# 4

# 원시 데이터형

원시 데이터는 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 같은 숫자나 "a", "b", "c" 같은 문자열처럼 간단한 글자나 키워드로 이루어진다. 앞 장에서 배운 것처럼 액션스크립트의 원시 데이터형에는 숫자, 문자열, 부울, undefined, 그리고 null 형이 있다. 이 장에서는 각 형의 데이터를 정의하고 검사하고 변경하는 방법에 대해 알아본다.

## 숫자형

숫자는 수를 세거나 수학 계산을 하거나 (무비 클립의 현재 프레임이나 스테이지 위치와 같은) 무비의 수치 속성을 추적하는 데 쓰인다. 액션스크립트에서 숫자를 정의하고 조작하는 방법을 살펴보자.

## 정수와 부동소수점수

대부분의 프로그래밍 언어에서는 '정수(integer)'와 '부동소수점수(floating-point number)'를 구분한다. 정수는 소수점 아래 부분이 없는 수이다. 정수에는 양수, 음수 그리고 0이 있다. 부동소수점수에는 0.56, 199.99, 3.14159 같이 소수점 아래 부분이 있는 수도 포함된다. 따라서 1, 34253, -3, 0, -9999999와 같은 수는 정수지만, 223.45, -0.56, 1/4 같은 수는 부동소수점수이다.

## 숫자 리터럴

하나의 고정된 데이터 값을 직접 표현한 것을 리터럴이라고 한다는 것은 이미 배웠다. 숫자형에서는 정수 리터럴, 부동소수점 리터럴, 특별 수치 리터럴의 세 가지 리터럴을 지원한다. 앞의 두 종류는 실제 수(고정된 수학적인 값을 가진 숫자)를 나타내지만, 셋째 리터럴은 무한대와 같은 수학적 개념을 표현하는 값으로 이루어진다.

### 정수 리터럴

1, 2, 3, 99, -200과 같은 정수 리터럴은 다음과 같은 규칙을 따른다.

- 정수에는 소수점이나 소수점 이하의 숫자가 포함되지 않는다.
- 정수는 액션스크립트에서 허용하는 최대값과 최소값 사이에 있어야 한다. 허용 가능한 값에 대해서는 '3부. 레퍼런스'의 Number 객체에 있는 MIN_VALUE와 MAX_VALUE 속성을 참조하기 바란다.
- 10진수 정수 앞에는 불필요한 0을 덧붙이면 안 된다(002, 000023, 05 같은 표현은 사용하지 않는다).

모든 정수 값이 10진수 정수인 것은 아니다. 액션스크립트에서는 8진수와 16진수 리터럴도 사용할 수 있다. 10진수, 8진수, 16진수에 관한 내용은 다음 웹사이트를 참조하기 바란다.

http://www.moock.org/asdg/technotes

8진수를 표기할 때는 앞에 0을 덧붙인다. 예를 들어 액션스크립트에서 8진수 723을 표기할 때는 다음과 같이 한다.

```
0723  // 10진수로는 467이 된다(7*64 + 2*8 + 3*1).
```

16진수 정수를 표기할 때는 아래와 같이 숫자 앞에 0x(또는 0X)를 덧붙인다.

```
0x723  // 10진수로는 1827이 된다(7*256 + 2*16 + 3*1).
0xFF   // 10진수로는 255가 된다(15*16 + 15*1).
```

숫자 값을 나타낼 때 16진수를 종종 사용하긴 하지만 대부분의 간단한 프로그램에서는 10진수만으로도 충분하다. 문자열을 숫자로 변환할 때는 〔예제 4-1〕에 나온 것처럼 숫자 앞에 있는 불필요한 0을 없앨 수 있도록 주의를 기울여야 한다.

***〔예제 4-1〕 앞에 있는 0 잘라내기***

```
function trimZeros(theString) {
  while (theString.charAt(0) == "0" || theString.charAt(0) == " ") {
    theString = theString.substring(1, theString.length);
  }
  return theString;
}

testString = "00377";
trace(trimZeros(testString));  // 377이 출력된다.
```

## 부동소수점 리터럴

부동소수점 리터럴은 소수점 아래 부분의 숫자까지 표기한다. 부동소수점 리터럴에는 아래 네 가지 성분 중 하나 이상이 포함된다.

10진수 정수

소수점(.)

소수점 아래의 수(10진수로 표기)

지수

위에서부터 세 개의 성분은 쉽게 이해할 수 있을 것이다. 예를 들면 3.14에서 "3"은 10진수 정수이고 "."은 소수점이며 "14"는 소수점 아래의 수이다. 하지만 넷째 성분(지수)에 대해서는 조금 더 설명이 필요하다.

매우 큰 양수 또는 음수를 부동소수점수로 표현할 때는 E(또는 e)를 이용하여 숫자에 지수를 추가할 수 있다. 지수가 있는 숫자 값은 E 앞에 있는 수에 10을 지수 번만큼 곱한 값이다. 예를 들면 다음과 같다.

```
12e2    // 1200 (10의 2승은 100이고, 12에 100을 곱하면 1200이 된다.)
143E-3  // 0.143 (10의 -3은 0.001이고, 여기에 143을 곱하면 0.143이 된다.)
```

이러한 형식은 과학 분야에서 보통 사용되는 표기법과 똑같다. 수학에 강하지 않다면 다음과 같이 생각하면 쉽다. 지수가 양수이면 그 수만큼 소수점을 오른쪽으로 옮기고 지수가 음수라면 그만큼 왼쪽으로 옮기면 된다.

값이 너무 크거나 작으면 액션스크립트에서 계산 결과를 지수가 포함된 숫자로 리턴하는 경우도 있다. 하지만 지수를 사용할 때 E(또는 e)를 이용하는 것은 단지 편리하게 표기하기 위한 것이다. 어떤 숫자를 여러 번 곱한 수(승수)를 구하고 싶다면 내장 함수인 Math.pow( ) 함수를 이용하면 된다. 이 함수에 대한 설명은 3부를 참조하기 바란다.

## 부동소수점수의 정확도

플래시에서는 약 15개의 유효 숫자 정도의 정확도를 제공하는 '배정도(double-precision)' 부동소수점수를 이용한다(숫자 앞에 있는 0, 뒤에 있는 0, 지수는 15개의 유효 숫자에 포함되지 않는다). 즉 플래시에서 123456789012345는 표기할 수 있지만, 1234567890123456은 표시할 수 없다. 정확도 때문에 표시할 수 있는 숫자의 크기가 제한되는 것은 아니지만, 숫자를 표기하는 정확도는 제한된다. 예를 들면 2e16은 123456789012345보다 더 큰 숫자이긴 하지만 유효 숫자의 개수는 단 한 개뿐이다.

액션스크립트에서는 계산한 결과가 의도와 다른 방식으로 반올림되어 0.143이 아닌 0.14300000000000001 같은 숫자가 나올 수도 있다. 이러한 현상은 컴퓨터에서는 어떤 진법을 사용한 숫자라도 내부적으로는 2진법으로 표기하며 이진수로는 무한소수와 같은 숫자가 나올 수 있기 때문에 발생한다(10진수에서 0.3333333... 같

은 숫자가 있는 것과 마찬가지다). 컴퓨터에서 표현할 수 있는 숫자의 정확도에는 한계가 있기 때문에, 이처럼 이진법으로 무한소수가 되는 수는 정확하게 표시할 수 없다. 이와 같이 값이 약간 차이가 날 수 있는 점을 감안하여 만약 값 차이 때문에 코드에 안 좋은 영향을 끼친다면, 숫자를 직접 수동으로 반올림해주는 것이 좋다. 예를 들어 아래 코드에서는 myNumber를 소수점 세 자리까지만 남기고 반올림한다.

```
myNumber = Math.round(myNumber * 1000) / 1000;
```

아래 예제는 어떤 숫자든지 주어진 소수점 자리까지 남기고 반올림하는 함수를 구현한 코드이다.

```
function trim(theNumber, decPlaces) {
  if (decPlaces >= 0) {
    var temp = Math.pow(10, decPlaces);
    return Math.round(theNumber * temp) / temp;
  }
}

// 소수점 둘째자리에서 반올림
trace(trim(1.12645, 2));  // Displays: 1.13
```

## 숫자 데이터형에 있는 특수 값

정수와 부동소수점 리터럴만 있으면 숫자 데이터형에 속하는 거의 모든 값을 표기할 수 있지만, 숫자가 아닌 것, 표기할 수 있는 제일 작은 값, 표기할 수 있는 제일 큰 값, 무한대, 음의 무한대 같은 수학적인 개념을 표기하기 위한 특별한 키워드 값은 따로 정해져 있다.

이러한 특수 값은 변수나 속성에 대입할 수 있으며, 다른 숫자 리터럴과 마찬가지로 아무 리터럴 표현식에서나 사용할 수 있다. 하지만 프로그래머가 이러한 숫자를 직접 사용하는 일은 거의 없고, 대개의 경우 인터프리터에서 어떤 표현식을 계산한 결과로 이러한 특수 값을 리턴한다.

## 숫자가 아닌 것

종종 숫자를 계산하거나 데이터형을 변환하다 보면 숫자가 아닌 값이 나오는 경우가 있다. 예를 들어 0/0 같은 것은 계산할 수가 없으며 아래 표현식은 유한 숫자로 변환되지 않는다.

```
23 - "go ahead and try!"
```

액션스크립트에서는 숫자형이긴 하지만 실제 수가 아닌 것을 나타낼 때 NaN 키워드 값을 이용한다. NaN은 어떤 수를 나타내는 것은 아니지만 아래 코드에서 볼 수 있듯이 숫자형 데이터로 간주된다.

```
x = 0/0;
trace(x);         // NaN이 출력된다.
trace(typeof x);  // "number"가 출력된다.
```

NaN은 유한한 숫자 값이 아니므로 NaN과 NaN을 비교하더라도 그 두 값이 같지는 않다. NaN 값을 가지는 두 개의 변수가 있을 때 그 두 변수는 같은 값을 가진 것으로 간주되지 않는다(언뜻 보면 같아 보이더라도 말이다). 이러한 문제를 해결하기 위해 isNaN()이라는 내장 함수를 이용하여 어떤 변수에 NaN 값이 들어 있는지 조사할 수도 있다.

```
x = 12 - "this doesn't make much sense";  // x는 NaN이다.
trace(isNaN(x));                          // true가 출력된다.
```

## 표현 가능한 최소, 최대값: MIN_VALUE와 MAX_VALUE

액션스크립트에서는 꽤 넓은 범위의 숫자를 표기할 수 있긴 하지만, 그렇다고 해서 그 범위가 무한한 것은 아니다. 사용할 수 있는 가장 큰 값은 1.7976931348623157e+308이며 가장 작은 값은 5e-324이다. 물론 이러한 숫자를 직접 입력하거나 기억해 두는 것이 불편하므로, Number.MAX_VALUE와 Number.MIN_VALUE라는 특수 값을 사용하면 된다.

