11

배열

'3장. 데이터와 데이터형'에서 원시 데이터형(문자열, 숫자, 부울, null과 undefined)으로 스크립트에서 기본적인 정보를 다룰 수 있다는 점을 배웠다. 또한 액션스크립트에서는 여러 조각의 데이터를 하나의 데이터로 묶을 수 있는 몇 가지 복합 데이터형을 제공한다는 것도 배웠다.

가장 먼저 배열에 대해 알아보자. 배열은 순서가 있는 정보의 리스트를 저장하고 조작하는 데 쓰이며 순서대로 처리되는 반복적인 프로그래밍에서 가장 기본적인 도구가 된다. 배열은 사용자 입력 폼에서 가져온 값을 저장하거나 풀다운 메뉴를 만드는 것부터 게임에서 적 우주선을 추적하는 데에 이르기까지 다양한 용도로 쓰인다. 가장 간단한 형태를 생각해 보면 배열은 시장에서 구입할 물건 리스트와 같은 아이템이라고 볼 수 있다.

배열이란 무엇인가?

배열은 건물이 여러 층으로 이루어지는 것처럼 여러 개의 개별 데이터 값을 포함할 수 있는 자료구조이다. 원시 데이터형과 달리 배열에는 두 개 이상의 데이터 값이 들어갈 수 있다. 아래 예에서는 우선 두 개의 문자열, 그리고 그 두 문자열을 포함하는 하나의 배열을 보여준다.

```
"oranges"              // 하나의 원시 문자열 값
"apples"               // 또 다른 원시 문자열 값
["oranges", "apples"]  // 두 문자열을 포함하고 있는 하나의 배열
```

배열은 범용 저장고이다. 배열에는 원하는 개수만큼의 아이템을, 데이터형에 구애받지 않고 저장할 수 있다. 심지어 배열에 다른 배열을 저장할 수도 있다. 아래의 예에는 문자열과 숫자를 동시에 저장하는 배열이 나와 있다. 아래와 같은 배열은 시장에서 구입할 물건과 그 개수를 나타낼 수도 있다.

배열을 서랍장에 비유해 보면 배열의 개념을 조금 더 쉽게 이해할 수 있다. 각 서랍에는 어떤 내용물(양말, 셔츠 등)이 들어 있다. 하지만 서랍장 자체에 어떤 내용물이 있는 것은 아니다. 서랍장에는 서랍만이 포함되어 있고 각 서랍에 내용물이 들어간다. 이러한 방식으로 서랍장은 여러 서랍을 하나의 단위로 모을 수 있다.

배열(서랍장)을 다룰 때는 대개 배열에 있는 값(서랍의 내용물)에 관심이 있게 마련이다. 배열에 있는 값이 바로 우리가 관리하고자 하는 정보이다. 하지만 그 값을 담고 있는 구조 자체를 조작해야 하는 경우도 있다. 예를 들면 배열의 크기를 변경하거나(서랍을 추가하거나 제거하는 것) 저장된 값의 순서를 변경(서랍의 내용물을 서로 바꾸는 것)할 수도 있다.

배열에는 값이 여러 개 저장되지만 배열 자체는 하나의 데이터이다. 여러 개의 문자를 포함하고 있는 문자열도 하나의 데이터이고 여러 개의 숫자를 포함하고 있는 수도 하나의 데이터인 것과 마찬가지로 생각하면 된다. 배열도 하나의 데이터이므로 어떤 변수에 대입하거나 복합 표현식의 일부로 사용할 수도 있다.

```
product = ["ladies downhill skis", 475]  // 배열을 변수에 저장한다.
display(product);                        // 배열을 함수에 전달한다.
```

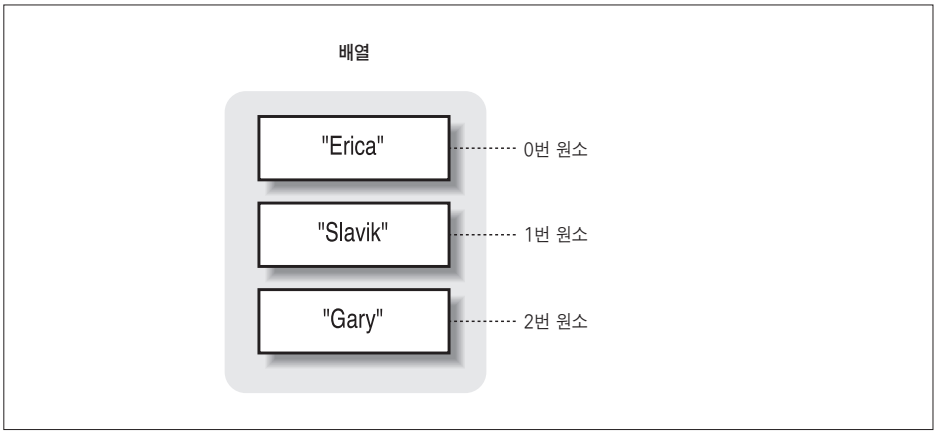
배열의 구조

배열에 저장된 각 아이템은 '원소(element)'라고 부르며, 각 원소에는 그 원소를 참조하는 데 쓰이는 고유 번호(인덱스)가 정해져 있다.

배열 원소

변수와 마찬가지로 배열의 각 원소에는 어떤 데이터라도 저장할 수 있다. 배열 전체는 결국 순서대로 이름이 붙어 있는 변수의 집합과 비슷하다. 하지만 배열에서는 아이템을 구분할 때 서로 다른 이름 대신 각각의 원소 번호(첫 번째 원소의 번호는 1이 아니라 0이다)를 이용한다.

〔그림 11-1〕은 세 개의 원소를 가지고 있는 배열 구조를 개념적으로 나타낸 그림이다. 0번 원소에는 "Erica", 1번 원소에는 "Slavik", 2번 원소에는 "Gary"가 저장되어 있다.



〔그림 11-1〕 배열 구조의 예

배열 원소의 값을 조작하려면 그 배열을 숫자를 이용해서 가리켜야 한다. 서랍장에 비유한다면 첫 번째 서랍장에 뭔가 집어넣고 두 번째 서랍장에 있는 물건을 꺼내라고 말하는 것과 비슷하다고 생각하면 된다.

배열 원소 인덱스

배열에서 원소의 위치는 '인덱스(index)'라고 부른다. 문자열에서 일곱 번째 문자를 따로 알아낼 수 있는 것처럼 배열에서도 인덱스를 이용하여 일곱 번째 원소(인덱스는 6이 된다)를 따로 구할 수 있다. 배열 원소의 값을 읽어들이거나 원소에 어떤 값을 대입할 때, 또는 여러 가지 다른 방법으로 배열 원소를 이용한 작업을 할 때는 그 원소의 인덱스를 사용한다. 예를 들어 배열을 다루는 함수 중에는 인덱스를 이용하여 처리할 원소의 범위를 지정해주는 것도 있다.

또한 배열의 맨 앞, 맨 뒤, 또는 중간에서 원소를 삽입 또는 삭제할 수 있다. 배열에는 빈틈이 있을 수도 있다(즉, 어떤 원소가 비어 있을 수도 있다). 0번 및 4번 위치에는 원소가 있는데, 1, 2, 3번 위치에는 아무것도 들어있지 않을 수도 있다. 이처럼 빈틈이 있는 배열을 '희소 배열(sparse array)'이라고 부른다.

배열의 크기

모든 배열에는 특정 개수의 원소가 들어있다. 배열에 저장할 수 있는 원소의 개수를 배열의 크기라고 한다. 자세한 내용은 잠시 후에 알아보기로 하자.

배열 생성

새로운 배열을 만들 때는 데이터 리터럴을 이용하거나(즉, 모든 원소를 직접 입력하거나) 배열 생성용 함수인 Array()라는 내장 함수를 이용하면 된다.

배열 생성자

Array() 생성자를 이용하여 배열을 만들 때는 new 연산자 뒤에 Array 키워드를 사용하고 그 뒤에 괄호를 붙이면 된다. 이렇게 하면 비어있는 배열(원소가 없는 배열)이 만들어진다. 이렇게 새로 만든 배열은 변수나 다른 데이터 저장소에 대입하여 나중에 참조할 수 있도록 만든다. 예를 들면 다음과 같다.

```
var myList = new Array();  // myList 변수에 비어있는 배열을 저장한다.
```

배열의 원소에 초기 값을 지정하고 싶을 때도 있다. 초기 값을 지정할 때는 Arrary( ) 생성자를 호출할 때 매개변수를 전달하면 된다. 매개변수에 따라 생성자 호출의 결과가 달라진다.

