JavaScript 튜토리얼

HandyPost는 한 도영(HDNua)이 작성하는 포스트 문서입니다.

1. 개요

jscc 개발을 위해 JavaScript를 프로그래밍 언어로써 사용하기 위한 토대를 마련한다. 이 문서에서는 웹브라우저로 Google Chrome[1]을 사용한다.

2. 프로젝트 준비

2.1) HTML, CSS, JavaScript

HTML은 HyperText Markup Language의 약자로, 웹 문서를 만들 때 사용하는 웹 페이지 프로그래밍 언어 중의 하나다. CSS는 Cascading Style Sheets의 약자로, 웹 문서의 스타일을 설정할 때 사용하는 프로그래밍 언어로, HTML과 결합하여 HTML로 작성한 페이지의 요소에 멋진 모양을 설정한다. 여기서는 HTML5, CSS3를 기준으로 HTML과 CSS를 설명한다.

그리고 여기서 보다 집중적으로 다룰 JavaScript는 웹 페이지의 동작을 결정하기 위해 사용하는 인터프리트 프로그래밍 언어다. 3장에서 컴파일러와 인터프리터의 차이를 설명했는데, C와 같은 언어는 컴파일 언어로 프로그램을 실행하기 전에 컴파일 과정을 거쳐야 한다. 인터프리터는 컴파일 과정 없이 소스 코드를 바로 해석하는 도구라고 설명한 바 있는데, JavaScript가 바로 그러한 언어 중의 하나라는 것이다. JavaScript는 아마 당신이 배운, 다른 어떠한 언어보다 놀랍도록 쉽고 편리할 것이다. 참고로 순수하게 JavaScript를 공부하려는 사람에게도 이 문서는 적합하다.

2.2) 개발 도구 선택: Brackets

Brackets는 Adobe가 제공하는 웹 코드 에디터다. 오픈 소스고, 무료이며, HTML, CSS 및 JavaScript에 초점이 맞춰져있다. 필자가 군대에서 JavaScript를 배울 때는 Handy HTML Maker라는 독자적인 개발 도구를 만들어 프로그래밍을 했는데, Windows, Mac과 Linux까지 지원되는 이렇게 편리한 웹 에디터가 있다는 사실을 알고 나선 이전의 프로그램을 사용할 이유가 없어졌다. 따라서 앞으로 프로젝트를 진행할 때는 Brackets를 사용할 것이다. 설치 및 사용 방법에 대해서는 인터넷에 정리된 문서가 많으니 이를 참조하라.

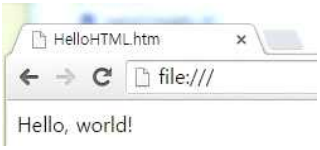
2.3) HTML 시작 예제: Hello, HTML!

우리가 언어를 배울 때 문법을 먼저 공부하지 않았듯, 앞으로의 설명은 언제나 실례를 먼저 보인 다음 문법을 설명하는 식으로 진행할 것이다. 그것이 이해도 빠르고 설명할 내용도 줄어든다. 다음은 가장 단순한 형태의 HTML 문서이다.

HelloHTML.htm

```
<html>
  <body>Hello, world!</body>
</html>
```

실행 결과



1) 글을 쓴 이 시점으로 43.0.2357.81 m 버전을 사용하고 있다.

HTML 코드에서 부등호로 둘러싸인 단어와 부등호를 포함한 문자열을 **태그(tag)**라고 한다. 이 예제에서 <body>라는 문자열을 보자. <body>와 </body>가 있고 그 사이에 실제 문서의 본문인 “Hello, world!"가 들어가 있는 것을 볼 수 있다. 이때 <body>를 **시작 태그(start tag)**, </body>을 **종료 태그(end tag)**라고 하며, 시작 태그와 종료 태그 안의 내용을 **컨텐츠(contents)**라고 한다. 시작 태그, 컨텐츠, 종료 태그를 하나로 묶어 **요소(element)**라고 한다. 즉 다음과 같다.

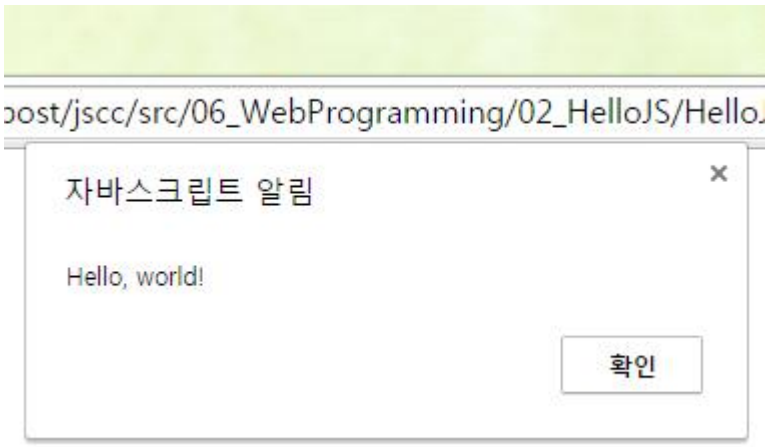
요소(element): 시작_태그(<body>) 컨텐츠(Hello, world!) 종료_태그(</body>)

HTML 문서는 요소들로 이루어진 문서다. 일단 우리는 JavaScript를 이용한 프로그래밍에 관심이 있으므로, JavaScript라는 것을 먼저 배우자. 다음은 "Hello, world!"를 경고 창으로 출력하는 코드다.

HelloJS.htm

```
<html>
  <body>
    <script>
      alert("Hello, world!");
    </script>
  </body>
</html>
```

실행 결과



방금까지는 웹브라우저에 문자열이 떴는데, 이제는 경고 창에만 뜬다. 그리고 <body> 요소 내부에 웬 <script>라는 녀석이 들어갔는데, 이 요소의 컨텐츠를 보니 우리가 자주 쓰던 프로그래밍 언어의 그것과 많이 닮아있다!

이로써 여러분은 JavaScript를 이용한 프로그래밍 세계에 첫 발을 내딛었다. C언어 책을 펴보면 나오는 수준의 "Hello, world!" 예제이므로, 잘 모르지만 alert가 경고 메시지를 출력하는 함수라는 사실은 알 것이다. 지금은 그것만 알면 된다. 이 문서에서는 jscc를 개발하기 위한 JavaScript를 가르칠 것이다.

그리고 이 문서를 보면 <html> 태그가 <body> 태그를 감싸고 있음을 알 수 있다. 이와 같은 관계를 부모-자식 관계로 본다면, <html> 태그는 <body> 태그의 부모 태그가 되고, <body> 태그는 <html> 태그의 자식 태그가 된다. <body> 태그와 <script> 태그 사이에도 같은 관계가 성립한다. <html> 태그와 <script> 태그의 경우는 물론 <html>을 부모 태그, <script>를 자식 태그라고 볼 수 있지만 이 경우는 서로 조상-후손의 관계로 본다. 즉 다음처럼 생각한다.

- 요소를 감싸는 요소에 대해 두 요소의 관계를 조상-후손 관계로 본다. 이때 요소를 직접 감싸는 요소가 있다면, 두 요소는 부모-자식 관계로 본다. 이때 자식 태그를 직계 자손이라고 한다.

