# 5

# 연산자

데이터를 조작하고 결합하고 변환하는 데 쓰이는 기호 또는 키워드를 '연산자(operator)'라고 한다. 프로그래밍을 처음 배우는 독자들도 +(덧셈) 또는 -(뺄셈) 같은 수학에서 쓰이는 연산자는 쉽게 알아볼 수 있을 것이다. 그러나 개념 자체는 익숙하지만 프로그래밍에서 사용하는 문법을 배워야 하는 경우도 있을 것이다. 예를 들어 두 개의 수를 곱할 때 액션스크립트에서는 일반적으로 학교에서 배우는 곱셈 기호(×) 대신 * 기호를 이용한다. 예를 들어 5 곱하기 6은 다음과 같이 표기한다.

```
5 * 6;
```

## 연산자의 일반적인 특징

각 연산자별로 특별한 용도가 있지만 모든 연산자에 적용되는 일반적인 성질이 있다. 각 연산자에 대해 알아보기 전에 일반적인 연산자의 특징을 알아보자.

## 연산과 표현식

연산자는 주어진 데이터 값(피연산자, operand)을 이용하여 작업을 수행한다. 예를 들어 5 * 6에서 5와 6이라는 숫자는 곱셈 연산자(*)의 피연산자이다. 아래 나온 것처럼 어떤 표현식이라도 피연산자로 사용할 수 있다.

```
player1score + bonusScore;              // 변수를 피연산자로 사용
(x + y) - (Math.PI * radius * radius);  // 복합 표현식을 피연산자로 사용
```

두 번째 예제에서는 왼쪽과 오른쪽 피연산자가 모두 표현식이고, 각 표현식에서 각자 필요한 연산을 처리해야 한다. 복합 표현식을 이용하여 더 긴 표현식을 만들 수도 있다.

```
((x + y) - (Math.PI * radius * radius)) / 2  // 전체를 2로 나눈다.
```

표현식이 너무 커지면 변수를 만들어서 중간 결과를 저장하는 것도 생각해 볼만하다. 이렇게 하면 프로그래밍을 하기도 좋고 코드의 의미가 더 명확하게 드러나므로 꽤 편리하다. 변수의 이름을 지을 때는 그 변수의 용도를 쉽게 알 수 있는 이름을 사용하는 것이 좋다.

```
var radius = 10;
var height = 25;
var circleArea = (Math.PI * radius * radius);
var cylinderVolume = circleArea * height;
```

## 피연산자의 개수

연산자를 분류할 때 그 연산자에서 사용하는 피연산자의 개수를 기준으로 분류하는 경우도 있다. 액션스크립트에는 한 개, 두 개, 또는 세 개의 피연산자가 필요한 연산자도 있다.

```
-x                                          // 연산자가 한 개
x * y                                       // 연산자가 두 개
(x == y) ? "true result" : "false result"   // 연산자가 세 개
```

피연산자가 한 개인 연산자는 '일항(unary)' 연산자, 피연산자가 두 개인 연산자는 '이항(binary)' 연산자, 피연산자가 세 개인 연산자는 '삼항(ternary)' 연산자라고 부른다. 여기서는 우리가 배우고자 하는 목적에 맞추어 피연산자의 개수가 아닌 연산자의 기능을 기준으로 연산자를 살펴보기로 하자.

## 연산자 우선 순위

연산자 '우선 순위(precedence)'는 여러 개의 연산자가 들어 있는 표현식에서 어떤 연산자를 먼저 실행할지를 결정한다. 예를 들어 한 표현식에 곱셈과 덧셈이 섞여 있으면 곱셈을 먼저 계산한다.

```
4 + 5 * 6  // 34가 된다. 4 + 30 = 34이기 때문이다.
```

* 연산자의 우선 순위가 + 연산자보다 높기 때문에, 4 + 5 * 6 은 4 + (5 * 6)으로 계산된다. 우선 순위를 정확하게 모르거나 연산 순서를 확실히 하고 싶다면 가장 높은 우선 순위를 가지는 괄호를 사용하면 된다.

```
(4 + 5) * 6  // 54가 된다. 9 * 6 = 54이기 때문이다.
```

꼭 써야 하는 것은 아니지만 괄호를 이용하면 복잡한 표현식을 조금 더 읽기 쉽도록 할 수 있다.

```
// x가 y보다 크거나 y가 z와 같은 경우
x > y || y == z
```

위 표현식은 우선 순위표를 찾아보기 전에는 확실히 이해하기 어렵다. 다음과 같이 괄호를 사용하면 코드를 이해하기가 훨씬 쉬워진다.

```
(x > y) || (y == z)  // 이게 훨씬 낫다.
```

〔표 5-1〕에 각 연산자의 우선 순위를 정리해 놓았다. 가장 높은 우선 순위를 가진 연산자(표의 맨 위에 있는 연산자)부터 차례로 실행된다. 우선 순위가 같은 연산자가 여러 개 있으면 왼쪽에 있는 것이 먼저 실행된다.

〔표 5-1〕 액션스크립트 연산자의 작용 결합 방향 및 우선 순위

| 연산자 | 우선 순위 | 결합 방향 | 설명 |
|---|---|---|---|
| x++ | 16 | → | 후치 증가 |
| x-- | 16 | → | 후치 감소 |
| . | 15 | → | 객체 속성 액세스 |
| [ ] | 15 | → | 배열 원소 액세스 |
| ( ) | 15 | → | 괄호 |
| function( ) | 15 | → | 함수 호출 |
| ++x | 14 | ← | 전치 증가 |
| --x | 14 | ← | 전치 감소 |
| - | 14 | ← | 부호 변경 |
| ~ | 14 | ← | 비트 NOT |
| ! | 14 | ← | 논리 NOT |
| new | 14 | ← | 객체/배열 생성 |
| delete | 14 | ← | 객체/속성/배열 원소 삭제 |
| typeof | 14 | ← | 데이터형 결정 |
| void | 14 | ← | undefined 값을 리턴 |
| * | 13 | → | 곱하기 |
| / | 13 | → | 나누기 |
| % | 13 | → | 나머지 |
| + | 12 | → | 더하기 또는 문자열 병합 |
| - | 12 | → | 빼기 |
| << | 11 | → | 비트 왼쪽 시프트 |
| >> | 11 | → | 비트 오른쪽 시프트(부호 있음) |
| >>> | 11 | → | 비트 오른쪽 시프트(부호 없음) |
| < | 10 | → | 미만 |
| <= | 10 | → | 이하 |
| > | 10 | → | 초과 |
| >= | 10 | → | 이상 |
| == | 9 | → | 등치 |
| != | 9 | → | 부등 |
| & | 8 | → | 비트 AND |
| ^ | 7 | → | 비트 XOR |
| \| | 6 | → | 비트 OR |
| && | 5 | → | 논리 AND |
| \|\| | 4 | → | 논리 OR |
| ?: | 3 | ← | 조건문 |

| 연산자 | 우선 순위 | 결합 방향 | 설명 |
|---|---|---|---|
| = | 2 | ← | 대입 |
| += | 2 | ← | 더해서 대입 |
| -= | 2 | ← | 빼서 대입 |
| *= | 2 | ← | 곱해서 대입 |
| /= | 2 | ← | 나눠서 대입 |
| %= | 2 | ← | 나머지 대입 |
| <<= | 2 | ← | 왼쪽 시프트해서 대입 |
| >>= | 2 | ← | 오른쪽 시프트해서 대입 |
| >>>= | 2 | ← | 오른쪽 시프트해서 대입(부호 없음) |
| &= | 2 | ← | 비트 AND 후 대입 |
| ^= | 2 | ← | 비트 XOR 후 대입 |
| \|= | 2 | ← | 비트 OR 후 대입 |
| , | 1 | → | 쉼표 |

## 연산자 결합 방향

방금 배운 것처럼 연산자 우선 순위는 연산자를 계산하는 순서를 나타낸다. 우선 순위가 높은 연산자를 우선 순위가 낮은 연산자보다 먼저 실행한다. 하지만 우선 순위가 같은 연산자가 여러 개 한꺼번에 사용되면 어떻게 될까? 그러한 경우에는 연산하는 방향을 나타내는 '연산자 결합 방향(operator associativity)' 규칙을 사용한다. 연산자에는 왼쪽 결합형(왼쪽에서 오른쪽으로 계산) 또는 오른쪽 결합형(오른쪽에서 왼쪽으로 결합)의 두 가지가 있다. 다음과 같은 표현식을 생각해 보자.

```
a = b * c / d
```

*와 / 연산자는 왼쪽 결합형이므로 왼쪽에 있는 * 연산자를 먼저 계산한다. 따라서 위 예제는 다음과 같은 식으로 계산된다.

```
a = (b * c) / d
```

반면에 = (대입) 연산자는 오른쪽 결합형이므로 다음과 같은 표현식은

```
a = b = c = d
```

'c에 d를 대입하고 b에 c를 대입하고 a에 b를 대입한다' 라는 뜻으로 해석할 수 있다.

```
a = (b = (c = d))
```

연산자 결합 방향은 직관적으로 알 수 있도록 정해져 있긴 하지만 복잡한 표현식에서 예상치 못한 결과가 나온다면 〔표 5-1〕의 내용을 다시 살펴보거나 미심쩍은 부분에 괄호를 덧붙이는 것이 좋다. 이 장을 계속 읽다 보면 결합 방향 때문에 오류가 발생하는 경우를 많이 볼 수 있을 것이다.

