14

렉시컬 구조

어떤 언어의 렉시컬 구조는 그 문법적 구성을 결정하는 일련의 규칙들을 의미한다. 스크립트의 소스 코드를 만들 때는 이러한 규칙을 따라야 한다.

공백

탭, 스페이스, 캐리지 리턴(줄바꿈) 문자는 우리말에서 띄어쓰기를 할 때 쓰이는 것과 마찬가지로 액션스크립트에서 각 단어들을 서로 구분하기 위해 쓰인다. 이렇게띄어쓰기를하지않으면읽기도힘들어지고이해하기도힘들다. 프로그래머들이 쓰는 용어로는 이러한 문자를 '공백(whitespace)'이라고 하며 소스 코드에서 서로 다른 '토큰(token, 키워드, 인식자 또는 우리말의 단어, 구문, 문장 따위와 비슷한 표현식)'을 분리하기 위해 쓰인다. 공백을 잘못 사용한 경우와 제대로 사용한 경우의 예는 다음과 같다.

```
varx   // 키워드인 var와 변수 x 사이에 공백이 없다.
var x  // 이렇게 해야 한다. 공백이 있기 때문에 인터프리터에서는
       // 이제 이 코드를 이해할 수 있다.
```

한 토큰이 끝나고 다른 토큰이 시작되는 것을 알려주는 다른 '구분자(delimiter, separator)'가 있다면 공백을 사용하지 않아도 된다. 아래 예에서는 =, +, /와 같은 연산자가 x, 10, 5, y를 서로 구분해주기 때문에 공백을 사용하지 않아도 된다.

```
x=10+5/y;         // 조금 빽빽해 보이긴 하지만 틀린 문장은 아니다.
x = 10 + 5 / y;  // 위 내용과 똑같은 선언문이지만 읽기가 더 편하다.
```

이와 마찬가지로 괄호, 중괄호, 대괄호, 쉼표, >, < 기호와 같이 구분자 역할을 하는 다른 문자가 있으면 공백을 사용하지 않아도 된다. 다음과 같은 문장은 너무 빽빽해 보이긴 하지만 문법적으로 전혀 문제가 없다.

```
for(var i=0;i<10;i++){trace(i);}
if(x==7){y=[1,2,3,4,5,6,7,8,9,10];}
myMeth=function(arg1,arg2,arg3){trace(arg1+arg2+arg3);};
```

공백을 추가하는 것은 사실 스타일 문제일 뿐이다. 어차피 액션스크립트 인터프리터에서는 불필요한 공백을 무시하기 때문이다. 하지만 코드의 가독성을 높이기 위해 따라야 하는 몇 가지 관행이 있다. 예를 들어 앞에서 사용한 표현식을 다음과 같이 써도 되지만 가독성은 많이 떨어진다.

```
x =
  10
  + 5
  / y;
```

선언문은 줄바꿈 문자가 아닌 세미콜론으로 끝마치도록 되어있다. 거의 모든 경우에 줄바꿈 문자는 선언문에 별 영향을 미치지 않고 선언문 종료자로 쓰이지도 않는다. 따라서 선언문을 읽기 쉽도록 하기 위해 줄바꿈 문자(탭 및 스페이스도 함께 사용)를 몇 개 정도 집어넣어도 된다.

```
myNestedArray = [[x, y, z],
                 [1, 2, 3],
                 ["joshua davis", "yugo nakamura", "james patterson"]];
// 다음과 같이 적는 것보다 훨씬 읽기가 좋다.
myNestedArray = [[x, y, z], [1, 2, 3],["joshua davis","yugo nakamura",
"james patterson"]];
// 아래와 같은 표현도 그리 읽기 좋은 것은 아니다.
myNestedArray = [[x, y, z],
[1, 2, 3],
["joshua davis","yugo nakamura","james patterson"]];
```

하나의 선언문을 여러 줄에 걸쳐 쓰는 데는 특별히 다른 작업이 필요하지 않다. 그냥 캐리지 리턴을 추가하고(즉 엔터키를 누르고) 계속해서 필요한 내용을 입력하기만 하면 된다. 코드의 가독성을 향상시키는 것과 관련된 자세한 내용은 스티브 맥코넬의 『*Code Complete*』(마이크로소프트 프레스)를 참조하기 바란다. 하지만 줄바꿈 문자가 선언문 종료자로 해석되는 경우도 있다. 이와 관련된 것은 다음 절에서 자세히 살펴보자.