Number.MAX_VALUE는 계산 결과가 실제 표기 가능한 양의 정수인지 확인할 때 유용하다.

```
z = x*y;
if (z <= Number.MAX_VALUE && z >= -Number.MAX_VALUE) {
  // 유효한 숫자
}
```

Number.MIN_VALUE는 사용할 수 있는 양수의 최소값이라는 점을 기억해 두자. 음수이면서 가장 절대값이 큰 수는 -Number.MAX_VALUE이다.

## 무한대와 음의 무한대: Infinity와 -Infinity

계산 결과가 Number.MAX_VALUE보다 큰 경우에 액션스크립트에서는 그 계산 결과를 Infinity라는 키워드로 표시한다. 마찬가지로 계산 결과가 가장 작은 음수보다 더 작은 경우에는 -Infinity라는 값을 이용하여 그 결과를 표시한다. Infinity와 -Infinity는 리터럴 숫자 표현식으로 직접 사용할 수도 있다.

## 무리수

NaN, Infinity, -Infinity, Number.MAX_VALUE, Number.MIN_VALUE 외에도 액션스크립트에서는 Math 객체를 이용하여 아래와 같은 수학 상수들을 간편하게 이용할 수 있다.

```
Math.E        // 자연로그 밑인 E 값
Math.LN10     // 10의 자연로그 값
Math.LN2      // 2의 자연로그 값
Math.LOG10E   // 10을 밑으로 할 때 E의 로그 값
Math.LOG2E    // 2를 밑으로 할 때의 E의 로그 값
Math.PI       // 파이(3.1415926...)
Math.SQRT1_2  // 1/2의 루트 값
Math.SQRT2    // 2의 루트 값(1.4142135...)
```

위 상수들은 자주 쓰이는 무리수의 부동소수점 값을 기억하기 좋게 지정해놓은 것이다. 다른 객체 속성을 사용할 때와 똑같은 방법으로 위와 같은 무리수를 사용하면 된다.

```
area = Math.PI*(radius*radius);
```

지원되는 모든 상수 목록을 보고 싶다면 3부에서 Math 객체 부분을 참고하기 바란다.

# 숫자 계산

숫자와 연산자를 이용하여 수식을 만들거나 복잡한 계산을 할 때는 내장 함수를 호출하는 방식으로 숫자 계산을 처리할 수 있다.

## 연산자 이용

기본적인 계산(더하기, 빼기, 곱하기, 나누기)은 +, -, *, / 연산자를 이용하여 처리하면 된다. 이 연산자들은 어떤 리터럴이나 변수와도 사용할 수 있다. 수식은 〔표 5-1〕에 나온 것과 같은 연산자 우선 순위에 따라 계산된다. 예를 들면 곱셈은 덧셈보다 먼저 계산된다. 아래 선언문은 모두 수학 연산자를 바르게 사용한 예이다.

```
x = 3 * 5;            // x에 15를 대입한다.
x = 1 + 2 - 3 / 4;    // x에 2.25를 대입한다.

x = 56;
y = 4 * 6 + x;        // y에 80을 대입한다.
y = x + (x * x) / x;  // y에 112를 대입한다.
```

## 내장 수학 함수

복잡한 계산을 처리할 때는 다음과 같이 Math 객체의 내장 수학 함수를 사용한다.

```
Math.abs(x)       // x의 절대값
Math.min(x, y)    // x와 y중에서 더 작은 값
Math.pow(x, y)    // x의 y승
Math.round(x)     // x의 반올림값
```

수학 함수에서는 실수를 사용할 때와 마찬가지로 표현식에서 사용할 수 있는 값을 리턴한다. 예를 들어 6개의 면이 있는 주사위를 흉내내는 경우를 생각해 보자. 이 때 random() 함수를 이용하여 0과 1 사이에서 무작위로 부동소수점수를 하나 구할 수 있다.

```
dieRoll = Math.random();
```

그리고 나서 이 값에 6을 곱하면 0과 5.999... 사이의 부동소수점수가 된다. 여기에 1을 더하자.

```
dieRoll = dieRoll * 6 + 1;  // dieRoll을 1과 6.999... 사이의 값으로 만든다.
```

마지막으로 floor() 함수를 이용하여 이 숫자의 소수점 이하 부분을 버려서 정수로 만든다.

```
dieRoll = Math.floor(dieRoll);  // dieRoll을 1과 6 사이의 정수로 만든다.
```

위 작업을 하나의 표현식으로 바꾸면 주사위를 던지는 것은 다음과 같은 하나의 식으로 표현할 수 있다.

```
// dieRoll을 1과 6 사이의 정수로 만든다.
dieRoll = Math.floor(Math.random() * 6 + 1);
```

## 문자열

문자열은 텍스트 데이터(문자, 문장부호 및 기타 글자)를 다루는 데 쓰이는 데이터형이다. 문자열 리터럴은 큰따옴표로 둘러싸인 임의의 글자 조합이면 된다.

```
"asdfksldfsdfeoif"  // 별 뜻 없는 문자열
"greetings"         // 의미가 있는 문자열
"moock@moock.org"   // 광고용 문자열
"123"               // 숫자 같지만 사실 문자열이다.
'singles'           // 작은따옴표도 사용할 수 있다.
```

문자열 리터럴을 구성하는 법을 배우기 전에 문자열에서 어떤 문자를 사용할 수 있는지 알아보자.

## 문자 인코딩

다른 모든 컴퓨터 데이터와 마찬가지로 텍스트 문자도 숫자로 된 코드 상태로 컴퓨터 내부에 저장된다. 저장하는 과정에서 인코딩되고 화면에 표시할 때는 실제 문자와 숫자로 된 코드를 연결해주는 '문자 세트(character set)'를 통해 디코딩된다.

문자 세트는 언어나 알파벳에 따라 달라진다. 예전에 서양에서 나온 애플리케이션에서는 128개의 문자(영문 알파벳, 숫자와 기본적인 문장부호)만을 포함하는 ASCII 문자 세트를 이용했다. 하지만 요즘 나오는 애플리케이션에서는 ISO-8859로 알려져 있는 문자 세트의 집합을 이용한다. 각 ISO-8859 문자 세트는 표준 라틴 알파벳('A'에서 'Z'까지)과 특정 언어에서 필요한 다양한 글자들을 포함하고 있다. 액션스크립트에서는 기본 문자 세트로 Latin 1이라고 알려져 있는 ISO-8859-1을 사용한다.

Latin 1 문자 세트에는 대부분의 서유럽 언어(불어, 독일어, 이탈리아어, 스페인어, 포르투칼어 등)가 포함되어 있긴 하지만, 그리스어, 터키어, 슬로바키아어, 러시아어 같은 언어는 포함되어 있지 않다. 수백만 문자까지 지원되는 국제 표준 문자 인코딩인 유니코드는 액션스크립트에서 지원되지 않는다(유니코드를 지원하다 보면 플래시 플레이어의 크기가 너무 커지기 때문이다). 하지만 액션스크립트에서는 일본어 문자를 지원하기 위한 Shift-JIS를 보조 문자 세트로 지원한다. 액션스크립트에서 텍스트 작업을 할 때는 Latin 1이나 Shift-JIS에 있는 모든 문자를 사용할 수 있다.

유니코드 자체는 지원되지 않지만 표준 유니코드 이스케이프 시퀀스를 이용하면 Latin 1이나 Shift-JIS에 있는 문자를 모두 표현할 수 있다. 또한 유니코드 스타일 함수를 이용하여 문자열을 조작할 수도 있다. 따라서 적어도 이론적으로 본다면 나중에 플래시에서 유니코드를 지원하게 되더라도 지금 만든 코드를 그대로 사용할 수 있다.

'부록 B. Latin 1 문자 범주 및 키코드'에 각 문자의 유니코드 코드 포인트(각 문자의 유니코드 번호)가 수록되어 있다. 스크립트에서 그러한 코드 포인트를 이용하여 문자를 조작하는 법도 배울 것이다.

## 문자열 리터럴

문자열을 만드는 가장 일반적인 방법은 Latin 1이나 Shift-JIS 문자 세트에 들어있는 문자들을 작은따옴표 또는 큰따옴표로 표시하는 방법이다.

```
"hello"
'Nice night for a walk.'
"The equation is 12 + 4 = 16, which programmers see as 12 + 4 == 16."
```

큰따옴표를 사용하여 문자열을 시작하는 경우에는 반드시 큰따옴표로 문자열을 마쳐야 한다. 마찬가지로 작은따옴표로 시작한 문자열은 작은따옴표로 마쳐야 한다. 하지만 큰따옴표로 둘러싼 문자열에 작은따옴표가 들어가거나 반대로 작은따옴표로 둘러싼 문자열에 큰따옴표가 들어가는 것은 가능하다. 아래 예에 나와 있는 문자열은 문법적으로 문제가 없다.

```
"Nice night, isn't it?"                 // 작은따옴표가
                                        // 큰따옴표 사이에 들어간 경우
'I said, "What a pleasant evening!"'    // 큰따옴표가 작은따옴표
                                        // 사이에 들어간 경우
```

## 비어있는 문자열

가장 짧은 문자열은 아무런 문자도 들어있지 않은 비어있는 문자열이다.

```
""
''
```

비어있는 문자열은 어떤 변수에 문자열이 저장되어 있는지 확인할 때 매우 유용하다.

```
if (firstName == "") {
  trace("You forgot to enter your name!");
}
```

하지만 어떤 변수를 “”와 비교하는 것이 원하는 대로 작동하지 않는 경우도 있다. “”는 숫자 0 또는 부울값 false와 같은 것으로 간주되기 때문이다(〔표 3-1〕 및 〔표 3-3〕 참조). 따라서 실제로 비어있는 문자열을 조사할 때는 우선 변수의 데이터형이 문자열인지 확인해야 한다.

```
if (typeof firstName == "string" && firstName == "") {
  trace("You forgot to enter your name!");
}
```

## 이스케이프 시퀀스

큰따옴표로 둘러싼 리터럴에서는 작은따옴표를 사용할 수 있고 작은따옴표로 둘러싼 리터럴에서는 큰따옴표를 사용할 수 있다는 것은 이미 배웠다. 하지만 다음과 같이 둘 다 사용하고 싶다면 어떻게 해야 할까?

```
'I remarked "Nice night, isn't it?"'
```

물론 위와 같은 코드를 그대로 사용하면 오류가 발생한다. 인터프리터에서는 "isn't"에 들어 있는 작은따옴표가 있는 위치에서 문자열이 끝난다고 인식하기 때문이다. 결국 인터프리터에서는 위 코드를 다음과 같이 해석한다.

```
'I remarked "Nice night, isn'  // 나머지는 잘못된 문장이라고 인식한다.
```

작은따옴표로 둘러싼 문자열에서 작은따옴표를 써야 하는 경우에는 '이스케이프 시퀀스(escape sequence)'를 이용해야 한다.