다른 프로그래밍 언어의 배열

거의 모든 고급 컴퓨터 언어에서는 배열 또는 배열 비슷한 것을 제공한다. 배열을 구현하는 방법은 언어에 따라 조금씩 다르다. 꽤 많은 언어에서 하나의 배열에 서로 다른 형의 데이터를 저장하지 못하게 되어 있다. 즉 어떤 배열에 숫자 또는 문자열을 저장할 수는 있지만, 하나의 배열에 숫자와 문자열을 섞어서 저장할 수는 없는 언어가 많다. 특이하게도 C에는 문자열 데이터형을 원시 데이터형으로 제공하지 않는다. 대신 C에서는 char라는 한 글자짜리 문자 데이터형을 제공하며 문자열은 복합 데이터형으로 간주하여 char 데이터형의 배열로 구현한다.

액션스크립트에서는 아이템을 삽입하거나 삭제하면 배열의 크기가 자동으로 변한다. 하지만 배열 크기를 배열을 처음 선언할 때 또는 배열의 차원이 결정될 때(즉 배열의 데이터를 저장할 메모리를 할당할 때) 지정해주어야 하는 언어도 많이 있다. 매크로미디어 디렉터의 스크립트 언어인 링고(Lingo)에서는 배열을 list라는 이름으로 부른다. 액션스크립트와 마찬가지로 링고에서는 서로 다른 형의 데이터를 하나의 배열에 저장할 수 있으며, 배열의 크기도 자동으로 바뀐다. 하지만 배열의 첫째 원소 번호가 0부터 시작하는 액션스크립트나 C와는 달리 링고에서는 첫째 원소 번호가 1부터 시작된다.

또한 배열의 범위를 벗어난 원소를 사용하려고 할 때의 반응도 각 언어에 따라 다르다. 액션스크립트와 링고에서는 배열의 끝을 넘어선 부분에 어떤 값을 대입하려고 하면 자동으로 새로운 원소를 추가한다. 배열 경계를 넘어선 위치에 있는 원소의 값을 구하면 액션스크립트에서는 undefined를 리턴하지만 링고에서는 오류가 발생한다. C에서는 제대로 된 원소 번호를 사용하는지에 대해 전혀 신경을 쓰지 않는다. C에서는 배열의 경계 밖에 있는 원소도 사용할 수 있지만 보통 이렇게 하면 메모리에 있는 다른 데이터를 덮어쓰거나 배열에 속하지 않는 무의미한 데이터를 액세스하게 된다(C에는 이처럼 프로그래머가 일일이 신경을 써야 하는 부분이 많이 있다).

Array() 생성자에 두 개 이상의 인자를 전달하거나 숫자가 아닌 인자를 하나 전달하는 경우에는 각 인자가 새로운 배열의 원소가 된다. 예를 들면 다음과 같다.

```
var frameLabels = new Array("intro", "section1", "section2", "home");
```

이렇게 하면 frameLabels에 저장된 배열은 다음과 같은 원소를 가지게 된다.

```
0: "intro"
1: "section1"
2: "section2"
3: "home"
```

Array() 생성자에 한 개의 숫자 인자만 전달하면 주어진 크기의 비어있는 원소가 있는 배열을 만든다.

```
var myList = new Array(14);       // 14개의 비어있는 원소를 가지는
                                  // 배열을 만든다.
```

Array() 생성자에는 제대로 된 표현식이라면 어떤 것도(복합 표현식 포함) 인자로 전달할 수 있다.

```
var x = 10;
var y = 5;
var myNumbers = new Array(x + 1, x * y, Math.random());
```

이렇게 하면 myNumbers 변수에는 다음과 같은 원소를 가진 배열이 저장된다.

```
0: 11
1: 50
2: 0에서 1 사이의 부동소수점수
```

배열 리터럴

경우에 따라 Array() 생성자보다는 배열 리터럴을 이용하여 배열을 만드는 것이 더 편리할수도 있다. 리터럴은 고정된 데이터를 직접 표기한 것이라는 점을 상기시키도록 하자.

```
"beaver"      // 문자열 리터럴
234.2034      // 숫자 리터럴
true          // 부울 리터럴
```

배열 리터럴에서는 대괄호로 배열의 시작과 끝을 표시한다. 대괄호 안에는 쉼표로 구분한 표현식의 리스트로 배열의 원소를 입력한다. 일반적인 문법은 다음과 같다.

```
[expression1, expression2, expression3]
```

위 표현식의 값을 구하고 그 값을 배열 원소로 저장한다. 여기서 쓰이는 표현식은 함수 호출이나 변수, 리터럴에서 다른 배열(배열 안에 들어있는 다른 배열은 중첩 배열(nested array)이라고 부른다)에 이르기까지 제대로 된 표현식이라면 어떤 표현식이라도 사용할 수 있다. 몇 가지 예를 들면 다음과 같다.

```
[4, 5, 63]                                  // 단순한 숫자 원소
["jeremy", "janice", "eman"]                // 단순한 문자열 원소
[1, 4, 6 + 10]                              // 연산이 포함된 숫자
[firstName, lastName, "tall", "skinny"]     // 변수와 문자열
["month end days", [31, 30, 28]]            // 중첩 배열 리터럴
```

두 가지 방법을 비교하기 위해 Array() 생성자를 이용하여 위와 똑같은 행렬을 만들어 보자.

```
new Array(4, 5, 63)
new Array("jeremy", "janice", "eman")
new Array(1, 4, 6 + 10)
new Array(firstName, lastName, "tall", "skinny")
new Array("month end days", new Array(31, 30, 28))
```

앞에서 이미 언급했듯이 액션스크립트에서는 하나의 배열에 서로 다른 형을 가진 데이터를 저장할 수 있다.

배열 원소 참조

배열을 만들고 나면 배열 원소로 저장된 값을 가져오거나 변경해야 하는 일이 생길 것이다. 이렇게 하려면 '5장. 연산자'에서 소개한 대괄호([], 배열 접근 연산자) 연산자를 이용해야 한다.

원소 값 가져오기

배열 원소의 값을 구하고 싶다면 다음과 같이 대괄호 안에 인덱스를 집어넣기만 하면 배열 원소를 참조할 수 있다.

```
arrayName[elementNumber]
```

arrayName은 배열의 이름이고 elementNumber는 숫자 값이 나오는 표현식이다. 첫 번째 원소는 0번이고 마지막 원소의 번호는 배열의 길이보다 1 작은 값이다. 마지막 원소 번호보다 더 큰 번호를 인덱스로 사용하면 인터프리터에서 undefined를 리턴한다. 예를 들면 다음과 같다.

```
// 배열 리터럴을 이용하여 배열을 만들고 trees에 저장한다.
var trees = ["birch", "maple", "oak", "cedar"];

// Output 창에 trees의 첫 번째 원소를 출력한다.
trace(trees[0]);  // Displays: "birch"

// favoriteTree 변수에 세 번째 원소의 값을 대입한다.
// (인덱스가 0부터 시작하므로 세 번째 원소의 인덱스는 2이다.)
var favoriteTree = trees[2];  // favoriteTree의 값이 "oak"가 된다.
```

이제 조금 더 복잡한 경우를 생각해 보자. 원소 인덱스에는 숫자가 나오는 표현식은 모두 사용할 수 있으므로 원소 인덱스가 들어갈 자리에 숫자 대신 변수를 사용해도 된다.

```
var i = 3;
var lastTree = trees[i];  // lastTree를 "cedar"로 설정한다.
```

심지어 배열 인덱스 대신 숫자를 리턴하는 함수 호출을 사용해도 된다.

```
// randomTree를 trees에서 임의로 선택한 원소로 설정한다.
// 이 때 0과 3 사이의 난수를 이용한다.
var randomTree = trees[Math.floor(Math.random() * 4)];
```

퀴즈 게임에서 배열에 저장된 문제를 임의로 선택하거나 카드를 나타내는 배열에서 임의로 한 장의 카드를 선택할 때 이와 비슷한 방법을 사용할 수 있다.

배열을 액세스하는 것은 변수 값을 액세스하는 것과 매우 비슷하다. 다음과 같이 배열 원소를 복합 표현식의 일부로 사용하는 것도 가능하다.

```
var myNums = [12, 4, 155, 90];
var myTotal = myNums[0] + myNums[1] + myNums[2] + myNums[3];   // 배열
원소의 합을 구한다.
```

이 예제에서 배열 원소에 저장된 값의 총합을 구하는 데 사용한 방법은 최적화된 코드라고 보기 힘들다. 잠시 후에 배열 원소를 순서대로 액세스하는 훨씬 빠르면서도 편리한 방법을 알아보기로 하자.