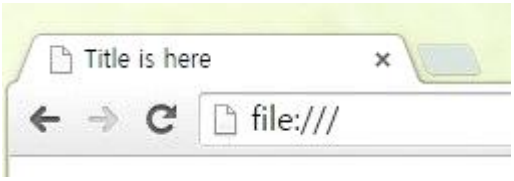
참고로 body 이외에 head라는 요소도 존재한다. 이를 이용하면 HTML 문서의 제목(파일 이름 말고)

을 설정할 수 있게 된다. 다음은 HTML 문서의 제목을 "Title is here"로 정하는 예제다.

HelloTitle.htm

```
<html>
  <head>
    <title>Title is here</title>
  </head>
</html>
```

실행 결과



<body> 태그가 사라지고 <head> 태그가 들어왔고, 또한 <head> 태그의 안에 <title>이라는 태그도 들어왔다. 그리고 실행 결과 화면을 통해 HTML 문서의 제목이 “Title is here"로 바뀌었음을 알 수 있다. 정리하면, HTML 문서의 제목을 설정할 때는 <title> 태그를 이용한다.

그리고 여기서 규칙을 정하겠다. 앞으로 굳이 설명할 필요가 없는 요소는 미리 정의되어있는 것으로 간주한다. 예를 들어 다음의 HTML 문서가 있다고 하자.

main.htm

```
<html>
  <head>
    <title>Title is here</title>
  </head>
  <body>
    <script>
      alert("Hello, world!");
    </script>
  </body>
</html>
```

이 경우 <html> 태그 내에 존재하는 요소에 대해 <html> 태그를 생략하고 다음과 같이 쓴다.

main.htm <html>

```
<head>
  <title>Title is here</title>
</head>
<body>
  <script>
    alert("Hello, world!");
  </script>
</body>
```

그리고 <html>의 직계 자손인 head와 body에 대해 다음과 같이 쓴다.

main.htm <html.head>

```
<title>Title is here</title>
```

main.htm <html.body>

```
<script>
  alert("Hello, world!");
</script>
```

이 방법에 따라 <title> 태그와 <script> 태그에 대해서도 동일하게 표현한다.

main.htm <html.head.title>

```
Title is here
```

main.htm <html.head.script>

```
alert("Hello, world!");
```

그리고 의미상 어떤 요소에 대해 작성된 코드인지가 명백한 경우 소속을 생략할 수 있다.

```
alert("Hello, world!");
```

끝으로, 이 문서에서 JavaScript를 이용하여 출력한 결과는 모두 텍스트이므로 다음과 같이 표현한다.

실행 결과

```
Hello, world!
```

그럼 이제 본격적으로 JavaScript를 배워보자. 당분간은 모든 코드를 html.body.style에 작성한다.

3. 문법

우리가 C++ 프로그래밍 언어를 알고 있으니, C++ 언어와의 차이점을 중심으로 JavaScript를 설명하겠다. 이 문서에서는 자바스크립트, JScript, JS라고 하면 모두 JavaScript를 말하는 것으로 하자.

3.1) 변수

JavaScript의 변수가 다른 언어와 크게 다른 점이라고 하면, JS에서는 변수를 선언하기 위해 자료형을 따로 지정하지 않는다는 것이다. JS의 모든 변수 선언은 var 키워드를 사용한다.

var.htm

```
var num;                    // 변수 num을 선언합니다.
num = 10;                   // 변수 num에 10을 대입합니다.
alert(num);                 // 변수 num을 출력합니다.

var str;                    // 변수 str을 선언합니다.
str = "Hello, world!";      // 변수 str에 문자열 "Hello, world!"를 대입합니다.
alert(str);                 // 변수 str을 출력합니다.
```

실행 결과
10
Hello, world!

첫 번째 창에서는 10을, 두 번째 창에서는 "Hello, world!"를 출력한다. num, str 변수 모두 var 키워드를 이용하여 선언된 변수다. 이처럼 JS에서는 var 키워드로 변수를 정의한다.

흥미로운 것은 선언한 변수에 어떠한 값이든 모두 넣을 수 있다는 것이다. 예를 들어 정수를 저장했던 변수에 문자열을 저장할 수 있다. 다음은 이에 대한 예제다.

var2.htm

```
var x = 10;                    // 변수 x를 선언하고 값을 10으로 초기화합니다.
alert(x);                      // x를 출력합니다.
x = "Hello, world!";           // x에 문자열을 대입합니다.
alert(x);                      // x를 출력합니다.
```

실행 결과는 위와 같다.

JS에서는 변수를 중복적으로 정의할 수 있다. 다음은 이에 대한 예제다.

var3.htm

```
var x = 10;                    // 변수 x를 선언하고 값을 10으로 초기화합니다.
alert(x);                      // x를 출력합니다.
var x = "Hello, world!";       // 변수 x를 선언하고 값을 문자열로 초기화합니다.
alert(x);                      // x를 출력합니다.
var x;                         // 변수 x를 선언합니다.
alert(x);                      // 변수가 재정의 되었다고 해서
                               // 이전 값이 사라지지 않습니다.
```

실행 결과
10
Hello, world!
Hello, world!

이러한 내용에만 주의하면, JS의 변수 사용은 여타 언어와 크게 다르지 않다. 한 줄에 여러 개의 변수를 선언하는 방법도, 변수를 선언하면서 초기화하는 방법도 같다.

3.2) 자료형(data type)

JavaScript의 변수는 값이 저장되는 순간 그 자료형이 결정된다. 다시 말해, JS의 변수는 내부적으로는 자료형을 저장하고 있다. JS의 자료형은 그 특성에 따라 기본 데이터 형식, 복합 데이터 형식, 특수 데이터 형식으로 나뉜다.

- 기본 데이터 형식

> Number

> String

> Boolean

- 복합 데이터 형식
> object
> Array
- 특수 데이터 형식
> Null
> Undefined

number, string과 boolean에 대해서는 이미 알고 있을 것이므로 object와 undefined에 대해 얘기해 보자. object는 JavaScript로 작성된 프로그램에 존재하는 모든 객체의 조상이다. number, string과 같은 기본 자료형을 제외하면, 우리는 C의 구조체와 같은 사용자 정의 자료형을 만들고, 만든 자료형의 인스턴스 또한 정의하고 사용할 수 있다. 이렇게 만든 모든 객체의 조상이 object가 되는 것이다. 잘 이해가 안 된다면 C++에서 int, double과 같은 기본 자료형을 빼고 모든 객체가 class object;라는 클래스를 상속하는 것처럼 이해하면 된다. 이에 대해서는 후에 다루겠다.

typeof 연산자를 이용하여 변수가 어떤 자료형을 가지고 있는지 확인할 수 있다. 다음을 보자.

datatype.htm

```
alert(typeof(1234));          // 수치 값 1234: number 출력
alert(typeof("hello"));       // 문자열 "hello": string 출력
alert(typeof(10==20));        // 부울 값 (10==20): boolean 출력
alert(typeof(undefined));     // undefined: undefined 출력
alert(typeof(null));          // null: object 출력
```

null의 경우 특수 데이터 형식이지만 typeof의 결과는 object를 반환함에 주의하라. 이와 같이 JS의 기본적인 자료형에 대해 알 수 있었다.