이스케이프 시퀀스에서는 역슬래시 문자(\) 뒤에 원하는 문자를 나타내는 코드 또는 원하는 문자 자체를 적는 방식으로 리터럴 문자열을 표시한다. 작은따옴표와 큰따옴표에 해당하는 이스케이프 시퀀스는 다음과 같다.

```
\'
\"
```

따라서 앞에서 입력한 인사말을 제대로 사용하려면 다음과 같이 써야 한다.

```
'I remarked "Nice night, isn\'t it?"'     // 작은따옴표는 이스케이프
                                          // 시퀀스로 입력한다.
```

몇 가지 특수문자 또는 미리 예약된 문자를 표시하기 위해 사용하는 이스케이프 시퀀스를 〔표 4-1〕에 정리해 두었다.

〔표 4-1〕 액션스크립트 이스케이프 시퀀스

| 이스케이프 시퀀스 | 의미 |
|---|---|
| \b | 역스페이스 문자(ASCII 8) |
| \f | 폼 피드 문자(ASCII 12) |
| \n | 줄바꿈 문자(ASCII 10) |
| \r | 캐리지 리턴(CR) 문자(ASCII 13) |
| \t | 탭 문자(ASCII 9) |
| \' | 작은따옴표 |
| \" | 큰따옴표 |
| \\ | 역슬래시 문자: \가 이스케이프 시퀀스의 시작부호로 해석되는 것을 막기 위해 역슬래시를 쓸 때는 이와 같은 이스케이프 시퀀스를 이용한다. |

## 유니코드 스타일 이스케이프 시퀀스

Latin 1 및 Shift-JIS 문자 중에는 키보드로 입력할 수 없는 것도 있다. 문자열에서 이러한 문자를 사용하려면 유니코드 스타일 이스케이프 시퀀스를 사용해야 한다. 플래시에서 실제로 유니코드를 지원하는 것은 아니고 단지 문법을 흉내낼 뿐이라는 점을 기억해 두자.

유니코드 스타일 이스케이프 시퀀스는 역슬래시와 소문자 u(즉 \u)로 시작하며, 그 뒤에 유니코드 문자의 코드 포인트에 해당하는 네 자리 16진수가 들어간다.

```
\u0040  // @ 표시
\u00A9  // 저작권 표시
\u0041  // 대문자 "A"
```

코드 포인트는 유니코드 문자 세트에서 각 문자에 할당된 고유번호이다. Latin 1 문자에 대한 유니코드 코드 포인트는 부록 B에 수록되어 있다. Shift-JIS 코드 포인트는 유니코드 컨소시엄 사이트에서 찾을 수 있다.

http://www.unicode.org/Public/MAPPINGS/OBSOLETE/EASTASIA/JIS/SHIFTJIS.TXT

Latin 1 문자 세트에 있는 문자만을 사용한다면 표준 유니코드 이스케이프 시퀀스의 축약형을 사용해도 된다. 축약형 이스케이프 시퀀스는 \x 뒤에 그 문자의 Lain 1 인코딩을 나타내는 두 자리 16진수를 덧붙인 것이다. Latin 1 코드 포인트는 유니코드의 맨 앞에 있는 256개의 코드 포인트와 같으므로 부록 B에 있는 도표를 사용해도 되지만 아래 예제에서 볼 수 있듯이 그냥 앞에 있는 u00만 제거해도 된다.

```
\u0040  // 유니코드 이스케이프 시퀀스
\x40    // \x 축약형
\u00A9  // 유니코드 이스케이프 시퀀스
\xA9    // 이런 식으로 축약형을 만든다.
```

유니코드 이스케이프 시퀀스 외에도 조금 번거롭긴 하지만 내장 함수인 fromCharCode()를 이용하면 어떤 문자든지 문자열에 집어넣을 수 있다. 이 함수에 대한 설명은 '문자 코드 함수'에서 찾을 수 있다. 유니코드 이스케이프 시퀀스나 fromCharCode() 함수를 이용하는 방법은 모두 Latin 1 또는 Shift-JIS 문자 세트에 포함된 문자의 코드 포인트만을 사용할 수 있다는 점에 주의하자. 다른 코드 포인트를 입력하면 플래시 차기 버전에서 또 다른 유니코드 레파토리를 지원하게 되는 경우를 제외한다면 원하는 유니코드 문자가 입력되지 않는다.

## 문자열 조작

문자열을 조작하면 사용자가 입력한 내용을 확인하는 프로그램에서 단어 섞기 게임까지 모든 프로그램을 만들 수 있다. 조금만 노력하면 깔끔한 텍스트 효과와 같은 재미있는 것도 만들 수 있다.

문자열을 조작할 때는 연산자 또는 내장 함수를 사용한다. 문자열 연산자를 이용하면 여러 개의 문자열을 하나로 합치거나 두 문자열을 비교할 수 있다. 내장 함수를 이용하면 문자열의 속성과 내용을 조사하거나 문자열의 일부를 추출하거나, 문자의 코드 포인트를 검사하거나 코드 포인트로부터 문자를 만들 수도 있고, 문자열의 모든 문자를 대문자 또는 소문자로 바꾸거나 문자열로부터 변수나 속성 이름을 만드는 것도 가능하다.

## 문자열 병합

문자열을 하나로 묶는 작업(두 개 이상의 문자열로부터 새로운 문자열을 만드는 것)을 '문자열 병합(concatenation)'이라고 부른다. 이미 살펴본 것처럼 덧셈 연산자(+)를 이용하여 두 개의 문자열을 합칠 수 있다.

```
"Macromedia" + "Flash"
```

위 코드를 실행하면 "MacromediaFlash"라는 하나의 문자열이 생긴다. 해놓고 보니 중간에 한 칸 띄우는 것을 깜빡했다. 공백을 추가하려면 두 문자열 중 한 쪽에 공백을 추가하기만 하면 된다.

```
"Macromedia " + "Flash"    // "Macromedia Flash"가 된다.
```

하지만 이러한 방법은 그다지 실용적이지 못하다. 사실 회사나 제품 이름에 공백을 붙이는 경우는 거의 없기 때문이다. 따라서 위 방법 대신 가운데 공백 한 칸으로 된 문자열을 하나 더 추가하여 세 개의 문자열을 하나로 묶으면 된다.

```
"Macromedia" + " " + "Flash"   // "Macromedia Flash"가 된다.
```

공백 문자열(" ")은 비어있는 문자열과 다르다는 점에 주의하자. 비어있는 문자에서는 두 따옴표 사이에 아무런 문자도(공백 문자도) 없다.

문자열 데이터가 들어 있는 변수를 합칠 수도 있다. 다음과 같은 코드를 생각해 보자.

```
var company = "Macromedia";
var product = "Flash";

// sectionTitle을 "Macromedia Flash"로 설정한다.
var sectionTitle = company + " " + product;
```

첫째와 둘째 줄에서는 문자열 값을 변수에 저장하였다. 그리고 나서 그 변수 사이에 공백을 포함시켜 두 값을 합쳤다. 두 개의 값은 변수에 저장되어 있지만 나머지 하나(공백)는 문자열 리터럴이다. 하지만 이렇게 해도 된다. 종종 이러한 코드를 써야 하는 경우도 있다.

원래 있던 문자열에 새로운 문자를 덧붙여야 하는 경우가 있다. 예를 들어 환영 인사를 다음과 같이 고치는 경우를 살펴보자.

```
var greeting = "Hello";      // 환영 인사
greeting = greeting + "?";  // 특이한 환영인사: "Hello?"
```

위와 같은 코드만으로도 필요한 작업은 처리할 수 있지만, 둘째 줄에서 greeting 이라는 변수를 두 번이나 적어야 한다. 더 효율적인 코드를 만들고 싶다면 += 연산자를 이용하면 된다. += 연산자는 오른쪽에 있는 문자열을 왼쪽에 있는 문자열에 덧붙이는 연산자이다.

```
var greeting = "Hello";     // 환영 인사
greeting += "?";            // 특이한 환영인사: "Hello?"
```

플래시 4 문자열 병합 연산자(&)가 플래시 5에서는 다른 용도(비트 AND)로 쓰인다. 플래시 4의 .swf 파일로 내보낼 때는 문자열을 합칠 때 add 연산자를 사용해야 한다. 플래시 5에서 add는 하위 호완성을 위해 쓰일 뿐 + 연산자를 기본으로 써야 한다는 것에 주의하자.

## concat() 함수

concat() 함수는 +=와 마찬가지로 새로운 문자열을 원래 있는 문자열에 합치는 기능을 한다. concat()는 함수이기 때문에 아래와 같이 점 연산자를 이용해야 한다.

```
var product = "Macromedia".concat("Flash");

var sentence = "How are you";
var question = sentence.concat("?")
```

하지만 조심해야 할 것이 하나 있다. += 연산자와는 달리 concat() 함수는 그 함수가 쓰이는 문자 자체를 바꾸지는 않는다. 단지 합친 문자열 값을 리턴할 뿐이다. 그 값을 이용하려면 어떤 변수나 다른 데이터를 저장할 수 있는 곳에 대입해야 한다. 아래 코드를 자세히 살펴보고 += 연산자와 concat() 함수의 차이점을 이해해 보자.

```
var greeting = "Hello";
greeting.concat("?");
```

```
trace(greeting);  // "Hello"가 출력된다. greeting 변수는 변경되지 않는다.

finalGreeting = greeting.concat("?");
trace(finalGreeting);  // "Hello?"가 출력된다.
```

concat() 함수에서는 여러 개의 인자도 받아들인다(즉 여러 개의 문자열을 쉼표로 구분하여 인자로 넘기면 그 문자열들을 모두 합친다).

```
firstName = "Karsten";

// finalGreeting를 "Hello Karsten?"으로 설정한다.
finalGreeting = greeting.concat(" ", firstName, "?");
```

위 코드는 결국 아래 코드와 같은 결과를 가져온다.

```
finalGreeting = greeting;
finalGreeting += " " + firstName + "?";
```

## 문자열 비교

두 개의 문자열이 같은지 아닌지를 확인할 때는 동치 연산자(==)나 부등 연산자(!=)를 이용한다. 조건에 따라 실행되는 코드에서 종종 문자열을 비교하게 된다. 예를 들면 사용자가 비밀번호를 입력하면 입력된 문자열과 실제 비밀번호를 비교하는 경우에 문자열 비교를 하게 된다. 그러면 비교 결과에 따라 코드의 실행 여부가 결정된다.