원소 값 설정

원소 값을 설정하려면 대입 연산자의 왼쪽 피연산자로 arrayName[elementNumber]를 사용하면 된다.

```
// 배열을 만든다.
var cities = ["Toronto", "Montreal", "Vancouver", "Waterloo"];
// cities is now: ["Toronto", "Montreal", "Vancouver", "Waterloo"]

// 배열의 첫 번째 원소 값을 설정한다.
// cities becomes ["London", "Montreal", "Vancouver", "Waterloo"]
cities[0] = "London";

// 배열의 네 번째 원소 값을 설정한다.
// cities becomes ["London", "Montreal", "Vancouver", "Hamburg"]
cities[3] = "Hamburg";

// 배열의 세 번째 원소 값을 설정한다.
// cities becomes ["London", "Montreal", 293.3, "Hamburg"]
cities[2] = 293.3;  // 데이터형이 바뀌어도 상관없다.
```

배열 원소를 설정할 때 인덱스에는 어떤 숫자 표현식이든 사용할 수 있다.

```
var i = 1;
// i번째 원소의 값을 설정한다.
// cities 배열은 ["London", "Prague", 293.3, "Hamburg"]로 바뀐다.
cities[i] = "Prague";
```

배열 크기 결정

모든 배열에는 현재 배열에 있는 원소의 개수(비어있는 원소도 포함)를 나타내는 length라는 내장 속성이 들어있다. 배열의 length 속성을 사용할 때는 다음과 같이 점 연산자를 사용하면 된다.

```
arrayName.length
```

배열의 length 속성을 이용하면 배열에 있는 원소의 개수를 알아낼 수 있다. 몇 가지 예를 들면 다음과 같다.

```
myList = [34, 45, 57];
trace(myList.length);  // Displays: 3

myWords = ["this", "that", "the other"];
trace(myWords.length);  // 단어나 문자 개수가 아닌 원소의 개수,
                        // 즉 3이 출력된다.

frameLabels = new Array(24);   // Array() 생성자에서 하나의 숫자 인자를
                               // 사용한 경우
trace(frameLabels.length);  // 24가 출력된다.
```

배열의 length는 그 배열의 마지막 원소의 인덱스에 1을 더한 값이다. 예를 들어 원소 인덱스가 0, 1, 2인 배열 크기는 3이 된다. 또한 원소의 인덱스가 0, 1, 2, 그리고 50인 배열의 크기는 51이다. 3번 인덱스부터 49번 인덱스까지 해당하는 모든 원소가 비어 있더라도 배열의 크기에는 그 부분도 모두 포함된다. 배열 인덱스 번호가 1이 아닌 0부터 시작하기 때문에 배열의 마지막 원소는 언제나 myArray[myArray.length - 1]이 된다.

배열에 원소를 추가하거나 배열에 있는 원소를 제거하면 배열 크기가 바뀌기 때문에 배열의 length 속성도 바뀐다. length 속성을 설정하여 배열 끝에 새로운 원소를 추가하거나 배열 끝에 있는 원소를 삭제할 수도 있다.

그런데 배열의 length 속성은 도대체 어디에 써먹을 수 있을까? 배열의 length 속성을 이용하면 〔예제 8-1〕에서 본 것처럼 배열의 모든 원소를 액세스하는 루프를 만들 수 있다. 배열의 모든 원소를 루프를 돌리면서 액세스하는 것은 프로그래밍에서 매우 기본적인 작업 중 하나이다. 루프와 배열을 결합하여 할 수 있는 일에 대해

알고 싶다면, soundtracks 배열을 쭉 훑어가면서 "hip hop"이라는 값이 저장된 위치를 찾아내는 〔예제 11-1〕을 자세히 살펴보자. 이 예제를 보면 '8장. 순환문'에서 배운 for 루프와 5장에서 배운 증가 연산자를 사용했다는 것을 알 수 있다. 또한 방금 배운 배열 액세스 코드도 포함되어 있다.

〔예제 11-1〕 배열 검색

```
// 배열 생성
var soundtracks = ["electronic", "hip hop", "pop", "alternative",
"classical"];

// 각 원소를 조사하여 "hip hop"이 들어있는지 확인한다.
for (var i = 0; i < soundtracks.length; i++) {
  trace("Now examining element: "+ i);
  if (soundtracks[i] == "hip hop") {
    trace("The location of 'hip hop' is index: "+ i);
    break;
  }
}
```

〔예제 11-1〕을 확장하여 〔예제 11-2〕에 나온 것처럼 임의의 배열에서 임의의 원소를 찾아낼 수 있는 일반화된 검색 함수를 만들 수도 있다. 이 검색 함수에서는 배열에서 그 원소를 발견한 위치를 리턴한다. 만약 원하는 원소를 찾지 못하면 null을 리턴한다.

〔예제 11-2〕 일반화된 배열 검색 함수

```
function searchArray (whichArray, searchElement) {
  // 각 원소를 조사하여 searchElement가 들어있는지 확인한다.
  for (var i = 0; i < whichArray.length; i++) {
    if (whichArray[i] == searchElement) {
      return i;
    }
  }
  return null;
}
```

위와 같은 검색 함수를 이용하여 다음과 같은 코드를 만들 수 있다. 아래 코드는 허가된 사용자명의 배열인 userNames 배열에 "Fritz"가 포함되어 있는지 확인하는 예이다.

```
if (searchArray (userNames, "Fritz") == null) {
  trace ("Sorry, that username wasn't found");
} else {
  trace ("Welcome to the game.");
}
```

이처럼 각각은 별로 대단한 것 같지 않더라도 모두 합쳐놓으면 강력하면서도 유연한 프로그램을 만들 수 있다. 알파벳에 있는 문자나 DNA의 아미노산 배열과 같이 사용자 마음대로 간단한 기본 구조로부터 상상할 수 있는 모든 것을 만들 수 있다. 이 장의 나머지 부분에서는 일반적인 작업을 수행하기 위한 내장 함수 사용법을 포함한 배열 조작 원리에 대해 자세히 알아보기로 하자.

이름이 있는 배열 원소

보통 배열의 원소는 숫자로 참조하지만 이름이 있는 원소를 만들 수도 있다. 이름이 있는 원소를 이용하면 '결합 배열(associative array)', 또는 '해시(hash)'라고 부르는 데이터형과 비슷한 기능을 구현할 수 있다. 하지만 이름이 있는 원소는 Array 메소드(push(), pop()과 같은 메소드. 잠시 후에 배우게 된다)에서 사용할 수도 없고 번호가 붙어 있는 원소 목록과는 별도의 원소로 처리된다. 이름이 있는 원소가 한 개, 숫자 인덱스를 사용하는 원소가 두 개 있는 배열의 length 속성은 3이 아니라 2가 된다. 따라서 배열에 있는 모든 이름이 있는 원소에 액세스하려면 이름이 있는 원소와 숫자 인덱스를 사용하는 원소 목록을 모두 보여줄 수 있는 for-in 루프(8장에서 배운 루프)를 이용해야 한다.

이름이 있는 배열 원소 생성 및 참조

이름이 있는 원소를 추가할 때는 미리 만들어둔 배열에서 대괄호 안에 숫자 대신 문자열을 이용하면 된다.

```
arrayName[elementName] = expression
```

여기서 elementName은 문자열이다. 예를 들면 다음과 같다.

```
var importantDates = new Array();
importantDates["dadsBirthday"] = "June 1";
importantDates["mumsBirthday"] = "January 16";
```

다음과 같이 점 연산자를 사용해도 된다.

```
arrayName.elementName = expression
```

이 때 elementName에는 문자열(따옴표로 둘러싼 것)은 사용할 수 없고 인식자를 사용해야 한다. 예를 들면 다음과 같다.

```
var importantDates = new Array();
importantDates.dadsBirthday = "June 1";
importantDates.mumsBirthday = "January 16";
```

어떤 원소의 인식자(예를 들면 importantDates 배열의 dadsBirthday같은 것)를 알고 있다면 아래에 나온 두 가지 방법 중 하나를 이용하여 그 원소를 사용할 수 있다.

```
var goShopping = importantDates["dadsBirthday"];
var goShopping = importantDates.dadsBirthday;
```

이름이 있는 원소에 어떤 값을 대입할 때와 마찬가지로 대괄호를 이용하여 원소를 액세스할 때는 elementName 자리에 문자열이나 문자열 표현식을 사용할 수 있다. 점 연산자를 이용할 때는 elementName이 문자열이 아닌 인식자(따옴표 없이 원소 이름만 적은 것)밖에 쓸 수 없다.