3.3) 문자열

다음은 JavaScript에서 기본 문자열을 정의하는 예제다.

string.htm

```
var s = "hello"; // 큰따옴표로 문자열을 정의할 수 있습니다.
s = 'world'; // 작은따옴표로도 문자열을 정의할 수 있습니다.
s = "Nice to 'meet' you!"; // 큰따옴표 문자열 안에 작은따옴표를 넣을 수 있습니다.
s = 'See "you" again!'; // 작은따옴표 문자열 안에 큰따옴표를 넣을 수 있습니다.
```

+ 연산자, += 연산자를 이용하여 문자열 두 개를 합칠 수 있다.

string_add.htm

```
var s = "Hello, " + 'world!'; // "Hello, world!" 출력
s += " Bye!"; // "Hello, world! Bye!" 출력
```

문자열의 길이를 얻으려면 다음과 같이 문자열 객체의 멤버 length를 이용한다.

string_len.htm

```
var str = 'hello';
alert(str.length); // 'hello'의 길이 5가 출력됩니다.
```

이와 같이 문자열의 기본적인 사용법을 알 수 있었다.

3.4) 사용자 입력을 받는 방법과 숫자-문자열 간 변환

사용자로부터 입력을 받으려면 prompt 함수를 이용한다. 다음은 이에 대한 예제이다.

prompt.htm

```
// prompt 함수를 호출하고 사용자가 입력한 값을 str에 저장합니다.
var str = prompt("Enter name", "Handy");
// 첫 번째 인자는 사용자에게 표시할 설명입니다.
// 두 번째 인자는 사용자가 입력할 값의 초기값입니다.
// prompt 함수는 사용자가 입력한 값을 문자열로 반환합니다.
alert("Hello, " + str + "!");
```

prompt 함수는 주석에도 설명했듯, 사용자가 입력한 값을 문자열로 반환함에 주의하라.

이전 절에서 문자열 두 개를 더할 때 + 연산자를 사용한다고 했다. 문자열과 숫자에 대해 + 연산자를 사용하면 숫자가 문자열 형식으로 변환된 다음 두 문자열을 더한 결과가 나타난다.

string_add2.htm

```
alert("Hello, " + 10 + "!"); // "Hello, 10!"이 출력됩니다.
```

그런데 prompt를 이용해 두 정수를 더하는 프로그램을 만들어보자.

string_noparse.htm

```
var num1 = prompt("Enter number 1: ", 10);
var num2 = prompt("Enter number 2: ", 20);
alert(num1 + num2); // num1=10, num2=20이면 출력 값은?
```

우리는 사용자가 입력한 두 값을 더한 결과가 30이기를 기대하고 있다. 그럼 실행 결과를 보자.

실행 결과

```
1020
```

결과는 아주 엉뚱하게도 1020이라는 수가 출력되었다! 무엇이 잘못되었는지 알겠는가?

방금 prompt가 문자열을 반환하는 함수라고 설명하였다. 따라서 num1과 num2에 저장된 값은 모두 수치 값이 아닌 문자열 값이다. 그리고 문자열 두 개에 + 연산자를 더하면 두 문자열이 합쳐진 결과가 나온다고도 설명하였다! 즉 1020이라는 수는 실제로 수가 아니라, 문자열 “10”과 문자열 “20”이 합쳐진 결과로 해석해야 한다.

그러므로 이를 계산하기 위해서는 저장된 문자열을 먼저 수치 형식으로 변환해야 한다. 어떻게 해야 할까? C에서 char 형식의 배열에 어떤 정수 문자열이 저장되어있고, 여기에 20을 더해 결과를 얻으려고 한다면, 우리는 atoi 함수를 호출하여 배열에 저장된 문자열을 정수로 먼저 변환한 다음 여기에 20을 더해 결과를 출력해야 한다. JS에서도 같은 일이 일어난다. 문자열을 저장하고 있는 변수를 정수로 변환한 값을 얻으려면 parseInt(), 실수로 변환한 값을 얻으려면 parseFloat()과 같은 함수를 사용해야 한다. 위 문제를 parseInt() 함수를 이용하여 수정한 코드는 다음과 같다.

string_parse.htm

```
var str1 = prompt("Enter number 1: ", 10);
var str2 = prompt("Enter number 2: ", 20);
var num1 = parseInt(str1);
var num2 = parseInt(str2);
alert(num1 + num2);
```

수를 문자열로 바꾸는 방법은 아주 쉬운데, 다음과 같이 빈 문자열을 더하면 된다.

numberToString.htm

```
var num1 = 10, num2 = 20;
alert("" + num1 + num2);
```

아니면 toString()이라는 메서드를 호출하는 방법도 있는데 다음과 같다.

```
var num1 = 10, num2 = 20;
alert(num1.toString() + num2.toString());
```

이와 같이 prompt 함수 및 숫자-문자열 간 변환 방법을 알 수 있었다.

3.5) 프로그램 흐름 제어

JS의 흐름 제어는 C/C++ 언어의 그것과 거의 같다. 예제를 살펴보자.

flow_control.htm

```
var foo = 1, bar = 2;

// 조건문 예제: if
if (foo == 1) {
  if (bar == 2)
    alert("num1 is 1 and num2 is 2");
  else
    alert("num1 is not 1 or num2 is not 2");
}

// 조건문 예제: switch 1
switch (foo) {
  case 0:
    alert("num1 is 0");
    break;
  case 1:
  case 2:
    alert("num1 is 1 or 2");
    break;
  default:
    alert("num1 is not 0, 1 and 2");
```

```
    break;
}

// 조건문 예제: switch 2
foo = 'hello';
switch (foo) { // switch 구문의 대상으로 문자열이 들어갈 수 있습니다.
  case 'hi': alert(0); break;
  case 'hello': alert(1); break;
  case 'good day': alert(2); break;
  default: alert(3); break;
}

// 반복문 예제: for
// 1부터 3까지 출력하는 예제입니다.
for (var i=0; i<10; ++i) {
  if (i > 3)
    break;
  alert("Hello, " + (i+1) + "!");
}



// 반복문 예제: while
// 5 이하의 짝수를 출력합니다.
foo = 5;
while (foo > 0) { // 양수인 foo에 대해
  if (foo % 2 == 1) { // foo가 홀수라면
    --foo;
    continue; // 그냥 지나갑니다.
  }

  alert(foo); // 지나가지 않은 수를 출력합니다.
  --foo; // foo를 하나 감소시킵니다.
}

// 반복문 예제: do-while
// 5 이하의 홀수를 출력합니다.
foo = 5;
do {
  if (foo % 2 == 0) {
    --foo;
    continue;
  }

  alert(foo); // 지나가지 않은 수를 출력합니다.
```

```
  --foo; // foo를 하나 감소시킵니다.
} while (foo > 0);
```

예제 코드를 보는 것만으로 JS의 조건문과 반복문을 이해할 수 있을 것이다. 참고로 경고 창이 많이 뜨는 데 불편함을 느끼고 있다고 해도 걱정하지 말자. 이 문제는 잠시 후에 개선할 것이다.

3.6) 배열

배열의 필요성에 대해서는 굳이 말하지 않겠다. JS에서도 배열을 지원한다. 그런데 JS의 배열은 기존에 우리가 알던 배열과는 선언과 활용, 내부 구조가 많이 다르다. 집중해서 살펴보자.