### == 연산자 및 += 연산자

동치 연산자에서는 두 개의 피연산자(왼쪽에 있는 것과 오른쪽에 있는 것)를 사용한다. 피연산자로는 문자열 리터럴이나 변수, 배열의 원소, 객체의 속성 또는 문자열로 변환되는 표현식을 사용할 수 있다.

```
"hello" == "goodbye"       // 두 개의 문자열 리터럴 비교
userGuess == "fat-cheeks"  // 변수와 문자열 리터럴 비교
userGuess == password      // 두 개의 변수 비교
```

왼쪽에 있는 피연산자가 오른쪽에 있는 피연산자와 같은 문자를 같은 순서로 늘어놓은 문자열이라면 두 문자열은 같은 것으로 간주되며, 연산 결과는 부울형 값인 true가 된다. 대문자와 소문자는 문자 세트에서 서로 다른 코드 포인트를 가지므로 서로 다른 문자로 처리된다. 다음 비교 결과는 모두 false이다.

```
"olive-orange"== "olive orange"   // 같지 않다.
"nighttime" == "night time"       // 같지 않다.
"Day 1" == "day 1"                // 같지 않다.
```

문자열 비교의 결과는 부울 값인 true 또는 false이므로, 아래와 같이 조건문이나 순환문에서 테스트용 표현식으로 사용할 수 있다.

```
if (userGuess == password) {
  gotoAndStop("classifiedContent");
}
```

표현식(userGuess == password)이 true라면 gotoAndStop("classifiedContent");라는 선언문이 실행된다. 만약 그 값이 false라면 gotoAndStop("classifiedContent");라는 선언문을 실행하지 않고 넘어가게 된다.

부울 값에 대해서는 잠시 후에 자세히 배울 것이다. 그리고 조건문에 대해서는 '7장. 조건문'에서 자세히 다룬다.

두 개의 문자가 서로 다른지 확인할 때는 동치 연산자와 반대의 결과를 리턴하는 부등 연산자를 이용한다. 예를 들어 아래 코드는 각각 false와 true가 된다.

```
"Jane" != "Jane"  // 두 개의 문자열이 같으므로 false.
"Jane" != "Biz"   // 두 개의 문자열이 다르므로 true.
```

아래 예에서는 두 개의 문자열이 서로 다른 경우에만 작업을 처리한다.

```
if (userGender != "boy") {
  // 여학생 전용 코드
}
```

플래시 4의 .swf 파일로 내보낼 때는 문자열을 비교할 때 ==와 != 대신 eq와 ne를 사용해야 한다. eq와 ne가 하위 호환성을 위해 지원되긴 하지만 플래시 5에서는 ==와 != 연산자를 써야 한다.

## 문자 순서와 알파벳 순서 비교

문자 순서를 바탕으로 두 개의 문자열을 비교할 수도 있다. 각 문자에는 숫자로 된 코드 포인트가 할당되어 있으며, 각 코드 포인트가 문자 세트에서 숫자 순서대로 정렬되어 있다는 것은 이미 배워서 알고 있을 것이다. 비교 연산자인 >, >=, <, <= 를 이용하여 어떤 문자가 더 앞에 있는지 알 수 있다. 모든 비교 연산자는 두 개의 피연산자를 비교한다.

```
"a" < "b"
"2" > "&"
"r" <= "R"
"$" >= "@"
```

동치 및 부등 표현식과 마찬가지로 비교 표현식의 결과는 피연산자의 관계에 따라 부울형 값이 true 또는 false로 나온다. 문자열 값을 가지는 것은 모두 피연산자로 사용할 수 있다.

Latin 1 문자 세트에서는 'A'에서 'Z'와 'a'에서 'z'가 알파벳 순서로 묶여 있기 때문에, 둘 중 어떤 글자가 알파벳 순으로 더 앞에 있는지 알아낼 때 문자순 비교를 사용한다. 하지만 Latin 1 문자 세트에서는 모든 소문자가 대문자 앞에 있다. 이러한 점을 깜빡 잊어버리면 다음과 같은 결과가 이해되지 않을 수도 있다.

```
"Z" < "a"       // true
"z" < "a"       // false
"Cow" < "bird"  // true
```

이제 각 비교 연산자를 자세히 살펴보자. 아래 설명에서 '비교 문자'는 두 개의 피연산자에서 제일 처음 나오는 서로 다른 문자를 뜻한다.

### > 연산자

왼쪽 피연산자의 비교 문자가 Latin 1이나 Shift-JIS 문자 순서로 볼 때 오른쪽 피연산자의 비교 문자보다 뒤에 있으면 true를 리턴한다. 두 피연산자가 똑같다면 false를 리턴한다.

```
"b" > "a"      // true
"a" > "b"      // false
"ab" > "ac"    // false (두 번째 문자가 비교 문자)
```

```
"abc" > "abc"  // false (똑같은 문자열)
"ab" > "a"     // true (b가 비교 문자)
"A" > "a"      // false (문자 순서상 "A"가 "a"보다 앞에 있다)
```

### >= 연산자

왼쪽 피연산자의 비교 문자가 오른쪽 피연산자의 비교 문자보다 뒤에 있거나 두 문자열이 똑같으면 true를 리턴한다.

```
"b" >= "a"  // true
"b" >= "b"  // true
"b" >= "c"  // false
"A" >= "a"  // false ("A"와 "a"의 코드 포인트가 다르다)
```

### < 연산자

왼쪽 피연산자의 비교 문자가 Latin 1이나 Shift-JIS 문자 순서로 볼 때 오른쪽 피연산자의 비교 문자보다 앞에 있으면 true를 리턴한다. 두 피연산자가 똑같다면 false를 리턴한다.

```
"a" < "b"      // true
"b" < "a"      // false
"az" < "aa"    // false (두 번째 문자가 비교 문자)
```

### <= 연산자

왼쪽 피연산자의 비교 문자가 오른쪽 피연산자의 비교 문자보다 앞에 있거나 두 문자열이 똑같으면 true를 리턴한다.

```
"a" <= "b"  // true
"a" <= "a"  // true
"z" <= "a"  // false
```

Latin 1 문자 세트에서 알파벳이 아닌 문자의 순서가 궁금하다면 부록 B를 참조하기 바란다.

아래 코드는 어떤 문자가 라틴 알파벳 문자인지 확인하는 예이다.

```
var theChar = "w";

if ((theChar >= "A" && theChar <= "Z") || (theChar >= "a" && theChar
<="z")) {
  trace("The character is in the Latin alphabet.");
}
```

논리 OR 연산자(||)를 이용하여 한꺼번에 두 가지 조건을 조사하는 방법을 익혀두자. OR 연산자에 대해서는 '5장. 연산자'에서 자세히 다룬다.

플래시 4 .swf 파일로 저장할 때는 gt, ge, lt, le 문자열 비교 연산자를 이용한다. 예전에 쓰이던 연산자도 하위 호환성을 위해 지원하긴 하지만 플래시 5에서는 >, >=, <, <= 연산자를 쓰는 것이 좋다.

## 내장 문자열 함수

concat() 함수를 제외하면 지금까지 사용한 문자열과 관련된 도구는 모두 연산자이다. 내장 함수를 이용하여 더 발전된 문자열 조작을 처리하는 법을 알아보자.

문자열에 대해 내장 함수를 사용하려면 아래와 같은 형태의 함수 호출을 수행해야 한다.

```
string.functionName(arguments)
```

예를 들어 myString이라는 문자열에 대해 charAt() 함수를 실행하려면 다음과 같이 한다.

```
myString.charAt(2)
```

문자열 함수에서는 원본 문자열과 관련된 데이터를 리턴한다. 이러한 리턴 값을 다른 변수나 객체 속성에 저장하여 나중에 사용할 수 있다.

```
thirdCharacter = myString.charAt(2);
```

## 문자 인덱스

많은 문자열 함수에서 문자의 인덱스(문자열의 첫 문자로부터의 거리로, 1이 아닌 0에서 시작한다)를 이용한다. 첫 문자는 0, 둘째 문자는 1, 셋째 문자는 2, 이런 식으로 인덱스가 붙게 된다. 예를 들어 "red"라는 문자열에서 r의 인덱스는 0, e의 인덱스는 1, d의 인덱스는 2이다.

문자열의 일부를 나타낼 때는 인덱스를 이용한다. 예를 들어 인터프리터에 '인덱스 3부터 인덱스 7까지 있는 문자는?' 또는 '인덱스가 5인 문자는?' 같은 질문을 던질 수도 있다.

## 문자열 조사

내장 속성인 length 또는 charAt(), indexOf(), lastIndexOf() 함수와 같은 내장 함수를 이용하여 문자열을 검사하거나 탐색할 수 있다.

### length 속성

length 속성은 문자열에 있는 문자의 개수(길이)를 알려주는 값이다. 함수가 아니라 속성이므로 그 값을 참조할 때는 괄호나 인자를 쓰지 않는다. 몇 가지 문자에 대한 length 값을 알아보자.

```
"Flash".length          // 5
"skip intro".length     // 10 (공백도 문자이므로 길이에 포함된다.)
"".length               // 비어있는 문자열이므로 길이는 0이다.

var axiom = "all that glisters will one day be obsolete";
axiom.length            // 42
```

문자 인덱스는 0에서 시작하므로 마지막 문자의 인덱스는 언제나 문자열의 length 속성에서 1을 뺀 숫자이다.

