

# JavaScript 1

정적인 HTML을 동적으로 표현하기 위해 경량의 프로그래밍 언어를 도입하기로 결정했다. 그래서 탄생한 것이 브렌던 아이크(Brendan Eich)가 개발한 자바스크립트

HTML, CSS와 함께 웹을 구성하는 요소 중 하나로 **웹 브라우저에서 동작하는 유일한 프로그래밍 언어**

개발자가 별도의 컴파일 작업을 수행하지 않는 **인터프리터 언어(Interpreter language)**

명령형(imperative), 함수형(functional), 프로토타입 기반(prototype-based) 객체지향 프로그래밍을 지원하는 멀티 패러다임 프로그래밍 언어

## 웹 브라우저

### 개발자 도구

window	F12 또는 Ctrl+shift+I
macOS	command+option+I

### 개발자 도구 기능

Elements	로딩된 웹페이지의 DOM과 CSS를 편집하여 렌더링된 뷰를 확인해 볼 수 있다. 단, 편집한 내용이 저장되지는 않는다. 웹 페이지가 의도된 대로 렌더링 되지 않았다면 이 패널을 확인하여 유용한 힌트를 얻을 수 있다.
Console	로딩된 웹페이지의 에러를 확인하거나 자바스크립트 소스코드에 포함시킨 console.log 네소드의 결과를 확인해 볼 수 있다.
Sources	로딩된 웹 페이지의 자바스크립트 코드를 디버깅할 수 있다
Network	로딩된 웹 페이지에 관련된 네트워크 요청 정보와 퍼포먼스를 확인할 수 있다.
Application	웹 스토리지, 세션, 쿠키를 확인하고 관리 할 수 있다

## 콘솔

자바스크립트 코드에서 에러가 발생하여 애플리케이션이 정상적으로 동작하지 않을 때 가장 우선적으로 살펴보아야 할 곳

구현 단계에서 디버깅을 실행하는 것보다 간편하게 값을 확인하며 개발을 진행하기 위해 console.log 함수를 사용하는 경우가 많다. console.log(...)는 소괄호 안에 있는 코드의 실행 결과를 콘솔에 출력하는 함수이다. 이 함수를 사용해 확인하고 싶은 값을 콘솔에 출력해 확인할 수 있다.

## 디버깅

에러 정보의 오른쪽에 에러 발생 위치를 나타내는 링크를 클릭해보자. 자바스크립트 코드를 디버깅을 할 수 있는 Sources 패널로 이동할 것이다.

Tools for Web Developers: 콘솔 사용과 Tools for Web Developers: Chrome DevTools에서 자바스크립트 디버깅 시작하기를 참고

## Node.js

클라이언트 사이드, 즉 웹 브라우저에서 동작하는 간단한 웹 애플리케이션은 브라우저만으로도 개발을 할 수 있다. 하지만 프로젝트의 규모가 커짐에 따라 React, JQuery와 같은 외부 라이브러리를 도입하거나 Babel, Webpack, ESLint등 여러가지 도구를 사용해야 할 필요가 있다. 이때 Node.js와 npm이 필요하다

## Node.js와 npm

Node.js는 Chrome V8 자바스크립트 엔진으로 빌드된 자바스크립트 런타임 환경(Runtime Environment)이다. 간단히 말해 브라우저에서만 동작하던 자바스크립트를 브라우저 이외의 환경에서 동작시킬 수 있는 자바스크립트 실행 환경이 Node.js이다.

npm(node package manager)은 자바스크립트 패키지 매니저이다. Node.js에서 사용할 수 있는 모듈들을 패키지화하여 모아둔 저장소 역할과 패키지 설치 및 관리를 위한 CLI(Command line interface)를 제공한다. 자신이 작성한 패키지를 공개할 수도 있고 필요한 패키지를 검색하여 재사용할 수도 있다.

**DOM** 각 태그가 들어있는 트리 (HTML)

**CSSOM** 각 태그에 대한 CSS 정보가 들어있는 태그 (CSS)

## 기본문법

### 변수

값을 저장(할당)하고 그 저장된 값을 참조하기 위해 사용한다. 한번 쓰고 버리는 값이 아닌 유지(캐싱)할 필요가 있는 값은 변수에 담아 사용한다.

변수를 선언할 때 `var` 키워드를 사용한다

### 값

프로그램에 의해 조작될 수 있는 대상

데이터 타입(Data type)	프로그래밍 언어에서 사용할 수 있는 값의 종류
변수(Variable)	값이 저장된 메모리 고악의 주소를 가리키는 식별자(identifier)
리터럴(literal)	소스코드 안에서 직접 만들어 낸 상수 값 자체를 말하며 값을 구성하는 최소단위

### 리터럴 표기법

```

// 숫자 리터럴
10.50
1001

// 문자열 리터럴
'Hello'
"World"

// 불리언 리터럴
true
false

// null 리터럴
null

// undefined 리터럴
undefined

// 객체 리터럴
{ name: 'Lee', gender: 'male' }

// 배열 리터럴
[ 1, 2, 3 ]

// 정규표현식 리터럴
/ab+c/

// 함수 리터럴
function() {}

```

## 데이터 타입

- 원시 타입 (primitive data type)
  - `number`
  - `string`
  - `boolean`
  - `null`
  - `undefined`
  - `symbol` (New in ECMAScript 6)
- 객체 타입 (Object data type)
  - `object`

```

// Number
var num1 = 1001;
var num2 = 10.50;

// String
var string1 = 'Hello';
var string2 = "World";

// Boolean

```

```

var bool = true;

// null
var foo = null;

// undefined
var bar;

// Object
var obj = { name: 'Lee', gender: 'male' };

// Array
var array = [ 1, 2, 3 ];

// function
var foo = function() {};

```

## 연산자

하나 이상의 표현식을 대상으로 산술, 할당, 비교, 논리, 타입 연산 등을 수행해 하나의 값을 만든다. 이때 연산의 대상을 피연산자라 한다.

```

// 산술 연산자
var area = 5 * 4; // 20

// 문자열 연결 연산자
var str = 'My name is ' + 'Lee'; // "My name is Lee"

// 할당 연산자
var color = 'red'; // "red"

// 비교 연산자
var foo = 3 > 5; // false

// 논리 연산자
var bar = (5 > 3) && (2 < 4); // true

// 타입 연산자
var type = typeof 'Hi'; // "string"

// 인스턴스 생성 연산자
var today = new Date(); // Sat Dec 01 2018 00:57:19 GMT+0900 (한국 표준시)

```

피연산자의 타입은 반드시 일치할 필요는 없다. 자바스크립트는 암묵적 타입 강제 변환을 통해 연산을 수행한다

## 키워드

수행할 동작을 규정한 것(ex. `var` 키워드는 새로운 변수를 생성할 것을 지시)

## 주석

작성된 코드의 의미를 설명하기 위해 사용

한줄주석 `//` 여러줄 주석 `/*` 과 `*/` 사이에 작성, 주석은 해석기가 무시하며 실행되지않음

## 문

프로그램은 컴퓨터에 의해 단계별로 수행될 명령들의 집합

각각의 명령을 문(statement)이라 하며 문이 실행되면 무슨 일인가가 일어나게 된다. 문은 리터럴, 연산자, 표현식, 키워드 등으로 구성되며 세미콜론(;)으로 끝나야한다.

문들은 일반적으로 위에서 아래로 순서대로 실행된다. 이러한 실행 순서는 조건문(if, switch)이나 반복문(while, for)의 사용으로 제어할 수 있다 이를 흐름제어(Control Flow)라 한다. 또는 함수 호출로 변경될 수 있다.

## 표현식

하나의 값으로 평가, 값(리터럴), 변수, 객체의 프로퍼티, 배열의 요소, 함수 호출, 메소드 호출, 피연산자와 연산자의 조합은 모두 표현식이며 하나의 값으로 평가(Evaluation)된다.

## 함수

어떤 작업을 수행하기 위해 필요한 문들의 집합을 정의한 코드 블록

함수는 호출에 의해 실행되는데 한번만 호출할 수 있는 것이 아니라 여러번 호출할 수 있다.(코드의 재사용이 유용)

## 객체

키와 값으로 구성된 프로퍼티의 집합

데이터(프로퍼티)와 그 데이터에 관련되는 동작(메소드)을 모두 포함할 수 있기 때문에 데이터와 동작을 하나의 단위로 구조화할 수 있어 유용

## 배열

1개의 변수에 여러 개의 값을 순차적으로 저장할 때 사용, 자바스크립트의 배열은 객체이며 유용한 내장 메소드를 포함

# 데이터 타입과 변수

## 데이터 타입

프로그래밍 언어에서 사용할 수 있는 데이터(숫자, 문자열, 불리언 등)의 종류

- 원시 타입 (primitive data type)

- `boolean`
- `null`
- `undefined`
- `number`

- `string`
- `symbol` (ES6에서 추가)
- 객체 타입 (object/reference type)
  - `object`

## 원시타입

변경 불가능한 값이며 pass-by-value(값에 의한 전달)

### number

자바스크립트는 하나의 숫자 타입만 존재, 모든 수를 실수로 처리하며 정수만을 표현하기 위한 데이터 타입은 없다.

자바스크립트는 2진수, 8진수, 16진수 데이터 타입을 제공하지 않기 때문에 이들 값을 참조하면 모두 10진수로 해석

- `Infinity` : 양의 무한대
- `Infinity` : 음의 무한대
- `NaN` : 산술 연산 불가(not-a-number)

```
var pInf = 10 / 0; // 양의 무한대
console.log(pInf); // Infinity

var nInf = 10 / -0; // 음의 무한대
console.log(nInf); // -Infinity

var nan = 1 * 'string'; // 산술 연산 불가
console.log(nan);      // NaN
```

### string

텍스트 데이터를 나타내는데 사용, 작은 따옴표(') 또는 큰 따옴표("") 안에 텍스트를 넣어 생성

자바스크립트의 문자열은 원시타입이며 변경 불가능, 한번 문자열이 생성되면 그 문자열은 변경할 수 없다.

문자열은 배열처럼 인덱스를 통해 접근할 수 있다. 이와 같은 특성을 갖는 데이터를 유사 배열 이라고 한다.

```
var str = 'string';
console.log(str); // string

str = 'String';
console.log(str); // String

str += ' test';
console.log(str); // String test

str = str.substring(0, 3);
console.log(str); // Str
```

```
str = str.toUpperCase();
console.log(str); // STR
```

## boolean

값은 논리적 참, 거짓을 나타내는 `true` 와 `false` 뿐이다.

비어있는 문자열과 `null`, `undefined`, 숫자 0은 `false` 로 간주된다.

## undefined

undefined타입의 값은 `undefined` 가 유일하다. 선언 이후 값을 할당하지 않은 변수는 undefined값을 가진다.

변수의 값이 없다는 것을 명시하고 싶은 경우 어떻게 하면 좋을까? 그런 경우는 undefined를 할당하는 것이 아니라 null을 할당한다.

## null

null타입의 값은 `null` 이 유일하다. 자바스크립트는 대소문자를 구별하므로 null,NULL 등과 다르다

`null` 은 의도적으로 변수에 값이 없다는 것을 명시할 때 사용한다. 이는 변수가 기억하는 메모리 어드레스의 참조 정보를 제거하는 것을 의미하며 자바스크립트 엔진은 누구도 참조하지 않는 메모리 영역에 대해 가비지 콜렉션을 수행할 것이다. 또는 함수가 호출되었으나 유효한 값을 반환할 수 없는 경우, 명시적으로 null을 반환하기도 한다.

null 타입을 확인할 때 `typeof` 연산자를 사용하면 안되고 일치 연산자(`===`)를 사용하여야 한다.

## symbol

변경 불가능한 원시 타입의 값, 주로 이름의 충돌 위험이 없는 유일한 객체의 프로퍼티 키를 만들기 위해 사용, 함수를 호출해 생성

## 객체타입

객체는 데이터와 그 데이터에 관련한 동작(절차, 방법,기능)을 모두 포함할 수 있는 개념적 존재이다. 프로퍼티와 메소드를 포함할 수 있는 독립적 주체, pass-by-reference(참조에 의한 전달) 방식으로 전달

## 변수

프로그램에서 사용되는 데이터를 일정 기간 동안 기억하여 필요할 때에 다시 사용하기 위해 데이터에 고유의 이름인 식별자를 명시한 것

`var`, `let`, `const` 키워드를 사용하여 **선언**하고 할당 연산자를 사용해 값을 **할당**한다. 그리고 식별자인 변수명을 사용해 변수에 저장된 값을 **참조**한다.

## 변수의 선언

변수명은 식별자로 불리기도 하며 명명 규칙이 존재한다.

- 반드시 영문자(특수문자 제외), underscore ( \_ ), 또는 달러 기호(\$)로 시작하여야 한다. 이어지는 문자에는 숫자(0~9)도 사용할 수 있다.
- 자바스크립트는 대/소문자를 구별하므로 사용할 수 있는 문자는 “A” ~ “Z” (대문자)와 “a” ~ “z” (소문자)이다.