JS의 배열은 대괄호('[', ']')를 이용하여 선언한다.

array.htm

```
var arr = [ 1, 2, 3, 4, 5 ]; // 배열 arr을 선언합니다.
for (var i=0; i<5; ++i) {
  alert(arr[i]); // 배열의 원소를 출력합니다.
}

// 배열에 값을 대입합니다.
arr[0] = 10;
for (var i=1; i<4; ++i) {
  arr[i] = 10 + i;
}
arr[4] = "End of array"; // 문자열도 대입할 수 있습니다.

for (var i=0; i<5; ++i) {
  alert(arr[i]); // 배열의 원소를 출력합니다.
}
```

혹시 문자열의 길이를 얻을 때 length라는 멤버를 이용했던 것을 기억하는가? 배열에도 같은 멤버가 존재한다. 즉 배열의 원소의 개수를 얻으려면 length 멤버를 사용한다.

array_len.htm

```
var arr = [ 1, 2, 3, 4, 5 ]; // 배열 arr을 선언합니다.
for (var i=0; i<arr.length; ++i) {
  alert(arr[i]); // 배열의 원소를 출력합니다.
}
```

배열의 멤버를 참조할 때 for-in이라는 새로운 반복문 키워드를 이용할 수 있다.

array_forin.htm

```
var arr = [ 1, 2, 3, 4, 5 ]; // 배열 arr을 선언합니다.
for (index in arr) { // 배열의 각 인덱스에 대해
  alert(arr[index]); // 인덱스를 이용하여 배열의 원소를 출력합니다.
}
```

이와 같이 배열을 사용하는 기본적인 예제를 살펴볼 수 있었다.

3.7) 함수

JS에서 함수를 정의하고 사용할 수 있다. 방법은 다음과 같다.

function.htm

```
// hello 함수를 정의합니다.
function hello() {
  alert("hello, world!");
}

// hello 함수를 호출합니다.
hello();
```

함수의 정의가 호출과 같은 영역에 있는 것을 이상하게 생각할 수 있으나, 이는 적법한 코드다. JS에서는 함수 내에 함수를 정의하고 호출하는 것이 가능하다. 놀라운 점은, 함수를 호출하기 전에 선언이나 정의가 반드시 앞에 있어야 할 필요가 없다는 것이다! 이에 대한 예제를 보이겠다.

function2.htm

```
// 1을 반환하는 one 함수를 정의합니다.
function one() { return 1; }

// num1과 num2를 더한 값을 출력합니다.
var num1 = one();
var num2 = two();
alert(num1 + num2);

// 2를 반환하는 two 함수를 정의합니다.
function two() { return 2; }
```

함수가 사용할 인자를 함수 정의에 넣을 수 있다.

function_param.htm

```
// a + b의 결과를 반환하는 sum 함수를 정의합니다.
function sum(a, b) { // JS의 변수가 모두 var 형식이므로 형식을 따로 정의하지 않습니다.
  return a + b;
}

alert(sum(1, 2)); // 두 정수를 + 연산자로 연산한 결과를 출력합니다.
alert(sum('hello, ', 'world!')); // 두 문자열을 + 연산자로 연산한 결과를 출력합니다.
alert(sum('hello', 1));
```

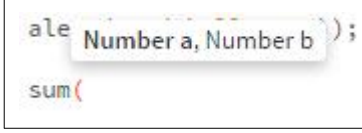
참고로 JS의 함수 정의에 인자의 형식을 넣을 수 없어 불편한 경우가 종종 있다. 다른 사람이 작성한 코드를 보는데 정수를 넘겨야 할지 문자열을 넘겨야 할지를 고민해야 하는 상황을 말하는데, 만약 형식을 표시하고 싶다면 정의한 함수 앞에 다음과 같은 형식으로 주석을 써주면 된다.[2)]

2) http://blog.brackets.io/2013/04/05/typeaware-javascript-code-intelligence/

function_codehint.htm

```
/** 주석을 만들 때 '*' 문자 2개를 써서 주석을 만듭니다.

- 함수 설명을 여기에 기록합니다.

- @param {<type>} <param_name>의 형식으로 기록합니다.
@param {number} a
@param {number} b
@return {number}

*/
function sum(a, b) {
  return a + b;
}
```

결과



이와 같이 코드 힌트가 잘 나타나는 것을 볼 수 있다.
JS에서는 함수도 일종의 객체로 본다. 그래서 JS에서는 선언된 변수에 함수를 대입할 수 있다.

function_delegate.htm

```
function sum(a, b) { return a + b; }
function mul(a, b) { return a * b; }

var f = sum; // 변수 f에 sum을 대입합니다.
alert(f(3, 5)); // 변수 f가 가리키는 함수를 호출합니다.
f = mul; // 변수 f에 mul을 대입합니다.
alert(f(3, 5)); // 변수 f가 가리키는 함수를 호출합니다.
```

실행 결과

```
8
15
```

함수를 만들면서 변수에 대입하는 놀라운 방법도 있다.

function3.htm

```
var f = function sum(a, b) { return a + b; }; // 변수 f에 sum을 대입합니다.
alert(f(3, 5)); // 변수 f가 가리키는 함수를 호출합니다.
f = function mul(a, b) { return a * b; }; // 변수 f에 mul을 대입합니다.
alert(f(3, 5)); // 변수 f가 가리키는 함수를 호출합니다.
```

그리고 이름 없이 함수를 정의하는 무명 함수(anonymous function)라는 문법도 있다.

function3.htm

```
var f = function (a, b) { return a + b; }; // 변수 f에 sum을 대입합니다.
alert(f(3, 5)); // 변수 f가 가리키는 함수를 호출합니다.
f = function (a, b) { return a * b; }; // 변수 f에 mul을 대입합니다.
alert(f(3, 5)); // 변수 f가 가리키는 함수를 호출합니다.
```

마지막으로, 함숫값을 반환하지 않는 함수의 반환 값은 undefined다.

function_noretval.htm

```
function hello() {
  var num = 1 + 2; // 값을 반환하지 않는 함수를 정의합니다.
}
alert(hello()); // hello() 함수를 호출하고 그 결과를 출력합니다.
```

이와 같이 JS의 함수에 대해 간단히 알아볼 수 있었다.

3.8) HTML 문서의 요소 다루기

3.5절의 예제를 테스트해보았다면 이미 느꼈겠지만, 경고 창이 여러 개 뜨는 데 이게 성가신 일이 아닐 수 없다. 여기서는 이 문제를 개선해보자.