이름이 있는 원소 제거

배열에서 이름이 있는 원소를 제거하려면 5장에서 배운 delete 연산자를 이용하면 된다.

```
delete arrayName.elementName
```

이름이 있는 원소를 삭제하면 원소의 값과 그 원소가 들어 있는 저장소가 모두 없어지므로 그 원소 및 그 안의 내용물이 차지하고 있는 메모리가 비워진다(이와는 달리 숫자 인덱스를 사용하는 원소에 대해 delete 연산을 적용시키면 값만 사라지고 저장소는 그대로 남는다).

배열에 원소 추가하기

새로운 원소의 값을 지정해주거나 배열의 length 속성을 증가시키거나 내장 배열 함수 중 하나를 이용하면 배열에 원소를 추가할 수 있다.

직접 새로운 원소 추가하기

이미 있는 배열의 특정 인덱스에 있는 원소에 값을 대입하기만 하면 새로운 원소를 추가할 수 있다.

```
// 배열을 만들고 세 개의 값을 대입한다.
var myList = ["apples", "oranges", "pears"];

// 네 번째 원소를 추가한다.
myList[3] = "tangerines";
```

새로운 원소를 반드시 원래 배열의 마지막 원소 바로 뒤에 추가할 필요는 없다. 새 원소를 마지막 원소에서 한 칸 이상 떨어진 위치에 추가하면 액션스크립트에서 자동으로 그 사이에 비어있는 원소를 추가해준다.

```
// 4번부터 38번 인덱스는 비워둔다.
myList[39] = "grapes";

trace (myList[12]); // 12번 원소는 undefined이므로
                    // 아무것도 출력되지 않는다.
```

length 속성을 이용하여 새로운 원소 추가하기

새로운 원소에 값을 대입하지 않고 배열을 확장하고 싶다면 length 속성만 증가시키면 액션스크립트에서 늘어난 길이에 맞게 새로운 원소를 추가한다.

```
// 세 개의 원소가 있는 배열을 만든다.
var myColors = ["green", "red", "blue"];

// 3번부터 49번까지 47개의 비어있는 원소를 추가한다.
myColors.length = 50;
```

학생들의 시험 성적과 같은 데이터를 나중에 새로 추가하기 위해 비어있는 원소를 만들고 싶다면 이러한 방법을 사용하면 된다.

Array 메소드를 이용하여 새로운 원소 추가하기

더 복잡한 원소 추가 작업을 처리할 때는 내장 배열 메소드를 이용하면 된다('12장. 객체와 클래스' 에서 배우겠지만, 메소드는 어떤 객체에 대해 작용하는 함수이다).

push() 메소드

push() 메소드에서는 배열 끝에 한 개 이상의 원소를 추가한다. 원래 있는 배열의 마지막 원소 뒤에 자동으로 데이터를 추가하므로 원래 배열 크기에는 신경 쓰지 않아도 된다. 또한 push 메소드를 이용하면 배열에 한꺼번에 여러 개의 원소를 추가할 수도 있다. 배열에 대해 push() 메소드를 호출하려면 배열 이름 뒤에 점을 찍고 push라는 키워드를 추가한 다음, 그 뒤 괄호 안에 매개변수를 입력하면 된다(매개변수는 하나도 없어도 된다).

```
arrayName.push(item1, item2,...itemn);
```

여기서 item1, item2,...itemn은 배열의 끝에 추가할 새 원소들을 각각 쉼표로 구분하여 늘어놓은 목록이다. 몇 가지 예를 들면 다음과 같다.

```
// 두 개의 원소가 있는 배열을 만든다.
var menuItems = ["home", "quit"];

// 새로운 원소를 추가한다.
// menuItems는 ["home", "quit", "products"]가 된다.
menuItems.push("products");

// 두 개의 원소를 더 추가한다.
// menuItems는 ["home", "quit", "products", "services",
// "contact"]가 된다.
menuItems.push("services", "contact");
```

인자 없이 호출하면 비어있는 원소를 추가한다.

```
menuItems.push();  // 배열 길이를 1 증가시킨다.

// 위의 코드와 똑같은 효과가 나타난다.
menuItems.length++;
```

push() 메소드에서는 새로 업데이트된 배열의 크기를 리턴한다.

```
var myList = [12, 23, 98];
trace(myList.push(28, 36));       // myList에 28과 36을 추가하고
                                  // 화면에는 5를 출력한다.
```

배열에 추가되는 아이템 자리에는 어떤 표현식이라도 사용할 수 있다. 표현식 값은 배열에 추가되기 전에 계산되어 배열에 대입된다.

```
var temperature = 22;
var sky = "sunny";
var weatherListing = new Array();
weatherListing.push(temperature, sky);
trace (weatherListing);   // "temperature,sky"가 아니라
                          // "22,sunny"가 출력된다.
```

푸시, 팝, 스택

push() 메소드의 이름은 '스택(stack)'이라는 프로그래밍 개념에서 나왔다. 스택은 접시를 쌓아놓은 것과 같은 수직 방향의 배열이라고 생각하면 된다. 카페테리아나 뷔페에 가면 스프링 받침대 위에 접시나 쟁반을 쌓아놓은 것을 본 적이 있을 것이다. 새로 씻은 접시를 여기에 추가할 때는 말 그대로 원래 있던 접시(stack) 위에 새로운 접시를 올려서 누르고(push), 그러면 원래 있던 접시들은 아래로 밀려 내려간다. 접시를 꺼낼(pop) 때는 가장 최근에 집어넣은 접시를 꺼내게 된다. 이러한 방식을 후입선출(LIFO, last-in-first-out)[1] 방식이라고 하며, 보통 히스토리 목록 같은 데서 많이 사용한다. 예를 들어 브라우저에서 '뒤로(Back)' 버튼을 누르면 바로 전에 열었던 페이지를 다시 열게 된다. '뒤로' 버튼을 한 번 더 누르면 그 전에 갔던 페이지를 여는 식으로 계속해서 뒤로 돌아간다. 이러한 기능을 구현할 때는 방문했던 페이지의 URL을 스택에 넣었다가(push) '뒤로' 버튼을 누를 때마다 그 URL들을 꺼내는(pop) 방법을 이용한다.

1) 역자주: 가장 나중에 집어넣은 것을 가장 먼저 꺼내는 방식

LIFO 스택의 예는 실생활에서도 쉽게 접할 수 있다. 비행기에서 짐을 마지막으로 집어넣은 승객은 대부분 나중에 짐을 찾을 때 가장 먼저 찾을 수 있다. 이는 짐을 집어넣은 순서와 반대 순서로 짐을 꺼내기 때문이다. 짐을 가장 먼저 맡긴 부지런한 승객은 오히려 짐을 찾을 때 가장 오래 기다려야 한다. 선입선출(FIFO, first-in-first-out)[1] 스택은 더 평등하다. 가장 먼저 온 승객이 가장 먼저 짐을 찾을 수 있기 때문이다. FIFO 스택은 은행에서 줄을 서는 경우와 비슷하다. FIFO 스택에서는 배열의 마지막 원소 대신 첫 번째 원소를 먼저 꺼낸다. 그리고 나서 배열의 첫 번째 원소를 지우면 줄에 서 있을 때 앞 사람이 빠져나가면(즉 앞 사람이 용무를 마치고 나면 앞으로 갈 수 있는 것처럼 두 번째 이후의 배열이 앞으로 이동한다. 따라서 일반적으로 '푸시(push)'는 LIFO 스택을 사용한다는 것을 의미하고 '추가(append)'는 FIFO 스택을 사용한다는 것을 의미한다. 어떤 경우든 새로운 원소는 스택의 맨 뒤에 추가된다. 다음에 원소를 꺼내는 위치가 다를 뿐이다.

unshift() 메소드

unshift() 메소드는 push()와 매우 비슷하지만 한 개 이상의 원소를 배열의 맨 앞에 추가하며 원래 있던 원소를 모두 뒤로 밀어낸다(즉 원래 있던 원소의 인덱스가 맨 앞에 추가된 원소의 개수만큼 커진다). unshift() 메소드의 문법은 다른 모든 배열 메소드와 같다.

```
arrayName.unshift(item1, item2,...itemn);
```

여기서 item1, item2,...itemn은 배열의 맨 앞에 추가할 새 원소들을 각각 쉼표로 구분하여 늘어놓은 목록이다. 이 때 새로운 원소는 입력한 순서 그대로 배열에 추가된다. 몇 가지 예를 들면 다음과 같다.

```
var flashVersions = new Array();
flashVersions[0] = 5;
flashVersions.unshift(4);    // flashVersions는 [4, 5]가 된다.
flashVersions.unshift(2,3);  // flashVersions는 [2, 3, 4, 5]가 된다.
```

1) 역자주: 가장 먼저 집어넣은 것을 가장 먼저 꺼내는 방식

unshift() 메소드도 shift() 메소드와 마찬가지로 새로운 원소가 추가된 배열의 크기를 리턴한다.