## 변수의 중복 선언

var 키워드로 선언한 변수는 중복 선언이 가능하다. 변수명이 같은 변수를 중복해 선언해도 에러가 발생하지 않는다.

## 동적타이핑(Dynamic Typing)

자바스크립트는 동적타입언어이다. 같은 변수에 여러타입의 값을 할당할 수 있다.

```
var foo;

console.log(typeof foo); // undefined

foo = null;
console.log(typeof foo); // object

foo = {};
console.log(typeof foo); // object

foo = 3;
console.log(typeof foo); // number

foo = 3.14;
console.log(typeof foo); // number

foo = 'Hi';
console.log(typeof foo); // string

foo = true;
console.log(typeof foo); // boolean
```

## 변수 호이스팅

var 선언문이나 function 선언문 등 모든 선언문이 해당 Scope의 선두로 옮겨진 것처럼 동작하는 특성을 말한다. 즉, 자바스크립트는 모든 선언문(var, let, const, function, function\*, class)이 선언되기 이전에 참조 가능하다.

### 변수 생성 3단계

#### 선언 단계(Declaration phase)

변수 객체(Variable Object)에 변수를 등록한다. 이 변수 객체는 스코프가 참조하는 대상이 된다.

#### 초기화 단계(Initialization phase)

변수 객체(Variable Object)에 등록된 변수를 메모리에 할당한다. 이 단계에서 변수는 undefined로 초기화된다.

#### 할당 단계(Assignment phase)



undefined로 초기화된 변수에 실제값을 할당한다.

```
console.log(foo); // ① undefined
var foo = 123;
console.log(foo); // ② 123
{
  var foo = 456;
}
console.log(foo); // ③ 456
```

①이 실행되기 이전에 `var foo = 123;` 이 호이스팅되어 ①구문 앞에 `var foo;` 가 옮겨진다.(실제로 변수 선언이 코드 레벨로 옮겨진 것은 아니고 변수 객체(Variable object)에 등록되고 undefined로 초기화된 것이다.) 하지만 변수 선언 단계와 초기화 단계가 할당 단계와 분리되어 진행되기 때문에 이 단계에서는 foo에는 undefined가 할당되어 있다. 변수 foo에 값이 할당되는 것은 2행에서 실시된다.

②에서는 변수에 값이 할당되었기 때문에 123이 출력된다.

### 함수 레벨 스코프(Function-level scope)

함수 내에서 선언된 변수는 함수 내에서만 유효하며 함수 외부에서는 참조할 수 없다. 즉, 함수 내부에서 선언한 변수는 지역 변수이며 함수 외부에서 선언한 변수는 모두 전역 변수이다. (`var`)

### 블록 레벨 스코프(Block-level scope)

코드 블록 내에서 선언된 변수는 코드 블록 내에서만 유효하며 코드 블록 외부에서는 참조할 수 없다. (`let`, `const`)

## var 키워드로 선언된 변수의 문제점

### 1. 함수 레벨 스코프(Function-level scope)

- 전역 변수의 남발
- for loop 초기화식에서 사용한 변수를 for loop 외부 또는 전역에서 참조할 수 있다.

### 2. var 키워드 생략 허용

- 의도하지 않은 변수의 전역화

### 3. 중복 선언 허용

- 의도하지 않은 변수값 변경

### 4. 변수 호이스팅

- 변수를 선언하기 전에 참조가 가능하다.

## 연산자

### 산술연산자

피연산자를 대상으로 수학적 계산을 수행해 새로운 숫자 값을 만든다. 산술 연산을 할 수 없는 경우에는 NaN을 반환

## 이항 산술연산자

+	덧셈
-	뺄셈
*	곱셈
/	나눗셈
%	나머지

## 단항 산술연산자


1개의 피연산자를 대상으로 연산한다.

++	증가
--	감소
+	어떠한 효과도 없다. 음수가 양수로 반전하지도 않는다.
-	양수를 음수로 음수를 양수로 반전한 값을 반환한다.

피연산자 앞에 위치한 전위 증가/감소 연산자(Prefix increment/decrement operator)는 먼저 피연산자의 값을 증가/감소시킨 후, 다른 연산을 수행

피연산자 뒤에 위치한 후위 증가/감소 연산자(Postfix increment/decrement operator)는 먼저 다른 연산을 수행한 후, 피연산자의 값을 증가/감소

## 문자열 연결연산자

 연산자는 피연산자 중 하나 이상이 문자열인 경우 문자열 연결 연산자로 동작

## 할당연산자

우항에 있는 피연산자의 평가 결과를 좌항에 있는 변수에 할당, 좌항의 변수에 값을 할당하므로 부수 효과

=	x=y	x=y
+=	x+=y	x=x+y
-=	x-=y	x=x-y
*=	x*=y	x=x*y
/=	x/=y	x=x/y
%=	x%=y	x=x%y

## 비교연산자

좌항과 우항의 피연산자를 비교하여 불리언 값을 반환, if문이나 for문과 같은 제어문의 조건식에서 주로 사용

## 동등/일치 비교연산자

--	--	--	--

==	동등비교	x==y	x와y의 값이 같음 (ex. NaN==NaN //false)
===	일치비교	x===y	x와y의 값과 타입이 같음 (타입까지 비교)
≠	부등비교	x≠y	x와 y의 값이 다름
≠=	불일치비교	x≠=y	x와 y의 값과 타입이 다름

## 대소 관계 비교 연산자

피연산자의 크기를 비교하여 불리언 값을 반환

>	x>y	x가 y보다 크다
<	x<y	x가 y보다 작다
≥	x≥y	x가 y보다 크거나 같다
≤	x≤y	x가 y보다 작거나 같다

## 삼항 조건연산자

조건식의 평가 결과에 따라 반환 할 값을 결정한다.

```
//표현식
조건식 ? 조건식이 true일때 반환할 값 : 조건식이 false일때 반환할 값
```

## 논리연산자

우항과 좌항의 피연산자(부정 논리 연산자의 경우, 우항의 피연산자)를 논리 연산

논리 부정(!) 연산자는 언제나 불리언 값을 반환한다. 하지만 논리합(||) 연산자와 논리곱(&&) 연산자는 일반적으로 불리언 값을 반환하지만 반드시 불리언 값을 반환해야 하는 것은 아니다.

	논리합(OR)
&&	논리곱(AND)
!	부정(NOT)

## 쉼표 연산자

왼쪽 피연산자부터 차례대로 피연산자를 평가하고 마지막 피연산자의 평가가 끝나면 마지막 피연산자의 평가 결과를 반환

```
var x, y, z;
x = 1, y = 2, z = 3; // 3
```

## 그룹 연산자

그룹 내의 표현식을 최우선으로 평가한다. 그룹 연산자를 사용하면 연산자의 우선 순위를 1순위로 높일 수 있다.

## typeof 연산자

자신의 뒤에 위치한 피연산자의 데이터 타입을 문자열로 반환

typeof 연산자는 7가지 문자열 "string", "number", "boolean", "undefined", "symbol", "object", "function" 중 하나를 반환한다. "null"을 반환하는 경우는 없으며 함수의 경우 "function"을 반환한다.

```
typeof ''           // "string"
typeof 1            // "number"
typeof NaN          // "number"
typeof true         // "boolean"
typeof undefined    // "undefined"
typeof Symbol()     // "symbol"
typeof null         // "object"
typeof []           // "object"
typeof {}           // "object"
typeof new Date()   // "object"
typeof /test/gi     // "object"
typeof function () {} // "function"
```

## 제어문

### 블록문

0개 이상의 문들을 중괄호로 묶은 것으로 코드 블록 또는 블록이라고 부르기도 한다. 블록문은 단독으로 사용할 수도 있으나 일반적으로 제어문이나 함수 선언문 등에서 사용한다. 문의 끝에는 세미 콜론(;)을 붙이는 것이 일반적이지만 블록문은 세미콜론을 붙이지 않는다.

### 조건문

주어진 조건식의 평가 결과에 따라 코드블럭의 실행을 결정, 불리언 값으로 평가될 수 있는 표현식

#### if/else문

주어진 조건식의 평가 결과, 즉 논리적 참, 거짓에 따라 실행할 코드 블록을 결정, 조건식의 평가 결과가 불리언 값이 아니면 불리언 값으로 강제 변환되어 논리적 참, 거짓을 구별

```
if (조건식1) {
  // 조건식1이 참이면 이 코드 블록이 실행된다.
} else if (조건식2) {
  // 조건식2이 참이면 이 코드 블록이 실행된다.
} else {
  // 조건식1과 조건식2가 모두 거짓이면 이 코드 블록이 실행된다.
}
```

### Switch문

switch문의 표현식을 평가하여 그 값과 일치하는 표현식을 갖는 case문으로 실행 순서를 이동, case 문은 상황을 의미하는 표현식을 지정하고 콘론으로 마친다. 그리고 그 뒤에 실행할 문들을 위치시킨다. switch문의 표현식과 일치하는 표현식을 갖는 case문이 없다면 실행 순서는 default문으로 이동한다.

```
switch (표현식) {  
  case 표현식1:  
    switch 문의 표현식과 표현식1이 일치하면 실행될 문;  
    break;  
  case 표현식2:  
    switch 문의 표현식과 표현식2가 일치하면 실행될 문;  
    break;  
  default:  
    switch 문의 표현식과 일치하는 표현식을 갖는 case 문이 없을 때 실행될 문;  
}
```

break 문이 없다면 case 문의 표현식과 일치하지 않더라도 실행 순서는 다음 case 문으로 연이어 이동한다.

## 반복문

주어진 조건식의 평가 결과가 참인 경우 코드블럭을 실행 그 후 조건식을 다시 검사하여 여전히 참인 경우 코드블럭을 다시 실행한다. 이는 조건식이 거짓일 때까지 반복된다.

### for문

조건식이 거짓으로 판별될 때까지 코드 블록을 반복 실행한다.

```
for (초기화식; 조건식; 증감식) {  
  조건식이 참인 경우 반복 실행될 문;  
}
```

```
      ②true  
      ⑤true      ④i=1  
      ⑧false     ⑦i=2  
for (var i = 0; i < 2; i++) {  
  
  console.log(i); ③i=0  
                  ⑥i=1  
}
```

어떤 식도 선언하지 않으면 무한 루프가 된다. `for (;;) { } // 무한루프`

### while문

주어진 조건식의 평가 결과가 참이면 코드 블록을 계속해서 실행한다. 조건문의 평가결과가 거짓이 되면 실행을 종료한다. 만약 조건식의 평가결과가 boolean값이 아니면 boolean값으로 강제 변환되어 논

리적 참,거짓을 구별한다.

```
var count = 0;

// count가 3보다 작을 때까지 코드 블록을 계속 반복 실행한다.
while (count < 3) {
  console.log(count);
  count++;
} // 0 1 2
```

조건식의 평가 결과가 참이면 무한루프가 된다. `while (true) { }// 무한루프`

## do/while문

코드 블록을 실행하고 조건식을 평가한다. 따라서 코드 블록은 무조건 한번 이상 실행된다.

```
var count = 0;

// count가 3보다 작을 때까지 코드 블록을 계속 반복 실행한다.
do {
  console.log(count);
  count++;
} while (count < 3); // 0 1 2
```

## break문

레이블 문, 반복문(for, for...in, for...of, while, do...while) 또는 switch 문의 코드 블록을 탈출한다. 레이블 문, 반복문, switch 문의 코드 블록 이외에 break 문을 사용하면 SyntaxError(문법 에러)가 발생한다.

## continue문

반복문(for, for...in, for...of, while, do...while)의 코드 블록 실행을 현 지점에서 중단하고 반복문의 증감식으로 이동한다. break 문처럼 반복문을 탈출하지는 않는다.

## 타입변환

개발자에 의해 의도적으로 값의 타입을 변환하는 것을 명시적 타입변환 또는 타입캐스팅이라한다.

```
var x = 10;

// 명시적 타입 변환
var str = x.toString(); // 숫자를 문자열로 타입 캐스팅한다.
console.log(typeof str); // string
```

개발자의 의도와는 상관없이 자바스크립트 엔진에 의해 암묵적으로 타입이 자동변환 되기도 하는데 이를 암묵적 타입변환 또는 타입 강제변환 이라고 한다.

```

var x = 10;

// 암묵적 타입 변환
// 숫자 타입 x의 값을 바탕으로 새로운 문자열 타입의 값을 생성해 표현식을 평가한다.
var str = x + '';

console.log(typeof str, str); // string 10

// 변수 x의 값이 변경된 것은 아니다.
console.log(x); // 10

```

## 암묵적 타입 변환

자바스크립트 엔진은 표현식을 평가할 때 문맥, 즉 컨텍스트에 고려하여 암묵적 타입 변환을 실행한다.

### 문자열 타입으로 변환

문자열 연결 연산자의 역할은 문자열 값을 만드는 것이다. 따라서 문자열 연결 연산자의 피연산자는 문맥, 즉 컨텍스트 상 문자열 타입이어야 한다.