HTML은 요소로 이루어진 문서라고 설명한 바 있다. HTML의 유용한 요소로 textarea 요소가 있다. 다음은 textarea 요소를 HTML 문서에 추가한 코드다.

textarea.htm

```
<html>
  <body>
    <textarea>hello, world!</textarea>
  </body>
</html>
```

실행 결과

textarea 요소는 이름 그대로 텍스트를 입력하는 공간이다. 태그 사이에 “hello, world!” 문자열을 입력했더니 요소 안에 내용이 들어갔다. 앞으로 우리가 출력을 확인할 때는 바로 이 textarea 요소의 내용에 문자열을 추가하는 식으로 출력 결과를 확인할 것이다. 그러면 더 이상 경고 창을 보지 않아도 되므로 프로그래밍하기 한결 수월해질 것이다.

textarea 요소의 내용을 변경하려면, 먼저 접근해야 한다. 어떻게 textarea 요소에 접근할까? textarea 요소에 이름을 붙이고, HTML 문서에서 해당 이름을 가진 요소를 불러오는 함수를 호출하면 되지 않을

까? 이렇게 생각했다면 정답이다. 그러면 먼저 textarea 요소에 이름을 붙이는 방법을 알아보자.

textarea와 같은 요소는 내부적으로 값을 여러 개 갖는다. 요소의 너비, 요소의 높이, 요소의 값과 같은 식으로 말이다. 요소에 대한 이러한 내부적인 값을 요소의 **속성(property)**이라고 한다. 그러므로 우리는 textarea 요소의 이름이 되는 속성을 우리가 원하는 값으로 설정하면 된다. 방법은 다음과 같다.

textarea_id.htm

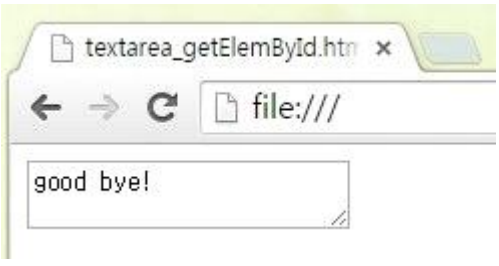
```
<html>
  <body>
    <textarea id='mytextarea'>hello, world!</textarea>
  </body>
</html>
```

textarea에 id='mytextarea'라는 구문이 추가되었다. 이는 id 속성을 'mytextarea'로 설정하겠다는 뜻이다. 요소에 대해 id는 유일하며, 우리는 id 요소의 값을 통해 요소를 획득할 수 있다. 그럼 이제 요소를 직접 얻어보자. 이는 다음의 코드에서 구현한다.

textarea_getElemById.htm

```
<html>
  <body>
    <textarea id='mytextarea'></textarea>
    <script>
      // document.getElementById 함수를 호출합니다.
      var myElem = document.getElementById('mytextarea');
      // value 속성을 good bye!로 변경합니다.
      myElem.value = 'good bye!';
    </script>
  </body>
</html>
```

실행 결과



document라는 녀석에 점(.)을 이용하여 멤버 함수를 호출하는 것처럼 getElementById라는 함수를 호출했다. document는 HTML 문서에 기본적으로 정의된 객체라고 생각하면 된다. 이렇게 기본적으로 정의된 객체는 document 외에도 window와 같이 여러 가지가 있는데, 이에 대해서는 나중에 다루겠다.

getElementById 메서드를 호출하여 요소를 획득하고 myElem에 저장한다. 그러면 myElem은 body 영역에 정의한 textarea 요소를 가리키는 상태가 된다. 이때 myElem의 속성인 value의 값을 변경하면 우리가 원하는 대로 문장이 변경되는 것을 볼 수 있다! 이렇게 하면 myElem의 value 속성에 문자열을 대입하는 것으로 textarea가 출력 창의 역할을 대신하게 할 수 있다.

textarea로 출력 창을 돌렸지만 매번 myElem을 얻어서 value를 설정하려니 불편하고, textarea 요소

도 크기가 너무 작다. 좀 더 쓸 만하게 이걸 수정해보자. 일단 답답하니 크기부터 늘이자. 요소의 내부 값을 변경하는 것이니 이것도 속성을 변경해야 한다. 요소의 모양을 설정할 때는 style이라는 속성의 값을 설정하면 된다.

textarea_style.htm

```
<html>
  <body>
    <textarea id='mytextarea'></textarea>
    <script>

      // mytextarea 요소를 가져옵니다.
      var myElem = document.getElementById('mytextarea');

      // 얻은 요소의 style 속성의 크기를 변경합니다.
      myElem.style.width = 800;
      myElem.style.height = 200;

    </script>
  </body>
</html>
```

실행 결과

이와 같이 요소의 크기를 변경할 수 있다. 참고로 화면에 꽉 차게 만들려면 width에 '100%', height에도 '100%' 값을 넣어주면 된다.

그럼 이제 여기에 문자열을 넣는 것을 생각하자. 물론 myElem.value 속성에 직접 접근하여 값을 수정하는 것도 방법이지만, 여기서는 Log라는 함수를 만들어서 이 함수가 그 일을 하도록 하겠다. 다시 말해 Log("Hello, world!")와 같이 호출하면 출력이 나온다.

다음은 이를 구현하는 코드이다.

textarea_log.htm

```
<html>
  <body>
    <textarea id='mytextarea'></textarea>
```

```
    <script>

      var myElem = document.getElementById('mytextarea');
      myElem.style.width = 800;
      myElem.style.height = 200;

      // 화면에 s가 가리키는 문자열을 출력하는 Log 함수를 정의합니다.
      function Log(s) {
        var myElem = document.getElementById('mytextarea');
        myElem.value += (s + '\n'); // 자동으로 개행이 되게 만듭니다.
      }

      Log("Hello, world!");
      Log("Nice to meet you!");

    </script>
  </body>
</html>
```

그리고 앞으로는 구현의 편의를 위해 기본적으로 HTML 문서에 다음이 작성되어있다고 가정한다.

main.htm

```
<html>
  <head>
    <script>

      /**
      프로그램의 주 진입점입니다.
      */
      function main() {
        Log("Hello, world!");
      }

      /**
      s가 가리키는 문자열을 HandyLogStream 요소에 출력합니다.
      @param {String} s
      */
      function Log(s) {
        var logObject = document.getElementById('HandyLogStream');
        logObject.value += (s + '\n');
      }

      /**
      main 메서드를 호출하기 전에 환경을 준비합니다.
      */
```

```
      function init() {
        var logObject = document.getElementById('HandyLogStream');
        logObject.style.width = 800;
        logObject.style.height = 200;
      }

    </script>
  </head>
  <body>
    <textarea id='HandyLogStream'></textarea>
    <script>
      init();
      main();
    </script>
  </body>
</html>
```

title을 넣을 때 사용하던 head 요소가 추가되었고, 그 내부에 script 요소가 새롭게 생겼다. 그리고 이 영역에 main, Log, init이라는 메서드 세 개가 정의되었고, body에는 textarea 요소를 정의한 다음 head에 정의한 init, main을 차례대로 호출하고 있다. 이 프로그램은 처음 보기엔 당황스럽지만 이전에 배운 지식을 토대로 뜯어보면 전혀 어려운 코드가 아니다. 앞으로는 main 메서드의 내부를 수정하여 출력을 textarea 요소에 뿌리는 식으로 출력을 확인할 것이다. 이와 같이 HTML 문서의 중요한 요소 중 하나인 textarea를 다루는 방법을 알아보고, 프로젝트를 보다 편리하게 개선하였다.

앞으로 학습을 진행할 때는 이 코드를 복사하고 붙여 넣은 다음 head 요소의 script를 수정하는 식으로 진행하겠다. 따라서 이 문서에서 코드 영역을 설명할 때도 기본적으로 <html.head.script> 내부에 코드를 작성하는 것으로 이해하면 된다.

3.9) 객체

JavaScript의 모든 것은 객체이다. 3.2절에서 자료형을 설명할 때 수치형, 문자열, Boolean형을 기본 자료형이라고 소개했는데, 사실 이들 또한 객체다. 이들은 변경 불가능한 객체(immutable object)라고 한다.