## 데이터형과 연산자

어떤 연산자에서는 피연산자로 여러 가지 데이터형을 사용할 수 있으며, 피연산자의 데이터형에 따라 연산 결과가 바뀔 수도 있다. 예를 들어 + 연산자는 숫자를 피연산자로 사용할 때는 덧셈을 처리하지만 문자열을 피연산자로 사용하면 문자열 병합 연산을 처리한다. 피연산자의 데이터형이 다르거나 잘못된 형의 데이터를 사용하면, '3장. 데이터와 데이터형' 에서 배운 규칙에 따라 형을 변환하는데, 이 과정에서 코드의 의미가 크게 달라질 수도 있다.

# 대입 연산자

대입 연산자는 이미 많이 사용해 보았다. 대입 연산자는 변수, 배열 원소 또는 객체 속성에 어떤 값을 집어넣는 데 쓰인다. 대입 연산은 다음과 같은 형식으로 사용한다.

```
identifier = expression
```

identifier는 값을 대입할 변수, 배열 원소 또는 객체 속성이고, expression은 저장하고자 하는 값(데이터)을 나타낸다. 예를 들면 다음과 같다.

```
x = 4;                  // 변수 x에 4를 대입한다.
x = y;                  // 변수 x에 y의 값을 대입한다.
name = "dj duong";      // name이라는 변수에 문자열을 대입한다.
products[3] = "Flash";  // products의 네 번째 원소에 문자열을 대입한다.
```

```
// square의 area 속성에 숫자를 대입한다.
square.area = square.width * 2;
```

다음과 같이 여러 개의 대입 연산을 한꺼번에 처리할 수도 있다.

```
x = y = 4;  // x와 y를 모두 4로 설정한다.
```

대입 연산자의 결합 방향은 오른쪽에서 왼쪽이므로, 4를 먼저 y에 대입하고 그 후에 y 값(4)을 x에 대입한다는 점을 기억해 두자.

## 대입과 연산의 결합

어떤 변수에 대입 연산을 할 때는 그 변수에 원래 있던 값을 기반으로 새로운 값을 대입하는 경우가 많이 있다.

```
counter = counter + 10;            // counter의 현재 값에 10을 더한다.
xPosition = xPosition + xVelocity; // xPosition 값에 xVelocity를 더한다.
score = score / 2;                 // score를 2로 나눈다.
```

액션스크립트에서는 +, -, /와 같은 연산자를 대입 연산자와 결합하여 '계산하여 대입하는' 연산을 처리하는 복합 대입 연산자를 지원한다. 따라서 덧셈과 대입을 결합하려면 += 연산자를 사용하면 된다. 나눗셈을 대입과 결합할 때는 /= 연산자를 이용한다. 따라서 앞에 나온 예제는 다음과 같이 복합 연산자로 표현하면 더 간단하게 만들 수 있다.

```
counter += 10;
xPosition += xVelocity;
score /= 2;
```

복합 대입 연산자의 목록은 〔표 5-1〕에 나와 있다.

## 수학 연산자

수학 연산자는 숫자로 된 피연산자에 대해 수학 연산을 처리하는 연산자이다. 숫자 연산자를 사용할 때 숫자가 아닌 피연산자를 사용하면 액션스크립트에서는 그 데이터를 숫자로 변환한다. 예를 들어 false - true를 계산하면 -1이 나오는데, 이는 false는 숫자 0으로, true는 숫자 1로 변환되기 때문이다. 마찬가지로 "3" * "5"를 계산하면 숫자 15가 나온다. 곱셈을 처리하기 전에 문자열 "3"과 "5"를 각각 숫자 3과 5로 변환하기 때문이다. 하지만 + 연산자는 조금 다르게 작동한다. + 연산자를 사용할 때 피연산자에 문자열이 하나라도 있으면 덧셈이 아니라 문자열 병합 작업을 처리한다.

숫자가 아닌 피연산자를 숫자로 변환할 수 없는 경우에는 그 피연산자가 특수 값인 NaN으로 설정된다. 이런 경우에는 전체 계산 결과가 NaN이 되어 버린다. 숫자로 변환하는 방법에 대한 내용은 〔표 3-1〕을 참조하기 바란다.

## 덧셈

덧셈 연산자는 두 피연산자의 합을 리턴한다.

```
operand1 + operand2
```

수학적으로 의미 있는 결과를 리턴하려면, + 연산자의 피연산자가 다음과 같이 숫자 값에 해당하는 표현식이어야 한다.

```
234 + 5          // 239를 리턴한다.
(2 * 3 * 4) + 5  // 29를 리턴한다.
```

덧셈 연산자는 계산 연산자 중에서 유일하게 피연산자 중 하나 이상이 문자열일 때 문자열을 합치는 연산을 수행하는 연산자이다. '4장. 원시 데이터형'의 '문자열 병합'을 참조하기 바란다.

## 증가

증가 연산자는 덧셈 연산자를 약간 변화시킨 연산자로, 하나의 피연산자만을 사용하며 현재 값에 1을 더한다. 증가 연산자는 두 가지 형식으로 사용할 수 있는데, 하나는 전치 증가(prefix increment)이고 다른 하나는 후치 증가(postfix increment)[1]이며, 다음과 같은 형식으로 사용하면 된다.

```
++operand    // 전치 증가
operand++    // 후치 증가
```

어떤 형식으로 사용하든 상관없이 변수나 배열의 원소 또는 객체 속성에 1을 더하는 기본 기능은 똑같이 작용한다.

```
var x = 1;
x = x + 1;  // x의 값이 2가 된다.
x++;        // 1을 더한다: x의 값이 3이 된다.
++x;        // 1을 더한다: x의 값이 4가 된다.
```

증가 연산자만 따로 사용할 때는 전치 증가나 후치 증가 어느 쪽을 사용해도 결과는 똑같다. 다만 관행적으로 후치 증가 연산자를 사용하는 것이 더 일반적이다.

하지만 복잡한 표현식에서는 결과가 달라진다. 전치 증가 연산자는 피연산자에 1을 더한 다음 1이 더해진 값을 리턴하며, 후치 증가 연산자는 피연산자에 1을 더하긴 하지만 1을 더하기 전의 값을 리턴한다.

```
var x = 1;
// 후치 증가: y를 1로 설정한 뒤 x의 값에 1을 더한다.
var y = x++;

var x = 1;
// 전치 증가: x의 값에 1을 더한 뒤에 y에 x를 대입하므로 y가 2가 된다.
var y = ++x;
```

증가 연산자는 '8장. 순환문' 에서 다시 다룰 것이다.

1) 역자주: 용어를 너무 어렵게 생각할 필요 없이 전치 증가는 ++ 연산자를 앞에 적는 것, 후치 증가는 ++ 연산자를 뒤에 적는 것이라고 생각하면 된다.

## 뺄셈

뺄셈 연산자는 첫째 피연산자에서 둘째 피연산자를 뺀다. 일반적인 형식은 다음과 같다.

```
operand1 - operand2
```

피연산자로는 모든 유효한 표현식을 사용할 수 있다. 만약 두 가지 중 하나의 피연산자라도 숫자형이 아니며 실수로 변환할 수도 없다면, 연산 결과는 NaN이 된다.

```
234 - 5  // 229
5 - 234  // -229
```

두 숫자 사이의 차이의 절대값을 구하려면 Math.abs() 메소드를 이용하면 된다. Math.abs() 메소드에 대한 설명은 '3부. 레퍼런스'에 나와 있다.