문자열 타입 아닌 값을 문자열 타입으로 암묵적 타입 변환을 수행할 때 아래와 같이 동작한다.

```

// 숫자 타입
0 + ''           // "0"
-0 + ''          // "0"
1 + ''           // "1"
-1 + ''          // "-1"
NaN + ''         // "NaN"
Infinity + ''    // "Infinity"
-Infinity + ''   // "-Infinity"
// 불리언 타입
true + ''        // "true"
false + ''       // "false"
// null 타입
null + ''        // "null"
// undefined 타입
undefined + ''   // "undefined"
// 심볼 타입
(Symbol()) + ''  // TypeError: Cannot convert a Symbol value to a string
// 객체 타입
({}) + ''        // "[object Object]"
Math + ''        // "[object Math]"
[] + ''          // ""
[10, 20] + ''    // "10,20"
(function(){}) + '' // "function(){}"
Array + ''       // "function Array() { [native code] }"

```

### 숫자 타입으로 변환

산술 연산자의 역할은 숫자 값을 만드는 것이다. 따라서 산술 연산자의 피연산자는 문맥, 즉 컨텍스트 상 숫자 타입이어야 한다.

자바스크립트 엔진은 숫자 타입 아닌 값을 숫자 타입으로 암묵적 타입 변환을 수행할 때 아래와 같이 동작한다. + 단항 연산자는 피연산자가 숫자 타입의 값이 아니면 숫자 타입의 값으로 암묵적 타입 변환

을 수행한다.

```
// 문자열 타입
+' '           // 0
+'0'           // 0
+'1'           // 1
+'string'      // NaN
// 불리언 타입
+true          // 1
+false         // 0
// null 타입
+null          // 0
// undefined 타입
+undefined     // NaN
// 심볼 타입
+Symbol()      // TypeError: Cannot convert a Symbol value to a number
// 객체 타입
+{}            // NaN
+[]            // 0
+[10, 20]      // NaN
+(function(){} ) // NaN
```

## 불리언 타입으로 변환

if 문이나 for 문과 같은 제어문의 조건식(conditional expression)은 불리언 값, 즉 논리적 참, 거짓을 반환해야 하는 표현식이다. 자바스크립트 엔진은 제어문의 조건식을 평가 결과를 불리언 타입으로 암묵적 타입 변환한다.

자바스크립트 엔진은 불리언 타입이 아닌 값을 **Truthy 값(참으로 인식할 값)** 또는 **Falsy 값(거짓으로 인식할 값)**으로 구분한다. 즉, Truthy 값은 true로, Falsy 값은 false로 변환된다.

아래 값들은 제어문의 조건식과 같이 불리언 값으로 평가되어야 할 컨텍스트에서 false로 평가되는 Falsy 값이다.

- false
- undefined
- null
- 0, -0
- NaN
- " (빈문자열)

## 명시적 타입 변환

### 문자열 타입으로 변환

문자열 타입이 아닌 값을 문자열 타입으로 변환하는 방법은 아래와 같다.

1. String 생성자 함수를 new 연산자 없이 호출하는 방법
2. Object.prototype.toString 메소드를 사용하는 방법



### 3. 문자열 연결 연산자를 이용하는 방법

```
// 1. String 생성자 함수를 new 연산자 없이 호출하는 방법
// 숫자 타입 => 문자열 타입
console.log(String(1));           // "1"
console.log(String(NaN));         // "NaN"
console.log(String(Infinity));    // "Infinity"
// 불리언 타입 => 문자열 타입
console.log(String(true));        // "true"
console.log(String(false));       // "false"

// 2. Object.prototype.toString 메소드를 사용하는 방법
// 숫자 타입 => 문자열 타입
console.log((1).toString());      // "1"
console.log((NaN).toString());    // "NaN"
console.log((Infinity).toString()); // "Infinity"
// 불리언 타입 => 문자열 타입
console.log((true).toString());   // "true"
console.log((false).toString());  // "false"

// 3. 문자열 연결 연산자를 이용하는 방법
// 숫자 타입 => 문자열 타입
console.log(1 + '');              // "1"
console.log(NaN + '');            // "NaN"
console.log(Infinity + '');       // "Infinity"
// 불리언 타입 => 문자열 타입
console.log(true + '');           // "true"
console.log(false + '');          // "false"
```

## 숫자 타입으로 변환

숫자 타입이 아닌 값을 숫자 타입으로 변환하는 방법은 아래와 같다.

1. Number 생성자 함수를 new 연산자 없이 호출하는 방법
2. parseInt, parseFloat 함수를 사용하는 방법(문자열만 변환 가능)
3. 단항 연결 연산자를 이용하는 방법
4. 산술 연산자를 이용하는 방법

```
// 1. Number 생성자 함수를 new 연산자 없이 호출하는 방법
// 문자열 타입 => 숫자 타입
console.log(Number('0'));         // 0
console.log(Number('-1'));        // -1
console.log(Number('10.53'));     // 10.53
// 불리언 타입 => 숫자 타입
console.log(Number(true));         // 1
console.log(Number(false));        // 0

// 2. parseInt, parseFloat 함수를 사용하는 방법(문자열만 변환 가능)
// 문자열 타입 => 숫자 타입
console.log(parseInt('0'));        // 0
console.log(parseInt('-1'));       // -1
console.log(parseFloat('10.53')); // 10.53

// 3. + 단항 연결 연산자를 이용하는 방법
// 문자열 타입 => 숫자 타입
console.log(+ '0');                // 0
```

```

console.log(+'-1');    // -1
console.log(+ '10.53'); // 10.53
// 불리언 타입 => 숫자 타입
console.log(+true);    // 1
console.log(+false);   // 0

// 4. * 산술 연산자를 이용하는 방법
// 문자열 타입 => 숫자 타입
console.log('0' * 1);    // 0
console.log('-1' * 1);   // -1
console.log('10.53' * 1); // 10.53
// 불리언 타입 => 숫자 타입
console.log(true * 1);   // 1
console.log(false * 1);  // 0

```

## 단축 평가

논리곱 연산자 `&&` 는 두개의 피연산자가 모두 `true` 로 평가될 때 `true` 를 반환한다. 대부분의 연산자가 그렇듯이 논리곱 연산자도 오른쪽에서 왼쪽으로 평가가 진행된다.

```

// 논리합(||) 연산자
'Cat' || 'Dog' // 'Cat'
false || 'Dog' // 'Dog'
'Cat' || false // 'Cat'

// 논리곱(&&) 연산자
'Cat' && 'Dog' // Dog
false && 'Dog' // false
'Cat' && false // false

```

- 객체가 null인지 확인하고 프로퍼티를 참조할 때

```

var elem = null;

console.log(elem.value); // TypeError: Cannot read property 'value' of null
console.log(elem && elem.value); // null

```

- 함수의 인수(argument)를 초기화할 때

```

// 단축 평가를 사용한 매개변수의 기본값 설정
function getStringLength(str) {
  str = str || '';
  return str.length;
}

getStringLength();    // 0
getStringLength('hi'); // 2

// ES6의 매개변수의 기본값 설정
function getStringLength(str = '') {
  return str.length;
}

```

```
getStringLength(); // 0
getStringLength('hi'); // 2
```

## 객체

자바스크립트의 객체는 키(key)과 값(value)으로 구성된 프로퍼티(Property)들의 집합이다.

### 프로퍼티

프로퍼티 키(이름)와 프로퍼티 값으로 구성된다. 프로퍼티 키는 프로퍼티를 식별하기 위한 식별자

- 프로퍼티 키 : 빈 문자열을 포함하는 모든 문자열 또는 symbol 값
- 프로퍼티 값 : 모든 값

### 메소드

프로퍼티 값이 함수일 경우, 일반 함수와 구분하기 위해 메소드라 부른다. 즉, 메소드는 객체에 제한되어 있는 함수를 의미한다.

## 객체 생성 방법

자바스크립트는 프로토타입 기반 객체 지향 언어로서 클래스라는 개념이 없고 별도의 객체 생성 방법이 존재한다.

### 객체 리터럴

중괄호({})를 사용하여 객체를 생성하는데 {} 내에 1개 이상의 프로퍼티를 기술하면 해당 프로퍼티가 추가된 객체를 생성할 수 있다. {} 내에 아무것도 기술하지 않으면 빈 객체가 생성된다.

```
var emptyObject = {};  
console.log(typeof emptyObject); // object  
  
var person = {  
  name: 'Lee',  
  gender: 'male',  
  sayHello: function () {  
    console.log('Hi! My name is ' + this.name);  
  }  
};  
  
console.log(typeof person); // object  
console.log(person); // {name: "Lee", gender: "male", sayHello: f}  
  
person.sayHello(); // Hi! My name is Lee
```

### object 생성자 함수

new 연산자와 Object 생성자 함수를 호출하여 빈 객체를 생성할 수 있다. 빈 객체 생성 이후 프로퍼티 또는 메소드를 추가하여 객체를 완성하는 방법이다. 생성자(constructor) 함수란 new 키워드와 함께 객체를 생성하고 초기화하는 함수를 말한다. 생성자 함수를 통해 생성된 객체를 인스턴스(instance)라

한다. 자바스크립트는 Object 생성자 함수 이외에도 String, Number, Boolean, Array, Date, RegExp 등의 빌트인 생성자 함수를 제공한다. 일반 함수와 생성자 함수를 구분하기 위해 생성자 함수의 이름은 파스칼 케이스(PascalCase)를 사용하는 것이 일반적이다.

```
// 빈 객체의 생성
var person = new Object();
// 프로퍼티 추가
person.name = 'Lee';
person.gender = 'male';
person.sayHello = function () {
  console.log('Hi! My name is ' + this.name);
};

console.log(typeof person); // object
console.log(person); // {name: "Lee", gender: "male", sayHello: f}

person.sayHello(); // Hi! My name is Lee
```

객체 리터럴 방식으로 생성된 객체는 결국 빌트인(Built-in) 함수인 Object 생성자 함수로 객체를 생성하는 것을 단순화시킨 축약 표현(short-hand)이다. 따라서 개발자가 일부러 Object 생성자 함수를 사용해 객체를 생성해야 할 일은 거의 없다.

## 생성자 함수

객체 리터럴 방식과 Object 생성자 함수 방식으로 객체를 생성하는 것은 프로퍼티 값만 다른 여러 개의 객체를 생성할 때 불편하다. 동일한 프로퍼티를 갖는 객체임에도 불구하고 매번 같은 프로퍼티를 기술해야 한다.

```
// 생성자 함수
function Person(name, gender) {
  this.name = name;
  this.gender = gender;
  this.sayHello = function(){
    console.log('Hi! My name is ' + this.name);
  };
}

// 인스턴스의 생성
var person1 = new Person('Lee', 'male');
var person2 = new Person('Kim', 'female');

console.log('person1: ', typeof person1);
console.log('person2: ', typeof person2);
console.log('person1: ', person1);
console.log('person2: ', person2);

person1.sayHello();
person2.sayHello();
```

- 생성자 함수 이름은 일반적으로 대문자로 시작한다. 이것은 생성자 함수임을 인식하도록 도움을 준다.

- 프로퍼티 또는 메소드명 앞에 기술한 `this` 는 생성자 함수가 생성할 **인스턴스(instance)**를 가리킨다.
- `this`에 연결(바인딩)되어 있는 프로퍼티와 메소드는 `public` (외부에서 참조 가능)하다.
- 생성자 함수 내에서 선언된 일반 변수는 `private` (외부에서 참조 불가능)하다. 즉, 생성자 함수 내부에서는 자유롭게 접근이 가능하나 외부에서 접근할 수 없다.

```
function Person(name, gender) {
  var married = true;          // private
  this.name = name;            // public
  this.gender = gender;        // public
  this.sayHello = function(){ // public
    console.log('Hi! My name is ' + this.name);
  };
}

var person = new Person('Lee', 'male');

console.log(typeof person); // object
console.log(person); // Person { name: 'Lee', gender: 'male', sayHello: [Function] }

console.log(person.gender); // 'male'
console.log(person.married); // undefined
```

## 객체 프로퍼티 접근

### 프로퍼티 키

일반적으로 문자열(빈 문자열 포함)을 지정한다. 프로퍼티 키에 문자열이나 symbol 값 이외의 값을 지정하면 암묵적으로 타입이 변환되어 문자열이 된다. 또한 문자열 타입의 값으로 수렴될 수 있는 표현식도 가능하다. **프로퍼티 키는 문자열이므로 따옴표(“ 또는 ”)를 사용한다.** 하지만 자바스크립트에서 사용 가능한 유효한 이름인 경우, 따옴표를 생략할 수 있다. 반대로 말하면 자바스크립트에서 사용 가능한 유효한 이름이 아닌 경우, 반드시 따옴표를 사용하여야 한다.

프로퍼티 값은 모든 값과 표현식이 올 수 있으며 프로퍼티 값이 함수인 경우 이를 메소드라 한다.

