15

고급 주제

이 장에서는 다양한 고급 액션스크립트 프로그래밍 기법과 지금까지 다뤄보지 않은 까다로운 문제에 대해 살펴보자.

데이터 복사, 비교 및 전달

데이터를 조작하는 데는 크게 세 가지 종류가 있다. 데이터를 복사(예: 변수 x의 값을 변수 y에 대입)하거나 비교(예: x와 y가 같은지 비교)하거나 전달(어떤 변수를 함수의 인자로 전달)하는 것이다. 원시 데이터는 복합 데이터와는 상당히 다른 방식으로 복사, 비교, 전달된다. 원시 데이터를 변수에 복사하면 그 변수에는 그 변수만의 복사본이 저장되므로, 메모리에서도 서로 다른 위치에 놓이게 된다. 따라서 다음과 같은 코드를 실행시키면 "Dave"라는 문자열이 메모리에 두 번 저장된다(하나는 name1 자리에, 나머지 하나는 name2 자리에).

```
name1 = "Dave";
name2 = name1;
```

원시 데이터를 복사하면 데이터의 리터럴 값이 새로운 변수에 할당된 메모리 공간에 저장되기 때문에 원시 데이터는 '값으로' 복사된다고 한다. 이와는 대조적으로 복합 데이터를 어떤 변수에 복사하면 그 데이터에 대한 레퍼런스만이 변수의 메모리 공간에 저장된다(실제 데이터가 그 자리에 저장되는 것은 아니다). 인터프리터에서는 이 레퍼런스를 이용하여 실제 데이터가 있는 위치(메모리 주소)를 알아낼 수 있다. 복합 데이터가 저장된 변수를 다른 변수에 복사하면 데이터 자체가 아니라 레퍼런스(포인터라고도 부른다)만 복사될 뿐이다. 따라서 복합 데이터는 '레퍼런스로' 복사된다고 한다.

커다란 배열이나 복합 데이터형을 복사할 때 실제 데이터를 복사하는 것은 너무 비효율적이라는 점을 고려하면 이런 방식을 사용하는 이유를 쉽게 이해할 수 있을 것이다. 실제로 프로그램을 짤 때는 이러한 점을 확실히 이해해야 한다. 똑같은 복합 데이터를 여러 변수에 대입한다면 각 변수에 그 데이터의 복사본이 저장되지 않는다(원시 데이터인 경우에는 실제로 데이터의 복사본이 저장된다). 대신 데이터 자체는 하나만 존재하고 모든 변수에서 같은 데이터를 가리킬 뿐이다. 하나의 데이터 값만 바꾸더라도 모든 변수가 업데이트된다.

실제 프로그래밍 과정에서 이러한 특성이 어떤 영향을 발휘하는지 알아보자. 만약 두 개의 변수가 똑같은 원시 데이터를 가리킨다면 각 변수에는 같은 값을 가지는 데이터의 복사본이 저장된다. 다음과 같이 변수 x에 12라는 값을 대입해 보자.

```
var x = 12;
```

이제 x 값을 새로운 변수인 y에 대입해 보자.

```
var y = x;
```

y 값이 12가 된다는 것은 쉽게 예상할 수 있을 것이다. 하지만 y에는 12라는 값을 가진 별도의 복사본이 저장되며, 그 데이터는 x에 저장된 값과는 서로 다르다. 따라서 x 값을 바꾸더라도 y 값은 변하지 않는다.

```
x = 15;
trace(x); // Output 창에 15가 출력된다.
trace(y); // Output 창에 12가 출력된다.
```

y에 x를 대입할 때 y에서는 숫자 12(즉 x에 저장된 원시 데이터)의 새로운 복사본을 받았기 때문에, x가 바뀌더라도 y는 바뀌지 않게 된다.

이제 복합 데이터를 이용하여 비슷한 작업을 해보자. 원소가 세 개인 배열을 만들고 그 변수를 변수 x에 저장하자.

```
var x = ["first element", 234, 18.5];
```

이제 앞에서 한 것처럼 y에 x를 대입하자.

```
var y = x;
```

이제 y 값은 x 값과 같아졌다. 하지만 x 값은 어떻게 될까? x는 복합 데이터인 배열을 참조하기 때문에 x 값은 사실 ["first element", 234, 18.5]라는 배열 자체가 아니라 그 데이터의 레퍼런스일 뿐이다. 따라서 y에 x를 대입하면 y에 저장되는 것은 배열 자체가 아니라 x에 저장되어 있던 배열에 대한 레퍼런스일 뿐이다. 따라서 x와 y는 메모리 어딘가에 기록된 같은 배열을 가리킨다.

변수 x를 통해 다음과 같이 배열을 바꾼다면

```
x[0] = "1st element";
```

그 결과는 y에도 바로 반영된다.

```
trace(y[0]);  // "1st element"가 출력된다.
```

이와 마찬가지로 y를 통해 배열을 바꾼다면 그 결과도 x를 통해 확인할 수 있다.

```
y[1] = "second element";
trace (x[1]);  // "second element"가 출력된다.
```

연결을 끊고 싶다면 slice() 함수를 이용하여 완전히 새로운 배열을 만들 수 있다.

```
var x = ["first element", 234, 18.5];
// x의 각 원소를 y에 저장된 새로운 배열에 저장한다.
var y = x.slice(0);
y[0] = "hi there";

trace(x[0]);  // "hi there"가 아닌 "first element"가 출력된다.
trace(y[0]);  // "first element"가 아닌 "hi there"가 출력된다.
```

이제 예제를 확장하여 원시 데이터 값과 복합 데이터 값을 비교하는 방법을 알아보자. 아래 코드에서는 x와 y에 똑같은 원시 값을 대입하여 두 개의 변수를 비교한다.

```
x = 10;
y = 10;
trace(x == y);  // true가 출력된다.
```

x와 y에는 원시 데이터가 들어가므로 그 두 데이터 값을 서로 비교한다. 값을 기반으로 하는 비교에서는 데이터를 있는 그대로 비교한다. x에 있는 10이라는 숫자는 y에 있는 10이라는 숫자와 같은 것으로 간주된다. 메모리에 저장된 내용이 똑같기 때문이다.