## 감소

감소 연산자는 증가 연산자와 거의 같으며 피연산자의 현재 값을 1 증가시키는 대신 1 감소시키는 연산자이다. 증가 연산자와 마찬가지로 감소 연산자에도 전치 감소와 후치 감소의 두 가지 형식이 있다.

```
--operand  // 전치 감소
operand--  // 후치 감소
```

둘 중 어떤 형식을 사용하더라도 모두 변수, 배열 원소 또는 객체 속성의 현재 값에서 1을 빼는 것은 똑같다. 전치 감소 연산자는 피연산자에서 1을 뺀 다음 그 값을 리턴하며, 후치 증가 연산자는 피연산자에서 1을 빼긴 하지만 1을 빼기 전의 값을 리턴한다. 증가 연산자와 마찬가지로 전치 감소와 후치 감소는 피연산자를 다른 표현식과 함께 사용할 때만 서로 다른 결과가 나올 수 있다.

```
var x = 10;
var y;
x = x - 1;    // x가 9가 된다.
x--;          // x가 8이 된다.
--x;          // x가 7이 된다.
y = --x;      // x가 6이 되고 y도 6이 된다.
y = x--;      // y는 여전히 6이고 x는 5가 된다.
```

## 곱셈

곱셈 연산자는 두 개의 숫자 피연산자를 곱하고 그 결과를 리턴한다. 곱셈은 일반적으로 다음과 같은 형식으로 쓰인다.

```
operand1 * operand2
```

유효한 표현식이라면 모두 피연산자로 사용할 수 있다. 곱셈에 쓰이는 * 기호는 일반적으로 수학에서 곱셈을 나타내는 × 대신 쓰이는 기호이다. 둘 중 한 피연산자라도 숫자형이 아니며 실수로 변환할 수도 없다면, 연산 결과는 NaN이 된다.

```
6 * 5  // Returns 30
```

## 나눗셈

나눗셈 연산자는 첫째 피연산자(분자, numerator)를 둘째 피연산자(분모, divisor)로 나누고 그 결과(몫, quotient)를 리턴하는 연산자이다. 나눗셈은 일반적으로 다음과 같은 형식으로 쓰인다.

```
operand1 / operand2
```

피연산자는 유효한 숫자 표현식이어야 한다. / 기호는 일반적으로 수학에서 나눗셈을 나타내는 ÷ 대신 쓰이는 기호이다. 둘 중 한 피연산자라도 숫자형이 아니며 실수로 변환할 수도 없다면, 연산 결과는 NaN이 된다. 계산 결과가 나누어 떨어지지 않는다면(소수 부분이 남는다면), 두 피연산자가 모두 정수인 경우에도 부동소수점 소수 값이 리턴된다.

```
20 / 5  // 4
5 / 4   // 1.25; 다른 언어에서는 보통 1.25가 아닌 1을 리턴한다.
```

디렉터의 링고와 같은 언어에서는 보통 두 피연산자가 모두 정수인 경우에는 결과를 정수로 리턴한다.

만약 분모가 0이라면(즉 0으로 나누는 경우에는), 분자가 0 또는 숫자가 아닌 값을 가질 때는 NaN, 분자가 양수일 때는 Infinity, 분자가 음수일 때는 -Infinity를 리턴한다. 예를 들면 다음과 같다.

```
trace (0/0); // NaN
trace ("a"/0); // NaN
trace (1/0); // Infinity
trace (-1/0); // -Infinity
```

만약 분모가 0이 될 가능성이 있다면, 다음과 같이 나눗셈을 하기 전에 값을 미리 검사하는 것이 좋다.

```
if (numItems != 0) {
  trace ("Average is"+ total / numItems);
else
  trace ("There are no items for which to calculate the average");
}
```

다른 언어에서는 0으로 나누면 오류가 발생하는 경우도 있다.

## 나머지

나머지 연산자는 '모듈 나눗셈(modulo division)'이라는 연산을 처리한다. 나머지 연산자를 사용하면 첫째 피연산자를 둘째 피연산자로 나눈 나머지를 리턴한다. 나머지 연산은 다음과 같은 형태로 쓰인다.

```
operand1 % operand2
```

예를 들면 14 % 4는 2를 리턴한다. 14를 4로 나눈 나머지는 2이기 때문이다.

C나 C++와는 달리 액션스크립트에서는 정수나 부동소수점 소수에 상관없이 유효한 숫자 표현식이라면, 나눗셈 연산의 피연산자로 사용할 수 있다. 예를 들면 5 % 4는 1이고, 5 % 4.5는 0.5가 된다. 만약 두 피연산자 중 하나라도 숫자형이 아니며 실수로 변환할 수도 없다면, 그 결과는 NaN이 된다.

모든 짝수를 2로 나눈 나머지는 0이다. [예제 5-1]은 이러한 원리를 이용해서 어떤 수가 짝수인지 홀수인지 알아내는 프로그램이다.

*〔예제 5-1〕 나머지 연산을 이용하여 짝수인지 알아내기*

```
var x = 3;
if (x%2 == 0) {
  trace("x is even");
} else {
  trace("x is odd");
}
```

## 부호 변경

부호 변경 연산자는 하나의 피연산자만을 취한다. 이 연산자는 피연산자의 부호를 바꿔준다(즉 음수는 양수가 되고 양수는 음수가 된다). 부호 변경 연산자는 다음과 같은 형식으로 쓰인다.

```
-operand
```

유효한 표현식이라면 모두 피연산자로 사용할 수 있다. 아래 예는 어떤 대상의 수평 위치가 양수 방향의 한계치보다 크거나 음의 방향의 한계치보다 작은지 검사하는 코드이다.

```
if (xPos > xBoundary || xPos < -xBoundary){
  // 영역을 벗어난 상태
}
```

# 동치 및 부등 연산자

두 표현식의 값이 같은지 검사할 때는 동치 연산자(==)를 이용한다. 동치 연산자는 다음과 같은 형식으로 쓰인다.

```
operand1 == operand2
```

operand1과 operand2 자리에는 임의의 유효한 표현식을 사용할 수 있다. 동치 연산자는 어떤 형의 피연산자도 비교할 수 있다. operand1과 operand2가 같으면 위 표현식에서 부울 값인 true를 리턴한다. 두 값이 다르면 false를 리턴한다.

```
var x = 2;
x == 1      // false
x == 2      // true
```

동치 연산자는 두 개의 등호(=)를 연속으로 놓은 연산자이다. 이 연산자는 두 개의 표현식이 같은지 확인하기 위해 쓰이며, 변수에 새로운 값을 대입하는 데 쓰이는 대입 연산자(=)와 혼동하지 않도록 주의해야 한다.

아래와 같은 예를 생각해 보자.

```
if (x = 5) {
  trace ("x is equal to 5")
}
```

위 예제에서는 x 값이 5인지 확인하는 대신 x에 5를 대입한다. 위 코드를 제대로 고치면 다음과 같다.

```
// = 대신 ==를 사용한다.
if (x == 5) {
  trace ("x is equal to 5")
}
```

## 원시 데이터형 동치

원시 데이터형에 대해서는 대부분의 동치 연산을 직관적으로 이해할 수 있다. [표 5-2]에 각 데이터형에 대한 동치 연산 규칙을 정리해 놓았다.

*[표 5-2] 원시 데이터형의 동치 조건*

| 데이터형 | 동치 조건(두 피연산자가 모두 같은 데이터형일 때) |
|---|---|
| 숫자 | operand1과 operand2가 같은 숫자이면 결과는 true이다. 두 피연산자가 모두 +Infinity 또는 -Infinitiy인 경우, -0 또는 +0일 때에도 결과는 true가 된다. 이를 제외한 모든 경우에는 두 피연산자가 모두 NaN인 경우에도 결과는 false가 된다.<br>1 == 4 // false<br>4 == 4 // true<br>NaN == NaN // false<br>+Infinity = -Infinity // false |

| 데이터형 | 동치 조건(두 피연산자가 모두 같은 데이터형일 때) |
|---|---|
| 문자열* | 대소문자를 구분하여 문자열을 비교한다. operand1과 operand2의 길이가 같고 똑같은 문자들을 같은 순서로 늘어놓은 문자열이라면 결과는 true가 된다. 그 외의 경우에는 false가 된다.<br>"Flash" == "Flash" // true<br>"O' Reilly" == "O Reilly" // false<br>"Moock" == "moock" // false ("m"과 "M"은 서로 다른 문자이다) |
| 부울 | 두 피연산자가 모두 true이거나 모두 false이면 결과는 true가 된다. 그 외에는 모두 false이다.<br>true == true // true<br>false == false // true<br>true == false // false |
| undefined | 두 피연산자가 모두 undefined이거나 하나는 undefined, 나머지 하나는 null인 경우에는 true가 된다. 그 외에는 모두 false이다. |
| null | 두 연산자가 모두 null이거나 둘 중 하나는 undefined, 나머지 하나는 null인 경우에는 true가 된다. 그 외에는 모두 false이다. |
| 복합 데이터형 | '복합 데이터형 동치' 절 참조 |

* 플래시 4의 문자열 동치 연산자는 eq이다. 하위 호환성을 위해 플래시 5에서도 eq가 지원되긴 하지만, 플래시 4 .swf 파일로 내보낼 파일이 아니라면 ==를 쓰는 것이 바람직하다.