문자열의 length 속성을 읽을 수는 있지만 다른 값으로 설정하는 것은 불가능하다. 다음과 같은 식으로 문자열의 길이를 늘릴 수는 없다.

```
axiom.length = 100;  // 괜찮은 생각이긴 한데 이러한 코드는 사용할 수 없다.
```

플래시 4 .swf 파일로 저장할 때는 아래에 나온 것처럼 length() 함수를 이용해야 한다. 이 함수는 하위 호환성을 위해 제공되는 함수이므로 플래시 5에서는 length 속성을 이용하는 것이 바람직하다.

다음과 같이 플래시 4의 length() 함수를 이용하여 "obsolete"[1]라는 단어의 문자 개수를 출력할 수도 있다.

```
trace (length("obsolete"));  // 8이 출력된다.
```

## charAt() 함수

다음과 같이 쓰이는 charAt() 함수를 이용하면 어떤 인덱스 위치에 있는 문자라도 알아낼 수 있다.

```
string.charAt(index)
```

여기서 string 자리에는 임의의 리터럴 문자열 또는 문자열을 포함하는 인식자[2]가 들어갈 수 있다. index는 구하고자 하는 문자의 위치를 나타내는 정수, 또는 정수 값이 나오는 표현식이다. index 값은 0에서 string.length - 1 사이에 있다. 만약 index가 이 범위 안에 들어오지 않는다면 비어있는 문자열이 리턴된다. 아래에 몇 가지 예가 나와 있다.

```
"It is 10:34 pm".charAt(1)  // 두 번째 문자인 "t"
var country = "Canada";
country.charAt(2);          // 세 번째 문자인 "n"
var x = 4;
fifthLetter = country.charAt(x);                  // fifthLetter는 "d"
lastLetter = country.charAt(country.length - 1);  // lastLetter는 "a"
```

## indexOf() 함수

문자열에서 어떤 문자를 찾을 때는 indexOf() 함수를 쓴다. 검색하려는 문자열에 원하는 문자들이 있으면 indexOf()에서는 그 문자들이 처음 나오는 인덱스(위

1) 역자주: "obsolete"는 '쓰이지 않는' 이라는 의미를 가진다. 여기서는 length() 함수가 플래시 5 이후에는 쓰이지 않는다는 것을 의미하기 위해 이 단어를 사용했다.

2) 역자주: 여기서 인식자는 변수, 배열 원소, 객체 속성과 같은 것을 의미한다.

치)를 리턴한다. 만약 원하는 문자가 없다면 -1을 리턴한다. indexOf()은 일반적으로 다음과 같은 형태로 쓰인다.

```
string.indexOf(character_sequence, start_index)
```

여기서 string 자리에는 리터럴 문자열 값이나 문자열이 저장된 인식자를 사용한다. character_sequence는 검색하고자 하는 문자열이며 문자열 리터럴이나 문자열을 사용하면 된다. start_index는 검색을 시작할 위치의 인덱스이다. start_index를 생략하면 string의 시작 부분부터 검색을 시작한다.

indexOf()를 이용하여 문자열에 W라는 문자가 있는지 조사해 보자.

```
"GWEN!".indexOf("W");  // 1을 리턴한다.
```

W는 "GWEN!"의 두 번째 문자이므로 W의 인덱스인 1이 리턴된다. 문자 인덱스는 0부터 시작되므로 두 번째 문자의 인덱스가 1이 된다는 점을 기억해 두자.

소문자 w를 찾으면 어떻게 될까?

```
"GWEN!".indexOf("w");  // -1을 리턴한다.
```

"GWEN!"에는 w가 없기 때문에 indexOf() 함수에서 -1을 리턴한다. 대문자와 소문자는 서로 다른 문자로 간주되기 때문에 indexOf()는 대소문자를 구분한다.

이메일 주소에 @ 표시가 들어있는지 확인하는 코드를 만들어 보자.

```
var email = "daniella2dancethenightaway.ca";  // 시프트키를 누르지 않은 경우

// @ 표시가 없으면 formStatus 텍스트 필드를 통해 사용자에게 주의를 준다.
if (email.indexOf("@") == -1) {
  formStatus = "The email address is not valid.";
}
```

하나의 문자만을 찾을 수 있는 것은 아니다. 문자열에 들어 있는 여러 개의 문자를 검색할 수도 있다. 회사 주소에서 "Canada"를 검색해 보자.

```
var iceAddress = "St. Clair Avenue, Toronto, Ontario, Canada";
iceAddress.indexOf("Canada");  // "C"의 인덱스인 36을 리턴한다.
```

indexOf() 함수에서는 "Canada"에서 첫 문자의 위치를 리턴한다. 이제 iceAddress.indexOf("Canada")의 리턴 값을 -1과 비교하여 회사의 국적을 저장하는 변수에 그 결과를 저장하자.

```
var isCanadian = iceAddress.indexOf("Canada") != -1;
```

만약 iceAddress.indexOf("Canada")가 -1이 아니면(즉 "Canada"를 찾을 수 있는 경우에는) iceAddress.indexOf("Canada") != -1의 값은 true이고 iceAddress.indexOf("Canada")가 -1이면(즉 "Canada"를 찾을 수 없으면), iceAddress.indexOf("Canada") != -1의 값은 false가 된다. 그리고 나서 이렇게 나온 부울 값을 isCanadian이라는 변수에 저장하여 북아메리카에 있는 국가별 우편 폼을 만들 수 있다.

```
if (isCanadian) {
  mailDesc = "Please enter your postal code.";    // 캐나다 우편 번호
} else {
  mailDesc = "Please enter your zip code.";       // 미국 우편 번호
}
```

indexOf() 함수는 문자열에서 추출할 부분을 찾을 때도 유용하다. substring() 함수에 대해 배울 때 어떤 식으로 쓰이는지 알아보자.

## lastIndexOf() 함수

indexOf() 함수는 문자 시퀀스가 처음으로 나오는 위치를 리턴한다. lastIndexOf() 함수는 문자 시퀀스가 마지막으로 나오는 위치를 리턴하고 시퀀스를 찾지 못하는 경우에는 -1을 리턴한다. lastIndexOf() 함수의 일반적인 형식은 indexOf() 함수의 형식과 똑같다.

```
string.lastIndexOf(character_sequence, start_index)
```

유일한 차이점은 lastIndexOf()에서는 문자열을 뒤에서부터 검색하기 때문에, start_index가 검색 대상 중에서 제일 왼쪽이 아니라 제일 오른쪽에 있는 문자라는 점이다. start_index를 생략하면 string.length - 1(문자열의 마지막 글자)이 기본값으로 쓰인다.

예를 들면 다음과 같이 사용할 수 있다.

```
paradox = "pain is pleasure, pleasure is pain";
paradox.lastIndexOf("pain");      // 30을 리턴한다. indexOf()를 사용하면
                                  // 0을 리턴한다.
```

아래 코드에서는 "pain"이 두 번째 나오는 위치 앞에서부터 검색을 시작하기 때문에 0("pain"이라는 단어가 처음 나오는 위치의 인덱스)이 리턴된다.

```
paradox.lastIndexOf("pain",29);  // 0이 리턴된다.
```

### 정규 표현식

정규 표현식(텍스트 데이터에서 패턴을 찾아내는 매우 강력한 도구)은 액션스크립트에서 지원되지 않는다.

## 문자열 추출

긴 문자열에 자주 사용하고자 하는 문자 시퀀스가 포함되는 경우가 있다. 예를 들어 "Steven Sid Mumby"라는 문자열에서 성 부분인 "Mumby"만 추출하고 싶은 경우를 생각해 보자. 문자열의 일부(부분 문자열, substring)을 추출할 때는 substring(), substr(), splice(), split()이라는 함수를 사용한다.

### substring() 함수

첫 문자와 마지막 문자의 인덱스를 기준으로 문자 시퀀스를 추출할 때는 substring() 함수를 이용한다. substring() 함수는 다음과 같은 형식으로 사용한다.

```
string.substring(start_index, end_index)
```

여기서 string은 문자열이 들어 있는 리터럴 문자열 값 또는 문자열이 포함된 인식자, start_index는 부분 문자열의 첫 글자의 인덱스, end_index는 추출하고자 하는 부분 문자열의 마지막 글자 다음 글자의 인덱스이다. end_index를 입력하지 않으면 string.length 값이 기본값으로 쓰인다.

```
var fullName = "Steven Sid Mumby";
middleName = fullName.substring(7, 10);  // middleName에 "Sid"를 대입한다.
firstName = fullName.substring(0, 6);    // firstName에 "Steven"을 대입한다.
lastName = fullName.substring(11);       // lastName에 "Mumby"를 대입한다.
```

하지만 실제 프로그래밍을 할 때는 이름, 중간 이름, 성이 어디서 시작하고 어디서 끝나는지 알 수 없기 때문에, 공백 문자와 같은 '구분자(delimiter)'를 이용하여 단어가 끊기는 부분을 찾아내야 한다. 아래 코드에서는 이름에서 마지막 공백 문자를 검색하여 그 뒤에 있는 부분이 성이라고 가정한다.

```
fullName = "Steven Sid Mumby";
lastSpace = fullName.lastIndexOf(" "); // 10을 리턴한다.

// 11에서 문자열의 끝까지가 성을 나타내는 부분이다.
lastName = fullName.substring(lastSpace+1);
trace ("Hello Mr. " + lastName);
```

start_index가 end_index보다 크면 함수를 실행하기 전에 자동으로 두 값을 서로 바꾼다. 아래와 같이 함수를 호출하면 결과는 똑같지만, substring() 함수를 사용할 때 인덱스를 바꿔 쓰면 나중에 코드를 읽기 힘들어지므로 인덱스를 제 순서대로 쓰는 습관을 기르는 것이 좋다.

```
fullName.substring(4, 6);  // "en"을 리턴한다.
fullName.substring(6, 4);  // "en"을 리턴한다.
```

## substr() 함수

substr() 함수는 시작 인덱스와 부분 문자열의 길이를 이용하여 문자 시퀀스를 추출한다(substring()에서는 시작 및 끝 인덱스를 사용한다). substr()의 일반적인 형식은 다음과 같다.

```
string.substr(start_index, length)
```

여기서 string은 리터럴 문자열 값 또는 문자열이 저장된 인식자이며, start_index는 부분 문자열의 첫 문자의 인덱스, length는 부분 문자열의 길이(start_index에서 시작하여 오른쪽으로 세는 길이)이다. length를 생략하면 start_index에서 시작하여 원본 문자열의 마지막 문자까지 추출한다. 이 함수를 사용한 예는 다음과 같다.

```
var fullName = "Steven Sid Mumby";
middleName = fullName.substr(7, 3);    // middleName에 "Sid"를 대입한다.
firstName = fullName.substr(0, 6);     // firstName에 "Steven"을 대입한다.
lastName = fullName.substr(11);        // lastName에 "Mumby"를 대입한다.
```

start_index에 음수를 사용하면 문자열의 끝을 기준으로 시작점을 지정할 수 있다. 마지막 문자는 -1, 마지막에서 두 번째 문제는 -2, 이런 식으로 시작점을 지정하면 된다. 따라서 앞에 있는 세 개의 substr() 예제는 다음과 같이 고쳐 쓸 수 있다.

```
middleName = fullName.substr(-9, 3);  // middleName에 "Sid"를 대입한다.
firstName = fullName.substr(-16, 6);  // firstName에 "Steven"을 대입한다.
lastName = fullName.substr(-5);       // lastName에 "Mumby"를 대입한다.
```

하지만 length에는 음수를 사용할 수 없다.