이제 x와 y에 같은 복합 데이터를 대입하고 두 변수를 다시 비교해 보자.

```
x = [10, "hi", 5];
y = [10, "hi", 5];
trace(x == y);  // false가 출력된다.
```

이번에는 x와 y에 복합 데이터가 저장되므로 두 값은 레퍼런스로 비교된다. x와 y에 대입한 배열에 저장된 값은 같지만 x와 y에서 같은 복합 데이터를 참조하는 것은 아니므로 두 변수를 비교할 때는 서로 다른 것으로 인식된다. 하지만 x에 y의 레퍼런스를 복사하면 어떻게 되는지 살펴보자.

```
x = y;
trace(x == y);  // true가 출력된다.
```

이제 레퍼런스가 똑같기 때문에 두 값은 같은 것으로 인식된다. 이처럼 비교 결과는 실제 배열 원소의 내용이 아니라 변수의 레퍼런스에 의해 결정된다.

'9장. 함수'의 '원시 데이터 및 복합 데이터 매개변수' 절에서 보았듯이 원시 데이터와 복합 데이터는 함수에 전달될 때도 서로 다른 방식으로 전달된다. 가장 크게 차이가 나는 점은 원시 변수를 함수에 인자로 전달하면 함수 안에서 그 데이터를 어떻게 변경하더라도 변수의 원본은 바뀌지 않는다는 점이다. 하지만 복합 변수를 전달할 때는 함수에서 변수를 조작하면 원본도 그대로 바뀐다. 즉 정수 변수 x를 함수에 전달하면 그 변수가 바뀐다고 하더라도 원본은 바뀌지 않는다. 하지만 y라는 배열을 함수에 전달하면 함수 안에서 그 배열을 조작한 내용이 함수 밖에 있는 원래

변수에도 그대로 반영된다(배열을 조작하면 y가 가리키는 위치에 있는 데이터가 바뀌기 때문이다).

비트 단위 프로그래밍

이제 방향을 약간 바꿔서 조금 생소한 주제를 다루어 보자. 이번 절에서 다룰 내용은 비트 연산자를 이용한 비트 단위 프로그래밍이다. 여기서 다루는 내용은 '5장. 연산자'에서는 너무 어려운 내용이라 건너뛰었지만, 경험을 쌓은 프로그래머나 용감한 초보자들을 위해 여기서 비트 단위 프로그래밍에 대해 살펴보겠다.

비트 연산자는 일련의 옵션을 최적화된 방법으로 추적하고 조작하기 위한 용도로 많이 사용한다. 기술적으로 본다면 비트 연산자도 수학 연산자에 속하지만 보통 수학적인 문맥보다는 논리적인 문맥에서 주로 쓰인다. 비트 연산자를 이용하면 정수에 있는 이진법 숫자(비트)를 하나씩 액세스할 수 있다. 비트 연산자의 작동 원리를 이해하려면 이진 형식으로 숫자를 표현하는 방법을 알아야 한다.

이진수는 이진법으로 숫자를 표기하는 1과 0만으로 이루어진 값으로 저장된다. 숫자의 각 자리수는 그 진법의 밑[1]을 곱한 횟수를 나타낸다. 이진법에서는 2를 밑으로 사용하므로 이진수의 맨 오른쪽부터 네 개의 자리는 각각 1(20)의 자리, 2(21)의 자리, 4(22)의 자리, 8(23)의 자리를 나타낸다. 아래 나온 이진수의 예를 통해 각 자리를 이용하여 이진수를 10진수로 어떻게 바꾸는지 알아보자.

```
1     // 10진수 1: (1 x 1) = 1
10    // 10진수 2: (1 x 2) + (0 x 1) = 2
11    // 10진수 3: (1 x 2) + (1 x 1) = 3
100   // 10진수 4: (1 x 4) + (0 x 2) + (0 x 1) = 4
1000  // 10진수 8: (1 x 8) + (0 x 4) + (0 x 2) + (0 x 1) = 8
1001  // 10진수 9: (1 x 8) + (0 x 4) + (0 x 2) + (1 x 1) = 9
```

이진수에서 각 자리는 비트("이진 숫자"를 의미하는 "binary digit"을 줄인 단어)라고 부른다. 예를 들어 4비트 숫자는 네 개의 숫자(각 자리에는 1이나 0이 들어감)로

1) 역자주: 이진법의 밑은 2, 8진법의 밑은 8, 10진법의 밑은 10과 같이 밑은 그 진법을 이루는 숫자의 개수라고 생각하면 된다.

이루어진 숫자이다. 가장 오른쪽에 있는 비트는 0, 그 왼쪽의 비트는 1, 그 왼쪽의 비트는 2와 같은 식으로 번호를 붙인다. 아래 숫자는 0, 6, 7 비트에 1이 들어있는 8 비트 숫자이다. 각 숫자 위에 있는 숫자는 비트 번호이다.

```
bit: 76543210
     11000001
```

어떤 비트에서 사용할 수 있는 가장 큰 숫자는 밑(radix라고 부르기도 한다)에서 1을 뺀 값이다. 예를 들어 10진법에 한 자리로 표현할 수 있는 가장 큰 숫자는 9이다. 이진법에서는 밑이 2이므로 각 비트에는 0 또는 1밖에 사용할 수 없다. 1과 0은 부울 값으로 따지면 각각 true와 false에 해당하므로 이진수를 일련의 on, off 스위치로 사용하면 매우 유용하다. 비트 연산자를 이용하여 처리하는 작업은 대부분 이러한 기능과 관련되어 있다.

어떤 비트를 1로 설정하면 on 또는 true 상태가 되며, 0으로 설정하면 off 또는 false 상태가 된다. 각 비트는 두 가지 가능한 상태(on/off 또는 true/false)를 나타낼 수 있기 때문에 '플래그(flag)' 또는 '스위치(switch)'로 생각할 수도 있다.

비트 단위 프로그래밍에서는 대개 일련의 속성을 on 또는 off로 설정하는 작업을 처리한다. 비트 연산자를 이용하면 여러 개의 변수를 사용하지 않고도 하나의 숫자 값으로 여러 옵션을 간단하게 나타낼 수 있다. 이렇게 하면 메모리 소모도 줄일 수 있고 속도도 향상시킬 수 있다.