## 복합 데이터형 동치

복합 데이터(객체, 배열, 함수, 무비 클립과 같은 데이터)를 저장하고 있는 변수에는 데이터 자체가 아니라 데이터에 대한 '레퍼런스(reference)'가 저장되므로 두 변수가 같은 아이템을 가리킬 수 없다. 이런 경우에는 두 개의 피연산자가 같은 내용을 담고 있는 두 개의 서로 다른 아이템이 아닌 같은 복합 데이터를 가리킬 때만 두 피연산자가 같다고 간주한다. 두 피연산자를 변환하여 같은 원시 데이터 값이 된다고 해도 두 피연산자가 같은 것으로 간주되지 않는다.

아래 예제에는 액션스크립트에서 데이터 자체가 아닌 같은 복합 데이터를 가리키는 레퍼런스를 비교하는 방법을 보여준다. 첫째 예에서는 같은 원소를 가지고 있긴 하지만, 서로 다른 두 개의 배열을 가리키는 피연산자(nameList1과 nameList2)를 사용한다. 따라서 레퍼런스가 다르기 때문에 비교 결과가 false로 나온다.

```
nameList1 = ["Linkovich", "Harris", "Sadler"];
nameList2 = ["Linkovich", "Harris", "Sadler"];
nameList1 == nameList2  // false
```

이번 예제에서는 cities와 canadianCities가 같은 배열을 가리킨다.

```
canadianCities = ["Toronto","Montreal","Vancouver"];
cities = canadianCities;
cities == canadianCities  // true
```

아래 예제에서 myFirstBall과 mySecondBall은 같은 생성자를 이용하여 만들어진(즉 같은 클래스에서 유도된 객체) 객체이지만 별개의(서로 다른) 인스턴스이다.

```
myFirstBall = new Ball();
mySecondBall = new Ball();
myFirstBall == mySecondBall  // false
```

따라서 복합 데이터 값에 대한 동치 비교는 값을 기준으로 하는 것이 아니라 레퍼런스를 기준으로 비교하는 것이다. 이 두 차이점에 대한 자세한 정보는 '15장. 고급 주제'의 '데이터 복사, 비교 및 전달' 부분에서 얻을 수 있다.

배열 레퍼런스를 복사하지 않고 배열 내용을 복사하려면, Array.slice() 메소드를 사용하면 된다. 아래 예에서는 dishes 배열로부터 kitchenItems 배열로 원소를 복사한다.

```
dishes = [ "cup", "plate", "spoon"];
kitchenItems = dishes.slice(0, dishes.length);
trace(kitchenItems == dishes);  // false가 출력된다.
```

이렇게 하면 kitchenItems와 dishes에는 서로 다른 복사본이 저장되므로 다른 복사본에 영향을 미치지 않고 내용을 마음대로 바꿀 수 있다.

## 동치와 데이터형 변환

두 피연산자가 같은 데이터형일 때 동치 검사를 한 결과는 이미 배웠다. 그런데 문자열과 숫자같이 서로 다른 데이터형을 가지는 피연산자를 비교할 때는 어떤 일이 일어날까?

```
"asdf" == 13;
```

피연산자의 데이터형이 서로 다르면 인터프리터에서는 비교 연산을 처리하기 전에 먼저 형을 변환한다. 인터프리터에서 사용하는 규칙은 다음과 같다.

1. 피연산자의 데이터형이 같으면 두 값을 비교하고 결과를 리턴한다(null과 undefined를 비교해도 true가 리턴된다).
2. 피연산자 중 하나는 숫자, 나머지 하나는 문자열이면 문자열을 숫자로 변환하고 1단계로 돌아간다.
3. 둘 중 하나가 부울이면 부울을 숫자(true = 1, false = 0)로 변환하고 1단계로 돌아간다.
4. 둘 중 하나가 객체이면 그 객체의 valueOf() 메소드를 호출하여 원시 데이터형으로 변환한다. 이렇게 할 수 없으면 false를 리턴한다. 데이터 변환이 성공하면 1단계로 돌아간다.
5. 위 과정을 모두 거쳐도 결과가 나오지 않으면 false를 리턴한다.

둘 중 하나는 객체이고 나머지 하나는 부울이면, 부울값이 숫자로 변환되고 그 숫자를 객체의 원시 데이터 값과 비교한다. 즉 someObject == true라는 표현식은 someObject라는 객체가 실제 존재한다고 하더라도 someObject는 비교 과정에서 숫자 또는 문자열로 변환되고 true는 숫자 1로 변환되기 때문에 보통 false일 가능성이 높다. 비교 과정에서 someObject를 부울형으로 다루고 싶다면 아래와 같이 Boolean() 함수를 사용하면 된다.

```
Boolean(someObject) == true    // someObject가 존재하면 true,
                               // 그렇지 않으면 false가 리턴된다.
```

동치 연산에 의한 변환은 주로 숫자형으로 변환된다. 방금 전에 설명한 다양한 형 변환의 결과를 잘 모르겠다면 3장의 '데이터형 변환'을 참조하기 바란다.

## 부등 연산자

부등 연산자는 동치 연산자와 반대의 부울형 결과를 리턴한다. 잠시 후에 나오는 [예제 5-2]에서도 볼 수 있듯이 상황에 따라 'x와 y가 같으면 아무것도 하지 않는다' 보다는 'x가 y와 같지 않으면 이 작업을 처리한다' 같은 표현이 더 적절할 때가 있다. 부등 연산자는 일반적으로 다음과 같은 형식으로 쓰인다.

```
operand1 != operand2
```

예를 들면 다음과 같이 쓸 수 있다.

```
var a = 5;
var b = 6;
var c = 6;
a != b  // true
b != c  // false
```

부등 연산자를 처리할 때는 동치 연산자와 똑같은 형 변환 규칙이 적용되며, 그 결과는 동치 연산자의 경우와 반대가 된다. 피연산자 중 하나가 NaN인 경우에도 마찬가지이다.

```
NaN != 7    // true
NaN != NaN  // true!
```

플래시 4 액션스크립트를 포함한 몇몇 언어에서는 부등 연산자로 != 대신 <> 를 사용하기도 한다. 잠시 후에 설명할 NOT 연산자(!)도 자세히 읽어보기 바란다.

## 일반적인 동치 연산 사용법

조건문에 들어가는 부울 표현식을 만들거나 변수에 부울 값을 대입할 때 동치 연산을 흔히 사용한다. [예제 5-2]에서 그러한 두 가지 경우와 != 와 == 연산자를 실제로 사용하는 예를 볼 수 있다.

***[예제 5-2] 동치 및 부등 연산자***

```
version = getVersion();       // 플레이어 버전을 확인한다.

// 버전에 "WIN"이라는 문자열이 있는지 확인한다.
// 들어 있으면 isWIN을 true로, 없으면 false로 설정한다.
isWin = (version.indexOf("WIN") != -1);

// isWIN이 true인 경우
if (isWin == true) {
  // MS 윈도우에서만 쓰이는 액션을 수행한다.
  trace("Please use IE4 or later on Windows.");
}
```

숙련된 프로그래머라면 if(isWIN == true) 대신 if(isWIN)을 사용하면 어떻게 될지 궁금할 것이다. isWIN도 부울 값을 가지므로 if 선언문에서 사용할 수 있기 때문이다. 물론 그렇게 하는 것도 괜찮지만 그렇게 하면 동치 연산자의 예를 만들 수 없지 않은가? 또한 초보 프로그래머에게는 위와 같이 조금 더 길더라도 자세하게 설명하는 것이 쉽게 이해될 것이다. if (isWIN == true)나 if(isWIN) 모두 문법적으로 전혀 문제가 없다. 이 부분에 대해서는 '7장. 조건문'에서 더 자세히 다루기로 하자.

## 비교 연산자

비교 연산자(또는 관계 연산자)는 두 값 중 어떤 값이 미리 정해진 순서를 기준으로 앞에 오는지 결정하는 연산자이다. 동치 및 부등 연산자와 마찬가지로 비교 연산자는 비교 연산자로 기술된 관계가 맞는지 틀리는지에 따라 부울 값인 true 또는 false 가운데 하나를 리턴한다.

비교 연산자는 문자열과 숫자에만 사용할 수 있다. 비교 연산자의 두 피연산자가 숫자인 경우에는 수학 시간에 배운 방법대로 비교 연산을 처리한다. 5<10은 true, -3<-6은 false, 이런 식으로 처리된다. 두 피연산자가 문자열이라면 '부록 B. Latin 1 문자 범주 및 키 코드'에 나온 문자 코드 포인트를 기준으로 비교 연산을 처리한다. 문자열 비교에 대한 자세한 내용은 4장의 '문자열 비교'를 참조하기 바란다.