JavaScript는 클래스 개념은 없다. JavaScript는 객체 지향 프로그래밍 언어이지만, 클래스 기반 언어가 아닌 프로토타입 기반 프로그래밍 언어다. **프로토타입 기반 프로그래밍(Prototype-based Programming)**이란 클래스 기반 언어에서 상속을 사용하는 것과 다르게, 객체를 원형으로 객체의 동작 방식을 복제하여 재사용하는 프로그래밍 방식을 말한다. 즉 클래스 기반 언어가 객체의 인스턴스를 만들 때 클래스를 참조한다면, 프로토타입 기반 언어는 객체를 참조한다.

JS에는 기본으로 정의된 내장 객체(built-in object)가 있고, 사용자가 만들어서 작성하는 사용자 정의 객체(custom object)가 있다. 여기서는 사용자 정의 객체를 먼저 알려주고 실습을 해본 다음 내장 객체의 종류를 알아보는 식으로 진행하는 것이 낫겠다. 참고로 객체 지향 프로그래밍에서 객체의 멤버 변수를 **필드(field)**, 멤버 함수를 **메서드(method)**라고 한다. 그리고 앞서 요소에 대한 내부적인 값을 속성이라고 했는데, 객체의 필드 또한 속성이라고 부른다. 앞으로 이 문서에서 JS의 객체의 멤버 변수와 멤버 함수는 모두 속성과 메서드라고 부를 것이므로 이 표현에 익숙해졌으면 한다.

3.9.1) 사용자 정의 객체

그러면 바로 사용자 정의 객체를 생성해보자. 사람을 의미하는 Person을 정의하고 필드로 이름과 나

이가 있다고 하자. JS에서는 객체를 만들기 위해 흥미롭게도 함수를 정의한다. 이때 객체를 만들기 위해 정의하는 함수를 **생성자(constructor)**라고 한다. 그러면 생성자를 만들고 이를 이용해 객체도 만들어 보자.

constructor.htm

```
// Person 객체를 만들기 위한 생성자를 정의합니다.
function Person(name, age) {
  // this 키워드를 이용하여 필드를 초기화합니다.
  this.myName = name;
  this.myAge = age;
}

function main() {
  // new 키워드와 생성자를 이용하여 Person 객체를 생성합니다.
  var person = new Person("Handy", 20);

  // person 객체의 필드를 가져와 출력합니다.
  Log(person.myName);
  Log(person.myAge);
}
```

실행 결과

```
Handy
20
```

new 키워드를 이용하여 객체를 생성하고 객체의 멤버에 접근하는 단순한 예제다. Java나 C#과 같은 고급 언어를 공부한 사람은 익숙한 방식이지만, C++만 공부한 사람은 new로 생성한 객체의 멤버에 접근할 때 화살표 연산자(->)가 아닌 점 연산자(.)를 쓰는 것을 의아하게 느낄 수 있다. JS의 객체의 멤버에 접근할 때는 점 연산자만 사용한다. JavaScript에서 생성자는 단지 new 연산자를 사용할 때 호출되는 함수이다.

생성자에 속성을 추가하는 방법은 알았지만 멤버 함수, 즉 메서드를 추가하는 방법은 아직 배우지 않았다. 그런데 이전 절에서 변수가 함수를 가리킬 수 있다고 설명한 것을 기억하는가? 이를 기억한다면 그냥 다음과 같이 쓰는 것만으로 메서드를 생성자에 추가하는 것이 가능함을 알 수 있다.

constructor_method.htm

```
function Person(name, age) {
  this.myName = name;
  this.myAge = age;

  // 자신의 정보를 문자열로 반환하는 myInfo 메서드를 정의합니다.
  this.myInfo = function() {
    // return 'Person: <name>, <age>'
    return 'Person: ' + this.myName + ', ' + this.myAge;
  };
```

```
}

function main() {
  // new 키워드와 생성자를 이용하여 Person 객체를 생성합니다.
  var person = new Person("Handy", 20);

  // person 객체의 myInfo 메서드를 호출하고 값을 infoString에 저장합니다.
  var infoString = person.myInfo();
  Log(infoString);
}
```

객체는 사용자가 정의한 속성이나 메서드 외에도 기본적으로 속성과 메서드를 가진다. 문자열의 길이를 얻기 위해 length 속성을 얻은 것이 기억나는가? 그와 같은 식이다. 기본으로 제공되는 메서드 중의 하나인 toString에 대해 살펴보자.

toString.htm

```
function Person(name, age) {
  this.myName = name;
  this.myAge = age;
  this.myInfo = function() {
    return 'Person: ' + this.myName + ', ' + this.myAge;
  };

  // 자신의 정보를 문자열로 반환하는 기본 메서드 toString을 재정의합니다.
  this.toString = function() { return this.myInfo(); }
}

function main() {
  // new 키워드와 생성자를 이용하여 Person 객체를 생성합니다.
  var person = new Person("Handy", 20);

  // person 객체를 Log 함수의 인자로 넘겨 출력합니다.
  Log(person);
}
```

실행 결과

```
Person: Handy, 20
```

Person 객체를 생성하고 person 객체에 대입했다. 그리고 Log 함수를 호출할 때 person을 문자열로 변환하는 함수를 별도로 호출하지 않았는데도 실행 결과가 잘 나온다. 어떻게 된 일일까?

JS에서는 어떤 객체를 문자열로 변환해야 하는 일이 생기면 내부적으로 toString 메서드를 호출한다. 이 경우 toString 메서드를 myInfo 메서드가 반환하는 값을 다시 반환하도록 정의했기 때문에 올바른 출력이 나온 것이다. 만약 'this.toString = ...'의 정의를 주석 처리하고 프로그램을 실행하면 다음 결과를 얻는다.

실행 결과

```
[object Object]
```

이전에 JS의 모든 객체의 조상이 object라고 설명한 바 있다. 이 결과는 toString을 재정의 하지 않았기 때문에 그 조상인 object에 정의된 기본 메서드 toString이 호출된 결과로 봐야 한다.

생성자를 이용하지 않고 바로 객체를 생성할 수도 있다.

singleton.htm

```
function main() {
  // 생성자를 이용하지 않고 객체를 생성합니다. 중괄호('{', '}')를 이용합니다.
  var person = {
    myName: 'HDNua',
    myAge: 24,
    toString: function() {
      return 'Person: ' + this.myName + ', ' + this.myAge;
    }
  };

  // person 객체를 Log 함수의 인자로 넘겨 출력합니다.
  Log(person);
}
```

이렇게 만든 객체는 프로그램에 단 하나만 생성되며, 이러한 객체를 **싱글톤(singleton)**이라고 한다.