자동차를 판매하는 플래시 사이트를 만든다고 가정해 보자. 여기서는 간단하게 설명하기 위해 한 종류의 자동차만을 팔지만, 사용자는 네 가지 옵션(에어콘, CD 플레이어, 선루프, 가죽 시트)을 마음대로 조합할 수 있다고 가정하자. 플래시 프로그램에서는 모든 옵션을 포함한 가격을 계산하고 서버측에 있는 프로그램에서는 이러한 정보를 사용자 프로파일의 일부로 저장해 둘 수 있다.

자동차의 옵션을 다음과 같이 네 개의 서로 다른 부울 변수로 저장할 수도 있다.

```
var hasAirCon = true;
var hasCD = true;
var hasSunRoof = true;
var hasLeather = true;
```

이렇게 하면 각각 고유의 값을 가지는 네 개(각 옵션마다 하나씩)의 스위치가 필요하다. 물론 이렇게 해도 별 문제는 없다. 하지만 메모리에 네 개의 변수를 저장해야 하고 서버에 있는 사용자 프로파일 데이터베이스에 네 개의 필드가 들어가야 한다. 하지만 자동차의 옵션을 각각 비트로 저장하면 모든 옵션을 4비트짜리 숫자 한 개에 저장할 수 있다. 에어컨은 0번 비트(1의 자리)에, CD 플레이어는 1번 비트(2의 자리)에, 선루프는 2번 비트(4의 자리)에, 가죽 시트는 3번 비트(8의 자리)에 넣을 수 있기 때문이다. 아래 예는 네 개의 옵션을 조합한 내용을 하나의 숫자로 저장하기 위한 설정법을 보여준다.

```
var options;
options = 1   // 1 == 0001; 0번 비트가 on인 경우: 에어컨
options = 2   // 2 == 0010; 1번 비트가 on인 경우: CD 플레이어
options = 4   // 4 == 0100; 2번 비트가 on인 경우: 선루프
options = 8   // 8 == 1000; 3번 비트가 on인 경우: 가죽 시트

// 옵션을 조합하는 방법
options = 5   // 5  == 0101: 에어컨(1)과 선루프(4)
options = 10  // 10 == 1010: CD 플레이어(2)와 가죽 시트(8)
options = 15  // 15 == 1111: 모든 옵션을 선택한 경우
```

어떤 옵션을 추가하거나 제거하고 싶다면 적당한 비트를 더하거나 빼기만 하면 된다.

```
var options = 0;  // 아무 옵션도 선택하지 않은 경우
options += 4;     // 선루프 추가(options는 4, 즉 0100이 된다).
options += 1;     // 에어컨 추가(options는 5, 즉 0101이 된다).
options += 2;     // CD 플레이어 추가(options는 7, 즉 0111이 된다).
options -= 4;     // 선루프 제거(options는 3, 즉 0011이 된다).
options += 8;     // 가죽 시트 추가(options는 11, 즉 1011이 된다).
```

이제 어떻게 여러 옵션을 하나의 숫자에 비트 단위로 저장하는지 알 수 있을 것이다. 그렇다면 자동차의 가격을 계산할 때는 그러한 가격을 어떻게 확인할 수 있을까? 이런 경우에 바로 비트 연산자를 사용해야 한다. 우선 각 연산자에 대해 알아본 후에 자동차 예제를 계속해서 만들어 보자.

비트 AND

비트 AND(&) 연산자는 숫자의 각 비트에 논리 AND 연산을 수행하여 두 숫자의 비트를 결합한다. 이 연산 결과로는 두 숫자를 조합한 값이 리턴된다.

비트 AND는 다음과 같은 형식으로 쓰인다.

```
operand1 & operand2
```

비트 AND의 피연산자로는 어떤 숫자도 사용할 수 있지만 연산을 처리하기 전에 항상 32비트 이진 정수로 변환된다. 만약 피연산자 중에 소수점 이하로 내려가는 숫자가 있다면 소수점 이하는 버려진다.

5장에서 배운 논리 AND 연산자에서 두 글자짜리 연산자인 &&를 이용하여 피연산자 전체를 다루는 반면에 비트 AND 연산자에서는 한 글자짜리 연산자인 &를 이용하며 피연산자의 각 비트에 AND 연산을 처리한다.

비트 AND에서는 두 피연산자인 operand1과 operand2의 각 비트를 하나씩 비교하여 만든 숫자를 리턴한다. 만약 두 피연산자의 특정 비트가 모두 1이라면 그 위치에 해당하는 비트는 1로 설정되며 그 외의 경우에는 0으로 설정한다.

비트 AND 연산자는 두 피연산자를 열을 맞추어 두 줄로 적어놓고 보면 쉽게 이해할 수 있다. 이렇게 하면 두 피연산자 모두 1이 들어있는 비트를 금방 확인할 수 있다.

아래 예에서는 두 피연산자의 2번 비트(오른쪽에서 세 번째 비트)가 모두 1이므로 결과에서도 2번 비트만 1이 된다. 다른 비트는 모두 0으로 설정된다.

```
  1111
& 0100
-------
  0100
```

다음 예에서는 두 피연산자의 0번과 3번 비트가 모두 1이므로 결과의 0, 3번 비트도 1이 된다. 하지만 1번과 2번 비트는 0으로 설정된다.

```
  1101
& 1011
-------
  1001
```

액션스크립트에서는 이진수 대신 10진수만을 사용하기 때문에 얼핏 보는 것만으로는 비트 연산을 알아보기 힘들다. 실제 코드에서는 위 예를 사용하려면 다음과 같은 식으로 써야 한다.

```
15 & 4   // 결과는 4가 된다.
13 & 11  // 결과는 9가 된다.
```

보통 실전에서는 주로 특정 플래그(비트)가 true인지 false인지 확인할 때 비트 AND를 많이 사용한다.