splice() 메소드

splice() 메소드는 배열에 새로운 원소를 추가하거나 배열로부터 원소를 제거하는 메소드이다. 보통 배열 중간에 원소를 끼워 넣거나(이 때 끼워 넣는 위치 뒤에 있는 원소들은 그만큼 뒤로 밀린다) 배열 중간에 있는 원소를 제거(이 때 없어지는 원소만큼 뒤에 있는 원소들을 앞으로 끌어당긴다)하기 위해 쓰인다. splice()로 이 두 가지 작업을 한꺼번에 처리하면 어떤 원소를 다른 원소(새로 들어가는 원소의 개수와 원래 들어 있던 원소의 개수가 달라도 된다)로 대체할 수도 있다. splice()는 다음과 같은 식으로 쓰인다.

```
arrayName.splice(startIndex, deleteCount, item1, item2,...itemn)
```

여기서 startIndex는 원소 제거를 시작하는 위치의 인덱스이고 deleteCount는 제거할 원소의 개수(이 때 startIndex에 있는 원소의 개수도 포함된다)이다. deleteCount를 사용하지 않으면 startIndex를 포함하여 그 이후에 있는 모든 원소가 제거된다. 그 뒤에 있는 item1, item2,...itemn은 필수적인 매개변수는 아니며, startIndex 자리부터 제거된 원소가 있던 위치에 들어갈 아이템을 쉼표로 구분하여 집어넣은 목록이다.

〔예제 11-3〕에 splice() 메소드를 사용하는 다양한 방법이 나와 있다.

〔예제 11-3〕 splice() 배열 메소드 사용법

```
// 배열을 만든다.
months = new Array("January", "Friday", "April", "May", "Sunday",
"Monday", "July");
// 배열에 뭔가 잘못된 부분이 있다. 고쳐보자.
// 우선 "Friday"를 지워보자.
months.splice(1,1);
  // months 배열은 이제 다음과 같이 바뀌었다.
  // ["January", "April", "May", "Sunday", "Monday", "July"]

// 이제 "April" 앞에 2월, 3월을 나타내는 "February", "March"를 추가한다.
// 여기에서는 deleteCount가 0이므로 아무런 원소도 삭제하지 않는다.
months.splice(1, 0, "February", "March");
```

```
// months는 이제 다음과 같이 된다.
// ["January", "February", "March", "April",
// "May", "Sunday", "Monday", "July"]

// 마지막으로 "Sunday", Monday"를 제거하고 "June"을 추가한다.
months.splice(5, 2, "june");
// months는 이제 다음과 같이 된다.
// ["January", "February", "March", "April", "May", "June", "July"]

// 이제 months 배열이 수정되었으므로 3번 인덱스("April") 이후에 있는 원소를
// 모두 제거하여 1년 중 4분의 1만 남긴다.
months.splice(3); // months는 ["January", "February", "March"]가 된다.
```

splice()에서는 지워진 원소로 이루어진 배열을 리턴한다. 이 기능을 활용하면 splice() 메소드를 이용하여 다음과 같이 배열에서 일련의 원소를 추출할 수 있다.

```
myList = ["a", "b", "c", "d"];
trace(myList.splice(1, 2));  // "b, c"가 출력된다.
                             // myList는 ["a", "d"]가 된다.
```

아무런 원소도 지워지지 않은 경우에는 비어있는 배열을 리턴한다.

concat() 메소드

push() 메소드와 마찬가지로 concat() 메소드에서는 배열 끝에 원소를 추가한다. 하지만 push()와는 달리 그 메소드를 호출한 배열을 변경하지 않고 새로운 배열을 리턴한다. 게다가 concat() 메소드에서는 인자로 주어진 배열을 각 원소별로 쪼갤 수 있기 때문에, 두 개의 배열을 하나의 새로운 배열로 합칠 수도 있다. concat() 메소드는 다음과 같이 쓰인다.

```
origArray.concat(elementList)
```

concat() 메소드에서는 elementList에 들어있는 원소를 하나씩 origArray에 붙이는데, origArray는 원래대로 둔 채로 새로운 배열을 리턴한다. 대개의 경우 이렇게 리턴된 배열은 변수에 저장한다. 아래 예에서는 간단한 숫자를 배열에 대입한다.

```
var list1 = new Array(11, 12, 13);
var list2 = list1.concat(14, 15);  // list2[11, 12, 13, 14, 15]
```

다음 예에서는 concat() 메소드를 이용하여 두 개의 배열을 합친다.

```
var guests = ["Panda", "Dave"];
var registeredPlayers = ["Gray", "Doomtrooper", "TRK9"];
var allUsers = registeredPlayers.concat(guests);
// allUsers는 ["Gray", "Doomtrooper", "TRK9", "Panda", "Dave"]가 된다.
```

concat() 메소드에서는 push() 메소드와는 달리 guests 배열의 원소를 모두 분리했다는 점을 위 예제를 통해 알 수 있다. 만약 다음과 같은 코드를 실행하면

```
var allUsers = registeredPlayers.push(guests);
```

다음과 같이 중첩 배열이 생긴다.

```
["Gray", "Shift", "TRK9", ["Panda", "Dave"]]
```

게다가 concat() 메소드와는 달리 push() 메소드를 사용하면 registeredPlayers 배열 자체가 변경된다.

하지만 concat()에서 중첩 배열(주 배열 안에 배열 형태로 들어있는 원소)까지 모두 분리하지는 않는다. 아래 코드에서 그러한 사실을 확인할 수 있다.

```
var x = [1, 2, 3];
var y = [[5, 6], [7, 8]];
var z = x.concat(y);  // z는 [1, 2, 3, [5, 6], [7, 8]]이 된다.
                      // y의 0번 및 1번 원소는 그대로 배열 형태로 합쳐진다.
```

배열에서 원소 제거하기

delete 연산자를 이용하거나 배열의 length 속성을 줄이거나 내장 배열 메소드를 이용하면 배열에서 원소를 제거할 수 있다.

delete 연산자를 이용한 원소 제거

delete 연산자는 다음과 같은 식으로 쓰이며 배열 원소를 undefined로 설정한다.

```
delete arrayName[index]
```

여기서 arrayName에는 배열 이름을, index에는 undefined로 설정하고자 하는 원소의 번호 또는 이름을 적으면 된다. 하지만 delete라는 연산자 이름은 약간 오해를 불러일으킬 소지가 있다. delete 연산자를 사용하더라도 배열에서 원소를 제거하는 것은 아니고 해당 원소의 값을 undefined로 설정하는 것뿐이기 때문이다. 따라서 delete 연산은 그 원소에 undefined 값을 대입하는 것과 똑같다. 어떤 배열의 원소 가운데 하나를 제거한 후에 length 속성을 조사해 보면 이러한 사실을 확인할 수 있다.

```
var myList = ["a", "b", "c"];
trace(myList.length);  // Displays: 3
delete myList[2];
trace(myList.length); // 여전히 3이 출력된다. 2번 인덱스에는 "c"가 아니라
                      // undefined가 들어 있지만 여전히 그 원소 자체는 존재한다.
```

원소를 완전히 없애려면 splice()(배열 중간에 있는 원소를 없앨 때)나 shift(), pop()(배열 맨 앞이나 맨 뒤에 있는 원소를 없앨 때) 메소드를 사용해야 한다. delete 연산자는 객체 속성, 이름이 있는 원소, 번호 인덱스를 사용하는 원소에 따라 각각 동작 방식이 다르다는 점에 주의하자. 이름이 있는 원소나 객체 속성에 대해 delete 연산을 적용하면 이름이 있는 원소나 객체 속성이 흔적도 없이 사라진다.

length 속성을 이용한 원소 제거

앞에서 length 속성을 증가시켜 배열에 새로운 원소를 추가하는 방법을 배웠다. 배열의 length 속성을 현재 크기보다 작게 설정하면 배열의 원소를 삭제할 수 있다(즉 배열의 끝을 잘라낼 수 있다).

```
var toppings = ["pepperoni", "tomatoes", "cheese", "green pepper",
"broccoli"];
toppings.length = 3;
trace(toppings);  // "pepperoni,tomatoes,cheese"가 출력된다.
                  // 3, 4번 원소(마지막 두 원소)를 잘라냈다.
```

배열 메소드를 이용한 원소 제거

배열에는 원소를 제거하기 위한 몇 가지 내장 메소드가 들어있다. splice() 메소드에서 배열 중간에 있는 원소를 제거하는 방법은 이미 배웠다. 배열의 맨 앞, 또는 맨 뒤에 있는 원소를 잘라낼 때는 pop()과 shift() 메소드를 이용한다.

pop() 메소드

pop() 메소드는 push() 메소드와 반대되는 작업을 처리한다. 즉 배열의 마지막 원소를 제거한다. pop()의 사용법은 매우 간단하다.

```
arrayName.pop()
```

pop() 메소드를 실행하면 배열의 길이가 1 줄어든다. pop() 메소드의 리턴 값은 그 메소드에서 제거한 원소 값이며, pop() 메소드를 사용한 예를 들면 다음과 같다.

```
x = [56, 57, 58];
x.pop();     // x is now [56, 57]
```

앞에서 배웠듯이 pop()은 보통 push()와 함께 LIFO 스택 작업을 처리하는 데 쓰인다. [예제 11-4]에서는 siteHistory 배열을 이용하여 사용자가 돌아다닌 사이트를 기록한다. 사용자가 새로운 프레임으로 이동하면 push() 메소드를 이용하여 그 위치를 배열에 추가한다. 사용자가 뒤로 돌아가면 siteHistory 배열에서 pop() 메소드를 통해 바로 전 위치를 알아내고 그 위치로 이동한다. [예제 11-4]는 온라인 코드 창고에서 다운로드할 수 있다.