JS에서는 기존 객체에 멤버를 추가할 수 있다. JS의 모든 객체는 확장 가능하다. 이는 C++과 같은 언어가 객체를 확장할 때 클래스부터 수정해나가야 한다는 점과 크게 대조적이다. JS의 객체는 그 자체로 하나의 딕셔너리 자료구조로 보는 것이 바람직하다. 실제로 빈 객체를 딕셔너리처럼 사용하는 방식을 jscc에서 사용하는데, 이는 다른 문서에서 보이겠다. 다음은 기존 객체에 멤버를 추가하는 코드다.

object_extension.htm

```
function main() {
  // 객체를 생성합니다.
  var person = {
    myName: 'HDNua',
    myAge: 24,
    toString: function() {
      return 'Person: ' + this.myName + ', ' + this.myAge;
    }
  };

  // 기존 객체에 속성을 추가합니다.
  person.myHeight = 100;

  // 기존 객체에 정의된 toString 메서드를 재정의 합니다.
  person.toString = function() {
```

```
    // return 'Person: <name>, <age>, <height>'
    return 'Person: ' + this.myName + ', ' + this.myAge + ', ' + this.myHeight;
  };

  // person 객체를 Log 함수의 인자로 넘겨 출력합니다.
  Log(person);
}
```

실행 결과

```
Person: HDNua, 24, 100
```

그런데 아주 재미있는 사실은, 객체의 멤버에 접근할 때 문자열을 사용할 수 있다는 것이다.

member_access_string.htm

```
function main() {
  // 객체를 생성합니다.
  var person = { myName: 'HDNua', myAge: 24 };

  // 문자열을 사용하여 객체의 멤버에 접근하고 값을 설정할 수 있다.
  var name = person['myName'];
  Log(name);
  person['myAge'] = 28;
  Log(person['myAge']);
}
```

실행 결과

```
HDNua
28
```

그래서 JS에서는 위와 같이 간단한 방법으로 객체의 멤버에 접근하는 것이 가능하다.

JS의 객체는 딕셔너리와 같다고 했다. 따라서 멤버의 이름을 **키(key)**, 멤버의 값을 **값(value)**이라고 할 수 있다. 객체의 키를 얻을 때 for-in 루프를 활용할 수 있다.

member_access_forin.htm

```
function main() {
  // 객체를 생성합니다.
  var person = { myName: 'HDNua', myAge: 24 };

  // 문자열을 사용하여 객체의 멤버에 접근하고 값을 설정할 수 있다.
  for (key in person) {
    var keyStr = 'Key: ' + key;
    var valStr = 'Value: ' + person[key];
```

```
    // 'Key: <key>, Value: <value>'
    Log(keyStr + ', ' + valStr);
  }
}
```

실행 결과

```
Key: myName, Value: HDNua
Key: myAge, Value: 24
```

C# 프로그래밍 언어를 배우다 온 사람이면 in의 왼편에 들어가는 것이 key라는 사실 때문에 많이 헛갈릴 텐데 이에 주의하기 바란다.

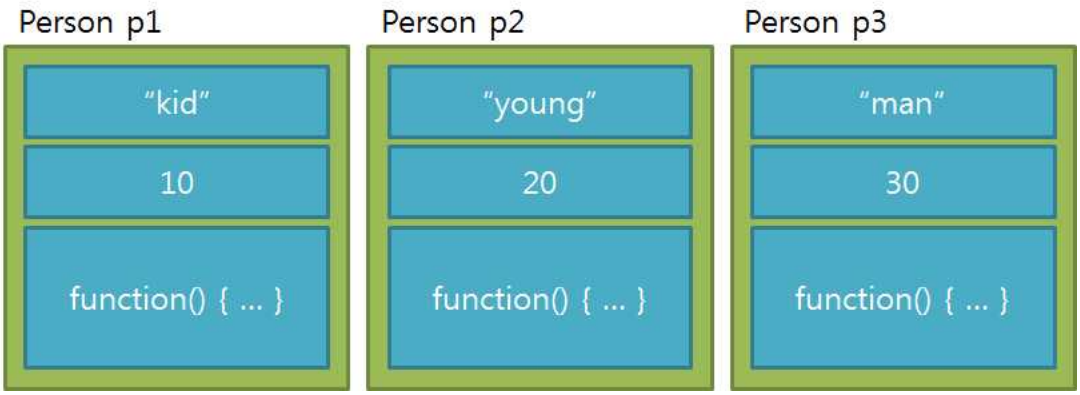
3.9.2) 프로토타입

다음은 Person 객체를 3개 정의하고 출력하는 예제다.

no_prototype.htm

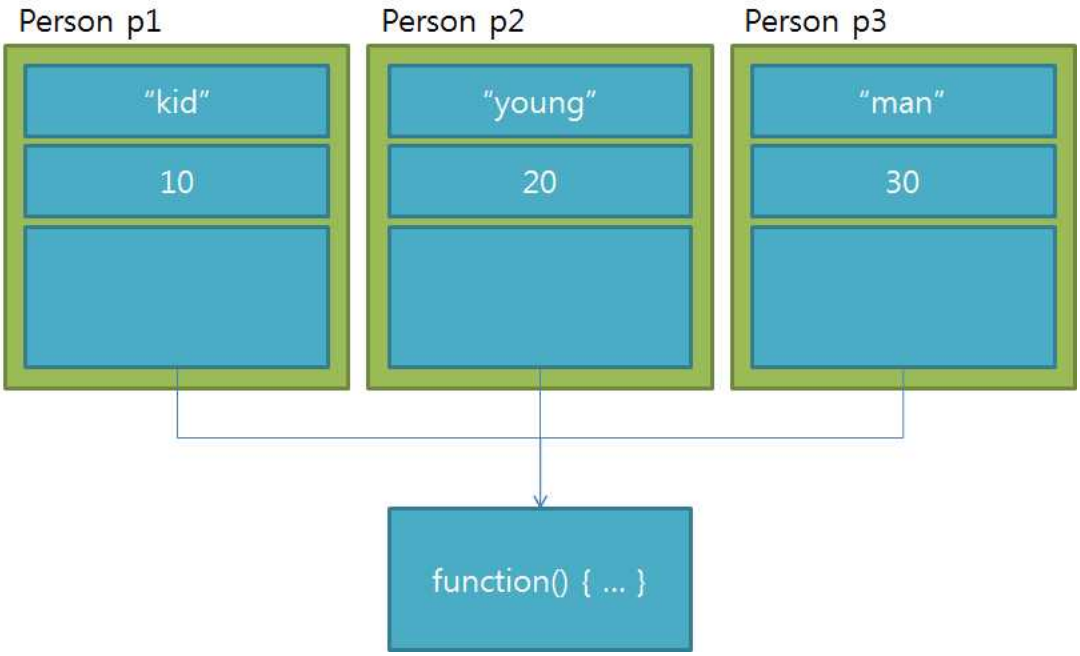
```
function Person(name, age) {
  this.myName = name;
  this.myAge = age;
  this.toString = function() {
    return 'Person: ' + this.myName + ', ' + this.myAge;
  };
}

function main() {
  var p1 = new Person("kid", 10);
  var p2 = new Person("young", 20);
  var p3 = new Person("man", 30);

  Log(p1.toString());
  Log(p2.toString());
  Log(p3.toString());
}
```

실행하는 데에는 전혀 문제가 없는 단순한 예제다. 그러나 이 경우 Person 생성자 안에 toString 메서드를 정의하는 부분은 문제가 될 수 있다. Person 객체가 생성될 때마다, 즉 Person 생성자가 호출될 때마다 각 객체는 자신만의 toString 객체를 별도로 가지게 된다. 무명 함수는 함수 객체를 생성해서 반환하기 때문이다. 그림으로 보면 지금 main 내부는 다음 상태와 같다.