플래시 4의 substring() 함수와 가장 비슷한 플래시 5 함수는 substr()이다. 플래시 4의 substring() 함수에서는 시작 인덱스와 길이를 이용하여 부분 문자열을 추출하기 때문이다.

## slice() 함수

substring()과 마찬가지로 slice()에서도 시작 인덱스와 끝 인덱스를 이용하여 문자 시퀀스를 추출한다. substring()에서는 원본 문자열의 시작 부분을 기준으로 인덱스를 지정해야 하지만, slice()에서는 문자열의 시작이나 끝을 기준으로 인덱스를 지정할 수 있다.

slice() 함수의 형식은 다음과 같다.

```
string.slice(start_index, end_index)
```

여기서 string은 리터럴 문자열 값이나 문자열이 저장된 인식자이며, start_index는 부분 문자열에 속하는 첫째 문자의 인덱스이다. start_index가 양수이면 일반적인 문자 인덱스가 되지만, 음수이면 문자열의 끝부터 세는 문자 인덱스가 된다(즉 string.length + start_index). 마지막으로 end_index는 부분 문자열의 마지막 문자 다음 문자의 인덱스이다. end_index를 적어주지 않으면 string.length가 기본값으

로 사용된다. end_index가 음수이면 문자열의 끝부터 세는 문자 인덱스가 된다(즉 string.length + start_index).

slice() 함수를 사용할 때 인덱스를 양수로 사용하면 substring()을 사용하는 것과 똑같이 작동한다. 음수 인덱스를 사용하더라도 순서가 거꾸로 된 문자열, 즉 end_index에서 start_index 방향으로 가는 문자열이 나오는 것이 아니라는 점을 기억해 두자. 음수 인덱스를 사용하면 단지 원본 문자열에서 인덱스를 세는 기준점이 문자열 끝으로 바뀔 뿐이다. 또한 문자열의 마지막 문자의 인덱스는 -1이며, end_index 인자는 원하는 부분 문자열의 마지막 글자 다음 글자의 인덱스이므로, 음수로 된 end_index를 사용하면 원본 문자열의 마지막 문자를 가리킬 수 없다. 아래 예에서 음수 인덱스를 사용하여 부분 문자열을 추출하는 법을 자세히 살펴보자.

```
var fullName = "Steven Sid Mumby";
middleName = fullName.slice(-9, -6);  // middleName에 "Sid"를 대입한다.
firstName = fullName.slice(-16, -10); // firstName에 "Steven"을 대입한다.
lastName = fullName.slice(-5, -1);    // lastName에 "Mumb"을 대입한다
                                      // 실제 원하는 문자열은 아니지만
                                      // 음수 end_index를 사용할 때는
                                      // 마지막 문자를 가리킬 수 없다.
lastName = fullName.slice(-5, 16)     // lastName에 "Mumby"를 대입한다.
                                      // 음수 인덱스와 양수 인덱스를
                                      // 섞어 써도 된다.
```

## split() 함수

지금까지 배운 문자열 추출 함수는 한 번에 하나의 문자 시퀀스만 추출할 수 있다. 한 번에 여러 개의 부분 문자열을 추출할 때는 split() 함수를 사용하면 된다(split() 함수에서는 배열을 사용해야 하므로 잠시 '11장. 배열'로 건너가서 배열에 대한 내용을 공부하고 다시 이 부분을 읽는 것도 좋다).

split() 함수는 하나의 문자열을 여러 개의 부분 문자열로 나누어서 각 부분 문자열을 배열로 저장한다. split() 함수는 다음과 같은 형식으로 쓰인다.

```
string.split(delimiter)
```

여기서 string은 리터럴 문자열 값이나 문자열이 저장된 인식자이며, delimiter는 string을 갈라낼 기준이 되는 문자 또는 문자열이다. 일반적으로 쉼표, 공백, 탭 등을 구분자로 사용한다. 쉼표를 기준으로 문자열을 분리할 때는 다음과 같이 하면 된다.

```
theString.split(",")
```

split() 함수를 이용하면 하나의 문장을 단어별로 분리하는 작업을 간단하게 처리할 수 있다. substring(), substr() 및 slice() 함수에서는 "Steven Sid Mumby"라는 문자열에서 각 단어를 일일이 수동으로 뽑아내야 한다. 하지만 split() 함수를 사용하고 공백("")을 delimiter로 지정하면 다음과 같이 간단하게 처리할 수 있다.

```
var fullName = "Steven Sid Mumby";
var names = fullName.split(" ");  // 정말 쉽다.

// 배열에 있는 이름을 각 변수에 대입한다.
firstName  = names[0];
middleName = names[1];
lastName   = names[2];
```

## 문자열 추출 성능

플래시 5에 있는 substr()과 slice() 함수는 사실 플래시 4에 있는 substring() 함수에 껍데기만 덧붙인 함수이므로 실행 속도가 약간 느리다. 속도 차이는 수 밀리초 단위에 불과하지만 문자열 작업을 많이 할 경우에는 눈에 띄게 속도가 느려질 수도 있다. 같은 작업을 여러 번 반복해야 한다면 substring() 함수를 사용하는 것이 가장 빠르다.

```
fullName = "Steven Sid Mumby";
// 플래시 5의 substr() 함수를 이용하여 middleName에 "Sid"를 대입한다.
middleName = fullName.substr(7, 3);

// 플래시 4의 substring() 함수를 이용하여 middleName에 "Sid"를 입력한다.
// 플래시 4의 substring()에서는 문자 인덱스가 1에서 시작한다는 점을 주의하자.
middleName = substring(fullname, 8, 3);
```

## 문자 검색과 문자 추출 결합

지금까지 문자열에 있는 부분 문자열을 찾는 방법과 추출하는 방법을 배웠다. 이 두 작업을 결합해서 쓰면 매우 강력한 도구가 된다.

지금까지 다룬 대부분의 예제에서는 다음과 같이 리터럴 표현식을 인자로 사용한다.

```
var msg = "Welcome to my website!";
var firstWord = msg.substring(0, 7);  // 0과 7은 숫자 인터럴
```

위 예제는 substring() 함수를 이용하는 방법을 제대로 보여주기는 하지만, 실전에서는 substring() 함수를 저런 식으로 사용하는 경우가 별로 없다. 대개의 경우 문자열의 내용을 미리 알 수 없기 때문에 인자를 그때 그때 만들어서 써야 한다. 예를 들어 "인덱스 0부터 인덱스 7까지의 부분 문자열"이라는 정적인 주문보다는 "첫째 문자부터 문자열의 첫째 공백까지의 부분 문자열"과 같은 동적인 주문을 더 자주 사용하게 마련이다. 이와 같은 방법을 이용하면 문자열의 내용을 미리 알지 못해도 상관없다. 아래 예에서는 substring()과 indexOf()를 결합하여 msg라는 변수의 첫째 단어를 추출한다.

```
var firstWord = msg.substring(0, msg.indexOf(" "));
```

msg.indexOf(" ") 라는 표현식은 msg에 있는 첫째 공백 문자의 인덱스를 리턴한다. 이와 같은 방법을 이용하면 공백의 위치에 구애받지 않고 원하는 작업을 할 수 있다. 즉 프로그램이 실행되는 중간에 내용이 변경되는 문자열을 가지고 작업할 수 있으며 프로그래머가 직접 문자 개수를 세지 않아도 되므로 오류가 발생할 수 있는 위험을 사전에 방지할 수 있다.

문자열 조사와 문자열 추출의 조합은 사실 무궁무진하다. [예제 4-2]에서는 문자 인덱스를 직접 코드에 입력할 필요 없이 msg 변수의 두 번째 단어를 추출할 수 있다. 우리말로 표현하자면 'msg에서 처음 나오는 공백 뒤에 있는 글자부터 두 번째 나오는 공백 앞에 있는 글자까지의 부분 문자열을 추출' 하는 작업을 하는 것이다. 여기서는 첫 번째와 두 번째 공백 위치를 변수에 저장하여 작업 과정을 더 보기 쉽도록 만들었다.

*[예제 4-2] 문자열의 두 번째 단어 추출하기*

```
var msg = "Welcome to my website!";
firstSpace = msg.indexOf("");                    // 첫 번째 공백
secondSpace = msg.indexOf("", firstSpace + 1);  // 두 번째 공백

// 두 번째 단어를 추출한다.
var secondWord = msg.substring(firstSpace + 1, secondSpace);
```

## 대소문자 변환

내장 함수인 toUpperCase()와 toLowerCase()를 이용하여 문자열을 모두 대문자 또는 모두 소문자로 변환할 수 있다. 이 함수들은 보통 문자열을 특정 형식으로 화면에 표시하거나 대소문자와 상관없이 두 개의 문자열을 비교할 때 많이 쓰인다.