아래 코드는 2번 비트(값이 4이다)가 true인지 확인하는 예이다.

```
if (x & 4) {
  // 필요한 작업 처리
}
```

2번 또는 3번 비트가 true인지 확인할 때는 다음과 같은 코드를 사용하면 된다.

```
if (x & (4|8)) {
  // 필요한 작업 처리
}
```

위 예에서는 2번 또는 3번 비트가 true로 설정되었는지 확인하기 위해 다음에 배울 | 연산자를 이용한다. 2번과 3번 비트가 모두 설정되었는지 확인할 때는 다음과 같은 코드를 사용해야 한다.

```
if (x & (4|8) == (4|8)) {
  // 필요한 작업 처리
}
```

비트 AND 연산자는 어떤 숫자에 있는 비트를 하나씩 false로 설정하는 데도 쓰일 수 있다. 자세한 내용은 뒤에 나오는 비트 NOT 연산자를 설명하는 부분에서 알아보자.

비트 OR

비트 OR 연산자(|)는 숫자에 있는 각 비트에 논리 OR 연산을 수행하여 두 개의 숫자에 있는 비트를 조합시킨다. 비트 AND와 마찬가지로 비트 OR에서도 조합한 결과를 숫자로 리턴한다. 비트 OR의 형식은 다음과 같다.

```
operand1 | operand2
```

피연산자로는 어떤 숫자를 사용해도 괜찮지만 피연산자로 쓰인 숫자는 연산이 수행되기 전에 32비트 이진수로 변환된다. 숫자에 있는 소수점 이하 부분은 모두 무시된다.

5장에서 배운 논리 OR 연산자에서는 두 글자짜리 연산자인 ||를 이용하여 피연산자 전체를 다루는 반면, 비트 OR 연산자에서는 한 글자짜리 연산자인 |를 이용하며 피연산자의 각 비트에 대해 OR 연산을 처리한다.

결과의 각 비트에는 두 피연산자에 있는 같은 비트에 논리 OR 연산을 처리한 결과가 저장된다. 따라서 어떤 비트가 operand1과 operand2 중 하나라도 1이라면 결과에서는 그 비트가 1이 된다. 아래 예제를 앞에 나온 비트 AND 연산자의 경우와 비교해 보자.

이 예에서는 두 피연산자에서 모두 0인 비트는 1번 비트이기 때문에, 그 비트만 0이 되고 나머지는 1이 된다.

```
  1101
| 0100
------
  1101
```

다음 예에서는 모든 비트가 두 피연산자 중 적어도 하나는 1이기 때문에, 결과적으로 모든 비트가 1이 된다.

```
  1101
| 1011
------
  1111
```

실제 코드에서는 위 예를 다음과 같은 식으로 적는다.

```
13 | 4    // 결과는 13이 된다.
13 | 11   // 결과는 15가 된다.
```

보통 각 옵션을 나타내는 여러 숫자를 결합하여 어떤 시스템의 모든 옵션을 나타내는 하나의 숫자 값을 구할 때 비트 OR를 사용한다. 예를 들어 다음과 같은 코드를 사용하면 2번 비트(4)와 3번 비트(8)를 결합한다.

```
options = 4 | 8;
```

어떤 값에서 특정 옵션을 true로 설정할 때도 비트 OR를 사용한다. 아래 코드에서는 3번 비트(8)로 표현되는 옵션을 true로 설정한다. 3번 비트에 있는 값이 이미 true인 경우에는 값이 변하지 않는다.

```
options = options | 8;
```

다음과 같이 한 번에 여러 비트를 설정할 수도 있다.

```
options = options | 4 | 8;
```

비트 XOR

이제 슬슬 이상하게 생긴 연산자가 나타나기 시작한다. 비트 XOR(eXclusive OR, 배타적 OR) 연산자는 캐럿 기호인 ^(대부분의 키보드에서 Shift-6)이다. 비트 XOR는 다음과 같은 형식으로 쓰인다.

```
operand1 ^ operand2
```

피연산자에는 어떤 숫자라도 사용할 수 있지만 연산을 처리하기 전에 모두 32비트 이진 정수로 변환된다. 소수점 이하 부분은 모두 무시된다.

비트 XOR는 어떤 비트가 두 피연산자 모두 1로 설정된 경우에는 결과가 1이 아니라 0이 된다는 점에서 비트 OR와 다르다. 즉 어떤 비트의 값이 두 피연산자 모두 같은 경우에는 XOR를 하면 그 비트가 0이 되고 두 피연산자의 해당 비트가 서로 다른 경우(둘 중 하나는 0, 나머지는 1인 경우)에는 1이 된다.

아래 예에서는 두 피연산자의 0번, 3번 비트가 서로 같기 때문에, 결과의 0번, 3번 비트는 0이 된다. 두 피연산자의 1번, 2번 비트는 서로 다르므로 결과의 1번, 2번 비트는 1이 된다.

```
  1011
^ 1101
------
  0110
```

아래 예에서는 두 피연산자의 모든 비트가 서로 같기 때문에 결과에서는 모든 비트가 0이 된다.

```
  0010
^ 0010
------
  0000
```

다음 예에서 두 피연산자의 0번, 2번, 3번 비트가 서로 다르므로 결과의 0번, 2번, 3번 비트는 1이 된다. 1번 비트는 두 피연산자 모두 1이기 때문에 결과의 1번 비트는 0이 된다.

```
  0110
^ 1011
------
  1101
```

십진수로 쓰면 위에 있는 세 개의 예제는 다음과 같이 쓰인다.

```
11 ^ 13    // 결과는 6이 된다.
2 ^ 2      // 결과는 0이 된다.
6 ^ 11     // 결과는 13이 된다.
```

비트 XOR 연산자는 어떤 옵션이 true이면 false로, false이면 true로 0과 1 사이를 토글(toggle)할 때 주로 사용한다. 2번 비트(4)로 표시되는 옵션을 토글하고 싶다면 다음과 같은 코드를 사용하면 된다.

```
options = options ^ 4;
```

비트 NOT

두 개의 숫자로부터 어떤 결과를 구하는 비트 AND, OR, XOR 연산자와는 달리 비트 NOT에서는 하나의 숫자 비트를 변경한다. 비트 NOT 연산자는 대부분 키보드의 왼쪽 맨 위에 있는 틸드 기호(~)로 표시하며 일반적인 사용법은 다음과 같다.

```
~operand
```

피연산자에는 어떤 숫자를 사용해도 괜찮지만 연산을 처리하기 전에 모든 숫자는 32비트 이진 정수로 변환된다. 소수점 이하 부분이 있다면 모두 무시된다.