### 프로퍼티 값 읽기

객체의 프로퍼티 값에 접근하는 방법은 `마침표(.) 표기법` 과 `대괄호([]) 표기법` 이 있다.

```
var person = {
  'first-name': 'Ung-mo',
  'last-name': 'Lee',
  gender: 'male',
  1: 10
};

console.log(person);

console.log(person.first-name); // NaN: undefined-undefined
console.log(person[first-name]); // ReferenceError: first is not defined
console.log(person['first-name']); // 'Ung-mo'
```

```

console.log(person.gender);    // 'male'
console.log(person[gender]);   // ReferenceError: gender is not defined
console.log(person['gender']); // 'male'

console.log(person['1']);      // 10
console.log(person[1]);        // 10 : person[1] -> person['1']
console.log(person.1);         // SyntaxError

```

프로퍼티 이름이 유효한 자바스크립트 이름이 아니거나 예약어인 경우 프로퍼티 값은 대괄호 표기법으로 읽어야 한다. 대괄호(`[]`) 표기법을 사용하는 경우, **대괄호 내에 들어가는 프로퍼티 이름은 반드시 문자열이어야 한다.**

객체에 존재하지 않는 프로퍼티를 참조하면 `undefined` 를 반환한다.

## 프로퍼티 값 갱신

객체가 소유하고 있는 프로퍼티에 새로운 값을 할당하면 프로퍼티 값은 갱신된다.

```

var person = {
  'first-name': 'Ung-mo',
  'last-name': 'Lee',
  gender: 'male',
};

person['first-name'] = 'Kim';
console.log(person['first-name'] ); // 'Kim'

```

## 프로퍼티 동적 생성

객체가 소유하고 있지 않은 프로퍼티 키에 값을 할당하면 하면 주어진 키와 값으로 프로퍼티를 생성하여 객체에 추가한다.

```

var person = {
  'first-name': 'Ung-mo',
  'last-name': 'Lee',
  gender: 'male',
};

person.age = 20;
console.log(person.age); // 20

```

## 프로퍼티 삭제

`delete` 연산자를 사용하면 객체의 프로퍼티를 삭제할 수 있다. 이때 피연산자는 프로퍼티 키이어야 한다.

```

var person = {
  'first-name': 'Ung-mo',
  'last-name': 'Lee',
  gender: 'male',
};

```

```
delete person.gender;
console.log(person.gender); // undefined

delete person;
console.log(person); // Object {first-name: 'Ung-mo', last-name: 'Lee'}
```

## for-in문

for-in 문을 사용하면 객체(배열 포함)에 포함된 모든 프로퍼티에 대해 루프를 수행할 수 있다.

```
var person = {
  'first-name': 'Ung-mo',
  'last-name': 'Lee',
  gender: 'male'
};

// prop에 객체의 프로퍼티 이름이 반환된다. 단, 순서는 보장되지 않는다.
for (var prop in person) {
  console.log(prop + ': ' + person[prop]);
}

/*
first-name: Ung-mo
last-name: Lee
gender: male
*/

var array = ['one', 'two'];

// index에 배열의 경우 인덱스가 반환된다
for (var index in array) {
  console.log(index + ': ' + array[index]);
}

/*
0: one
1: two
*/
```

for-in 문은 객체의 문자열 키(key)를 순회하기 위한 문법이다. 배열에는 사용하지 않는 것이 좋다. 이유는 아래와 같다.

1. 객체의 경우, 프로퍼티의 순서가 보장되지 않는다. 그 이유는 원래 객체의 프로퍼티에는 순서가 없기 때문이다. 배열은 순서를 보장하는 데이터 구조이지만 객체와 마찬가지로 순서를 보장하지 않는다.
2. 배열 요소들만을 순회하지 않는다.

for-in 문의 단점을 극복하기 위해 ES6에서 for-of 문이 추가되었다.

```
const array = [1, 2, 3];
array.name = 'my array';

for (const value of array) {
  console.log(value);
}
```

```

/*
1
2
3
*/

for (const [index, value] of array.entries()) {
  console.log(index, value);
}

/*
0 1
1 2
2 3
*/

```

for-in 문은 객체의 프로퍼티를 순회하기 위해 사용하고 for-of 문은 배열의 요소를 순회하기 위해 사용한다.

## Pass-by-reference

객체 타입은 동적으로 변화할 수 있으므로 어느 정도의 메모리 공간을 확보해야 하는지 예측할 수 없기 때문에 런타임에 메모리 공간을 확보하고 메모리의 힙 영역(Heap Segment)에 저장된다. 이에 반해 원시 타입은 값(value)으로 전달된다. 즉, 복사되어 전달된다. 이를 pass-by-value라 한다.

```

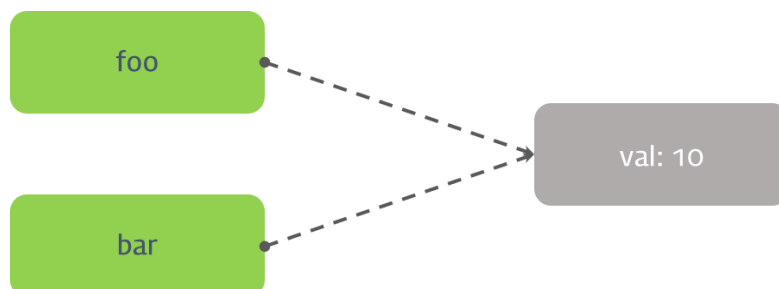
// Pass-by-reference
var foo = {
  val: 10
}

var bar = foo;
console.log(foo.val, bar.val); // 10 10
console.log(foo === bar);     // true

bar.val = 20;
console.log(foo.val, bar.val); // 20 20
console.log(foo === bar);     // true

```

foo 객체를 객체 리터럴 방식으로 생성하였다. 이때 변수 foo는 객체 자체를 저장하고 있는 것이 아니라 생성된 객체의 참조값(address)을 저장하고 있다.





```
var a = {}, b = {}, c = {}; // a, b, c는 각각 다른 빈 객체를 참조
console.log(a === b, a === c, b === c); // false false false

a = b = c = {}; // a, b, c는 모두 같은 빈 객체를 참조
console.log(a === b, a === c, b === c); // true true true
```

## Pass-by-value

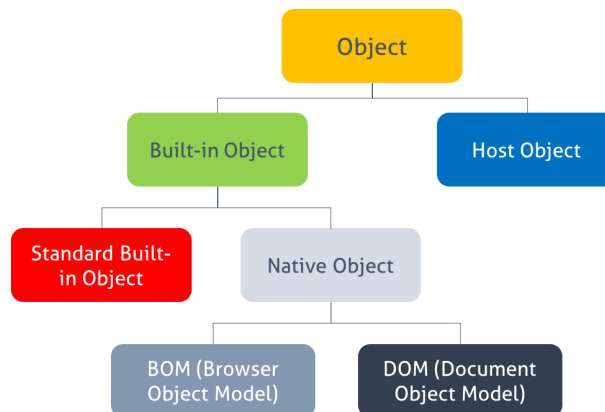
원시 타입은 값(value)으로 전달된다. 즉, 값이 복사되어 전달된다. 이를 pass-by-value(값에 의한 전달)라 한다. 원시 타입은 값이 한번 정해지면 변경할 수 없다.(immutable) 또한 이들 값은 런타임(변수 할당 시점)에 메모리의 스택 영역(Stack Segment)에 고정된 메모리 영역을 점유하고 저장된다.

```
// Pass-by-value
var a = 1;
var b = a;

console.log(a, b); // 1 1
console.log(a === b); // true

a = 10;
console.log(a, b); // 1 10
console.log(a === b); // false
```

## 객체의 분류



## 객체와 변경불가성

Immutability(변경불가성)는 객체가 생성된 이후 그 상태를 변경할 수 없는 디자인 패턴을 의미한다. Immutability은 함수형 프로그래밍의 핵심 원리이다.

## immutable value vs. mutable value

Javascript의 원시 타입(primitive data type)은 변경 불가능한 값(immutable value)이다.

- Boolean

- null
- undefined
- Number
- String
- Symbol (New in ECMAScript 6)

원시 타입 이외의 모든 값은 객체(Object) 타입이며 객체 타입은 변경 가능한 값(mutable value)이다. 즉, 객체는 새로운 값을 다시 만들 필요없이 직접 변경이 가능하다는 것이다.

```
var arr = [];
console.log(arr.length); // 0

var v2 = arr.push(2); // arr.push()는 메소드 실행 후 arr의 length를 반환
console.log(arr.length); // 1
```

처리 후 결과의 복사본을 리턴하는 문자열의 메소드 slice()와는 달리 배열(객체)의 메소드 push()는 **직접 대상 배열을 변경** 한다. 그 이유는 배열은 객체이고 객체는 immutable value가 아닌 변경 가능한 값이기 때문이다.

```
var user = {
  name: 'Lee',
  address: {
    city: 'Seoul'
  }
};

var myName = user.name; // 변수 myName은 string 타입이다
user.name = 'Kim';
console.log(myName); // Lee

myName = user.name; // 재할당
console.log(myName); // Kim
```

```
var user1 = {
  name: 'Lee',
  address: {
    city: 'Seoul'
  }
};

var user2 = user1; // 변수 user2는 객체 타입이다.

user2.name = 'Kim';

console.log(user1.name); // Kim
console.log(user2.name); // Kim
```

## 불변 데이터 패턴

객체를 불변객체로 만들어 프로퍼티의 변경을 방지하며 객체의 변경이 필요한 경우에는 참조가 아닌 객체의 방어적 복사(defensive copy)를 통해 새로운 객체를 생성한 후 변경한다.

이를 정리하면 아래와 같다.

- 객체의 방어적 복사(defensive copy) Object.assign
- 불변객체화를 통한 객체 변경 방지 Object.freeze

## object.assign

타겟 객체로 소스 객체의 프로퍼티를 복사한다. 이때 소스 객체의 프로퍼티와 동일한 프로퍼티를 가진 타겟 객체의 프로퍼티들은 소스 객체의 프로퍼티로 덮어쓰기된다. 리턴값으로 타겟 객체를 반환한다.

```
// Syntax
Object.assign(target, ...sources)
```

```
// Copy
const obj = { a: 1 };
const copy = Object.assign({}, obj);
console.log(copy); // { a: 1 }
console.log(obj == copy); // false

// Merge
const o1 = { a: 1 };
const o2 = { b: 2 };
const o3 = { c: 3 };

const merge1 = Object.assign(o1, o2, o3);

console.log(merge1); // { a: 1, b: 2, c: 3 }
console.log(o1);     // { a: 1, b: 2, c: 3 }, 타겟 객체가 변경된다!

// Merge
const o4 = { a: 1 };
const o5 = { b: 2 };
const o6 = { c: 3 };

const merge2 = Object.assign({}, o4, o5, o6);

console.log(merge2); // { a: 1, b: 2, c: 3 }
console.log(o4);     // { a: 1 }
```

```
const user1 = {
  name: 'Lee',
  address: {
    city: 'Seoul'
  }
};

// 새로운 빈 객체에 user1을 copy한다.
const user2 = Object.assign({}, user1);
// user1과 user2는 참조값이 다르다.
console.log(user1 === user2); // false

user2.name = 'Kim';
console.log(user1.name); // Lee
console.log(user2.name); // Kim

// 객체 내부의 객체(Nested Object)는 Shallow copy된다.
console.log(user1.address === user2.address); // true

user1.address.city = 'Busan';
console.log(user1.address.city); // Busan
console.log(user2.address.city); // Busan
```

## Object.freeze

불변(immutable)객체로 만들수 있다.

```
const user1 = {
  name: 'Lee',
  address: {
    city: 'Seoul'
  }
};

// Object.assign은 완전한 deep copy를 지원하지 않는다.
const user2 = Object.assign({}, user1, {name: 'Kim'});

console.log(user1.name); // Lee
console.log(user2.name); // Kim

Object.freeze(user1);

user1.name = 'Kim'; // 무시된다!

console.log(user1); // { name: 'Lee', address: { city: 'Seoul' } }

console.log(Object.isFrozen(user1)); // true
```

하지만 객체 내부의 객체(Nested Object)는 변경가능하다. 내부 객체까지 변경 불가능하게 만들려면 Deep freeze를 하여야한다

```
const user = {
  name: 'Lee',
  address: {
    city: 'Seoul'
  }
};

Object.freeze(user);

user.address.city = 'Busan'; // 변경된다!
console.log(user);
// { name: 'Lee', address: { city: 'Busan' } }
```

```
function deepFreeze(obj) {
  const props = Object.getOwnPropertyNames(obj);

  props.forEach((name) => {
    const prop = obj[name];
    if(typeof prop === 'object' && prop !== null)
      deepFreeze(prop);
  });
  return Object.freeze(obj);
}

const user = {
  name: 'Lee',
  address: {
    city: 'Seoul'
  }
};

deepFreeze(user);

user.name = 'Kim'; // 무시된다
user.address.city = 'Busan'; // 무시된다

console.log(user);
// { name: 'Lee', address: { city: 'Seoul' } }
```

## Immutable.js

List, Stack, Map, OrderedMap, Set, OrderedSet, Record와 같은 영구 불변 (Permit Immutable) 데이터 구조를 제공한다.

npm을 사용하여 Immutable.js를 설치한다. npm을 사용하여 Immutable.js를 설치한다. `$npm install immutable`

Immutable.js의 Map 모듈을 임포트하여 사용한다.

```
const { Map } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = map1.set('b', 50)
map1.get('b') // 2
map2.get('b') // 50
```

## 함수

어떤 작업을 수행하기 위해 필요한 문(statement)들의 집합을 정의한 코드 블록

```
// 함수의 정의(함수 선언문)
function square(number) {
  return number * number;
}

// 함수의 호출 (여러번 호출 가능)
square(2); // 4
```

## 함수정의

함수를 정의하는 방식은 3가지가 있다.