문자열도 아니고 숫자도 아닌 데이터 값을 비교 연산에서 사용하면, 인터프리터는 그 값을 문자열 또는 숫자 데이터형으로 변환하려고 시도하게 된다. 비교 연산자에 대한 설명을 마치고 나서 비교 연산에서 데이터형 변환의 효과에 대해서도 알아보자.

### 〈 연산자

< 연산자는 다음과 같은 형식으로 쓰인다.

```
operand1 < operand2
```

피연산자가 숫자일 때는 operand1이 operand2보다 작으면 true를, 그렇지 않으면 false를 리턴한다.

```
5 < 6        // true
5 < 5        // false; 두 값이 같으므로 5는 5보다 작지 않다.
-3 < -6      // false; -3은 -6보다 크다.
-6 < -3      // true;  -6은 -3보다 작다
```

피연산자가 문자열이라면 operand1이 알파벳순으로 볼 때 operand2보다 앞에 있으면(부록 B 참조) true를, 그렇지 않으면 false를 리턴한다.

```
"a" < "z"         // true; 소문자 "a"는 소문자 "z"보다 앞에 있다.
"A" < "a"         // true; 대문자는 소문자보다 앞에 있다.
"Z" < "a"         // true; 대문자는 소문자보다 앞에 있다.
"hello" < "hi"    // true; "e"가 "i"보다 앞에 있다.
```

## 〉 연산자

> 연산자는 다음과 같은 형식으로 쓰인다.

```
operand1 > operand2
```

피연산자가 숫자일 때는 operand1이 operand2보다 크면 true를, 그렇지 않으면 false를 리턴한다.

```
5 > 6        // false
5 > 5        // false; 두 값이 같으므로 5는 5보다 크지 않다.
-3 > -6      // true; -3은 -6보다 크다.
-6 > -3      // false; -6은 -3보다 크지 않다.
```

피연산자가 문자열이라면 operand1이 알파벳순으로 볼 때 operand2보다 뒤에 있으면(부록 B 참조) true를, 그렇지 않으면 false를 리턴한다.

```
"a" > "z"         // false; 소문자 "a"는 소문자 "z"보다 앞에 있다.
"A" > "a"         // false; 대문자는 소문자보다 뒤에 있지 않다.
"Z" > "a"         // false; 대문자는 소문자보다 뒤에 있지 않다.
"hello" > "hi"    // false; "e"는 "i"보다 앞에 있다.
```

## 〈= 연산자

<= 연산자는 다음과 같은 형식으로 쓰인다.

```
operand1 <= operand2
```

피연산자가 숫자일 때는 operand1이 operand2 이하이면 true를, 그렇지 않으면 false를 리턴한다.

```
5 <= 6       // true
5 <= 5       // true; 5 < 5 의 결과와는 다르다.
-3 <= -6     // false
-6 <= -3     // true
```

피연산자가 문자열이라면 operand1이 알파벳순으로 볼 때 operand2보다 앞에 있거나 4장의 '문자열 비교'에서 설명한 규칙에 따라 두 문자열이 같으면 true를, 그렇지 않으면 false를 리턴한다.

```
"a" <= "z"        // true; 소문자 "a"는 소문자 "z"보다 앞에 있다.
"A" <= "a"        // true; 같지는 않지만 "A"는 "a"보다 앞에 있다.
"Z" <= "a"        // true; 대문자는 소문자보다 앞에 있다.
"hello" <= "hi"   // true; "e"는 "i"보다 앞에 있다.
```

<= 연산자를 사용할 때는 < 기호 뒤에 = 기호를 쓴다는 점에 주의하자. =<와 같이 생긴 연산자는 없다.

## 〉= 연산자

>= 연산자는 다음과 같은 형식으로 쓰인다.

```
operand1 >= operand2
```

피연산자가 숫자일 때는 operand1이 operand2 이상이면 true를, 그렇지 않으면 false를 리턴한다.

```
5 >= 6       // false
5 >= 5       // true; 5 > 5 의 결과와는 다르다.
-3 >= -6     // true
-6 >= -3     // false
```

피연산자가 문자열이라면 operand1이 알파벳순으로 볼 때 operand2보다 뒤에 있거나 4장의 '문자열 비교'에서 설명한 규칙에 따라 두 문자열이 같으면 true를, 그렇지 않으면 false를 리턴한다.

```
"a" >= "z"        // false; 소문자 "a"는 소문자 "z"보다 앞에 있다.
"A" >= "a"        // false; "A"는 "a"보다 앞에 있으며 둘은 같은 문자열이
아니다.
"Z" >= "a"        // false; 대문자는 소문자보다 앞에 있다.
"hello" >= "hi"   // false; "e"는 "i"보다 앞에 있다.
```

>= 연산자를 사용할 때도 > 기호 뒤에 = 기호를 쓴다는 점에 주의하자. => 같이 생긴 연산자는 없다.

## 비교 연산과 데이터형 변환

비교 연산자는 대개의 경우 숫자를 비교할 때 사용한다. 형 변환은 비교 연산자에 의해 시작되므로 숫자를 위주로 처리한다. 어떤 비교 연산자라도 피연산자의 데이터형이 서로 다르면, 또는 피연산자가 숫자도 아니고 문자열도 아니면 다음과 같은 단계를 거쳐서 형 변환을 시도한다.

1. 두 피연산자가 모두 숫자이면 두 피연산자를 수학적으로 비교하여 결과를 리턴한다. 둘 중 하나라도 NaN이 있으면 != 연산을 제외한 모든 비교 연산에서 false를 리턴한다.

2. 두 연산자가 모두 문자열이면 부록 B에 나온 코드 포인트를 기준으로 알파벳 순서대로 문자열을 비교하여 그 결과를 리턴한다.

3. 하나는 숫자, 나머지 하나는 문자열이면 문자열을 숫자로 변환하고 1단계로 돌아간다.

4. 둘 중 하나라도 부울, null 또는 undefined 값이 있으면 그 피연산자를 숫자로 변환하고 1단계로 돌아간다.

5. 둘 중 하나라도 객체가 있으면 그 객체의 valueOf() 메소드를 호출하여 원시 데이터값으로 변환하고 1단계로 돌아간다. valueOf() 메소드를 실행할 수 없거나 원시 데이터값이 리턴되지 않으면 false를 리턴한다.

6. false를 리턴한다.

비교 과정에서 형 변환을 한다고 해서 원래 아이템에 저장된 값이나 데이터형이 바뀌지는 않는다는 점을 알아두자. 임시로 값을 변환한 결과는 표현식을 처리하고 나면 사라진다.

다음은 두 개의 부울 값을 비교하는 간단한 예이다.

```
false < true      // true: 0은 1보다 작다.
```

비교 연산자에서는 언제나 복합 데이터형을 문자열 또는 숫자로 변환하여 연산을 처리한다. 아래 예에서 someObj와 someOtherObj는 모두 Object 클래스의 멤버이며 문자열 값도 "[object, Object]"로 똑같다.

```
someObj = new Object();
someOtherObj = new Object();
someObj <= someOtherObj;    // true!
```

다음 예에서 "A"의 코드 포인트는 65지만, "A"를 숫자로 변환하면 NaN이 되므로 표현식 값이 false가 된다. 문자열의 코드 포인트를 조사할 때는 charCodeAt() 함수를 사용한다.

```
"A" <= 9999                 // false
"A".charCodeAt(0) < 9999    // true
```

## 문자열 연산자

플래시 4에서 문자열 데이터를 사용할 때는 별도의 문자열 연산자(합치기(&), 동치(eq), 부등(ne), 비교(lt, gt, le, ge))가 필요하다. 플래시 5 이후부터는 이와 같은 특별한 문자열 연산자가 폐기되었다(여전히 지원되긴 하지만 새로 만드는 프로그램에서는 사용하지 않는 것이 좋다). 플래시 4를 위한 .swf 형식으로 내보낼 프로그램에서는 예전에 사용하던 연산자(eq, ne, lt, gt, le, ge)를 사용해야 하지만, 그렇지 않다면 ==, !=, <, >, <=, >= 연산자를 사용하자. 마찬가지로 문자열 병합 연산을 할 때도 플래시 4 형식으로 내보내야 한다면 & 대신 add를 사용해야 하겠지만(& 연

산자는 플래시 5 이후에서 비트 AND 연산자로 쓰인다), 그렇지 않다면 + 연산자를 사용하자.

플래시 4와 플래시 5의 문자열 연산자에 대한 자세한 내용은 4장의 '문자열 조작' 및 〔표 4-2〕에서 찾아볼 수 있다.

## 논리 연산자

4장에서는 부울 값을 이용하여 논리적인 결정을 내리는 법을 살펴보았다. 그 때 우리가 배운 내용은 하나의 요인만을 가지고 어떤 결정을 내리는 것(무비 로딩이 아직 끝나지 않았다면 무비를 시작하지 않는다. 우주선에 두 배의 파워를 가진 미사일이 있으면 명중했을 때 데미지를 키운다...)이었다. 하지만 모든 프로그래밍 논리가 그리 간단한 것은 아니다. 의사결정과 관련된 논리적인 결정을 내리다 보면 여러 가지 요인을 고려해야 하는 경우가 많이 있다.