비트 NOT에서는 피연산자의 모든 비트를 뒤집는다. 예를 들면 다음과 같다.

```
~00000000000000000000000000000010
  // 결과는 11111111111111111111111111111101이 된다.

~11111111111111111111111111111010
  // 결과는 00000000000000000000000000000101이 된다.
```

10진수로 표현하면 다음과 같이 쓸 수 있다.

```
~2    // 결과는 -3이 된다. 아래 설명 참조
~-6   // 결과는 -5가 된다. 아래 설명 참조
```

여기서는 음수를 이진수로 표기하는 방법에 대해서 자세히 다루지는 않겠다. 하지만 어느 정도 경험이 있는 프로그래머라면 비트 연산에서는 음의 이진수를 2의 보수 표기법으로 나타낸다는 것 정도는 알아야 한다. 이러한 표기법을 잘 모르는 독자라면 비트 NOT 연산자의 리턴 값이 원래 피연산자 값의 부호를 바꾼 것보다 1 작은 값이라는 정도만 기억해두면 된다. 예를 들면 다음과 같다.

```
~-10  // -10의 부호를 바꾸면 10이 되고 그 값에서 1을 빼면 9가 된다.
```

비트 NOT 연산자는 보통 비트 AND 연산자와 함께 사용하여 특정 비트를 0으로 설정할 때 쓰인다. 예를 들어 2번 비트를 0으로 설정할 때는 다음과 같이 하면 된다.

```
options = options & ~4;
```

~4는 2번 비트에 있는 한 개의 0을 제외하면 나머지는 모두 1로 채워져 있는 32비트 정수를 리턴한다. 이 숫자와 oprions 변수에 대해 비트 AND 연산을 적용하면, options의 2번 비트가 0으로 설정되고 나머지 비트는 영향을 받지 않는다. 위 코드를 조금 더 간단하게 쓰면 다음과 같다.

```
options &= ~4;
```

똑같은 방법으로 여러 개의 비트를 한꺼번에 0으로 설정할 수 있다. 아래 코드를 이용하면 2번과 3번 비트를 동시에 0으로 설정할 수 있다.

```
options &= ~(4 | 8);
```

비트 시프트 연산자

비트 단위 프로그래밍에서는 이진수를 일련의 스위치인 것처럼 다룬다. 이러한 스위치를 여기저기로 옮겨야 하는 경우도 있다. 예를 들어 0번 비트가 on으로 설정되어 있는데, 0번 비트를 off로 설정하고 2번 비트를 on으로 설정하고 싶다면 0번 비트를 왼쪽으로 두 칸 움직이기만 하면 된다. 또는 어떤 숫자의 5번 비트가 on인지 off인지 알고 싶다면 그 비트를 다섯 칸 오른쪽으로 옮기고 나서 0번 비트의 값을 확인해도 된다. 비트 시프트 연산자를 이용하면 이러한 작업을 할 수 있다.

비트 시프트 연산자를 이용하면 2의 멱수[1]로 곱하거나 나누는 것을 매우 빠르게 처리할 수 있다. 어떤 10진수를 10으로 나눌 때는 소수점을 한 칸 왼쪽으로 옮기기만 하면 된다. 또한 어떤 10진수에 10을 곱할 때는 소수점을 한 칸 오른쪽으로 옮기면 된다. 만약 103(즉 1000)을 곱하고 싶다면 소수점을 세 칸 오른쪽으로 옮기면 된다. 비트 시프트 연산을 이용하면 이진수에 대해 이와 비슷한 작업을 처리할 수 있다. 비트를 오른쪽으로 시프트하면 매번 시프트할 때마다 숫자를 2로 나누게 된다. 또한 비트를 왼쪽으로 시프트하면 매번 시프트할 때마다 숫자에 2를 곱하게 된다.

1) 역자주: 2를 여러 번(정수 번) 곱한 수. 2, 4, 8, 16, 32, 64,...는 모두 2의 멱수이다.

부호가 있는 오른쪽 시프트

부호가 있는 오른쪽 시프트를 이용하면 2의 멱수로 피연산자를 나눌 수 있다. 부호가 있는 오른쪽 시프트에서는 >> 부호(> 표시를 두 번 겹쳐 쓴 것)를 이용하며 일반적인 형식은 다음과 같다.

```
operand >> n
```

여기서 n은 operand의 비트를 몇 번 오른쪽으로 시프트할지를 지정해주는 정수이다. 이 연산의 결과는 operand를 2n으로 나눈 값과 같다. 나눗셈을 했을 때 나머지가 생기면 그 나머지는 모두 무시된다. 이 연산의 작동 방법은 다음과 같다.

모든 비트는 n으로 지정한 자리 수만큼 오른쪽으로 시프트된다. 숫자의 오른쪽 끝을 넘어가는 숫자는 모두 무시된다. 빈 자리를 채우기 위해 왼쪽 끝에 새로운 비트가 추가된다. operand가 양수인 경우에는 새로 추가되는 비트가 0이고 음수인 경우에는 새로 추가되는 비트가 1이 된다(음수는 2의 보수 표현법으로 표기되기 때문이다). 유사코드로 적어보면 다음과 같이 된다.

```
// 맨 오른쪽 비트(0)이 없어지고 왼쪽에 0이 추가된다.
// 결과는 00000000000000000000000000000100이 된다.
00000000000000000000000000001000 >> 1
```

어떤 숫자를 오른쪽으로 한 비트 시프트하면 그 숫자를 21(즉 2)으로 나누는 셈이 된다. 10진수로 쓴다면 다음과 같이 된다.

```
8 >> 1  // 결과는 4가 된다.
```

나눗셈을 했을 때 나머지가 생긴다면 그 값은 그냥 버려진다.

```
9 >> 1 // 이 경우에도 결과는 4가 된다.
```

음수의 경우에도 한 비트씩 시프트할 때마다 숫자 값을 2로 나눈다.

```
-16 >> 2  // 결과는 -4가 된다(-16을 2의 제곱으로 나눈 값).
```

부호가 없는 오른쪽 시프트

부호가 없는 오른쪽 시프트는 > 기호를 세 번 연속으로 적어서 표기하며 일반적으로 다음과 같은 형식으로 쓰인다.

```
operand >>> n
```

시프트 결과로 생기는 빈 자리를 무조건(operand의 부호에 상관없이) 0으로 채운다는 점을 제외하면 부호가 있는 오른쪽 시프트와 다를 것은 없다. 양수의 경우에는 부호가 있는 오른쪽 시프트와 부호가 없는 오른쪽 시프트를 사용한 결과가 완전히 똑같다.