- 함수 선언문
- 함수 표현식
- Function 생성자 함수

## 함수선언문

### 함수명

함수 선언문의 경우, 함수명은 생략할 수 없다. 함수명은 함수 몸체에서 자신을 재귀적(recursive) 호출 하거나 자바스크립트 디버거가 해당 함수를 구분할 수 있는 식별자이다.

### 매개변수 목록

0개 이상의 목록으로 괄호로 감싸고 콤마로 분리한다. 다른 언어와의 차이점은 매개변수의 타입을 기술하지 않는다는 것이다. 이 때문에 함수 몸체 내에서 매개변수의 타입 체크가 필요할 수 있다.

### 함수 몸체

함수가 호출되었을 때 실행되는 문들의 집합이다. 종괄호({ })로 문들을 감싸고 `return` 문으로 결과값을 반환할 수 있다. 이를 반환값(return value)라 한다.

```
// 함수 선언문
function square(number) {
  return number * number;
}
```

## 함수 표현식

일급객체 다른 객체들에 일반적으로 적용 가능한 연산을 모두 지원하는 객체를 가르킴

함수 표현식 방식으로 정의한 함수는 함수명을 생략할 수 있다. 이러한 함수를 **익명 함수 (anonymous function)**이라 한다. 함수 표현식에서는 함수명을 생략하는 것이 일반적이다.

```
// 기명 함수 표현식(named function expression)
var foo = function multiply(a, b) {
  return a * b;
};

// 익명 함수 표현식(anonymous function expression)
var bar = function(a, b) {
  return a * b;
};

console.log(foo(10, 5)); // 50
console.log(multiply(10, 5)); // Uncaught ReferenceError: multiply is not defined
```

## Function 생성자 함수

함수 선언문과 함수 표현식은 모두 함수 리터럴 방식으로 함수를 정의하는데 이것은 결국 내장함수 `function` 생성자 함수로 함수를 생성하는 것을 단순화 시킨 `short-hand`(축약법)이다. `Function` 생성자 함수는 `Function.prototype.constructor` 프로퍼티로 접근 할수있다.

```
new Function(arg1, arg2, ... argN, functionBody)
```

```
var square = new Function('number', 'return number * number');
console.log(square(10)); // 100
```

## 함수 호이스팅

```
var res = square(5);

function square(number) {
  return number * number;
}
```

자바스크립트는 ES6의 `let`, `const`를 포함하여 모든 선언(`var`, `let`, `const`, `function`, `function*`, `class`)을 호이스팅(Hoisting)한다.

호이스팅이란 var 선언문이나 function 선언문 등 모든 선언문이 해당 Scope의 선두로 옮겨진 것처럼 동작하는 특성을 말한다. 즉, 자바스크립트는 모든 선언문(var, let, const, function, function\*, class)이 선언되기 이전에 참조 가능하다. 함수 선언문으로 정의된 함수는 자바스크립트 엔진이 스크립트가 로딩되는 시점에 바로 초기화하고 이를 VO(variable object)에 저장한다. 즉, **함수 선언, 초기화, 할당이 한번에 이루어진다**. 그렇기 때문에 함수 선언의 위치와는 상관없이 소스 내 어느 곳에서든지 호출이 가능하다.

## First-class object (일급객체)

생성,대입,연산,인자 또는 반환값으로서의 전달 등 프로그래밍 언어의 기본적 조작을 제한없이 사용할 수 있는 대상을 의미

다음 조건을 만족하면 일급 객체로 간주한다.

1. 무명의 리터럴로 표현이 가능하다.
2. 변수나 자료 구조(객체, 배열...)에 저장할 수 있다.
3. 함수의 파라미터로 전달할 수 있다.
4. 반환값(return value)으로 사용할 수 있다.

```
// 1. 무명의 리터럴로 표현이 가능하다.
// 2. 변수나 자료 구조에 저장할 수 있다.
var increase = function (num) {
  return ++num;
};

var decrease = function (num) {
  return --num;
};

var predicates = { increase, decrease };

// 3. 함수의 매개변수에 전달할 수 있다.
// 4. 반환값으로 사용할 수 있다.
function makeCounter(predicate) {
  var num = 0;

  return function () {
    num = predicate(num);
    return num;
  };
}

var increaser = makeCounter(predicates.increase);
console.log(increaser()); // 1
console.log(increaser()); // 2

var decreaser = makeCounter(predicates.decrease);
console.log(decreaser()); // -1
console.log(decreaser()); // -2
```

## 매개변수(Parameter,인자)

함수의 작업 실행을 위해 추가적인 정보가 필요할 경우, 매개변수를 지정한다. 매개변수는 함수 내에서 변수와 동일하게 동작한다.

## 매개변수(parameter,인자) vs 인수(argument)

매개변수는 함수 내에서 변수와 동일하게 메모리 공간을 확보하며 함수에 전달한 인수는 매개변수에 할당된다. 만약 인수를 전달하지 않으면 매개변수는 undefined로 초기화된다

```
var foo = function (p1, p2) {  
  console.log(p1, p2);  
};  
  
foo(1); // 1 undefined
```

## call-by-value

원시 타입 인수는 call-by-value(값에 의한 호출)로 동작한다. 이는 함수 호출 시 원시 타입 인수를 함수에 매개변수로 전달할 때 매개변수에 값을 복사하여 함수로 전달하는 방식이다. 이때 함수 내에서 매개변수를 통해 값이 변경되어도 전달이 완료된 원시 타입 값은 변경되지 않는다.

```
function foo(primitive) {  
  primitive += 1;  
  return primitive;  
}  
  
var x = 0;  
  
console.log(foo(x)); // 1  
console.log(x);      // 0
```

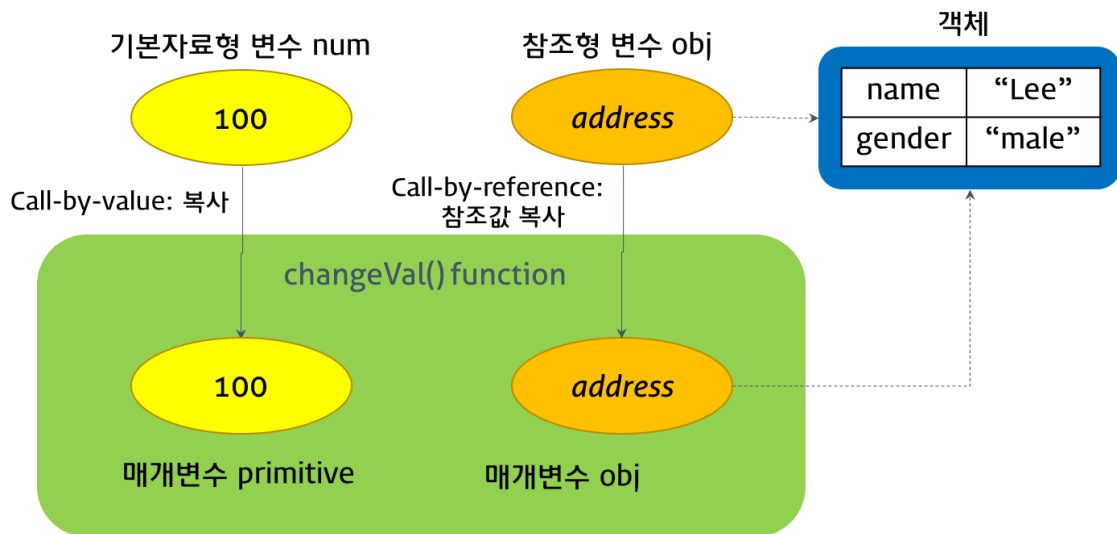
## call-by-reference

객체형(참조형) 인수는 call-by-reference(참조에 의한 호출)로 동작한다. 이는 함수 호출시 참조 타입 인수를 함수에 매개변수로 전달할 때 매개변수에 값이 복사되지 않고 객체의 참조값이 매개변수에 저장되어 함수로 전달되는 방식이다.

```
function changeVal(primitive, obj) {  
  primitive += 100;  
  obj.name = 'Kim';  
  obj.gender = 'female';  
}  
  
var num = 100;  
var obj = {  
  name: 'Lee',  
  gender: 'male'  
};  
  
console.log(num); // 100  
console.log(obj); // Object {name: 'Lee', gender: 'male'}  
  
changeVal(num, obj);
```



```
console.log(num); // 100
console.log(obj); // Object {name: 'Kim', gender: 'female'}
```



changeVal 함수는 원시 타입과 객체 타입 인수를 전달 받아 함수 몸체에서 매개변수의 값을 변경하였다. 이때 원시 타입 인수는 값을 복사하여 매개변수에 전달하기 때문에 함수 몸체에서 그 값을 변경하여도 어떠한 부수 효과(side-effect)도 발생시키지 않는다.

## 반환값

함수는 자신을 호출한 코드에게 수행한 결과를 반환(return)할 수 있다. 이때 반환된 값을 반환값(return value)이라 한다.

- **return** 키워드는 함수를 호출한 코드(caller)에게 값을 반환할 때 사용한다.
- 함수는 배열 등을 이용하여 한 번에 여러 개의 값을 리턴할 수 있다.
- 함수는 반환을 생략할 수 있다. 이때 함수는 암묵적으로 undefined를 반환한다.
- 자바스크립트 해석기는 **return** 키워드를 만나면 함수의 실행을 중단한 후, 함수를 호출한 코드로 되돌아간다. 만일 **return** 키워드 이후에 다른 구문이 존재하면 그 구문은 실행되지 않는다.

```
function calculateArea(width, height) {
  var area = width * height;
  return area; // 단일 값의 반환
}
console.log(calculateArea(3, 5)); // 15
console.log(calculateArea(8, 5)); // 40

function getSize(width, height, depth) {
  var area = width * height;
  var volume = width * height * depth;
  return [area, volume]; // 복수 값의 반환
}

console.log('area is ' + getSize(3, 2, 3)[0]); // area is 6
console.log('volume is ' + getSize(3, 2, 3)[1]); // volume is 18
```

## 함수 객체의 프로퍼티

함수는 일반 객체와는 다른 함수만의 프로퍼티를 갖는다.

```
▼ function square(number)
  arguments: null
  caller: null
  length: 1
  name: "square"
  ▶ prototype: Object
  ▼ __proto__: function ()
    ▶ apply: function apply()
      arguments: (...)
    ▶ get arguments: function ThrowTypeError()
    ▶ set arguments: function ThrowTypeError()
    ▶ bind: function bind()
    ▶ call: function call()
      caller: (...)
    ▶ get caller: function ThrowTypeError()
    ▶ set caller: function ThrowTypeError()
    ▶ constructor: function Function()
      length: 0
      name: ""
    ▶ toString: function toString()
    ▶ Symbol(Symbol.hasInstance): function [Symbol.hasInstance]()
    ▶ __proto__: Object
  ▶ <function scope>
  ▼ <function scope>
    ▶ Global: Window
```

Square 함수 객체의 속성

## caller 프로퍼티

자신을 호출한 함수를 의미한다.

```
function foo(func) {
  var res = func();
  return res;
}

function bar() {
  return 'caller : ' + bar.caller;
}

console.log(foo(bar)); // caller : function foo(func) {...}
console.log(bar());   // null (browser에서의 실행 결과)
```

## length 프로퍼티

함수 정의 시 작성된 매개변수 갯수를 의미한다.

```
function foo() {}
console.log(foo.length); // 0

function bar(x) {
  return x;
}
console.log(bar.length); // 1
```

```
function baz(x, y) {
  return x * y;
}
console.log(baz.length); // 2
```

## name 프로퍼티

함수명을 나타낸다. 기명함수의 경우 함수명을 값으로 갖고 익명함수의 경우 빈문자열을 값으로 갖는다.

```
// 기명 함수 표현식(named function expression)
var namedFunc = function multiply(a, b) {
  return a * b;
};
// 익명 함수 표현식(anonymous function expression)
var anonymousFunc = function(a, b) {
  return a * b;
};

console.log(namedFunc.name); // multiply
console.log(anonymousFunc.name); // ''
```