예를 들어 사용자가 로그인해야 하는 플래시 사이트를 만든다고 가정해 보자. 사용자가 손님으로 로그인하면 볼 수 있는 내용이 제한된다. 등록된 사용자가 로그인하면 무비가 재생되는 동안 그 사용자가 특별한 내용을 볼 수 있다는 것을 나타내는 변수를 설정한다.

```
var userStatus = "registered";
```

제한된 영역에 접근하는 것을 허용할지 결정하는 과정에서 다음과 같이 부울형 검사를 해야 한다.

```
if (userStatus == "registered") {
  // 등록된 사용자이므로 사용할 수 있다.
}
```

등록된 사용자만 볼 수 있는 내용은 공개하지 않으면서 예비 투자자들에게 등록하지 않고도 사이트의 주요 내용을 볼 수 있게 하고 싶다면, 새로운 사용자 범주인 "specialGuest"를 만들 수도 있다. 어떤 사람을 특별한 손님으로 처리하려면 userStatus 변수를 "specialGuest"로 설정해야 한다.

```
if (userStatus == "registered") {
  // 등록된 사용자만 볼 수 있는 내용
}

if (userStatus == "specialGuest") {
  // 위와 같은 등록된 사용자를 위한 내용
}
```

물론 프로그램을 이렇게 만들면 정말 골치 아픈 문제가 많이 생긴다. 매번 스크립트를 작성할 때마다 해야 할 일이 두 배로 늘어나고 버전을 관리하기도 어려워진다.

따라서 '사용자가 등록된 사용자이거나 특별한 손님인 경우에는 접근을 허락한다' 는 내용의 부울 검사를 처리하는 것이 좋다. 부울 논리 연산자를 이용하면 그러한 복합 논리 구문을 만들 수 있다. 여기서는 논리 OR 연산자(||)를 이용하면 여러 개의 부울 연산을 하나의 표현식으로 처리할 수 있다.

```
if (userStatus == "registered" || userStatus == "specialGuest") {
  // 등록된 사용자를 위한 내용
}
```

멋지지 않은가? 이렇게 하면 여러 가지 문제를 깔끔하게 해결할 수 있다.

## 논리 OR

논리 OR 연산자는 둘 중 적어도 하나의 조건을 만족하며 어떤 작업을 처리하도록 할 때 가장 많이 쓰인다. 예를 들어 '배가 고프거나 목이 마르면 부엌으로 간다' 같은 것도 논리 OR이 들어가는 문장이다. 논리 OR의 기호는 두 개의 수직 바를 붙여놓은 기호(||)이다. | 문자는 일반적으로 대부분의 키보드에서 시프트키를 누른 채로 오른쪽 위에 있는 역슬래시[1]를 누르면 입력되는 문자로 키보드에서는 가운데가 약간 끊어진 형태로 표시되어 있다. 논리 OR는 다음과 같은 형식으로 쓰인다.

```
operand1 || operand2
```

1) 역자주: 한글 키보드에서는 보통 역슬래시가 아닌 원 기호(₩)로 표시되어 있다.

두 피연산자가 모두 부울 표현식일 때 둘 중 하나라도 true이면 true를 리턴하고 두 피연산자가 모두 false이면 false를 리턴한다. 모든 조건을 따져보면 다음과 같다.

```
true  || false   // 첫 번째 피연산자가 true이므로 true
false || true    // 두 번째 피연산자가 true이므로 true
true  || true    // true (둘 중 하나만 true여도 true가 된다)
false || false   // 둘 다 false이므로 false
```

위에 나온 조건은 두 피연산자가 모두 부울형인 경우만을 고려했지만 부울형이 아니더라도 제대로 된 표현식이라면 모두 피연산자로 쓰일 수 있다. 부울형이 아닌 피연산자를 사용하면 논리 OR 연산의 리턴 값이 반드시 부울형(true 또는 false)인 것은 아니다. 기술적으로 볼 때 논리 OR를 계산할 때 operand1을 부울형으로 변환하는 단계가 가장 먼저 수행된다. 변환한 결과가 true이면 논리 OR에서는 operand1의 값을 리턴한다. 그렇지 않으면(즉 operand1을 부울형으로 변환한 결과가 true가 아니면), operand2의 값을 리턴한다. 몇 가지 예를 들어보면 다음과 같다.

```
null || "hi there!"  // operand1이 true로 변환되지 않으므로 operand2의
                     // 값인 "hi there!"가 리턴된다.

true || false        // operand1이 true이므로 operand1의 값인
                     // true가 리턴된다.

"hey" || "dude"      // operand1이 비어있지 않은 문자열이므로 false이다.
                     // 따라서 operand2의 값인 "dude"가 리턴된다.

false || 5 + 5       // operand1이 true가 아니므로 operand2의 값인
                     // 10이 리턴된다.
```

사실 논리 OR에서 리턴되는 부울이 아닌 값을 사용할 일은 거의 없다. 따라서 그러한 결과를 보통 부울형 결정을 내려야 하는 조건문에서 사용한다. [예제 5-3]을 살펴보자.

*[예제 5-3]*

```
var x = 10;
var y = 15;
if (x || y) {
  trace("One of either x or y is not zero");
}
```

셋째 줄에서 if 선언문의 부울 값을 사용하는 위치에 (x || y)라는 논리 OR 연산을 사용한다. x || y의 값을 결정하는 첫 단계는 10(첫째 피연산자인 x 값)을 부울형으로 변환하는 것이다. 〔표 3-3〕에서 볼 수 있듯이 임의의 0이 아닌 유한한 숫자를 부울형으로 변환하면 true가 된다. 첫째 피연산자가 true로 변환되는 값이면 논리 OR에서는 첫째 피연산자의 값을 리턴한다. 이 경우에는 10이 리턴된다. 따라서 인터프리터 입장에서 보면 위에 있는 if 선언문은 다음과 같이 된다.

```
if (10) {
  trace("One of either x or y is not zero");
}
```

하지만 10은 부울이 아닌 숫자이다. 그렇다면 어떤 일이 일어날까? if 선언문에서는 논리 OR 연산에서 리턴된 값을 다시 부울형으로 변환한다. 이 경우에는 10이 부울 값인 true(이번에도 〔표 3-3〕의 규칙이 적용된다)로 변환되므로 인터프리터에서는 결국 코드를 다음과 같이 인식할 것이다.

```
if (true) {
  trace("One of either x or y is not zero");
}
```

결국 위와 같은 표현식이 되므로 주어진 조건이 true가 되어 중괄호 안에 들어있는 trace( ) 선언문이 실행된다.

논리 OR 연산의 첫째 피연산자가 true이면 둘째 피연산자가 필요없으므로 둘째 피연산자의 값을 조사하는 것은 사실 비효율적이다. 따라서 액션스크립트에서는 첫째 피연산자를 부울형으로 변환한 값이 false인 경우에만 둘째 피연산자를 조사한다. 이러한 사실은 첫째 피연산자가 false가 아니면 둘째 피연산자를 계산하지 않기를 원하는 경우에 유용하게 써먹을 수 있다. 아래 예제에서는 주어진 숫자가 제한된 영역을 벗어나는지 조사한다. 만약 숫자가 너무 작다면 숫자가 너무 큰지 조사하는 둘째 조건은 검사해볼 필요도 없다.

```
if (xPos < 0 || xPos > 100) {
  trace ("xPos is not between 0 and 100 inclusive.");
}
```

## 논리 AND

논리 OR 연산자와 마찬가지로 논리 AND는 주로 조건에 따라 코드 블록을 실행할 때 쓰이는데, 이 경우에는 두 조건이 모두 만족되는 경우에만 true가 된다. 논리 AND 연산자는 다음과 같은 형식으로 쓰인다.

```
operand1 && operand2
```

제대로 된 표현식이라면 operand1과 operand2에 모두 사용할 수 있다. 가장 간단한 경우는 두 피연산자가 모두 부울 표현식인 경우인데, 두 피연산자 가운데 하나라도 false가 있으면 false를 리턴하고 둘 다 true인 경우에만 true를 리턴한다.

```
true  && false   // 두 번째 피연산자가 false이므로 false
false && true    // 첫 번째 피연산자가 false이므로 false
true  && true    // 둘 다 true이므로 true
false && false   // 둘 다 false이므로 false
```

두 개의 실례를 통해 논리 AND 연산자를 이용하는 법을 알아보자. 〔예제 5-4〕에서는 두 개의 변수가 모두 50보다 클 때 trace() 선언문을 실행한다.