왼쪽 시프트

왼쪽 시프트 연산자는 어떤 수에 2의 멱승을 곱하는 데 쓰인다. 이 연산자는 << 부호(< 부호를 두 번 연속으로 적는다)로 표기하며 일반적인 형식은 다음과 같다.

```
operand1 << n
```

여기서 n은 operand를 시프트시킬 비트 수이다. 이 연산을 처리한 결과는 operand에 2n을 곱한 것과 같다. 작동 방법은 다음과 같다.

모든 비트는 n으로 지정한 자리수만큼 왼쪽으로 시프트된다. 왼쪽 끝을 넘어가는 비트는 모두 무시된다. 모든 비트를 왼쪽으로 움직임에 따라 오른쪽 끝에 생기는 공백에는 0이 채워진다. 예를 들면 다음과 같다.

```
01000000000000000000000000001001 << 4
  // 결과는 00000000000000000000000010010000이 된다.
```

어떤 숫자를 왼쪽으로 4비트 시프트시키면 24(즉 16)을 곱한 것과 같은 값이 나온다. 위 유사코드를 10진수로 적으면 다음과 같이 된다.

```
9 << 4  // 9 * 16 = 144
```

지금까지 사용한 예에서는 특정 비트에 해당하는 값(0번 비트는 1, 1번 비트는 2, 2번 비트는 4, 3번 비트는 8...)을 직접 계산하였다. 하지만 왼쪽 시프트 연산자를 이용하면 어떤 비트의 값을 간단하게 계산할 수 있다.

```
(1 << 0)    // 0번 비트: 1
(1 << 1)    // 1번 비트: 2
(1 << 2)    // 2번 비트: 4
(1 << 3)    // 3번 비트: 8
(1 << 15)   // 15번 비트가 32768이라는 것을 외우는 것보다는 훨씬 간편하다.
```

또한 비트 값 대신 숫자 인덱스를 이용하여 동적으로 비트를 선택할 때도 왼쪽 시프트가 유용하게 쓰인다. 〔예제 15-1〕은 어떤 숫자를 이진수로 표현했을 때 1의 개수를 알아내는 코드이다.

〔예제 15-1〕 왼쪽 시프트를 이용하여 1로 설정된 비트의 개수를 알아낸다.

```
myNumber = 27583;  // 1의 개수를 세고자 하는 숫자
count = 0;
for (var i=0; i < 32; i++) {
  if (myNumber & (1 << i)) {
    count++;
  }
}
```

〔예제 15-2〕는 오른쪽 시프트 연산을 이용하여 〔예제 15-1〕과 똑같은 기능을 하도록 만든 코드이다. 여기에서는 각 비트에 해당하는 값을 계산하는 대신 주어진 값을 반복하여 오른쪽 시프트 시킴으로써 그 값에 들어있는 1의 개수를 센다.

```
myNumber = 27583;
count = 0;
temp = myNumber;    // 임시 복사본을 만든다.
for (var i = 0; i < 32; i++) {
  if (temp & 1) {
    count++;
  }
  temp = temp >> 1;
}
```

오른쪽 시프트를 하고 나면 시프트하기 전의 값을 알아낼 수 없기 때문에, myNumber를 임시 변수인 temp에 복사해야 하며 이 코드를 실행하고 나면 temp 변수의 값은 0이 된다.

비트 단위 연산자 응용

맨 처음에 비트 단위 연산자를 시작할 때 자동차를 판매하는 플래시 사이트의 예를 사용하였다. 이제 비트 단위 연산자를 모두 배웠으니 〔예제 15-3〕에 나온 것처럼 이 연산자들을 이용하여 자동차의 가격을 계산해 보자. 이 예제의 .fla 파일은 온라인 코드 창고에서 다운로드할 수 있다.

〔예제 15-3〕 비트 연산자를 응용한 예제

```
// 우선 각 옵션을 설정한다(보통 사용자가 폼에 입력한 내용을 기준으로
// 값을 더하거나 빼서 옵션을 설정하지만 여기에서는 직접 코드에 넣는다).
var hasAirCon   = (1<<0)    // 0번 비트: 0-no, 1-yes
var hasCDplayer = (0<<1)    // 1번 비트: 0-no, 2-yes
var hasSunRoof  = (1<<2)    // 2번 비트: 0-no, 4-yes
var hasLeather  = (1<<3)    // 3번 비트: 0-no, 8-yes

// 비트 OR를 이용하여 여러 옵션을 조합한다.
var carOptions = hasAirCon | hasCDplayer | hasSunRoof | hasLeather;

// 가격을 계산하는 함수
function totalPrice(carOptions) {
  var price = 0;
  if (carOptions & 1) {  // 첫 번째 비트가 1이면
    price += 1000;       // $1000을 더한다.
  }
  if (carOptions & 2) {  // 두 번째 비트가 1이면
    price += 500;        // $500을 더한다.
  }
  if (carOptions & 4) {  // 세 번째 비트가 1이면
    price += 1200;       // $1200을 더한다.
  }
  if (carOptions & 8) {  // 네 번째 비트가 1이면
    price += 800;        // $800을 더한다.
  }

  return price;
}

// 함수를 호출하여 제대로 작동하는지 확인한다.
trace(totalPrice(carOptions));  // 3000이 리턴된다.
                                // (CD 플레이어를 제외한 옵션 가격의 총합)
```

코드 전체에서 비트 값을 코드에 일일이 넣는 대신 특정 옵션에 해당하는 비트 값을 다음과 같이 변수에 저장하는 것이 좋다.

```
var airConFLAG   = 1 << 0;  // 0번 비트, 1
var cdPlayerFLAG = 1 << 1;  // 1번 비트, 2
var sunroofFLAG  = 1 << 2;  // 2번 비트, 4
var leatherFLAG  = 1 << 3;  // 3번 비트, 8
```

연습 문제: 각 옵션을 나타내는 값을 직접 코드에 넣는 대신 변수와 왼쪽 시프트 연산자만을 이용하여 〔예제 15-3〕과 같은 코드를 다시 만들어 보자.