[예제 11-4] 히스토리 기능을 이용한 Back 버튼

```
// 무비의 1번 프레임에 들어갈 코드
stop();
var siteHistory = new Array();

function goto(theLabel) {
  // 아직 요청받은 프레임으로 이동하지 않았다면
  if (theLabel != siteHistory[siteHistory.length - 1]) {
    // 요청받은 사이트를 히스토리에 저장하고 그 사이트로 이동한다.
    siteHistory.push(theLabel);
    gotoAndStop(siteHistory[siteHistory.length - 1]);
```

```
  }
  trace(siteHistory);
}

function goBack() {
  // 히스토리의 마지막 아이템을 제거한다.
  siteHistory.pop();
  // 히스토리에 남은 사이트가 있다면...
  if (siteHistory.length > 0) {
    // 가장 최근 프레임으로 이동한다.
    gotoAndStop(siteHistory[siteHistory.length - 1]);
  } else {
    // 시작 페이지로 이동한다.
    gotoAndStop("home");
  }
  trace(siteHistory);
}

// 이동 버튼에 들어가는 코드
on (release) {
  goto("gallery");
}

// Back 버튼에 들어가는 코드
on (release) {
  goBack();
}
```

shift() 메소드

shift() 메소드는 배열의 맨 앞에 새로운 원소를 추가하는 데 사용했던 unshift()와 반대되는 역할을 하여 배열의 맨 앞에서 원소를 빼낸다.

```
arrayName.shift()
```

pop() 메소드와 마찬가지로 shift() 메소드도 자신이 제거한 원소의 값을 리턴한다. 나머지 원소는 모두 배열의 앞쪽으로 한 칸씩 움직인다. 예를 들면 다음과 같다.

```
var sports = ["hackey sack", "snowboarding", "inline skating"];
sports.shift();  // ["snowboarding", "inline skating"]이 된다.
sports.shift();  // ["inline skating"]이 된다.
```

shift() 메소드는 원소 값만 지우는 것이 아니라 실제로 원소 자체를 없애버리므로 배열의 첫 번째 원소를 없앨 때는 delete 연산자보다는 shift() 메소드를 사용하는 것이 낫다. shift()를 이용하여 목록의 범위를 제한할 수도 있다. 예를 들어 무비의 프레임 속도를 계산하는 경우를 생각해 보자. 매번 프레임이 렌더링될 때마다 push() 메소드를 이용하여 현재 시각을 배열에 저장한다. 배열에 가장 최근 10개 프레임이 렌더링된 시각만 저장하고 싶다면 오래된 시각은 shift() 메소드를 이용하여 제거하면 된다. 그리고 나서 프레임 속도를 계산할 때는 배열에 있는 시간 간격의 평균을 구하면 된다. [예제 11-5]에 이러한 방법이 나와 있다.

[예제 11-5] 무비의 프레임 속도 계산

```
// 무비 클립에서 시간 측정용 배열을 만든다.
onClipEvent(load) {
  var elapsedTime = new Array();
}

// enterFrame 클립 이벤트를 이용하여 매 프레임이 렌더링되는 시각을 측정한다.
onClipEvent(enterFrame) {
  // 현재 시각을 elapsedTime에 추가한다.
  elapsedTime.push(getTimer());

  // 평균 값을 계산할 만큼 충분한 샘플이 있을 때
  if (elapsedTime.length > 10) {
    // elapsedTime에서 제일 오래된 시각을 제거한다.
    elapsedTime.shift();

    // 프레임마다 그 사이에 걸리는 시간의 평균을 계산한다.
    elapsedAverage = (elapsedTime[elapsedTime.length - 1] -
                      elapsedTime[0]) / elapsedTime.length;

    // 초당 프레임 수를 계산하려면 1초(1000ms)를 경과시간의 평균으로 나눈다.[1)]
    fps = 1000 / elapsedAverage;
    trace("current fps " + fps);
  }
}
```

1) 역자주: getTimer() 메소드에서는 무비가 시작한 후로 경과한 시간을 밀리초(1/1000초) 단위로 리턴하므로, elapsedAverage 값을 1000으로 나누면 한 프레임당 걸리는 시간이 나오고 이 역수(즉 1000/elapsedAverage)는 초당 프레임 수가 된다.

splice() 메소드

splice() 메소드를 이용하면 배열에 원소를 추가할 수도 있고 배열에서 원소를 제거할 수도 있다는 것을 이미 배웠다. splice()는 이미 자세히 다루었으므로 여기에서는 그냥 넘어가도록 하겠다. splice()를 원소를 지우는 데 사용하는 예는 다음과 같다.

```
var x = ["a", "b", "c", "d", "e", "f"];
x.splice(1,3); // 1, 2, and 3번 원소를 제거하여 ["a", "e", "f"]만 남긴다.
x.splice(1);   // 1번부터 끝까지 원소를 모두 제거하여 ["a"]만 남긴다.
```

범용 배열 조작 도구

앞에서 배열 메소드를 이용하여 배열에서 원소를 제거하거나 배열에 원소를 추가하는 방법을 알아보았다. 이 외에도 액션스크립트에는 배열의 순서를 바꾸거나 원소를 정렬하거나 배열 원소를 문자열로 바꾸거나 어떤 배열에서 다른 배열을 추출하는 내장 함수가 들어있다.

reverse() 메소드

이름에서 알 수 있듯이 reverse() 메소드는 배열의 원소 순서를 반대로 뒤집는다. 문법은 다음과 같다.

```
arrayName.reverse()
```

이 메소드를 적용한 예는 다음과 같다.

```
var x = [1, 2, 3, 4];
x.reverse();
trace(x);  // "4,3,2,1"이 출력된다.
```

reverse()는 보통 정렬된 목록의 순서를 뒤집을 때 많이 사용한다. 예를 들어 가격을 기준으로 오름차순으로 정렬한 상품 목록이 있으면 상품 목록을 가장 싼 것부터 가장 비싼 것까지 출력할 수 있다. 가장 비싼 물건부터 가장 싼 물건 순으로 목록을 출력하려면 배열의 순서를 뒤집어주면 된다.

연습 문제: 배열에 있는 원소 순서를 뒤집는 함수를 직접 만들어 보자. 생각보다 어려울 뿐만 아니라 reverse() 메소드가 더 빠르다는 것을 느낄 수 있을 것이다.

sort() 메소드

sort() 메소드에서는 배열에 있는 원소의 순서를 사용자가 정하는 규칙에 따라 다시 정렬한다. 아무런 규칙도 정해주지 않으면 sort()에서는 기본적으로 알파벳 순서로 원소를 정렬한다. 배열을 알파벳순으로 정렬하는 것이 가장 쉬우므로 우선 알파벳순 정렬부터 시작해 보자.

```
arrayName.sort()
```

배열에서 sort() 메소드를 호출할 때 아무런 인자도 사용하지 않으면 원소를 임시로 문자열로 바꾼 후에 '부록 B. Latin 1 문자 범주 및 키코드' (자세한 내용은 '4장. 원시 데이터형' 의 '문자 순서와 알파벳 순서 비교' 에서 볼 수 있다)에 나온 코드 포인트 순서에 따라 정렬한다.

```
// 아래 결과는 쉽게 예상할 수 있다.
var animals = ["zebra", "ape"];
animals.sort();
trace(animals);  // "ape,zebra"가 출력된다.
                 // 간단한 메소드 하나로 정렬을 할 수 있다.

// 정렬 순서는 우리가 생각하는 알파벳 순서와는 약간 다르다.
// Zebra의 대문자 Z는 ape의 소문자 a보다 앞에 나온다.
var animals = ["Zebra", "ape"];
animals.sort();
trace(animals);  // "Zebra,ape"가 출력된다. 부록 B 참조.
```

sort() 메소드를 이용하여 직접 선택한 규칙에 따라 배열 원소를 정렬할 수도 있다. 이 방법은 약간 까다롭긴 하지만 매우 유용하다. 우선 인터프리터에서 배열의 두 원소를 비교할 때 사용할 '비교 함수(compare function)' 를 만드는 것부터 시작하자. 그리고 나서 다음과 같이 sort() 메소드를 호출할 때 그 함수를 인자로 전달한다.

```
arrayName.sort(compareFunction)
```

여기서 compareFunction 자리에는 순서를 정하는 방법이 들어있는 비교 함수의 이름이 들어간다.