*〔예제 5-4〕 조건 검사 표현식에서 쓰이는 논리 AND 연산*

```
x = 100;
y = 51;
if (x>50 && y>50) {
  trace("Both x and y are greater than 50");
}
```

x>50과 y>50은 모두 true이므로 trace() 선언문이 실행된다.

〔예제 5-5〕는 등록된 사용자와 특별한 손님에게만 접근을 허용하는 웹사이트의 예를 약간 변경한 것이다. 이번에는 1월 1일이 아닌 경우에만 사용자가 로그인할 수 있도록 한다. 여기서는 Date 객체를 이용하여 현재 날짜를 확인한다. getMonth()의 리턴 값이 0이면 1월을 의미한다.

*〔예제 5-5〕 복합 논리 표현식*

```
userStatus = "registered";
now = new Date();         // 새로운 Date 객체 생성
day = now.getDate();      // 1 이상 31 이하의 정수를 리턴한다(날짜).
month = now.getMonth();   // 0 이상 11 이하의 정수를 리턴한다(달).

// 들여쓰기와 괄호 부분을 자세히 보자. 가독성을 향상시키기
// 위해 if 선언문을 여러 줄에 걸쳐서 쓰는 경우도 종종 있다.
if ( ((userStatus == "registered") || (userStatus == "specialGuest"))
    && (month + day) > 1) {
  // 사용자에게 접근을 허가한다.
}
```

논리 AND 연산자는 논리 OR 연산자와 기술적으로 볼 때 거의 비슷하게 작동한다. 우선 operand1을 부울형으로 변환한다. 그렇게 변환한 값이 false이면 operand1의 값이 리턴된다. 결과가 true이면 operand2의 값이 리턴된다.

논리 AND 연산의 첫째 피연산자를 부울형으로 변환한 값이 false라면 둘째 피연산자는 계산할 필요도 없고 괜히 계산하면 시간만 더 잡아먹게 된다. 따라서 액션스크립트에서는 첫째 피연산자가 true인 경우에만 둘째 피연산자를 계산한다. 이러한 사실은 첫째 피연산자가 true가 아니면 둘째 피연산자를 계산하고 싶지 않은 경우에 유용하게 써먹을 수 있다. 아래 예제에서는 분모가 0이 아닌 경우에만 나눗셈을 계산한다.

```
if (numItems != 0 && total / numItems > 3) {
  trace ("You've averaged more than 3 dollars per transaction");
}
```

## 논리 NOT

논리 NOT 연산자(!)는 일항 연산자이며 피연산자의 부울 값의 반대되는 값을 리턴한다. 일반적인 사용법은 다음과 같다.

```
!operand
```

제대로 된 표현식이라면 어떤 표현식이라도 operand로 사용할 수 있다. operand가 true라면 논리 NOT에서는 false를 리턴한다. 만약 operand가 false라면

true를 리턴한다. operand가 부울형이 아니면 그 값을 부울형으로 변환한 다음, 그 값과 반대되는 값을 리턴한다.

부등 연산자(!=)와 마찬가지로 논리 NOT은 '이런 조건' 보다는 '이렇지 않은 조건' 을 조사할 때 편리하게 쓰인다. 날짜가 1월 1일이 아닌 경우에만 어떤 작업을 처리하도록 했던 〔예제 5-5〕를 떠올려보면 이해하는 데 도움이 될 것이다. 처음 볼 때는 다음과 같은 표현을 쓰는 것이 맞을 것 같다는 생각도 든다.

```
month != 0 && day != 1
```

하지만 다시 생각해 보면 month != 0은 1월을 제외한 모든 경우에 true이다(여기까지는 괜찮다). 그런데 day != 1은 매달 첫 날에 false가 된다. 따라서 위 표현식을 사용하면 1월 1일뿐만 아니라 매월 1일마다 위 표현식이 false가 되므로 우리의 의도와는 다르게 작동한다.

오늘이 1월 1일인지 조사하는 것은 훨씬 쉽다.

```
month == 0 && day == 1
```

위 표현식은 1월 1일에만 true가 된다. 따라서 이 표현식에 NOT 연산자를 추가하면 날짜가 1월 1일이 아닌지를 확인할 수 있다.

```
!(month==0 && day==1)
```

이렇게 하면 month+day > 1이라는 식으로 날짜를 조사하는 것보다 코드를 작성한 의도가 훨씬 명확하게 드러난다.

NOT 연산자를 사용하는 또 다른 전형적인 예는 한 변수를 true에서 false로, 또는 그 반대로 바꾸는 경우이다. 예를 들어 사운드를 켜고 끄는 데 쓰이는 버튼이 하나 있다고 가정해 보자. 그러면 다음과 같이 코드를 만들 수 있다.

```
soundState = !soundState  // 현재 사운드 상태를 반대로 돌린다.
if (soundState) {
  // 사운드를 켜면 소리가 나도록 한다.
} else {
  // 사운드를 끄면 볼륨을 0으로 만들어서 소리가 나지 않게 만든다.
}
```

! 기호는 부등 연산자(!=)에서도 쓰인다는 점에 주의하자. 프로그래밍 토큰(기호)으로서 ! 문자는 보통 "반대", "부정"의 의미로 쓰인다. 일반적으로 수학에서 팩토리얼을 나타낼 때 사용하는 ! 기호와는 전혀 상관없다.

## 논리 표현식과 데이터 값

논리 연산을 이용하여 부울 결과를 계산하는 것은 수학 연산자를 이용하여 숫자로 된 결과를 계산하는 것과 비슷하다. 예를 들어 아래 있는 네 개의 피연산자와 세 개의 연산자를 이용하면 하나의 결과가 나온다.

```
userStatus == "registered" || userStatus == "specialGuest"
```

사용자가 등록된 사용자는 아니지만 특별한 손님이면 위 표현식은 다음과 같이 쓸 수 있다.

```
false || true
```

따라서 위 표현식은 true가 된다.

계산하고 나면 하나의 숫자가 되는 다음과 같은 수학 연산과 위 부울 연산을 비교해 보자.

```
(1 + 2) * (3 + 4)  // 1, 2, 3, 4라는 네 개의 값이
3 * 7              // 3, 7 이라는 두 개의 값이 되고
21                 // 결국은 21이라는 하나의 값이 된다.
```

계산하고 나면 하나의 부울 값을 리턴하는 복합 표현식은 부울 값을 사용할 수 있는 곳(조건문의 조건을 나타내는 표현식 같은)이라면 어디든지 사용할 수 있다.

## 그룹 연산자

함수 호출 이외에도 괄호 기호인 ()는 코드 구문을 하나의 그룹으로 묶어서 다양한 연산자의 우선 순위를 변경하는 데도 쓰인다. 또한 if와 같이 괄호를 반드시 사용해야 하는 선언문도 있다. 괄호는 일반적으로 다음과 같은 형식으로 쓰인다.

```
(expression)
```

이렇게 하면 괄호 안에 있는 expression을 계산하여 그 결과를 리턴한다. 몇 가지 예를 들면 다음과 같다.

```
if (x == y) {                    // if 선언문에서는 반드시 괄호를
                                 // 사용해야 한다.
  trace("x and y are equal");    // 함수를 호출할 때도 괄호를 사용한다.
}
(5 + 6) * 7                      // 계산 순서를 바꿀 수 있다.
(x >= 100) || (y <= 50)          // 괄호가 꼭 필요한 건 아니지만
                                 // 가독성을 위해 사용한다.
x >= 100 || y <= 50              // 괄호가 없으면 약간 읽기가 힘들어진다.
```

## 쉼표 연산자

쉼표 연산자(,)는 그다지 흔하게 쓰이지는 않지만 한 개의 표현식을 쓸 자리에 두 개의 표현식을 쓰고자 할 때 사용한다. 쉼표 연산자는 보통 다음과 같은 형식으로 쓰인다.

```
operand1, operand2
```

제대로 된 표현식이라면 모두 피연산자로 사용할 수 있다. 쉼표 연산자를 실행하면 operand1을 계산하고 operand2를 평가한 다음 operand2의 값을 리턴한다. 일반적으로 쉼표 연산자의 리턴 값은 쓰이지 않는다.

쉼표 연산자는 〔예제 5-6〕에 나온 것처럼 for 순환문에서 여러 개의 변수를 한꺼번에 초기화하여 사용할 때 주로 쓰인다. 예제의 첫 줄에서 i=0과 j=10은 각각 쉼표 연산자의 피연산자이다(이와 같은 표현식을 사용하면 i와 j의 초기 값이 각각 0과

10으로 설정된다). 또한 두 번째 쉼표 연산자에서는 i++와 j--라는 표현식이 피연산자가 된다(이렇게 하면 매번 루프를 돌 때마다 i의 값은 1씩 증가하고 j의 값은 1씩 감소한다). 이처럼 for 순환문에서는 문법상 제한된 공간에 여러 개의 표현식을 집어넣기 위해 쉼표 연산자를 사용한다.