왜 비트 단위 연산을 사용할까?

〔예제 15-3〕보다는 부울 연산자를 여러 번 사용하는 것이 더 이해하기 쉽다. 하지만 비트 단위 연산을 이용하면 속도가 엄청나게 개선되며 용량도 크게 줄일 수 있다. 컴퓨터에서 이해하기 좋도록 이진법을 사용하면 메모리도 절약할 수 있고 속도도 향상시킬 수 있다.

부울 연산과 비트 단위 연산을 비교하기 위해 사용자 프로파일을 관리하는 데 각 사용자마다 on 또는 off로 설정될 수 있는 32개 항목이 있다고 가정해 보자. 일반적인 데이터베이스를 사용한다면 각 사용자마다 32개의 필드가 필요하다. 사용자 수가 100만명 정도 된다면 32개의 필드가 100만 개 필요하다. 하지만 비트 단위 프로그래밍을 사용하면 32개의 항목을 하나의 숫자에 저장할 수 있기 때문에 각 사용자의 데이터베이스에 하나씩의 필드만 있으면 된다. 이렇게 하면 디스크 공간을 절약할 수 있을 뿐만 아니라 사용자의 프로파일을 액세스할 때마다 32개의 부울 변수 대신 한 개의 정수만을 사용하면 된다. 수백만 개의 트랜잭션을 처리하는 경우에는 각 트랜잭션마다 몇 밀리초씩만 절약해도 시스템 성능이 크게 개선될 수 있다.

비트 단위 연산에 대해 자세히 배우고 싶다면 진 마이어스가 쓴 'Becoming Bit Wise'를 참조하기 바란다(http://www.cscene.org/CS9/CS9-02.html).[1)]

1) 역자주: C 프로그래머를 대상으로 쓴 글이긴 하지만 뒤에 나오는 코드를 제외하면 나머지 내용은 액션스크립트 프로그래머라도 읽어볼만한 글이다.

고급 함수 영역 문제

이 절의 내용은 함수 영역을 처음 배우는 독자들에게는 너무 어렵기 때문에, 앞에서 그냥 넘어갔던 주제에 대해 살펴보자. 이제 무비 클립, 함수 영역, 객체에 대한 내용을 모두 배웠으니 함수 영역과 관련된 고급 주제를 배울 준비가 갖춰진 셈이다.

9장에서 배웠듯이 함수 영역 사슬은 일반적으로 함수를 선언한 선언문에 대해 상대적으로 결정된다. 하지만 이 규칙에는 미묘하게 확장된 부분이 있다. 한 타임라인에 있는 함수를 다른 무비 클립의 타임라인에 있는 변수에 대입하면 함수의 영역 사슬도 영향을 받는다. 원래 함수를 직접 호출하면 그 영역 사슬에는 원래의 타임라인이 포함되지만 다른 타임라인의 변수를 통해 그 함수를 호출하면 그 영역 사슬에는 변수의 타임라인이 포함된다.

예를 들어 현재 클립을 회전시키면서 크기를 조절하는 transformClip()이라는 함수를 만든다고 생각해 보자. 클립을 회전시키고 크기를 조절하는 정도는 rotateAmount와 widthAmount라는 변수에 저장한다.

```
var rotateAmount = 45;
var widthAmount = 50;

function transformClip () {
  _rotation = rotateAmount;
  _xscale = widthAmount;
}

// 함수를 호출한다.
transformClip();
```

이제 이 transformClip() 함수를 rect라는 클립의 tc라는 변수에 대입하자.

```
rect.tc = transformClip;
```

다음과 같이 rect.tc를 통해 transformClip() 함수를 호출하면 아무런 일도 일어나지 않는다.

```
rect.tc();
```

왜 아무런 일도 일어나지 않을까? tc에 저장된 함수는 원래 함수의 타임라인이 아닌 rect의 타임라인에 해당하는 영역 사슬에 속하므로 rotateAmount와 widthAmount

변수가 없기 때문이다. 하지만 rect에도 rotateAmount와 widthAmount 변수를 추가하면 rect.tc에서도 그 변수를 찾을 수 있기 때문에 함수가 제대로 작동한다.

```
rect.widthAmount = 10;
rect.rotateAmount = 15;
rect.tc();  // rect의 너비를 10퍼센트로 축소시키고 15도 회전시킨다.
```

하지만 이와는 달리 같은 타임라인에 있는 일반적인 데이터 객체에 함수를 대입하면 함수의 영역 사슬이 바뀌지 않는다. 대신 영역 사슬이 함수 선언에 따라 영구적으로 결정된다. 〔예제 15-4〕를 살펴보자.

〔예제 15-4〕 객체 메소드의 고정된 영역

```
// 변수 설정
var rotateAmount = 45;
var widthAmount = 50;

// rotateAmount와 widthAmount의 값을 출력하는
// transformClip() 함수를 만든다.
function transformClip () {
  trace(rotateAmount);
  trace(widthAmount);
}

// 앞 예제에서 사용한 rect 클립에 해당하는
// 객체를 만든다.
var rectObj = new Object();

// transformClip을 rectObj의 속성에 복사한다.
rectObj.tc = transformClip;

// rectObj의 rotateAmount와 widthAmount를 설정한다.
rectObj.rotateAmount = 15;
rectObj.widthAmount = 10;

// rectObj.tc를 호출한다.
rectObj.tc();  // 15와 10이 아닌 45와 50이 출력된다.
               // rectObj.tc의 영역은 transformClip()의 영역과 같다.
```

어떤 함수를 객체 속성으로 대입하면 그 함수의 영역은 함수를 선언한 부분의 타임라인에 속하게 된다. 하지만 원격 무비 클립에 대입하면 그 함수의 영역은 원격 클립의 타임라인에 속한다.