## \_\_proto\_\_ 접근자 프로퍼티

모든 객체는 `[[Prototype]]`이라는 내부 슬롯이 있다. `[[Prototype]]` 내부 슬롯은 프로토타입 객체를 가리킨다. 프로토타입 객체란 프로토타입 기반 객체 지향 프로그래밍의 근간을 이루는 객체로서 객체간의 상속(Inheritance)을 구현하기 위해 사용된다. 즉, 프로토타입 객체는 다른 객체에 공유 프로퍼티를 제공하는 객체를 말한다.

```
// __proto__ 접근자 프로퍼티를 통해 자신의 프로토타입 객체에 접근할 수 있다.
// 객체 리터럴로 생성한 객체의 프로토타입 객체는 Object.prototype이다.
console.log({}.__proto__ === Object.prototype); // true
// 객체는 __proto__ 프로퍼티를 소유하지 않는다.
console.log(Object.getOwnPropertyDescriptor({}, '__proto__'));
// undefined

// __proto__ 프로퍼티는 모든 객체의 프로토타입 객체인 Object.prototype의 접근자 프로퍼티이다.
console.log(Object.getOwnPropertyDescriptor(Object.prototype, '__proto__'));
// {get: f, set: f, enumerable: false, configurable: true}

// 모든 객체는 Object.prototype의 접근자 프로퍼티 __proto__를 상속받아 사용할 수 있다.
console.log({}.__proto__ === Object.prototype); // true

// 함수 객체의 프로토타입 객체는 Function.prototype이다.
console.log((function() {}).__proto__ === Function.prototype); // true
```

## prototype 프로퍼티

함수 객체만이 소유하는 프로퍼티이다. 즉 일반 객체에는 `prototype` 프로퍼티가 없다.

```
// 함수 객체는 prototype 프로퍼티를 소유한다.
console.log(Object.getOwnPropertyDescriptor(function() {}, 'prototype'));
// {value: {...}, writable: true, enumerable: false, configurable: false}
```

```
// 일반 객체는 prototype 프로퍼티를 소유하지 않는다.
console.log(Object.getOwnPropertyDescriptor({}, 'prototype'));
// undefined
```

prototype 프로퍼티는 함수가 객체를 생성하는 생성자 함수로 사용될 때, 생성자 함수가 생성한 인스턴스의 프로토타입 객체를 가리킨다.

## 함수의 다양한 형태

### 즉시 실행 함수

함수의 정의와 동시에 실행되는 함수를 즉시 실행 함수(IIFE, Immediately Invoke Function Expression)라고 한다. 최초 한번만 호출되며 다시 호출할 수는 없다.

```
// 기명 즉시 실행 함수(named immediately-invoked function expression)
(function myFunction() {
  var a = 3;
  var b = 5;
  return a * b;
})();

// 익명 즉시 실행 함수(immediately-invoked function expression)
(function () {
  var a = 3;
  var b = 5;
  return a * b;
})();

// SyntaxError: Unexpected token (
// 함수선언문은 자바스크립트 엔진에 의해 함수 몸체를 닫는 중괄호 뒤에 ;가 자동 추가된다.
function () {
  // ...
}(); // => };();

// 따라서 즉시 실행 함수는 소괄호로 감싸준다.
(function () {
  // ...
})();

(function () {
  // ...
})();
```

### 내부 함수

함수 내부에 정의된 함수를 내부함수라 한다.

내부함수 child는 자신을 포함하고 있는 부모 함수 parent의 변수에 접근할 수 있다. 하지만 부모함수는 자식함수(내부함수)의 변수에 접근할 수 없다.

```
function parent(param) {
  var parentVar = param;
  function child() {
    var childVar = 'lee';
```

```

    console.log(parentVar + ' ' + childVar); // Hello lee
  }
  child();
  console.log(parentVar + ' ' + childVar);
  // Uncaught ReferenceError: childVar is not defined
}
parent('Hello');

```

내부함수는 부모함수의 외부에서 접근할 수 없다.

```

function sayHello(name){
  var text = 'Hello ' + name;
  var logHello = function(){ console.log(text); }
  logHello();
}

sayHello('lee'); // Hello lee
logHello('lee'); // logHello is not defined

```

## 재귀함수

자기 자신을 호출하는 함수

```

// 피보나치 수열
// 피보나치 수는 0과 1로 시작하며, 다음 피보나치 수는 바로 앞의 두 피보나치 수의 합이 된다.
// 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
function fibonacci(n) {
  if (n < 2) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(0)); // 0
console.log(fibonacci(1)); // 1
console.log(fibonacci(2)); // 1
console.log(fibonacci(3)); // 2
console.log(fibonacci(4)); // 3
console.log(fibonacci(5)); // 5
console.log(fibonacci(6)); // 8

// 팩토리얼
// 팩토리얼(계승)은 1부터 자신까지의 모든 양의 정수의 곱이다.
// n! = 1 * 2 * ... * (n-1) * n
function factorial(n) {
  if (n < 2) return 1;
  return factorial(n - 1) * n;
}

console.log(factorial(0)); // 1
console.log(factorial(1)); // 1
console.log(factorial(2)); // 2
console.log(factorial(3)); // 6
console.log(factorial(4)); // 24
console.log(factorial(5)); // 120
console.log(factorial(6)); // 720

```

자신을 무한히 연쇄 호출하므로 호출을 멈출 수 있는 탈출 조건을 반드시 만들어야 한다. 탈출 조건이 없는 경우, 함수가 무한 호출되어 stackoverflow에러가 발생한다. 대부분의 재귀 함수는 for나 while문으로 구현이 가능하다.

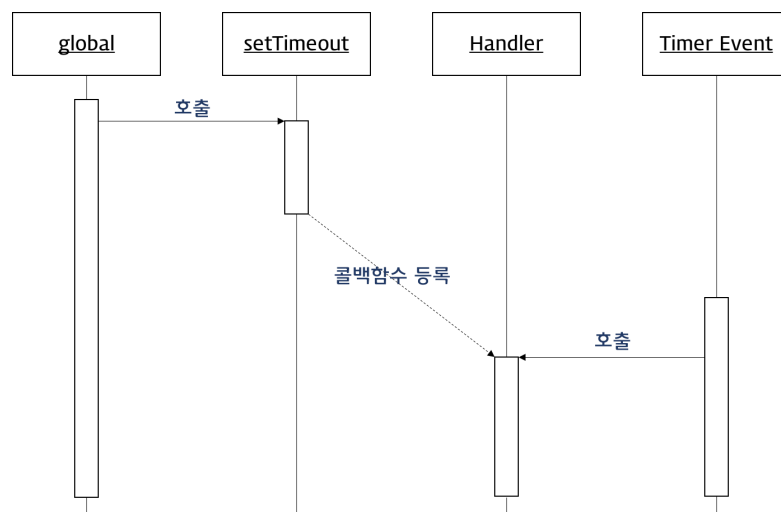
## 콜백 함수

함수를 명시적으로 호출하는 방식이 아니라 특정 이벤트가 발생했을 때 시스템에 의해 호출되는 함수 콜백 함수가 자주 사용되는 대표적인 예는 이벤트 핸들러 처리이다

```
<!DOCTYPE html>
<html>
<body>
  <button id="myButton">Click me</button>
  <script>
    var button = document.getElementById('myButton');
    button.addEventListener('click', function() {
      console.log('button clicked!');
    });
  </script>
</body>
</html>
```

콜백 함수는 매개변수를 통해 전달되고 전달받은 함수의 내부에서 어느 특정시점에 실행된다. setTimeout()의 콜백 함수에서 두번째 매개변수에 전달된 시간이 경과되면 첫번째 매개변수에 전달한 콜백함수가 호출된다.

```
setTimeout(function () {
  console.log('1초 후 출력된다. ');
}, 1000);
```



주로 비동기식 처리 모델에 사용된다. 비동기식 처리 모델이랑 처리가 종료하면 호출될 함수(콜백함수)를 미리 매개변수에 전달하고 처리가 종료하면 콜백함수를 호출하는 것이다.

콜백함수는 콜백 큐에 들어가 있다가 해당 이벤트가 발생하면 호출된다. 콜백함수는 클로저이므로 콜백 큐에 단독으로 존재하다가 호출되어도 콜백함수를 전달받은 함수의 변수에 접근할 수 있다.

```
function doSomething() {
  var name = 'Lee';

  setTimeout(function () {
    console.log('My name is ' + name);
  }, 100);
}

doSomething(); // My name is Lee
```

## 타입 체크

### typeof

타입 연산자 `typeof`는 피연산자의 데이터 타입을 문자열로 반환한다.

```
typeof '';           // string
typeof 1;            // number
typeof NaN;          // number
typeof true;         // boolean
typeof [];           // object
typeof {};           // object
typeof new String(); // object
typeof new Date();   // object
typeof /test/gi;     // object
typeof function () {}; // function
typeof undefined;    // undefined
typeof null;         // object (설계적 결함)
typeof undeclared;   // undefined (설계적 결함)
```

### Object.prototype.toString

객체를 나타내는 문자열 반환

```
var obj = new Object();
obj.toString(); // [object Object]
```

`Function.prototype.call` 메소드를 사용하면 모든 타입의 값의 타입을 알아낼 수 있다.

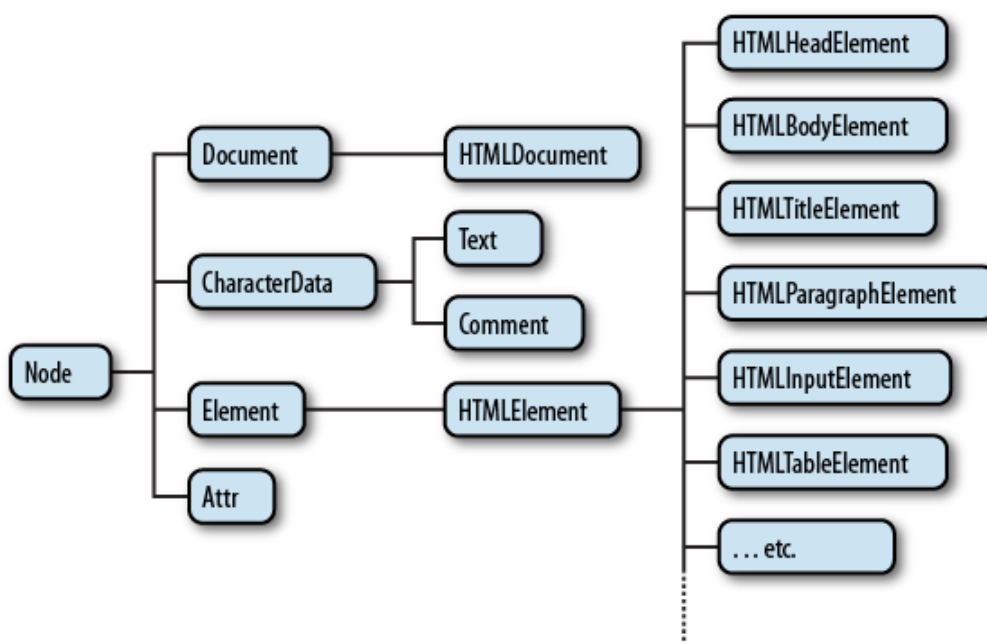
```
Object.prototype.toString.call('');           // [object String]
Object.prototype.toString.call(new String()); // [object String]
Object.prototype.toString.call(1);            // [object Number]
Object.prototype.toString.call(new Number()); // [object Number]
Object.prototype.toString.call(NaN);          // [object Number]
Object.prototype.toString.call(Infinity);     // [object Number]
Object.prototype.toString.call(true);         // [object Boolean]
Object.prototype.toString.call(undefined);    // [object Undefined]
```

```

Object.prototype.toString.call();           // [object Undefined]
Object.prototype.toString.call(null);       // [object Null]
Object.prototype.toString.call([]);         // [object Array]
Object.prototype.toString.call({});        // [object Object]
Object.prototype.toString.call(new Date()); // [object Date]
Object.prototype.toString.call(Math);      // [object Math]
Object.prototype.toString.call(/test/i);   // [object RegExp]
Object.prototype.toString.call(function () {}); // [object Function]
Object.prototype.toString.call(document);  // [object HTMLDocument]
Object.prototype.toString.call(argument);  // [object Arguments]
Object.prototype.toString.call(undeclared); // ReferenceError

```

## instanceof



DOM tree의 객체 구성

instanceof 연산자는 피연산자인 객체가 우항에 명시한 타입의 인스턴스인지 여부를 알려준다. 이때 타입이란 constructor를 말하며 프로토타입 체인에 존재하는 모든 constructor를 검색하여 일치하는 constructor가 있다면 true를 반환한다.

```

function Person() {}
const person = new Person();

console.log(person instanceof Person); // true
console.log(person instanceof Object); // true