인용부호 밖에 있는 공백은 무시되지만 인용부호 안에 있는 공백은 무시되지 않는다. 아래 두 예를 비교해 보자.

```
x = 5;
trace("The value of x is"  +   x); // "The value of x is5"가 출력된다.
trace("The value of x is "+x);     // "The value of x is 5"가 출력된다.
```

선언문 종료자(세미콜론)

'6장. 선언문'에서 배웠듯이 세미콜론은 액션스크립트에서 한 선언문이 끝났음을 의미한다. 관용적으로 선언문이 끝나는 곳에는 항상 세미콜론을 추가해야 하지만 액션스크립트에서는 반드시 그런 것은 아니다. 액션스크립트 인터프리터에서는 세미콜론이 생략된 경우에도 선언문이 끝나는 곳을 적절히 유추해낸다.

```
// 이렇게 하는 것이 더 바람직하다.
var x = 4;
var y = 5;
// 하지만 다음과 같이 해도 된다.
var x = 4
var y = 5
```

액션스크립트 인터프리터에서는 위와 같은 경우에 줄바꿈 문자를 선언문 종료자로 간주한다(C와 같이 더 엄격한 언어의 컴파일러에서는 오류가 발생한다). 하지만 코드에 있는 선언문에서 세미콜론을 빼먹는 것은 글을 쓸 때 마침표를 빼먹는 것과 비슷하다. 독자가 대부분의 문장을 이해할 수는 있겠지만 혼동을 일으킬만한 부분이 생길 수도 있고 가독성도 많이 떨어진다. 예를 들어 return 선언문 뒤에 세미콜론을 빼먹으면 어떤 결과가 나타날지 생각해 보자.

```
function addOne (value) {
  return
  value + 1
}
```

위와 같은 구문이 있으면 액션스크립트에서는 다음과 같은 내용으로 간주한다.

```
function addOne (value) {
  return;
  value + 1;
}
```

따라서 value + 1 대신 undefined가 리턴된다. return 키워드만 사용하는 것도 문법적으로 문제가 없기 때문이다. 어떤 줄에 return만 덩그러니 있고 value + 1 뒤에 세미콜론이 있는 경우에도 return 선언문은 return 키워드만으로 이루어진(즉 그 뒤에 아무 것도 없는) 하나의 선언문으로 인식된다.

이러한 문제를 해결하기 위해서는 세미콜론을 항상 적어주는 습관을 기르는 것이 좋다. 또한 return 선언문과 같은 경우에는 return 키워드 뒤에 오는 표현식을 다른 줄에 적지 않도록 주의해야 한다. 그렇게 하면 선언문의 의미 자체가 달라지기 때문이다. 따라서 위 예는 다음과 같이 써야 한다.

```
function addOne (value) {
  return value + 1;
}
```

세미콜론은 개별 선언문이 끝나는 것을 표시하긴 하지만 선언문 블록 구분자(중괄호)가 있는 곳에는 사용하지 않아도 된다.

```
for (var i=0;i<10;i++) {  // 세미콜론을 쓰지 않는다.
  trace(i);               // 여기에는 세미콜론이 들어간다.
}                         // 세미콜론을 쓰지 않는다.

if(x == 10) {             // 세미콜론을 쓰지 않는다.
  trace("x is ten");      // 여기에는 세미콜론이 들어간다.
} else {                  // 세미콜론을 쓰지 않는다.
  trace("x is not ten"); // 여기에는 세미콜론이 들어간다.
}                         // 세미콜론을 쓰지 않는다.
[_Code_]
on (release) {            // 세미콜론을 쓰지 않는다.
```

```
    trace("Click");          // 여기에는 세미콜론이 들어간다.
}                            // 세미콜론을 쓰지 않는다.
```

하지만 함수 리터럴 뒤에는 반드시 세미콜론을 사용해야 한다.

```
function (param1, param2, ... paramn) { statements };
```

#include 명령어에서는 세미콜론을 사용할 수 없기 때문에 세미콜론을 적으면 오류가 발생한다. '3부. 레퍼런스' 의 #include 부분을 참조하기 바란다.