그런데 생각해보자. toString 메서드는 자신의 정보를 출력하는 메서드다. 객체가 가지고 있는 속성의 값이 달라질지언정, 객체가 달라진다고 출력하는 논리가 달라지지는 않는다. 따라서 같은 내용의 함수를 객체마다 가지고 있는 건 사실 불필요한 것이다. 후에 객체가 수십, 수백 개로 늘어나게 된다면 필요 없는 부분에 그만큼 메모리를 사용하게 되므로, 이는 개선해야만 하는 문제가 된다.

JavaScript의 모든 객체는 prototype이라는 숨겨진 객체를 가지고 있는데, 이 객체를 이용해 멤버를 공유할 수 있다. 이를 이용해 toString 메서드가 하나의 메서드를 가리키게 만든다면 위에서 보인 문제는 다음과 같이 해결이 된다.



그럼 바로 적용해보자.

prototype.htm

```
function Person(name, age) {
  this.myName = name;
  this.myAge = age;
}
// prototype의 메서드로 toString을 정의합니다.
Person.prototype.toString = function() {
  return 'Person: ' + this.myName + ', ' + this.myAge;
};
```

```
function main() {
  var p1 = new Person("kid", 10);
  var p2 = new Person("young", 20);
  var p3 = new Person("man", 30);

  Log(p1.toString());
  Log(p2.toString());
  Log(p3.toString());
}
```

이와 같이 프로토타입의 필요성을 이해하고 사용 방법을 알 수 있었다.

3.9.3) 프로토타입 체인

JavaScript에서 속성이나 메서드를 참조하게 되면, 먼저 자신 안에 멤버가 정의되어있는지 찾아본 다음, 발견하지 못하면 그 프로토타입으로 이동하여 해당 프로토타입 객체 내에서 멤버를 찾는다. 이는 멤버를 찾거나, 멤버를 찾지 못하고 null을 반환하고서야 비로소 끝나는데, 이러한 객체들의 연쇄를 가리켜 **프로토타입 체인(prototype chain)**이라고 한다.

프로토타입 객체는 생성한 각각의 객체에서부터 최상위 객체인 Object의 프로토타입까지 연결되어있다. 다음의 예제를 살펴보자.

prototype_chain.htm

```
function Person(name, age, greeting) {
  this.myName = name;
  this.myAge = age;

  // 사람마다 인사하는 방법이 다르다고 하자.
  this.greeting = greeting;
}
// 사람을 표현하는 방법은 모두 같다.
Person.prototype.myInfo = function() {
  return 'Person: ' + this.myName + ', ' + this.myAge;
};

function main() {
  var person = new Person("kid", 10, function() { Log("I'm a boy!"); });

  // 1.1) person 객체의 greeting 멤버를 찾습니다.
  // 1.2) greeting 멤버가 있으므로 이를 호출합니다.
  person.greeting();

  // 2.1) person 객체에서 myInfo 멤버를 찾습니다.
  // 2.2) person.prototype 객체로 이동하여 myInfo 멤버를 찾습니다.
  // 2.3) myInfo 멤버가 있으므로 이를 호출합니다.
  Log(person.myInfo());
```

```
  // 3.1) person 객체에서 toString 멤버를 찾습니다.
  // 3.2) person.prototype 객체로 이동하여 toString 멤버를 찾습니다.
  // 3.3) Object.prototype 객체로 이동하여 toString 멤버를 찾습니다.
  // 3.4) toString 멤버가 있으므로 이를 호출합니다.
  Log(person.toString());

  // 3.1) person 객체에서 wrong 멤버를 찾습니다.
  // 3.2) person.prototype 객체로 이동하여 wrong 멤버를 찾습니다.
  // 3.3) Object.prototype 객체로 이동하여 wrong 멤버를 찾습니다.
  // 3.4) wrong 멤버가 없으므로 undefined를 반환합니다.
  Log(person.wrong);
}
```

실행 결과

```
I'm a boy!
Person: kid, 10
[object Object]
undefined
```

실행 결과를 참고하여 주석을 읽어보면 프로토타입 체인을 이해할 수 있을 것이다.

3.10) 예외 처리

JavaScript 또한 예외 처리 문법을 지원한다. C++과 다르지 않으므로 예제만 보이겠다.

exception.htm

```
/**
인자로 받은 수의 제곱을 반환합니다.
n: 분자(numerator), d: 분모(denominator)

@param {number} n
@param {number} d
@return {number}
*/
function div(n, d) {
  // 분모가 0이라면 예외 처리합니다.
  if (d == 0)
    throw "Denominator is 0.";
  return n / d;
}

function main() {
  // try-catch 구문을 이용한다.
  try {
    var numerator, denominator;
```

```
    // (10 / 20)의 결과를 출력합니다.
    numerator = 10;
    denominator = 20;
    Log(div(numerator, denominator));

    // (10 / 0)의 결과를 출력합니다.
    numerator = 10;
    denominator = 0;
    Log(div(numerator, denominator));

    Log("Program ended");
  } catch (ex) {
    Log('Exception occurred: ' + ex);
  }
}
```

실행 결과

```
0.5
Exception occurred: Denominator is 0.
```

이와 같이 JS의 문법에 대해 알아볼 수 있었다.

4. 단원 마무리

처음에는 욕심이 있어 HTML, CSS, JavaScript, 문서 객체 모델(DOM)과 브라우저 객체 모델(BOM)까지 모두 설명하려고 했지만, 문서 하나에 지나치게 많은 내용을 담는 것도 무리가 있어 이쯤에서 마무리하고자 한다. 사실 프로젝트가 jscc이니만큼 JS에 집중적인 문서가 되어야지, HTML과 CSS는 부수적인 요소이므로 굳이 여기에서 지면을 할애하여 설명할 필요가 없을 것이라고 판단했다. 여기서 제시한 설명과 예제는 앞으로 진행할 내용을 위한 것일 뿐, 실제 웹 페이지를 만들 때는 이 문서보다 정확하고 풍부한 내용이 담긴 사이트나 책을 찾는 것이 현명하다. 물론 필요한 내용은 이후에도 계속 설명할 것이다.

원래 이 문서는 두 파트로 나누어, 첫 번째를 HTML, CSS 및 JavaScript에 대한 개념을 세우고 두 번째 문서에서 Handy HTML Maker를 직접 설계하면서 감각을 키울 생각이었는데, 서문에서도 말했지만 이미 사회에는 Brackets라는 좋은 프로그램이 있어서, 굳이 원래 프로젝트와 관계없는 것을 끌어들여 집중력을 낮출 이유가 없었다. 그래서 Handy HTML Maker는 후에 JSCC와 별개로 진행할 생각이다.

다음은 드디어 원래 목적이었던 jscc를 공부한다. 실행기를 만들고, 링커를 만든 다음 컴파일러를 만드는 식으로 진행할 것이다. 1장과 2장을 빼면 사실 컴파일러와 직접적으로 관계있는 어떤 것을 배운 느낌이 아니지만, 다음 문서를 보게 되면 우리가 어째서 그것들을 모두 알아야만 했는지 이해하게 될 것이라고 장담한다. 애초에 jscc는 JavaScript로 개발하는 프로젝트였고, 컴파일러를 공부한다면 어셈블리와 컴파일러, 인터프리터에 대한 이해는 필수적인 것이었다. 지금까지 따라와 준 사람이 있다면 정말 대단한 것이라고 자축해도 좋다, 필자는 진심으로 그렇게 생각한다.