```

## 유사 배열 객체



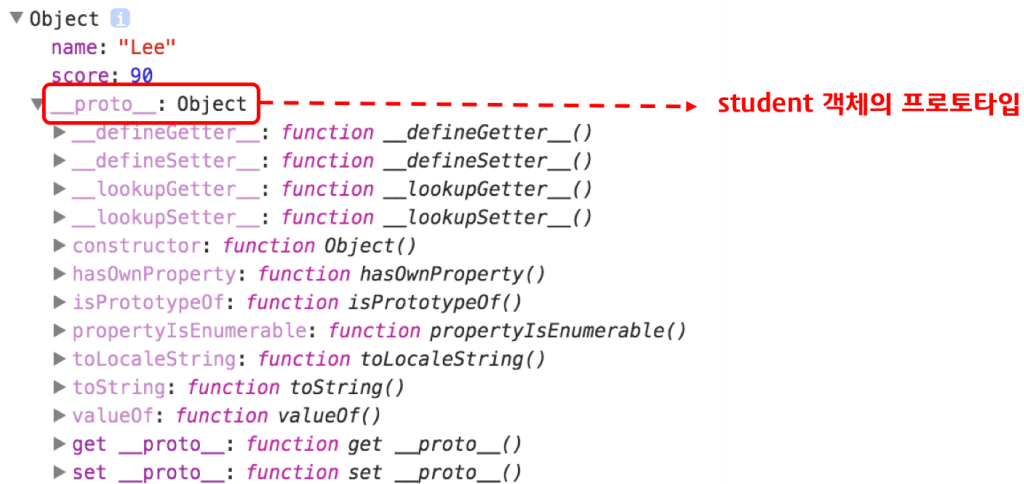
배열인지 체크하기 위해서는 Array.isArray 메소드를 사용한다.

```
console.log(Array.isArray([])); // true
console.log(Array.isArray({})); // false
console.log(Array.isArray('123')); // false
```

## 프로토타입

자바스크립트의 객체 생성 방법

자바스크립트의 모든 객체는 자신의 부모 역할을 담당하는 객체와 연결되어 있다. 이것은 마치 객체 지향의 상속 개념과 같이 부모 객체의 프로퍼티 또는 메소드를 상속받아 사용할수 있게 한다. 이러한 부모객체를 프로토타입 이라 한다.



```
var student = {
  name: 'Lee',
  score: 90
}
console.log(student.__proto__ === Object.prototype); // true
```

## [[Prototype]] vs prototype 프로퍼티

모든 객체는 자신의 프로토타입 객체를 가리키는 [[Prototype]] 인터널 슬롯을 갖으며 상속을 위해 사용된다. 함수도 객체이므로 [[Prototype]] 인터널 슬롯을 갖는다. 그런데 함수 객체는 일반 객체와 달리 prototype 프로퍼티도 소유하게 된다.

```
function Person(name) {
  this.name = name;
}

var foo = new Person('Lee');
```

```
console.dir(Person); // prototype 프로퍼티가 있다.  
console.dir(foo);    // prototype 프로퍼티가 없다.
```

## [[Prototype]]

- 함수를 포함한 모든 객체가 가지고 있는 인터널 슬롯이다.
- 객체의 입장에서 자신의 부모 역할을 하는 프로토타입 객체를 가리키며 함수 객체의 경우 **Function.prototype**를 가리킨다. 그 이유에 대해서는 [4.2 생성자 함수로 생성된 객체의 프로토타입 체인](#)을 참조하기 바란다.

```
console.log(Person.__proto__ === Function.prototype);
```

## prototype 프로퍼티

- 함수 객체만 가지고 있는 프로퍼티이다.
- 함수 객체가 생성자로 사용될 때 이 함수를 통해 생성될 객체의 부모 역할을 하는 객체(프로토타입 객체)를 가리킨다.

```
console.log(Person.prototype === foo.__proto__);
```

## constructor 프로퍼티

객체의 입장에서 자신을 생성한 객체를 가르킨다.

```
function Person(name) {  
  this.name = name;  
}  
  
var foo = new Person('Lee');  
  
// Person() 생성자 함수에 의해 생성된 객체를 생성한 객체는 Person() 생성자 함수이다.  
console.log(Person.prototype.constructor === Person); //true  
  
// foo 객체를 생성한 객체는 Person() 생성자 함수이다.  
console.log(foo.constructor === Person); //true  
  
// Person() 생성자 함수를 생성한 객체는 Function() 생성자 함수이다.  
console.log(Person.constructor === Function); //true
```

## Prototype chain

자바스크립트는 특정 객체의 프로퍼티나 메소드에 접근하려고 할 때 해당 객체에 접근하려는 프로퍼티 또는 메소드가 없다면 [[Prototype]]이 가리키는 링크를 따라 자신의 부모 역할을 하는 프로토타입 객체의 프로퍼티나 메소드를 차례대로 검색한다. 이것을 프로토타입 체인이라 한다.

```

var student = {
  name: 'Lee',
  score: 90
}
console.dir(student);
console.log(student.hasOwnProperty('name')); // true
console.log(student.__proto__ === Object.prototype); // true
console.log(Object.prototype.hasOwnProperty('hasOwnProperty')); // true

```

student 객체는 hasOwnProperty 메소드를 가지고 있지 않으므로 에러가 발생하여야 하나 정상적으로 결과가 출력되었다. 이는 student 객체의 [[Prototype]]이 가리키는 링크를 따라가서 student 객체의 부모 역할을 하는 프로토타입 객체(Object.prototype)의 메소드 hasOwnProperty를 호출하였기 때문에 가능한 것이다.

## 객체 리터럴 방식으로 생성된 객체의 프로토타입 체인

- prototype 프로퍼티는 함수 객체가 생성자로 사용될 때 이 함수를 통해 생성된 객체의 부모 역할을 하는 객체, 즉 프로토타입 객체를 가리킨다.
- [[Prototype]]은 객체의 입장에서 자신의 부모 역할을 하는 객체, 즉 프로토타입 객체를 가리킨다.

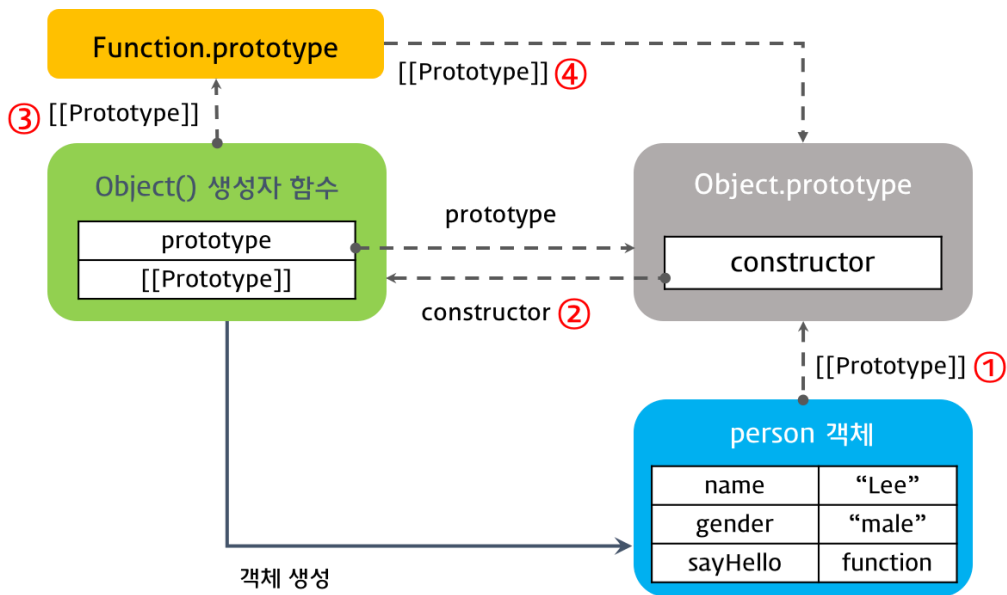
```

var person = {
  name: 'Lee',
  gender: 'male',
  sayHello: function(){
    console.log('Hi! my name is ' + this.name);
  }
};

console.dir(person);

console.log(person.__proto__ === Object.prototype); // ① true
console.log(Object.prototype.constructor === Object); // ② true
console.log(Object.__proto__ === Function.prototype); // ③ true
console.log(Function.prototype.__proto__ === Object.prototype); // ④ true

```



결론적으로 객체 리터럴을 사용하여 객체를 생성한 경우, 그 객체의 프로토타입 객체는 `Object.prototype`이다.

## 생성자 함수로 생성된 객체의 프로토타입 체인

함수를 정의하는 방식

- 함수선언식(Function declaration)
- 함수표현식(Function expression)
- `Function()` 생성자 함수

3가지 함수 정의 방식은 결국 `Function()` 생성자 함수를 통해 함수 객체를 생성한다, 따라서 어떠한 방식으로 함수 객체를 생성하여도 모든 함수 객체의 `prototype` 객체는 `Function.prototype`이다. 생성자 함수도 함수 객체이므로 생성자 함수의 `prototype` 객체는 `Function.prototype`이다.

객체 생성 방식	엔진의 객체 생성	인스턴스의 prototype 객체
객체 리터럴	<code>Object()</code> 생성자 함수	<code>Object.prototype</code>
<code>Object()</code> 생성자 함수	<code>Object()</code> 생성자 함수	<code>Object.prototype</code>
생성자 함수	생성자 함수	생성자 함수 이름.prototype

```
function Person(name, gender) {
  this.name = name;
  this.gender = gender;
  this.sayHello = function(){
    console.log('Hi! my name is ' + this.name);
  };
}

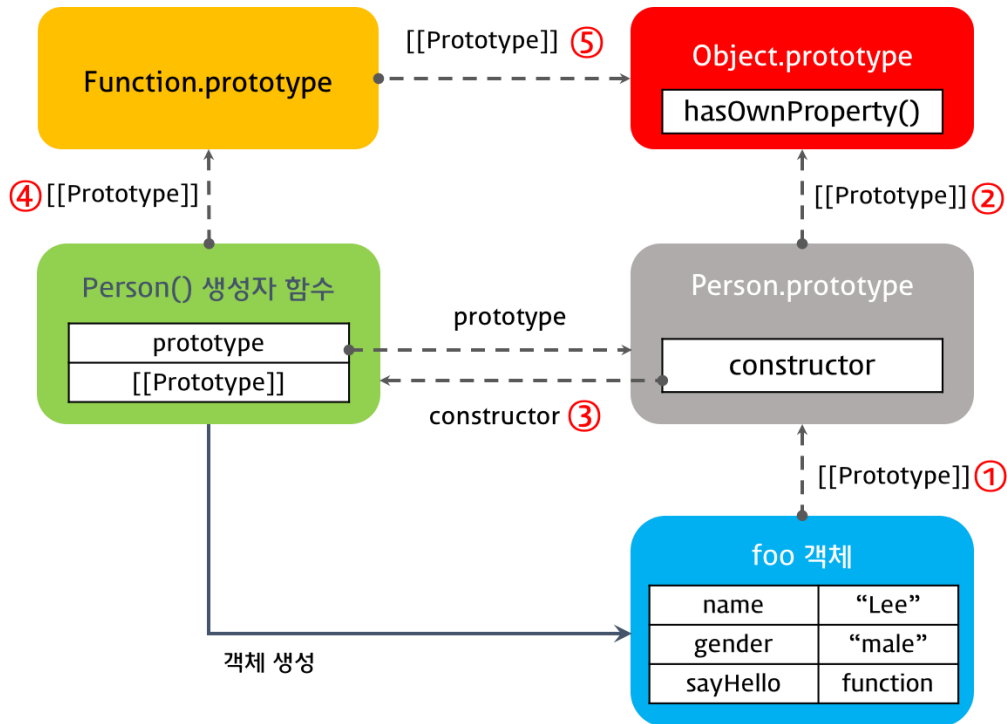
var foo = new Person('Lee', 'male');

console.dir(Person);
console.dir(foo);
```

```

console.log(foo.__proto__ === Person.prototype); // ① true
console.log(Person.prototype.__proto__ === Object.prototype); // ② true
console.log(Person.prototype.constructor === Person); // ③ true
console.log(Person.__proto__ === Function.prototype); // ④ true
console.log(Function.prototype.__proto__ === Object.prototype); // ⑤ true

```



모든 객체의 부모 객체인 Object.prototype 객체에서 프로토타입 체인이 끝나기 때문에 이때 Object.prototype 객체를 프로토타입 체인의 종점(End of prototype chain)이라 한다.