주석

'주석(comment)' 은 인터프리터에서는 무시하지만 사람들이 읽기 위한 목적으로 추가하는 텍스트이다. 코드의 내용을 설명하거나 버전 관련 정보를 제공하거나 자료 구조를 설명한다거나, 그러한 프로그래밍 방법을 사용한 이유를 남겨두기 위해 주석을 자세하게 적어주는 것이 좋다. 또한 주석에서 단순하게 코드 문법 자체를 설명할 필요는 없고 그 코드와 관련된 개념을 확실하게 적어줄 필요가 있다. 예를 들어 다음과 같은 주석은 사실 그다지 필요가 없다.

```
// i에 5를 대입한다.
i = 5;
```

하지만 아래 주석에서는 i에 5를 대입하는 이유를 설명하고 있기 때문에 코드의 흐름을 따라가는 데 크게 도움이 된다.

```
// 5번 인덱스에서 시작하는 비밀번호 문자열을
// 검색하기 위해 카운터를 초기화한다.
var i = 5;
```

또한 아예 변수나 핸들러의 이름을 이해하기 쉽게 만들어서 그러한 이름 자체가 주석 역할을 할 수 있도록 하는 습관을 기르는 것도 좋다. 아래 두 예 중 어느 쪽이 더 이해하기 좋은지 비교해 보자.

```
x = y / z;                       // 이해하기 힘들다.
average = sum / numberOfItems;   // 따로 설명할 필요 없이 쉽게
                                 // 이해된다.
```

액션스크립트에서는 한 줄짜리 주석과 여러 줄짜리 주석을 모두 지원한다. 한 줄짜리 주석은 지금까지 많이 본 두 개의 슬래시를 연속으로 적은 형태의 주석이다.

```
// 이 부분은 사람들만 읽을 수 있고 인터프리터에서는 무시하는 부분이다.
```

한 줄짜리 주석은 그 줄이 끝나면 자동으로 끝난다. 여러 줄에 걸쳐서 주석을 쓰고 싶다면 다음 줄에도 // 문자를 추가해야 한다.

```
// 여기는 주석을 시작하는 부분이다.
// 여기에도 주석을 더 적을 수 있다.
```

또한 다음과 같이 코드 뒤에 주석을 추가할 수도 있다.

```
var x;     // 코드와 같은 줄에 주석을 달아도 된다.
```

액션스크립트에서는 여러 줄짜리 주석도 지원한다. 보통 큰 코드 블록에 대한 주석을 달 때 사용하며, 이 주석은 줄바꿈 문자로 끝나지 않는다. 여러 줄짜리 주석은 /* 로 시작하고 */ 가 나타날 때까지 계속 주석으로 처리된다.

```
/* -----BEGIN VERSION CONTROL INFO----
  Name: MyApplication, Version 1.3.1
  Author: Killa Programma
  Last Modified: August 07, 2000
   ------END VERSION CONTROL INFO-----
*/
```

여러 줄짜리 주석은 플래시 5 액션스크립트 에디터의 전문가 모드나 #include 명령어를 이용해 포함할 수 있는 .as 소스 파일에서만 사용할 수 있다. 전문가 모드에서 일반 모드로 전환하면 여러 줄짜리 주석이 한 줄짜리 주석으로 변환된다. 다음과 같이 주석을 겹쳐서 사용하는 것도 가능하다.

```
/* 이 주석은
/* 겹쳐 있는 // 주석입니다 */
```

다음과 같은 코드는 관행적으로 사용하지 않지만 문법적으로 문제가 있는 것은 아니다.

```
/* 이 주석 뒤에는 실제 코드가 들어간다 */ var x = 5;
```

어떤 코드를 고치고 나면(많이 고치거나 적게 고치거나 크게 상관없이) 다음과 같이 날짜와 수정한 사람의 이니셜을 추가하는 것이 보통이다.

```
// Fixed error in financial calculation. BAE 12-04-00. V1.01
```