비교 함수를 만들 때는 우선 두 개의 인자(이 두 인자는 배열에 있는 두 개의 비교 대상 원소를 의미한다)를 받아들이는 함수를 새로 만든다. 함수의 본체에서는 sort() 메소드를 실행시켰을 때 배열의 앞쪽으로 갈 원소를 결정한다. 첫째 원소가 둘째 원소보다 앞으로 가도록 하고 싶다면 함수에서 음수를 리턴한다. 첫 번째 원소가 두 번째 원소보다 뒤로 가도록 할 생각이라면 양수를 리턴한다. 원래 위치에 그대로 남아 있도록 할 때는 0을 리턴한다. 유사코드로 표현한다면 다음과 같이 입력할 수 있다.

```
function compareElements (element1, element2) {
  if (element1이 element2의 앞으로 가는 경우) {
    return -1;
  } else if (element1이 element2의 뒤로 가는 경우) {
    return 1;
  } else {
    return 0;  // 원소를 그대로 두는 경우
  }
}
```

예를 들어 원소들을 숫자를 기준으로 오름차순으로 정렬하려면 다음과 같은 함수를 사용하면 된다.

```
function sortAscendingNumbers (element1, element2) {
  if (element1 < element2) {
    return -1;
  } else if (element1 > element2) {
    return 1;
  } else {
    return 0;  // 두 원소가 같은 경우
  }
}

// 비교 함수를 만들었으므로 직접 테스트해보자.
var x = [34, 55, 33, 1, 100];
x.sort(sortAscendingNumbers);
trace(x);    // "1,33,34,55,100"이 출력된다.
```

숫자 크기를 기준으로 정렬하는 비교 함수는 훨씬 더 간단하게 만들 수 있다. 앞에서 만든 sortAscendingNumbers() 함수는 다음과 같이 쓸 수 있다.

```
function sortAscendingNumbers (element1, element2) {
  return element1 - element2;
}
```

이렇게 하면 element1이 element2보다 작은 경우에는 음수가 리턴되고 element1이 element2보다 큰 경우에는 양수가 리턴되며 두 수가 같은 경우에는 0이 리턴된다. 정말 간결하면서도 우아한 코드이다. 내림차순 정렬을 할 때는 다음과 같이 하면 된다.

```
function sortDescendingNumbers (element1, element2) {
  return element2 - element1;
}
```

〔예제 11-6〕에는 sort()의 기본 알파벳순 비교 방법을 약간 수정하여 대소문자를 구분하지 않고 정렬할 수 있도록 하는 비교 함수가 나와 있다.

〔예제 11-6〕 대소문자를 구분하지 않는 알파벳순 배열 정렬

```
var animals = ["Zebra", "ape"];

function sortAscendingAlpha (element1, element2) {
 return (element2.toLowerCase() < element1.toLowerCase());
}

animals.sort(sortAscendingAlpha);
trace(animals);  // "ape,Zebra"가 출력된다.
```

물론 비교 함수에서 간단한 문자열이나 숫자만을 다룰 수 있는 것은 아니다. 다음 예에서는 픽셀 영역을 기준으로 무비 클립을 오름차순으로 정렬한다.

```
var clips = [square1, square2, square3];

function sortByClipArea (clip1, clip2) {
  clip1area = clip1._width * clip1._height;
  clip2area = clip2._width * clip2._height
  return clip1area - clip2area;
}

clips.sort(sortByClipArea);
```

정말 강력하고도 멋진 정렬 메소드의 위력을 볼 수 있다.

slice() 메소드

slice() 메소드는 splice() 메소드의 일부분을 구현한 형태의 메소드이며, 배열에서 연속된 원소들을 추출하는 기능을 가지고 있다. 하지만 splice()와는 달리 slice()에서는 원소를 추출하는 기능만을 수행할 뿐 그 원소들을 지우거나 다른 원소들을 그 자리에 집어넣지는 않는다. slice() 메소드에서는 새로운 배열을 만들고 그 대신 원래 배열을 변경시키지 않는다. 이 메소드를 사용하는 방법은 다음과 같다.

```
origArray.slice(startIndex, endIndex)
```

startIndex는 가져올 원소 중 첫 번째 원소의 인덱스를, endIndex는 마지막 원소 바로 다음 원소의 인덱스를 나타낸다. slice() 메소드에서는 origArray[startIndex]부터 origArray[endIndex - 1]까지의 원소를 복사한 새로운 배열을 리턴한다. endIndex를 생략하면 그 자리에 origArray.length가 들어간 것처럼 동작하며 origArray[startIndex]에서부터 origArray[origArray.length - 1]까지의 원소를 복사한 배열을 리턴한다. 몇 가지 예를 들면 다음과 같다.

```
var myList = ["a", "b", "c", "d", "e", "f"];
myList.slice(1, 3);  // ["b", "c", "d"]가 아닌 ["b", "c"]를 리턴한다.
myList.slice(2);     // ["c", "d", "e", "f"]를 리턴한다.
```

join() 메소드

배열의 모든 원소를 나타내는 문자열을 만들고 싶다면 join() 메소드를 이용하면 된다. join() 메소드에서는 우선 주어진 배열의 각 원소를 문자열로 변환한다. 비어있는 원소는 비어있는 문자열("")로 변환된다. 그리고 나서 변환한 모든 문자열을 하나의 긴 문자열로 변환한다. 이 때 각 문자열은 구분자(delimiter)라는 특정한 문자(또는 문자열)를 이용하여 구분한다. 이 모든 작업이 끝나면 join()에서는 새로 만든 문자열을 리턴한다. join() 메소드의 사용법은 다음과 같다.

```
arrayName.join(delimiter)
```

delimiter는 arrayName의 원소를 변환한 문자열을 구분하는 데 쓰이는 문자열이다. delimiter를 지정해 주지 않으면 기본값인 쉼표(,)가 구분자로 쓰인다. 다음 예제를 보면 join() 선언문을 쉽게 이해할 수 있을 것이다.

```
var siteSections = ["animation", "short films", "games"];

// siteTitle을 "animation>> short films>> games"로 설정한다.
var siteTitle = siteSections.join(">> ");

// siteTitle을 "animation:short films:games"로 설정한다.
var siteTitle = siteSections.join(":");
```

join()에서는 그 메소드를 호출한 배열을 건드리지 않고 대신 그 배열을 바탕으로 만든 문자열을 리턴할 뿐이다.

join() 메소드를 호출할 때 구분자를 전달하지 않으면 toString()을 사용한 경우와 똑같은 결과를 리턴한다. 하지만 toString()에서는 쉼표 뒤에 공백을 집어넣지 않으므로 조금 더 보기 좋은 결과를 원한다면 join() 메소드를 사용하는 것이 좋다.

```
var x = [1, 2, 3];
trace(x.join(", "));  // "1,2,3"이 아니라 "1, 2, 3"이 리턴된다.
```

배열의 원소 중에 또 다른 배열이 있는 경우에는 이상한 결과가 나올 수도 있다. join()에서는 원소를 문자열로 변환하고, 배열은 toString() 메소드를 통해 문자열로 변환되므로 원소중에 배열이 있다면(즉 중첩 배열이 있다면), 그러한 중첩 배열은 join() 메소드를 호출할 때 사용한 구분자가 아닌 쉼표를 구분자로 사용하여 문자열로 변환된다. 즉 join() 메소드를 호출할 때 다른 구분자를 사용하더라도 중첩 배열에 대해서는 그 구분자가 적용되지 않는다. 예를 들면 다음과 같다.

```
var x = [1, [2, 3, 4], 5];
x.join("|");  // "1|2,3,4|5"가 리턴된다.
```

toString() 메소드

12장에서 배우겠지만 toString()은 임의의 객체의 문자열 표현을 리턴하는 범용 객체 메소드이다. Array 객체의 경우에는 toString() 메소드를 호출하면 배열의 모든 원소를 쉼표로 구분한 목록을 문자열 형태로 리턴한다. toString() 메소드는 다음과 같이 직접 호출할 수 있다.

```
arrayName.toString()
```

