두 개의 피연산자를 더하는 역할을 한다. 마찬가지로 플래시 5에서는 두 값이 같은지 다른지를 알아내기 위한 연산자인 ==와 !=를 문자열이나 숫자 및 그 외의 데이터형에 대해서도 사용할 수 있다.

플래시 5에서는 여러 개의 데이터형에 사용할 수 있는 연산자가 많이 있지만 플래시 4에서는 그렇지 않기 때문에, 플래시 4 파일을 플래시 5에서 불러올 때 문제가 생길 수도 있다. 따라서 플래시 5에서 플래시 4 파일을 가져올 때는 아래와 같이 혼동을 일으킬 수 있는 연산자의 피연산자로 쓰인 숫자에는 모두 Number( ) 함수를 자동으로 삽입한다(피연산자가 숫자 리터럴인 경우는 제외).

+, ==, !=, <>, <, >, >=, <=

또한 플래시 4 파일을 플래시 5 파일로 변환하면 문자열 병합 연산자인 & 연산자도 새로 도입된 add 연산자로 치환된다. 〔표 3-5〕에 플래시 4 연산자를 플래시 5 연산자로 변환한 예가 나와 있다.

〔표 3-5〕 플래시 4에서 플래시 5 연산자로 변환한 예

플래시 4 문법	플래시 5 문법
Loop While (count <= numRecords)	while (Number(count)<= Number(numRecords))
If (x = 15)	if(Number(x) == 15)
If (y < > 20)	if(Number(y) != 20)
Set Variable: "lastName" = "kavanagh"	lastName = "kavanagh"
Set Variable: "name" = "molly" & lastName	name = "molly" add lastName

데이터형 확인법

주어진 표현식의 데이터형을 알고 싶다면 다음과 같이 typeof 연산자를 이용하면 된다.

```
typeof 표현식;
```

typeof 연산자는 주어진 표현식의 데이터형을 알려주는 문자열을 리턴한다. 리턴 값은 〔표 3-6〕에 나와 있다.

〔표 3-6〕 typeof 연산자의 리턴 값

원본 데이터형	연산자의 리턴 값
숫자	"number"
문자	"string"
부울형	"boolean"
객체	"object"
배열	"object"
null	"null"
무비클립	"movieclip"
함수	"function"
undefined	"undefined"

예를 들면 다음과 같다.

```
trace(typeof "game over");    // "string"이 Output 창에 출력된다.

var x = 5;
trace(typeof x);              // "number"가 출력된다.

var now = new Date();
trace(typeof now);            // "object"가 출력된다.
```

〔예제 3-1〕에 나온 것처럼 typeof를 for-in 선언문과 함께 사용하면 타임라인에 있는 모든 무비 클립 인스턴스를 손쉽게 찾아낼 수 있다. 일단 모든 클립을 찾아내면 나중에 프로그래밍을 통해 처리하기 좋도록 각 클립을 배열에 저장할 수도 있다(지금 〔예제 3-1〕을 제대로 이해할 수 없다면 '1부. 액션스크립트 기초'를 모두 마친 후에 다시 읽어보면 된다).

〔예제 3-1〕 동적으로 확인한 무비 클립을 배열에 채우기

```
var childClip = new Array();
var childClipCount = 0;

for (i in _root) {
   thisItem = _root[i];
   if (typeof thisItem == "movieclip") {
      // 변수 뒤에 ++ 연산자를 붙인 점에 주의
      childClip[childClipCount++] = thisItem;
   }
}

// 배열을 모두 채웠으므로 배열을 이용하여
// 그 안에 들어 있는 클립을 조작할 수 있다.
childClip[0]._x = 0;  // 첫째 클립을 스테이지 왼쪽에 놓는다.
childClip[1]._y = 0;  // 둘째 클립을 스테이지 맨 위에 놓는다.
```

원시 데이터와 복합 데이터

지금까지는 가장 흔한 '원시(primitive)' 데이터형인 숫자와 문자열을 주로 다루었다. 원시 데이터형은 언어의 기본 단위에 해당하며 각 원시 값에는 하나의 데이터가 있는 그대로 저장된다(여러 개의 아이템이 저장되는 배열과는 대조된다). 따라서 원시 데이터는 매우 직접적이다.

액션스크립트에서는 숫자, 문자열, 부울형, undefined, null 데이터형을 지원한다. 하지만 C나 C++에서 쓰이는 것과 같은 문자 한 개에 해당하는 데이터형(char형)은 지원하지 않는다.

원시 데이터형은 그 이름에서 알 수 있듯이 매우 간단하다. 이 데이터형에는 텍스트 메시지, 프레임 번호, 무비 클립 크기 같은 정보를 저장할 수 있지만, 더 높은 수준의 복잡한 데이터는 저장할 수 없다. 더 정교한 데이터 처리(수십 개의 공이 튕기는 것을 물리적으로 시뮬레이션하거나 500개 정도의 문제와 답을 관리해야 하는 퀴즈 같은 경우)가 필요하다면 '복합(composite)' 데이터형을 사용해야 한다. 복합 데이터를 이용하면 여러 개의 연관된 데이터들을 하나의 데이터처럼 관리할 수 있다.

액션스크립트에서는 배열, 객체, 무비 클립 이렇게 세 가지 복합 데이터형을 지원한다. 함수도 기술적으로 보면 객체의 일종이므로 복합 데이터로 간주할 수 있다. 하지만 함수를 복합 데이터인 것처럼 조작하는 경우는 거의 없다. 데이터형으로서의 함수에 대한 자세한 내용은 '9장. 함수'를 참조하기 바란다.

한 개의 숫자는 원시 데이터지만 여러 개의 숫자 목록(즉 배열)은 복합 데이터이다. 다음 예를 보면 복합 데이터가 얼마나 유용한지 알 수 있다. Derek이라는 고객의 프로파일을 추척하는 작업을 생각해 보자. Derek의 특징을 원시 데이터 값으로 저장하는 변수를 여러 개 만들 수 있다.

```
var custName = "Derek";
var custTitle = "Coding Genius";
var custAge = 30;
var custPhone = "416-222-3333";
```

하지만 몇 명의 고객만 더 추가하려고 해도 이러한 형식을 이용하면 작업하기가 귀찮아진다. 따라서 필요한 정보(cust1Name, cust2Name, cust1Title, cust2Title 따위)

를 추적하기 위해 순서대로 이름이 붙은 변수를 사용해야 한다. 하지만 배열을 이용하면 이보다 훨씬 효과적으로 정보를 저장할 수 있다.

```
cust1 = ["Derek", "Coding Genius", 30, "416-222-3333"];
```

새로운 고객을 추가하고 싶다면 새로운 배열을 만들기만 하면 된다.

```
cust2 = ["Komlos", "Comic Artist", 28, "515-515-3333"];
cust3 = ["Porter", "Chef", 51, "515-999-3333"];
```

이렇게 하면 깔끔하고 멋져 보인다. 복합 데이터형에 대해서는 잠시 후에 더 자세히 배우게 될 것이다.

앞으로 배울 내용

이 장에서 액션스크립트의 데이터에 대해 기초적인 내용을 배웠으므로 이제 더 심도 있는 내용을 배울 준비가 된 것이다. 4장에서는 숫자, 문자열, 부울형, undefined와 null 데이터형에 대해 배울 것이다. '5장. 연산자'에서는 데이터 조작법을 배운다. 그 다음에는 무비 클립, 배열 및 객체 같은 복합 데이터형을 배운다.