이러한 성질은 함수의 영역이 함수를 선언한 부분의 객체에 속하는 자바스크립트와는 다르다는 것을 알 수 있다. 예를 들어 HTML 프레임셋 중 한 프레임이 플래시 무비의 클립과 비슷하다고 볼 때 이 차이점을 분명하게 이해할 수 있다. 자바스크립트에서는 원격 프레임에 어떤 함수를 대입하더라도 그 함수의 영역은 원격 프레임이 아닌 함수를 선언한 프레임에 속한다. 〔예제 15-5〕에서 이러한 성질을 확인할 수 있다.

〔예제 15-5〕 자바스크립트의 정적 함수 영역

```
// 프레임셋의 0번 프레임에 들어갈 코드
// 프레임 0의 변수를 만든다.
var myVar = "frame 0";

// 프레임셋의 1번 프레임에서 함수를 설정하고 그 함수를
// 0번 프레임으로 복사한다.
parent.frames[1].myMeth = function () { alert(myVar); };
myMeth = parent.frames[1].myMeth;

// 0번 프레임에서 myMeth() 함수를 호출한다.
myMeth(); // "frame 0"이 출력된다.

// 같은 프레임셋의 1번 프레임에 있는 버튼에 들어갈 코드
<FORM>
  <INPUT type="button" value="click me" onClick="myMeth();">
</FORM>

// 1번 프레임에서 버튼을 클릭하여 myMeth()를 호출한다.
// "frame 0"이 출력된다.
// myMeth()의 영역은 1번 프레임이 아니라 함수를 선언한 선언문이
// 들어있는 0번 프레임이 된다.
```

movieclip 데이터형

이번에도 액션스크립트의 기본적인 내용을 바탕으로 조금 더 복잡한 주제에 대해 알아보자. '13장. 무비 클립'에서 무비 클립은 대부분의 경우에 객체와 똑같은 식으로 작동한다는 것을 배웠다. 하지만 무비 클립은 클래스의 일종이 아니라 하나의 독립적인 데이터형이다. 액션스크립트를 만든 게리 그로스먼은 무비 클립과 객체 데이터형의 내부 구현법의 차이점을 다음과 같이 설명한다.

> 액션스크립트에서 무비 클립과 객체를 거의 똑같이 다룰 수 있지만, 무비 클립은 플레이어 내부에서 객체와는 다른 방법으로 구현된다. 가장 큰 차이점은 할당하는 법, 그리고 제거하는 법에서 찾을 수 있다. 일반적인 객체는 레퍼런스 카운트를 이용하여 가비지 컬렉션을 수행하지만 무비 클립은 타임라인에 의해 할당 및 제거가 결정되며, duplicateMovieClip()과 removeMovieClip()을 통해 직접 제어된다.
>
> x = new Array()라고 변수를 선언한 다음 바로 x = null 이라는 선언문을 실행하면 액션스크립트에서는 그 배열 객체에 대한 레퍼런스가 더 이상 남아있지 않다는 것(즉 어떤 변수도 그 객체를 참조하지 않음)을 알아내고 가비지 컬렉션 기능을 실행시킨다(즉 그 객체가 차지하고 있는 메모리를 제거한다). 주기적인 마크 앤드 스윕 가비지 컬렉션을 이용하여 순환적으로 참조되어 있는 객체도 제거할 수 있다(즉 두 개의 더 이상 쓰이지 않는 객체가 서로를 참조하는 경우에도 메모리에서 그 객체를 제거할 수 있는 고급 기법을 사용한다).
>
> 하지만 무비 클립은 그런 식으로 동작하지 않는다. 타임라인에서 객체를 추가하는 것에 따라 생성되거나 없어진다. 만약 무비 클립이 (duplicateMovieClip()과 같은 함수를 통해) 동적으로 생성되었다면, 그 클립은 removeMovieClip() 함수를 이용하기 전에는 없어지지 않는다.
>
> 객체에 대한 레퍼런스는 포인터(메모리 주소 레퍼런스)이다. 따라서 레퍼런스를 추적하고 가비지 컬렉션을 사용하면 엉뚱한 곳을 가리키는 포인터에 의한 메모리 손실을 방지할 수 있다. 하지만 무비 클립에 대한 레퍼런스는 소프트 레퍼런스(대상의 절대 경로를 저장하는 레퍼런스)이다. foo라는 무비 클립이 있는데 x = foo(x가 foo 클립에 대한 레퍼런스가 된다)라고 대입하고 나서 removeMovieClip()을 이용하여 foo를 삭제한 다음, foo라는 이름의 클립을 새로 만들면 x는 이번에도 새로 만든 foo 클립을 참조한다.
>
> 하지만 일반적인 객체는 다르다. 어떤 객체에 대한 레퍼런스가 있으면 그 객체가 바로 제거되지 않는다. 따라서 무비 클립이 객체였다면 removeMovieClip() 함수를 실행시키더라도 x가 그 객체를 가리키고 있는 동안에는 객체가 메모리에서 제거되지 않는다. 게다가 foo라는 무비 클립을 새로 만들면 원래 있던 foo와 새로운 foo가 동시에 존재할 수는 있지만, 원래 만든 foo는 더 이상 렌더링되지 않는다.

객체형과의 차이점이 매우 중요하기 때문에 movieclip이라는 형이 따로 있어야 한다. 마찬가지 이유로 함수는 객체와 여러 면에서 비슷하지만 함수에 typeof 연산자를 적용시키면 "function"이 리턴된다.

앞으로 배울 내용

드디어 '1부. 액션스크립트 기초'가 끝났다. '2부, 액션스크립트 응용'에서는 몇 가지 중요한 실전 액션스크립트 제작 기법 및 플래시 폼을 만드는 법, 코드를 디버그하는 법에 대해 살펴보자. 액션스크립트를 능숙하게 사용하려면 '3부. 레퍼런스'에 나온 내장 함수, 속성, 객체, 클래스와 같은 것을 익혀야 한다.