*〔예제 5-6〕 for 순환문에서 쉼표 연산자를 사용하는 법*

```
for (var i=0, j=10; i!=j; i++, j--) {
  trace ("i: " + i + " j: " + j);
}

// 결과는 다음과 같다.
 i: 0 j: 10
 i: 1 j: 9
 i: 2 j: 8
 i: 3 j: 7
 i: 4 j: 6
```

## void 연산자

void 연산자는 아무런 값도 리턴하지 않고 어떤 표현식을 계산할 때 사용된다. 문법은 다음과 같다.

```
void expression
```

expression에는 계산하고자 하는 표현식이 들어간다. 자바스크립트에서는 결과를 브라우저의 상태 표시줄에 보여주지 않으면서 하이퍼텍스트 링크 안에서 자바스크립트 표현식을 계산하기 위해 void를 사용한다. 액션스크립트에서는 void 연산자를 사용할 일이 거의 없지만 ECMA-262와의 호환성을 위해 void 연산자가 포함되어 있다.

# 기타 연산자

나머지 연산자들은 다른 장에서 배울 주제와 관련된 연산자이다. 여기서는 간단히 참고할만한 내용만 설명하고 자세한 사용법은 직접 관련된 내용을 다루는 장에서 설명할 것이다.

## 비트 연산자

메모리, 계산 속도, 전송 속도 등을 최적화시켜 성능을 크게 개선할 수 있는 대형 시스템을 개발하려면 15장에 있는 비트 연산자에 대한 내용을 읽어보는 것이 도움이 된다. 그럴 필요가 없다면 최적화에는 안 좋지만 훨씬 간단하게 비트 연산자와 똑같은 작업을 처리할 수 있는 부울 논리 연산자를 사용하는 것이 낫다.

## typeof 연산자

typeof 연산자는 표현식의 데이터형을 결정하는 데 쓰인다. typeof 연산자에서는 다음과 같이 하나의 피연산자만을 사용한다.

```
typeof operand;
```

제대로 된 표현식이라면 어떤 표현식도 operand 자리에 사용할 수 있다. typeof 연산의 리턴 값은 oeprand를 계산한 결과의 데이터형을 나타내는 문자열이다. 자세한 내용은 3장에서 찾아볼 수 있다.

## new 연산자

new 연산자는 새로운 복합 데이터(배열이나 객체)를 만드는 연산자이다. 이 때 객체는 내장 클래스 또는 사용자 정의 클래스의 멤버이다. new 연산자는 다음과 같이 사용한다.

```
new constructor
```

constructor는 새로 만들 객체의 속성을 정의하는 함수이다. '11장. 배열'과 '12장. 객체와 클래스'를 참조하기 바란다.

## delete 연산자

객체, 객체 속성, 배열 원소 또는 변수를 스크립트에서 제거할 때 delete 연산자를 사용한다. delete 연산자는 다음과 같이 쓰인다.

```
delete identifier
```

identifier가 데이터 컨테이너(변수, 속성 또는 배열)가 아니라서 delete 연산을 처리할 수 없으면 false를 리턴한다. 작업을 제대로 처리한 경우에는 true를 리턴한다(11장 참조).

## 배열 원소/객체 속성 연산자

11장 및 12장에서 배우게 되겠지만 배열 원소나 객체 속성을 읽어들이거나 설정할 때는 [] 연산자를 사용한다. 배열을 액세스할 때는 다음과 같이 사용한다.

```
array[element]
```

이 때 array는 배열의 이름 또는 배열 리터럴이고 element는 0부터 시작하는 양의 정수로 이루어진 인덱스로, 사용하고자 하는 배열 원소를 가리키는 값이다(또는 계산 결과가 그러한 인덱스가 되는 표현식을 사용해도 된다).

객체에 접근할 때는 다음과 같은 형식을 사용한다.

```
object[property]
```

object는 객체의 이름 또는 객체 리터럴이고 property는 사용하고자 하는 객체 속성의 이름을 나타내는 문자열을 뜻하는 표현식이다.

대입 연산자(=)의 왼쪽에 사용하면, 배열 원소가 객체 속성에 대입 연산자의 오른쪽에 있는 값을 대입한다.

```
var colors = new Array();  // 새로운 배열을 만든다.
colors[0] = "orange";      // 첫 번째 원소를 설정한다.
colors[1] = "green";       // 두 번째 원소를 설정한다.

var ball = new Object();   // 새로운 객체를 만든다.
var myProp = "xVelocity";  // 변수에 문자열을 저장한다.
ball["radius"] = 150;      // radius(반지름) 속성을 설정한다.
ball[myProp] = 10;         // myProp 변수를 통해 xVelocity 속성을 설정한다.
```

다른 곳에서 사용할 때는 주어진 원소 또는 속성 값을 리턴한다.

```
diameter = ball["radius"] * 2;  // diameter를 300으로 설정한다.
trace(colors[0]);               // "orange"가 출력된다.
```

## 점 연산자

점 연산자는 객체 속성 및 여러 단계로 이루어진 무비 클립을 참조할 때 가장 많이 쓰이는 방법이다. 기능적으로 볼 때 점 연산자는 [] 연산자와 똑같은 역할을 한다. 즉 객체 속성 값을 읽어들이거나 설정하는 기능을 가지고 있다. 하지만 두 연산자의 구조가 서로 다르기 때문에 용도가 다르다. 점 연산자의 일반적인 사용법은 다음과 같다.

```
object.property
```

object는 객체의 이름 또는 객체 리터럴이며 property는 object의 속성을 나타내는 인식자이다. property에 임의의 표현식이나 문자열 리터럴을 사용할 수 없다는 점에 주의하자. property에는 속성 이름만을 사용할 수 있다. 배열 원소는 숫자이므로 이름이 없기 때문에, 배열 원소를 사용할 때는 점 연산자를 쓸 수 없다.

대입 연산에서 왼쪽에 있는 피연산자로 사용되면 점 연산자를 통해 속성 값에 새로운 값을 대입할 수 있다.

```
var ball = new Object();
ball.radius = 150;
ball.xVelocity = 10;
```

다른 곳에서 쓰이면 주어진 객체 속성 값을 리턴한다.

```
diameter = ball.radius;
newX = ball.xPosition + ball.xVelocity;
```

'12장. 객체와 클래스', '13장. 무비 클립'을 참조하기 바란다.

## 조건 연산자

조건 연산자는 조건 선언문을 간편하게 표기하기 위한 편리한 연산자이다. 이 연산자에서는 다음과 같이 세 개의 피연산자를 사용한다.

```
condition ? result_if_true : result_if_false;
```

조건 연산을 실행하면 가장 먼저 첫째 피연산자(condition)의 값이 계산된다. condition이 true이거나 변환했을 때 true가 되는 값이라면, 둘째 피연산자(result_if_true)가 리턴된다. 그렇지 않으면 셋째 피연산자(result_if_false)가 리턴된다(7장 참조).

## 함수 호출 연산자

trace( ) 함수나 문자열 조작 함수에서 이미 사용해 보았듯이 함수 호출 연산자인 ( )는 함수를 호출하는 데 쓰인다. 함수 호출은 일반적으로 다음과 같이 표현한다.

```
function_name(argument_list)
```

첫째 피연산자인 function_name은 함수의 이름이며 반드시 표현식이 아닌 인식자를 사용해야 한다. 만약 그러한 함수가 없다면 인터프리터에서 오류가 발생한다. argument_list는 함수에 전달되는 0개 또는 그 이상의 인자(즉 인자가 없을 수도 있다)를 나타내며, 인자가 여러 개 있으면 각 인자를 쉼표로 구분한다. 함수 호출 연산의 리턴 값은 함수 자체에 의해 제공되는 리턴 값이다.

함수 호출 연산자를 이용하면 어떤 내장 함수 또는 사용자 정의 함수라도 호출할 수 있다.

```
trace("an internal function");  // 내장 함수. 인자는 1개.
myClip.play();                  // 무비 클립의 메소드. 인자는 없다.
myRectangle.area(6, 9);         // 사용자 정의 함수. 인자는 2개.
init();                         // 사용자 정의 함수. 인자는 없다.
```

## 앞으로 배울 내용

다른 모든 언어와 마찬가지로 액션스크립트에도 문법이 있고 말하는 방법이라는 것이 있다. 지금까지 문장 구조, 명사, 형용사, 접속사 등에 대해 배웠다. 이제 실제로 작업에 필요한 동사에 해당하는 선언문을 알아보자. 선언문까지 배우고 나면 문장과 비슷한 구조를 가지는 명령어를 만들기 위한 준비를 끝마치게 된다.