하지만 대개의 경우에 배열에 toString() 메소드를 직접 사용할 필요는 없다. 어떤 배열의 이름을 문자열이 들어갈 자리에 사용하면 자동으로 toString() 메소드가 호출되기 때문이다. 예를 들면 trace(arrayName)이라고 쓰면 쉼표로 구분한 값의 목록이 Output 창에 출력된다. 이 때 trace(arrayName)은 사실 trace(arrayName.toString())과 같다. 디버깅 과정에서 배열의 원소를 간단하게 확인하는 경우에는 toString() 메소드가 유용하게 쓰인다. 예를 들면 다음과 같다.

```
var sites = ["www.moock.org", "www.macromedia.com", "www.oreilly.com"];
// 배열을 텍스트 필드에 출력한다.
debugOutput = "the sites array is" + sites.toString();
```

객체로서의 배열

12장을 읽고 나면 배열도 Array 클래스의 인스턴스로 객체의 일종이라는 것을 알 수 있을 것이다. 다른 객체와 마찬가지로 배열 객체에도 속성을 추가할 수 있다. 배열에 이름이 있는 원소를 추가할 때는 사실 Array 객체에 속성을 추가하는 것뿐이다. 하지만 앞에서 배운 것처럼 Array 클래스의 내장 함수는 숫자 인덱스를 사용한 원소에 대해서만 사용할 수 있다.

'9장. 함수'에서 배열과 객체를 혼합한 인자 객체에 대해 알아보았다. 인자 객체는 번호가 있는 원소와 속성을 혼합한 것이다(인자 객체에는 callee라는 속성이 있어서 매개변수를 배열 형태로 저장한다는 점을 떠올려 보자). 이제 모든 배열에는 번호가 붙은 원소 목록, 내장 속성인 length, 그리고 사용자가 마음대로 추가할 수 있는 다른 속성을 저장할 수 있다는 점을 확실히 이해할 수 있을 것이다.

다차원 배열

지금까지는 스프레드시트의 한 행 또는 한 열과 같은 일차원 배열만을 알아보았다. 하지만 스프레드시트와 마찬가지로 행과 열이 모두 있는 배열을 만들 수 있을까? 그렇게 하려면 하나의 차원을 추가해야 한다. 액션스크립트에서는 원래 일차원 배열만을 지원하지만 배열 안에 다른 배열을 만들면 다차원 배열과 비슷한 효과를 만들 수 있다. 즉 배열을 다른 배열의 원소로 사용하면 된다(중첩 배열이라고 부른다).

가장 간단한 다차원 배열은 이차원 배열인데, 행(row)과 열(column)로 원소들을 정리해 놓은 것이라고 생각하면 된다. 이 때 행은 배열의 첫 번째 차원, 열은 두 번째 차원에 해당한다.

실질적인 예를 통해 이차원 배열을 어떤 식으로 사용하는지 알아보자. 각각 수량과 가격이 포함된 세 개의 제품에 대한 주문을 처리하는 경우를 가정해 보자. 이 때 세 개의 행(한 제품에 한 행)과 두 개의 열(하나는 수량, 나머지 하나는 가격)이 있는 스프레드시트와 같은 구조를 만들자. 각 행마다 하나씩 배열을 만들어 두 원소를 각각 첫째 열과 둘째 열에 해당시킨다고 볼 수 있다.

```
var row1 = [6, 2.99];   // 6개, 가격은 2.99
var row2 = [4, 9.99];   // 4개, 가격은 9.99
var row3 = [1, 59.99];  // 1개, 가격은 59.99
```

그리고 나서 각 행을 spreadsheet라는 배열에 집어넣는다.

```
var spreadsheet = [row1, row2, row3];
```

이렇게 하면 각 행의 수량과 가격을 곱하고 그 값을 모두 합하면 전체 가격을 구할 수 있다. 이차원 배열의 원소는 두 개의 인덱스(각각 행과 열을 나타냄)를 이용하여 액세스할 수 있다. 예를 들어 spreadsheet[0]은 첫 번째 행의 2열짜리 배열을 나타낸다. 따라서 첫 번째 행의 두 번째 열에 액세스하려면 spreadsheet[0][1]이라고 적으면 된다.

```
// 주문 총액을 저장할 변수를 만든다.
var total;

// 주문 총액을 계산한다. 각 행에 있는 두 원소를 곱한 후
// 그 값을 total에 더한다.
for (var i = 0; i < spreadsheet.length; i++) {
  total += spreadsheet[i][0] * spreadsheet[i][1];
}

trace(total);  // 117.89가 출력된다.
```

저장하는 법과 액세스하는 법을 제외하면 다차원 배열도 일반적인 배열과 똑같이 작동한다. 따라서 다차원 배열에 들어있는 데이터를 다룰 때도 Array 메소드를 사용할 수 있다.

객관식 퀴즈, 버전 3

〔예제 9-10〕에서는 〔예제 1-1〕에 있는 객관식 퀴즈를 개조했다. 그 때는 재사용할 수 있는 집중화된 함수를 이용하여 퀴즈 코드를 개선하였다. 지금까지 배운 배열에 대한 내용을 바탕으로 앞에서 만든 퀴즈 예제를 조금 더 향상시켜보자. 이번에는 각 문제별로 사용자가 선택한 답과 정답을 배열에 저장하여 코드를 더 간결하고 읽기 쉽게 만들고 새로운 문제를 추가할 때도 확장하기 좋도록 만들어보자.

이번에 고칠 부분은 메인 퀴즈 코드가 있는 1번 프레임의 scripts 레이어이다. 이전에 만들었던 버전과 이번에 만들 버전은 모두 온라인 코드 창고에서 구할 수 있다. 이번에 어떤 점을 향상시키는지 알고 싶다면 〔예제 11-7〕의 코드를 자세히 살펴보자. 특히 주석 부분에 주의를 기울이자.

〔예제 11-7〕 객관식 퀴즈, 버전 3

```
stop();
//  *** 메인 타임라인 변수를 초기화한다.
var displayTotal;                // 사용자의 최종 점수를 출력하기 위한 텍스트
필드
var numQuestions = 2;            // 퀴즈에 있는 문제 수
var totalCorrect = 0;            // 정답 개수
var userAnswers = new Array();   // 사용자가 선택한 답을 저장할 배열
var correctAnswers = [3, 2];     // 각 문제의 정답이 있는 배열
// q1answer, q2answer나 correctAnswer1, correctAnswer2와 같은
// 변수를 사용하지 않는다. 이러한 정보는 간단하게 배열을 이용하여
// 저장한다.

// *** 사용자가 선택한 답을 등록하기 위한 함수
function answer (choice) {
  // userAnswers의 length 속성을 이용하여 사용자가 답한 문제의 수를
  // 확인할 수 있으므로 answer.currentAnswer 속성을 이용하여 직접
  // 문제수를 계산하지 않아도 된다. 또한 사용자가 선택한 답을 동적으로
  // 이름을 붙인 변수에 저장하기 위해 set 선언문을 사용하지 않아도 된다.

  // 사용자가 선택한 답을 배열에 저장한다.
  userAnswers.push(choice);
  // 문제를 푼 개수에 따라 다음 이동할 클립을 선택한다.
  if ((userAnswers.length) == numQuestions) {
    gotoAndStop ("quizEnd");
  } else {
```

```
      gotoAndStop ("q"+ (userAnswers.length + 1));
    }
  }

  // *** 사용자의 점수를 계산하는 함수
  function gradeUser() {
    // 정답의 개수를 센다. userAnswer와 correctAnswer의 원소를 비교하면
    // 이전 버전에서 eval() 함수를 이용하는 것보다 훨씬 깔끔하게 정답의
    // 개수를 확인할 수 있다.
    for (var i = 0; i < userAnswers.length; i++) {
      if (userAnswers[i] == correctAnswers[i]) {
        totalCorrect++;
      }
    }
    // 사용자의 점수를 동적인 텍스트 필드에 출력한다.
    displayTotal = totalCorrect;
  }
```

연습 문제: numQeusteions 대신 correctAnswers.length를 이용하여 위 코드를 더 일반화시켜 보자.

앞으로 배울 내용

이 장에서는 첫째 복합 데이터형인 배열에 대해 기초적인 내용부터 시작하여 다양한 응용법까지 알아보았다. 여러 조각의 데이터를 하나의 데이터로 저장하는 법을 제대로 이해했다면 다음 장을 공부할 준비가 끝난 셈이다. 다음 장에서는 더 강력한 복합 데이터형인(그리고 액션스크립트의 근본적인 구성요소인) 객체에 대해 살펴보기로 하자.