코드의 일부를 잠시 사용하지 않거나(또는 어떤 코드를 더 이상 사용하지 않을 계획이더라도) 코드를 그냥 지워버리는 대신 주석으로 활용할 수 있다. 이렇게 코드를 주석으로 만들어 인터프리터에서 그냥 건너 뛰도록 하는 것을 코드를 주석 처리한다고 말한다. 이렇게 주석 처리한 코드를 나중에 다시 사용하고 싶으면 주석 구분자만 지워버리면 된다.

```
/* 이 코드 블록을 일단 사용하지 않도록 한다.
duplicateMovieClip("character", "newCharacter", 1);
newCharacter._rotation = 90;
*/
```

한 줄만 주석 처리하고 싶다면 다음과 같이 슬래시를 두 개 사용해도 된다.

```
// newCharacter._rotation = 90;
```

예약어

액션스크립트에서는 몇 가지 '예약어(reserved word)'를 사용하여 선언문이나 연산자와 같이 몇 가지 내장된 기능을 표기한다. 예약어는 인터프리터에서 사용하기 위해 예약해 둔 것이므로 코드에서 인식자로 사용하지 않도록 주의해야 한다. 예약어를 원래 기능이 아닌 다른 기능으로 사용하려고 한다면 대부분 오류가 발생하게 된다. 액션스크립트의 예약어는 〔표 14-1〕에 나와 있다.

〔표 14-1〕 액션스크립트 예약어

add*	for	lt*	tellTarget*
and*	function	ne*	this
break	ge*	new	typeof
continue	gt*	not*	var
delete	if	on	void

do	ifFrameLoaded*	onClipEvent	while
else	in	or*	with
eq*	le*	return	

* 플래시 4의 예약어. 플래시 5에서는 더 이상 쓰이지 않는다.

또한 〔표 14-2〕에 열거한 키워드도 사용하지 않는 것이 좋다. 〔표 14-2〕에 나온 키워드는 플래시 5 액션스크립트에는 포함되지 않지만 ECMA-262에서 앞으로 사용할 수 있는 키워드이기 때문에, 나중에 액션스크립트에서도 쓰일 가능성이 있다.

〔표 14-2〕 나중에 도입될 가능성이 있는 예약어

abstract	extends	private
boolean	final	protected
byte	finally	public
case	float	short
catch	goto	static
char	implements	super
class	import	switch
const	instanceof	synchronized
debugger	int	throws
default	interface	transient
double	long	try
enum	native	volatile
export	package	

확실히 정의된 키워드 외에도 내장 속성이나 메소드, 객체의 이름을 인식자로 사용하는 것은 피하는 것이 좋다. 그렇게 하면 내장 속성, 메소드, 객체를 무효화시킬 수도 있기 때문이다. 예를 들면 다음과 같다.

```
Date = new Object();  // 이렇게 하면 Date 생성자가 무효화된다.
```

위와 같이 하면 더 이상 Date 객체를 만들 수 없다.

```
var now = new Date();  // now가 undefined로 설정된다.
trace(now);            // 현재 시각과 날짜 대신 비어있는 문자열이 출력된다.
```

인식자

모든 변수, 함수, 객체 속성의 이름은 인식자이다. 액션스크립트의 인식자는 다음과 같은 규칙을 따라야 한다.

- 인식자에는 문자(A-Z, a-z), 숫자, 밑줄, 달러 표시만을 사용할 수 있다. 인식자에 스페이스, 마침표, 역슬래시 같은 다른 문장부호를 사용하지 않도록 각별한 주의를 기울여야 한다.
- 인식자는 문자, 밑줄 또는 달러 표시로 시작해야 한다(숫자는 맨 앞에 올 수 없다).
- 예약어와 같은 인식자를 사용할 수 없다.

꼭 그래야 하는 것은 아니지만 무비 클립 인스턴스 이름이나 프레임 레이블, 레이어 이름을 만들 때도 위와 같은 규칙을 따르는 습관을 기르는 것이 좋다.