### toUpperCase() 함수

toUpperCase() 함수는 문자열에 있는 모든 문자를 대문자로 바꾸고 그렇게 변환한 문자열을 리턴한다. 주어진 문자에 대한 대문자가 존재하지 않는 경우에는 그대로 남겨 둔다. toUpperCase()는 일반적으로 다음과 같은 형식으로 쓰인다.

```
string.toUpperCase()
```

여기서 string은 임의의 리터럴 문자열 값 또는 문자열이 저장된 인식자이다.

```
"listen to me".toUpperCase();       // "LISTEN TO ME"라는 문자열이 생긴다.
var msg1 = "Your Final Score: 234";
var msg2 = msg1.toUpperCase();   // "YOUR FINAL SCORE: 234"를 msg2에 저장
```

toUpperCase()를 호출할 때 사용한 문자열 자체는 변환되지 않는다는 점에 유의하자. 단지 대문자로 이루어진 문자열 복사본을 리턴할 뿐이다. 아래 예에서 그 차이점을 확인할 수 있다.

```
var msg = "Forgive me, I forgot to bring my spectacles.";
msg.toUpperCase();
trace(msg);// "Forgive me, I forgot to bring my spectacles."가 출력된다.
          // toUpperCase()를 호출하더라도 msg 변수는 변경되지 않는다.
```

## toLowerCase() 함수

toLowerCase() 함수는 문자열에 있는 모든 대문자를 소문자로 바꾼다.

```
// normal을 "this sentence has mixed caps!"로 바꾼다.
normal = "ThiS SenTencE Has MixED CaPs!".toLowerCase();
```

대소문자를 구분하지 않고 두 개의 문자열을 비교할 때는 다음과 같이 두 문자열을 모두 대문자 또는 소문자로 바꿔서 비교해야 한다.

```
if (userEntry.toLowerCase() == password.toLowerCase()) {
  // 비밀 정보를 이용할 수 있다.
}
```

〔예제 4-3〕의 프로그램은 대소문자 변환과 문자열 함수를 이용하여 텍스트 필드에 있는 텍스트를 애니메이션처럼 움직이게 해주는 프로그램이다. 이 프로그램을 이용하려면 msgOutput이라는 텍스트 필드가 레이어 위에 있고 scripts 레이어에 〔예제 4-3〕의 코드가 들어 있는 세 개의 프레임이 있는 무비가 필요하다.

*〔예제 4-3〕 대소문자 애니메이션*

```
// 1번 프레임의 코드
var i = 0;
var msg = "my what fat cheeks you have";

function caseAni () {
  var part1 = msg.slice(0, i);
  var part2 = msg.charAt(i);
  var part2 = part2.toUpperCase();
  var part3 = msg.slice(i+1, msg.length);
  msg = part1 + part2 + part3;
  msgOutput = msg;
  msg = msg.toLowerCase();
  i++;

  if (i > (msg.length - 1)) {
    i=0;
  }
}
```

```
// 2번 프레임의 코드
caseAni();

// 3번 프레임의 코드
gotoAndPlay(2);
```

# 문자 코드 함수

'유니코드 스타일 이스케이프 시퀀스' 절에서 이스케이프 시퀀스 형태로 문자를 삽입하는 방법을 배웠다. 이외에도 액션스크립트에서는 문자열에 있는 문자 코드를 다루기 위한 방법으로 fromCharCode()와 charCodeAt()이라는 두 개의 함수를 지원한다.

## fromCharCode() 함수

fromCharCode() 함수를 호출하여 임의의 문자 또는 문자열을 만들 수 있다. 다른 문자열 함수와는 달리 fromCharCode() 함수를 사용할 때는 문자열 리터럴이나 문자열을 포함하는 인식자에 대해 호출하지 않고 아래에서 볼 수 있듯이 String이라는 특별한 객체의 메소드로 호출한다.

```
String.fromCharCode(code_ point1, code_ point2, ...)
```

fromCharCode()를 호출할 때는 언제나 String.fromCharCode로 시작한다. 또한 한 개 이상의 코드 포인트(만들고자 하는 문자를 나타내는 코드 포인트)를 인자로 사용한다. 유니코드 스타일 이스케이프 시퀀스와는 달리 fromCharCode()를 호출할 때는 코드 포인트를 16진수가 아닌 10진수로 표기한다. 따라서 16진수에 익숙하지 않다면 유니코드 스타일 이스케이프 시퀀스보다는 fromCharCode() 함수를 사용하는 것이 더 쉽다. 이 함수는 다음과 같이 사용하면 된다.

```
// lastName을 "moock"로 설정한다.
lastName = String.fromCharCode(109, 111, 111, 99, 107);

// 유니코드 스타일 이스케이프 시퀀스를 이용하면 다음과 같이 입력하면 된다.
lastName = "\u006D\u006F\u006F\u0063\u006B"  ;
```

```
// 저작권 표시("ⓒ 2001")는 다음과 같이 만들 수 있다.
copyNotice = String.fromCharCode(169) + "2001";
```

플래시 4 .swf 파일로 내보낼 때에는 플래시 4에서 쓰이던 문자 생성 함수인 chr() 또는 mbchr() 함수를 써야 한다. 이 함수도 하위호환성을 위해 지원하긴 하지만 플래시 5에서는 fromCharCode()를 사용하는 것이 바람직하다.

## charCodeAt() 함수

문자열에 있는 임의의 문자의 코드 포인트를 알고 싶다면 아래와 같은 형식으로 쓰이는 charCodeAt( ) 함수를 사용하면 된다.

```
string.charCodeAt(index)
```

여기서 string은 리터럴 문자열 값이나 문자열이 저장된 인식자이며, index는 코드 포인트를 알아내고자 하는 문자의 위치를 가리킨다. charCodeAt( ) 함수는 index 위치에 있는 문자의 유니코드 코드 포인트에 해당하는 10진수를 리턴한다.

```
var msg = "A is the first letter of the Latin alphabet.";
trace(msg.charCodeAt(0));  // 65가 출력된다("A"의 코드 포인트).
trace(msg.charCodeAt(1));  // 32가 출력된다(스페이스의 코드 포인트).
```

charCodeAt( ) 함수는 보통 키보드만으로는 입력할 수 없는 문자를 이용하여 문자열을 처리할 때 쓰인다. 아래 예제에서는 주어진 문자가 저작권 표시인지 확인한다.

```
msg = String.fromCharCode(169) + "2000";
if (msg.charCodeAt(0) == 169) {
  trace("The first character of msg is a copyright symbol.");
}
```

플래시 4 .swf 파일로 내보낼 때는 ord()와 mbord()라는 플래시 4 코드 포인트 함수를 써야 한다. 이 함수들도 하위 호환성을 위해 지원하긴 하지만 플래시 5에서는 charCodeAt()를 쓰는 것이 바람직하다.

## eval을 이용한 문자열에 있는 코드 실행

eval() 함수는 액션스크립트에서 문자열을 인식자로 변환하는 함수이다. 하지만 액션스크립트의 eval() 함수를 제대로 이해하려면 자바스크립트의 eval() 함수가 작동하는 방법을 먼저 이해하는 것이 좋다. 자바스크립트에서 eval()은 임의의 문자열을 코드 블록으로 변환한 다음, 그 코드 블록을 실행하는 최상위 내장 함수이다. 자바스크립트의 eval() 함수는 다음과 같은 식으로 사용한다.

```
eval(string)
```

자바스크립트에서 eval()을 실행하면 인터프리터에서는 string을 코드로 변환한 다음, 그 코드를 실행시키고 (어떤 값이 나온다면) 그 결과를 리턴한다. 아래와 같은 자바스크립트 예제를 살펴보자.

```
eval("parseInt('1.5')");  // parseInt() 함수를 호출한다.
                          // 이 경우에는 1이 리턴된다.
eval("var x = 5");        // x라는 새로운 변수를 만들고
                          // 그 변수에 5를 대입한다.
```

전에 eval() 함수를 본 적이 없다면, '코드가 들어 있는 문자열을 사용하게 될까? 왜 코드를 그냥 쓰지 않지?' 하는 의문이 들 수도 있다. 이러한 함수를 사용하는 이유는 eval() 함수를 이용하면 필요에 따라 동적으로 코드를 만들어낼 수 있기 때문이다. 예를 들어 func1, func2, func3, ..., func10과 같이 순서대로 번호가 매겨진 함수가 10개 있다고 가정해 보자. 아래와 같이 10개의 함수 호출 선언문을 이용하여 모든 함수를 실행할 수도 있다.

```
func1();
func2();
func3();
// 기타 등등...
```

하지만 아래와 같이 순환문 안에서 eval() 함수를 이용하면 훨씬 간편하게 모든 함수를 호출할 수 있다.

```
for (i = 1; i <= 10; i++){
  eval("func" + i + "()");
}
```

액션스크립트의 eval() 함수는 자바스크립트의 eval() 함수에서 지원하는 기능의 일부만을 지원한다. 액션스크립트에서는 인자가 인식자인 경우에만 작동한다. 따라서 액션스크립트의 eval() 함수는 아래 예에서 볼 수 있듯이 특정 인식자에 있는 데이터를 가져오는 역할밖에 할 수 없다.

```
var num = 1;
var person1 = "Eugene";
trace (eval("person" + num));  // "Eugene"이 출력된다.
```

이렇게 기능의 일부만을 사용할 수 있지만, 그래도 eval() 함수는 꽤 유용하다. 아래 예에서는 순환문을 이용하여 일련의 무비 클립을 만들어낸다. 그리고 나서 각 클립을 eval()을 이용하여 배열에 저장한다.

```
for (var i = 0; i < 10; i++) {
  duplicateMovieClip("ballParent", "ball" + i, i);
  balls[i] = eval("ball" + i);
}
```

하지만 eval()을 이용하면 속도가 느려질 수 있다는 것에 주의하자. 더 강도 높은 작업을 할 때는 동적으로 클립을 참조할 때 아래와 같이 배열 접근 연산자를 이용하는 것이 낫다.

```
duplicateMovieClip("ballParent", "ball" + i , i);
balls[ballCount] = _root ["ball" + i];
```

플래시 4에서는 동적으로 변수 이름을 만들고 그러한 변수 이름을 참조하여 배열과 비슷한 효과를 얻는 데 eval() 함수를 이용하는 경우가 많았다. 하지만 플래시 5에서는 배열을 기본으로 지원하기 때문에 이러한 기법이 권장되지도 않으며 굳이 이러한 방법을 사용할 필요도 없다. 자세한 내용은 '2장. 변수'의 '동적으로 이름이 주어지는 변수 만들기'에서 찾아보기 바란다.

# 플래시 4와 플래시 5의 문자열 연산자 및 함수

지금까지 문자열 연산자와 함수를 설명하면서 각 연산자와 함수에 해당하는 플래시 4 테크닉에 대해서도 잠깐씩 알아보았다. 플래시 5를 이용하여 플래시 4 무비를 만들 때는 플래시 4의 문자열 연산자와 함수를 사용해야 한다. 하지만 플래시 5 무비를 만들 때는 그보다 좋은 플래시 5 연산자를 사용하는 것이 좋다. 플래시 4 문법에 익숙한 독자라면 〔표 4-2〕를 참조하여 플래시 5의 문법을 익혀두는 것이 좋다.