대문자와 소문자

대소문자를 확실히 구분하는 언어에서는 그 언어에서 쓰이는 모든 토큰(인식자와 키워드 포함)을 대소문자를 가려서 입력해야 한다. 다음과 같은 예를 생각해 보자.

```
If (x == 5) {
  x = 10;
}
```

만약 대소문자를 구분하는 언어에서 위와 같은 선언문을 사용한다면, if라는 키워드를 If로 썼기 때문에 오류가 발생한다. 게다가 대소문자를 구분하는 언어에서는 다음과 같은 두 선언문에서 서로 다른 변수를 정의하게 된다(첫 번째 선언문에서는 firstName, 두 번째 선언문에서는 firstname이라는 변수를 정의한다).

```
var firstName = "doug";
var firstname = "terry";
```

액션스크립트의 기반이 되는 ECMA-262 스펙에서는 대소문자를 철저하게 구분하도록 되어있다. 하지만 액션스크립트에서는 플래시 4 무비와의 하위 호환성을 유

지하기 위해 이 표준 스펙을 따르지 않는다. 액션스크립트에서 〔표 14-1〕에 있는 키워드에 대해서는 대소문자를 구분하지만, 인식자에 대해서는 대소문자를 구분하지 않는다. 예를 들어 아래 코드에서는 onClipEvent 키워드를 onclipevent라고 적었기 때문에 오류가 발생한다.

```
onclipevent (enterFrame)  // onClipEvent (enterFrame)이 맞다.
```

하지만 키워드와는 달리 인식자에서는 대소문자를 구분하지 않으므로 아래 선언문에서는 똑같은 변수에 다른 값을 대입한다.

```
var firstName = "margaret";
var firstname = "michael";
trace(firstName);         // "michael"이 출력된다.
trace(firstname);         // 이번에도 "michael"이 출력된다
                          // (두 변수는 같은 변수이다).
```

액션스크립트에서는 속성 이름이나 함수 이름과 같은 내장 인식자에서도 대소문자를 구분하지 않는다. 다음과 같은 코드를 사용하면 자바스크립트에서는 오류가 발생하지만 액션스크립트에서는 전혀 문제 없이 실행된다.

```
myList = new array();   // new Array();라고 하는 것이 바람직하다.
```

만약 자바스크립트 코드를 액션스크립트로 옮긴다거나 자바스크립트 프로그래머가 액션스크립트를 배운다면 이러한 점에서 문제가 생길 수도 있다. 자바스크립트에서는 같은 이름을 용도에 따라 대소문자를 다르게 해서 서로 다른 변수로 사용하는 경우가 종종 있기 때문이다. 예를 들어 다음과 같은 선언문을 생각해 보자.

```
date = new Date();   // 자바스크립트에서는 제대로 동작하지만
                     // 액션스크립트에서는 안 된다.
```

위 선언문을 쓰면 액션스크립트에서는 객체 클래스인 Date와 인식자 date를 서로 구분할 수 없기 때문에, 심각한 문제가 생길 수도 있다. 액션스크립트에서 위와 같은 코드를 사용하면 내장 클래스인 Date를 못 쓰게 된다. 따라서 인식자를 정의할 때 Date나 Array와 같이 미리 정의된 인식자와 적어도 하나 이상의 문자가 다른 인식자를 사용해야 한다. 위에 있는 코드를 액션스크립트에서 사용할 때는 다음과 같이 하면 된다.

```
myDate = new Date(); // 액션스크립트에서는 이렇게 하는 것이 좋다.
```

어떠한 상황에서도 가장 중요한 것은 일관성을 유지하는 것이다(언어 자체에서 일관성을 유지하지 않아도 되는 경우라도 프로그래머가 알아서 일관성을 유지하는 것이 좋다). 변수, 함수, 인스턴스 및 기타 아이템을 사용할 때 대소문자를 확실히 구분해 주면 가독성도 향상되고, 액션스크립트에서 대소문자를 철저하게 구분하는 새로운 규칙을 적용하는 날이 오더라도 코드를 많이 고치지 않아도 된다.

앞으로 배울 내용

이제 액션스크립트 언어의 기본적인 내용은 모두 배웠다. 액션스크립트 코드를 이해하고 직접 코드를 작성하는 데 필요한 지식을 모두 익힌 것이다. 다음 장에서는 앞에서 간단하게 짚고 넘어갔던 고급 주제와 각종 비법에 대해 살펴보자.