***〔표 4-2〕 플래시 4와 플래시 5의 연산자 및 함수***

| 플래시 4 문법 | 플래시 5 문법 | 설명 |
|---|---|---|
| “” | “” 또는 ‘’ | 문자열 리터럴 |
| & | +(하위 호환성을 위해 add를 사용해도 된다) | 문자열 병합 연산자 |
| eq | == | 동치 연산자 |
| ge | >= | 비교(~ 이상) |
| gt | > | 비교(~ 초과) |
| le | <= | 비교(~ 이하) |
| lt | < | 비교(~ 미만) |
| ne | != | 부등 연산자(같지 않음) |
| chr() 또는 mbchr() | fromCharCode()* | 인코딩된 숫자로부터 문자 생성 |
| length() 또는 mblength() | length* | 플래시 4에서는 함수, 플래시 5에서는 속성이며 문자열에 있는 문자의 개수를 리턴한다. |
| mbsubstring() | substr() | 문자열로부터 문자 시퀀스 추출 |
| ord() 또는 mbord() | charCodeAt()* | 지정된 문자의 코드 포인트 |
| substring() | substr() | 문자열로부터 문자 시퀀스 추출 |

* 플래시 5 문자열 연산자와 함수는 모두 멀티바이트 문자를 지원하므로 플래시 4와 같이 강제로 싱글바이트 연산만을 처리하도록 할 수 없다. 예를 들어 플래시 5의 fromCharCode() 함수는 mbchr() 함수와 거의 같다. 마찬가지로 mblength()와 length, mbsubstring()과 substr(), mbord()와 charCodeAt()도 거의 같다.

## 부울형

부울 데이터는 참과 거짓이라는 두 가지 논리 상태를 표시하는 데 쓰인다. 따라서 부울 데이터형에서는 true와 false라는 두 개의 값만 사용한다. 부울 데이터는 문자열 데이터가 아니므로 true와 false를 따옴표로 감싸지 않는다는 점을 염두에 두어야 한다. true와 false는 예약되어 있는 원시 데이터 값이므로 변수나 인식자 이름으로 사용할 수 없다.

부울 값을 이용하여 코드를 실행할 때 논리적인 부분을 추가할 수 있다. 예를 들어 우주선의 화력 상태를 추적하는 변수에 true를 대입할 수 있다.

```
shipHasDoubleShots = true;
```

shipHasDoubleShots 변수를 부울 리터럴인 true와 비교하면 우주선에서 발사한 미사일이 명중할 때 상대방에게 입히는 데미지를 결정할 수 있다.

```
if (shipHasDoubleShots == true) {
  // 두 배의 파워로 공격한다.
  // 비교 결과가 true일 때 여기에 있는 코드를 실행한다.
} else {
  // 보통 파워로 공격한다.
  // 비교 결과가 false일 때 여기에 있는 코드를 실행한다.
}
```

파워가 두 배인 미사일이 다 떨어지면 그 변수를 false로 설정한다.

```
shipHasDoubleShots = false;
```

이렇게 하면 shipHasDoubleShots == true라는 표현식 값이 false가 되므로 미사일이 적에게 명중했을 때 보통 파워만큼의 데미지만을 입히게 된다.

모든 비교 연산자의 연산 결과는 부울 값으로 리턴된다. '사용자가 입력한 비밀번호가 실제 비밀번호와 같은가?' 라는 질문을 던졌을 때 그 대답은 부울형이 된다.

```
// userGuess == password 는 true 아니면 false 값을 가진다.
if (userGuess == password) {
  gotoAndStop("secretContent");
}
```

'무비 클립이 90도 이상 회전되었는가?' 라는 질문을 던졌을 때도 그 답은 부울형으로 주어진다.

```
// myClip._rotation > 90 은 true 아니면 false 값을 가진다.
if (myClip._rotation > 90) {
  // 클립이 90도 이상 회전되었으면 클립을 흐리게 표시한다.
  myClip._alpha = 50;
}
```

액션스크립트의 속성 및 메소드 중에도 플래시 무비 환경을 부울 값으로 표현하는 것들이 많이 있다. 예를 들어 스페이스바가 눌러져 있는지를 물어본다면 인터프리터에서는 부울 값인 true(예) 또는 false(아니오)로 대답한다.

```
// Key.isDown() 함수는 true 또는 false를 리턴한다.
if (Key.isDown(Key.SPACE)) {
  // 스페이스바가 눌려 있으므로 우주선에서 미사일을 발사한다.
}
```

부울 연산자를 이용하여 복잡한 논리 표현식을 만드는 법은 5장에서 알아보도록 하자.

## 부울 값을 이용하여 프리로더 만들기

부울형을 응용한 고급 예제에 대해 생각해 보자. 프레임이 500개나 있고 꽤 많은 내용물이 담겨 있는 문서가 있다고 가정하자. 오프닝 시퀀스의 시작 부분인 20번 프레임의 레이블은 intro이다. 이 무비의 메인 타임라인의 2번 프레임에 다음과 같은 코드를 집어넣는다.

```
if (_framesloaded >= _totalframes) {
  gotoAndPlay("intro");
} else {
  gotoAndPlay(1);
}
```

무비가 시작되면 플레이헤드가 2번 프레임에 들어간다. 액션스크립트 인터프리터에서는 위 조건문에 도착하게 되면 _framesloaded 〉= _totalframes라는 부울 표현식을 조사한다. 아직 무비를 로딩하는 중이라면 _framesloaded가 무비에 있는 전

체 프레임 수(_totalframes)보다 작다. 만약 _framesloaded가 _totalframes보다 크거나 같지 않으면, _framesloaded 〉= _totalframes는 false 값을 가지게 된다. 따라서 gotoAndPlay("intro")는 건너 뛰고 대신 gotoAndPlay(1)을 실행한다. gotoAndPlay(1) 선언문에서는 플레이헤드를 다시 1번 프레임으로 돌려보내서 무비를 다시 재생한다. 플레이헤드가 2번 프레임에 들어가면 위 코드가 다시 실행된다. 이처럼 _framesloaded 〉= _totalframes가 true가 되기 전까지(즉 모든 프레임이 다 로딩될 때까지)는 플레이헤드가 1번과 2번 프레임 사이만 맴돌게 된다. 이 부울 표현식이 true가 되면 gotoAndPlay("intro")가 실행되어 플레이헤드가 intro 레이블로 이동한다. 모든 프레임을 로딩하는 작업이 끝나고 나면 무비를 시작해도 좋기 때문이다.

놀랍지 않은가! 부울 표현식만 가지고 프리로더를 만들었다. 조건문과 부울 값을 이용하여 무비를 제어하는 방법에 대한 내용은 7장에서 매우 자세히 다룰 것이다.

## Undefined

지금까지 살펴본 데이터형은 거의 모두 정보를 저장하고 조작하는 데 쓰이는 데이터형이다. undefined 데이터형은 그 활용 범위가 매우 좁다. 이 데이터형은 어떤 변수가 존재하는지, 또는 그 변수에 값을 대입한 적이 있는지 확인하기 위한 용도로 쓰인다. undefined 데이터형에는 값이 undefined 하나밖에 없다.

변수를 처음 선언하면 기본값으로 undefined가 저장된다.

```
var velocity;
```

인터프리터는 위 선언문을 다음과 같이 해석한다.

```
var velocity = undefined;
```

어떤 변수에 값이 있는지 확인할 때는 다음과 같이 그 변수를 undefined와 비교하면 된다.

```
if (myVariable != undefined) {
  // myVariable에 어떤 값이 저장되어 있으므로 필요한 작업을 처리한다.
}
```

undefined를 문자열로 사용하면 비어있는 문자열로 변환된다는 점을 주의하자. 예를 들어 firstName이 undefined이면 아래와 같은 trace() 선언문에서는 “”(비어있는 문자열)을 출력한다(즉 아무것도 출력되지 않는다).

```
var firstName;
trace(firstName);  // 아무것도 출력되지 않는다(비어있는 문자열).
```

똑같은 코드를 자바스크립트에서 실행한다면 비어있는 문자열 대신 “undefined”가 출력된다. 액션스크립트에서는 하위 호환성을 위해 undefined를 “”로 변환한다.

플래시 4 액션스크립트에는 undefined 형이 없기 때문에, 플래시 4 프로그램 중에는 비어있는 문자열을 이용하여 어떤 변수에 유용한 값이 들어있는지 확인하는 것들이 많이 있다. 따라서 아래와 같은 코드가 흔히 사용되었다.

```
if (myVar eq "") {
  // myVar 변수가 아직 정의되지 않은 상태이므로
  // 아무 것도 하지 않는 것이 좋다.
}
```

플래시 5에서 undefined를 문자열로 변환할 때 “”가 아닌 다른 문자열로 변환한다면, 위와 같은 코드를 플래시 5 플레이어에서 재생할 때 엉뚱한 결과가 나올 것이다.

액션스크립트에서는 선언되었지만 아무런 값도 할당받지 않은 변수와 존재하지 않는 변수에 모두 undefined 값을 리턴한다는 점에 유의하자. 이 부분도 존재하지 않는 변수를 참조하면 오류가 발생하는 자바스크립트와는 차이점을 보인다.

## Null

사실 생각해 보면 null 형은 undefined 형과 거의 똑같다. undefined 데이터형과 마찬가지로 null 데이터형도 데이터가 없음을 나타내는 데 쓰이며, 한 가지 원시 값인 null밖에 사용할 수 없다. 하지만 null 값은 인터프리터에서 자동으로 대입하는 값이 아니라 사용자가 직접 대입하는 값이다.

변수나 배열 원소, 객체 속성에 유효한 숫자, 문자열, 부울, 배열 또는 객체 값이 들어있지 않다는 것을 나타낼 때 null을 대입한다.

null과 비교했을 때 같은 것으로 간주되는 것은 null 자신과 undefined 뿐이다.

```
null == undefined;   // true
null == null;        // true
```

## 앞으로 배울 내용

지금까지 생각보다 상당히 많은 내용을 배웠다. 앞으로 배우게 될 고급 주제들을 이해할 수 있는 탄탄한 기반을 닦을 수 있었을 것이다. 아직 자세한 내용을 다 익히지 못했더라도 너무 걱정하지 않아도 된다. 필요할 때 다시 공부해도 된다. 데이터를 합치고 변환하는 방법을 배우는 다음 장에서도 계속해서 열심히 공부하기 바란다. 그 뒤에는 좀더 모양새를 갖춘 응용 예제를 다룰 것이다.