## 프로토타입 객체의 확장

프로토타입도 객체이므로 일반 객체와 같이 프로퍼티를 추가/삭제 할수 있다. 이렇게 추가/삭제된 프로퍼티는 즉시 프로토타입 체인에 반영된다.

```

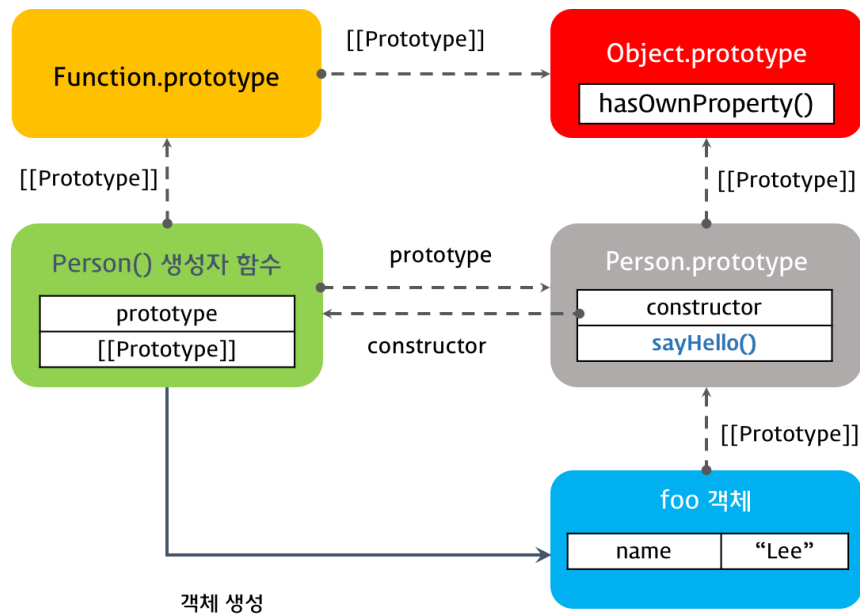
function Person(name) {
  this.name = name;
}

var foo = new Person('Lee');

Person.prototype.sayHello = function(){
  console.log('Hi! my name is ' + this.name);
};

foo.sayHello();

```



## 원시 타입(Primitive data type)의 확장

원시 타입으로 프로퍼티나 메소드를 호출할 때 원시 타입과 연관된 객체로 일시적으로 변환되어 프로토타입 객체를 공유하게 된다.

원시 타입은 객체가 아니므로 프로퍼티나 메소드를 직접 추가할 수 없다.

```
var str = 'test';

// 에러가 발생하지 않는다.
str.myMethod = function () {
  console.log('str.myMethod');
};

str.myMethod(); // Uncaught TypeError: str.myMethod is not a function
```

하지만 String 객체의 프로토타입 객체 String.prototype에 메소드를 추가하면 원시타입, 객체 모두 메소드를 사용할 수 있다.

```
var str = 'test';

String.prototype.myMethod = function () {
  return 'myMethod';
};

console.log(str.myMethod()); // myMethod
console.log('string'.myMethod()); // myMethod
console.dir(String.prototype);
```

Built-in object(내장 객체)의 Global objects(Standard Built-in Objects)인 String, Number, Array 객체 등이 가지고 있는 표준 메소드는 프로토타입 객체인 String.prototype, Number.prototype, Array.prototype 등에 정의되어 있다. 이들 프로토타입 객체 또한 Object.prototype를 프로토타입 체

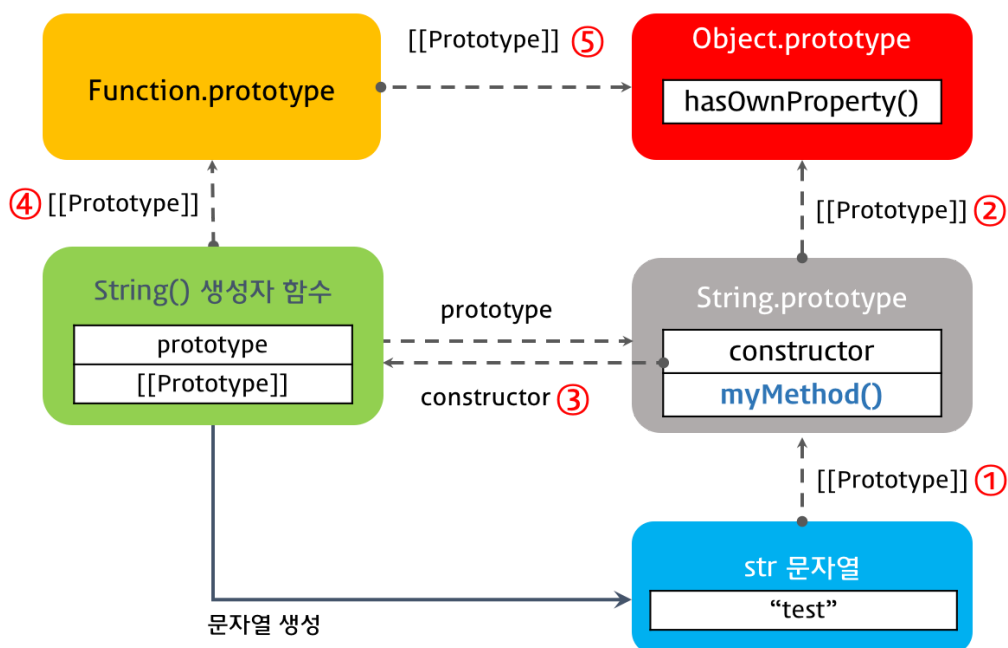
인에 의해 자시닉 프로토타입 객체로 연결한다. 자바스크립트는 표준 내장 객체의 프로토타입 객체에 정의한 메소드의 추가를 허용한다.

```
var str = 'test';

String.prototype.myMethod = function() {
  return 'myMethod';
}

console.log(str.myMethod());
console.dir(String.prototype);

console.log(str.__proto__ === String.prototype); // ① true
console.log(String.prototype.__proto__ === Object.prototype); // ② true
console.log(String.prototype.constructor === String); // ③ true
console.log(String.__proto__ === Function.prototype); // ④ true
console.log(Function.prototype.__proto__ === Object.prototype); // ⑤ true
```



## 프로토타입 객체의 변경

결정된 프로토타입 객체는 다른 임의의 객체로 변경할 수 있다. 이것은 부모 객체인 프로토타입을 동적으로 변경할 수 있다는 것을 의미한다.

주의할 것은 프로토타입 객체를 변경하면

- 프로토타입 객체 변경 시점 이전에 생성된 객체기존 프로토타입 객체를 **[[Prototype]]**에 바인딩한다.
- 프로토타입 객체 변경 시점 이후에 생성된 객체변경된 프로토타입 객체를 **[[Prototype]]**에 바인딩한다.

```
function Person(name) {
  this.name = name;
}
```

```

}

var foo = new Person('Lee');

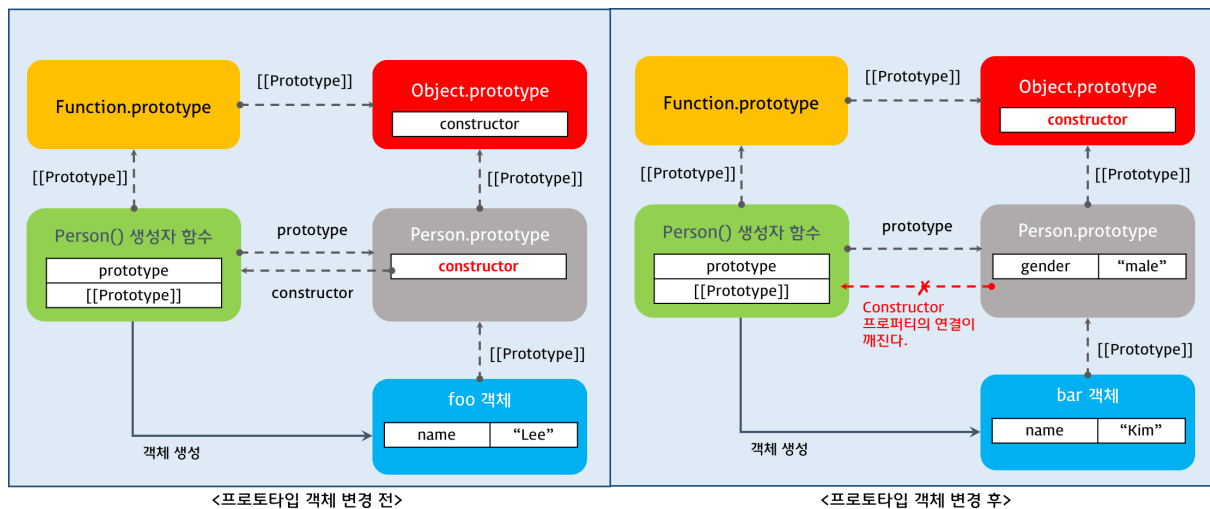
// 프로토타입 객체의 변경
Person.prototype = { gender: 'male' };

var bar = new Person('Kim');

console.log(foo.gender); // undefined
console.log(bar.gender); // 'male'

console.log(foo.constructor); // ① Person(name)
console.log(bar.constructor); // ② Object()

```



- ① `constructor` 프로퍼티는 `Person()` 생성자 함수를 가리킨다.
- ② 프로토타입 객체 변경 후, `Person()` 생성자 함수의 `Prototype` 프로퍼티가 가리키는 프로토타입 객체를 일반 객체로 변경하면서 `Person.prototype.constructor` 프로퍼티도 삭제되었다. 따라서 프로토타입 체인에 의해 `bar.constructor`의 값은 프로토타입 체이닝에 의해 `Object.prototype.constructor` 즉 `Object()` 생성자 함수가 된다.

## 프로토타입 체인 동작 조건

객체의 프로퍼티를 참조하는 경우, 해당 객체에 프로퍼티가 없는 경우, 프로토타입 체인이 동작한다.

객체의 프로퍼티에 값을 할당하는 경우, 프로토타입 체인이 동작하지 않는다. 이는 객체에 해당 프로퍼티가 있는 경우— 값을 재할당하고 해당 프로퍼티가 없는 경우 해당 객체에 프로퍼티를 동적으로 추가 하기 때문이다.

```

function Person(name) {
  this.name = name;
}

Person.prototype.gender = 'male'; // ①

var foo = new Person('Lee');
var bar = new Person('Kim');

```



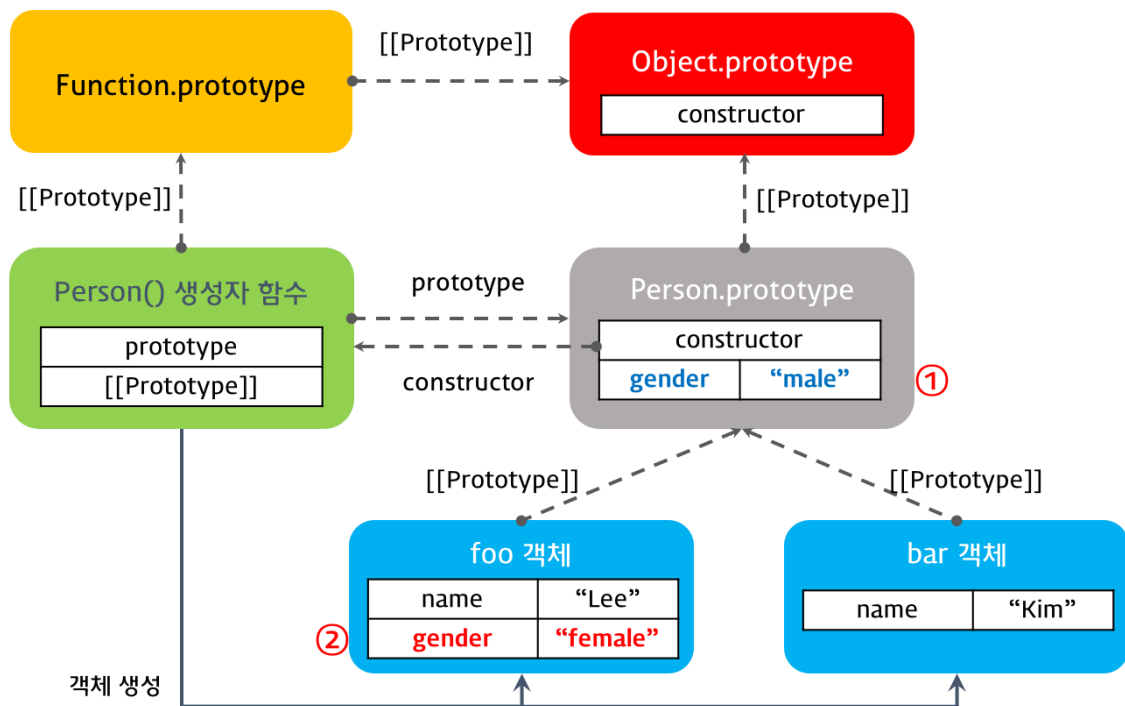
```

console.log(foo.gender); // ① 'male'
console.log(bar.gender); // ① 'male'

// 1. foo 객체에 gender 프로퍼티가 없으면 프로퍼티 동적 추가
// 2. foo 객체에 gender 프로퍼티가 있으면 해당 프로퍼티에 값 할당
foo.gender = 'female'; // ②

console.log(foo.gender); // ② 'female'
console.log(bar.gender); // ① 'male'

```



foo 객체의 gender 프로퍼티에 값을 할당하면 프로토타입 체인이 발생하여 Person.prototype 객체의 gender 프로퍼티에 값을 할당하는 것이 아니라 foo 객체에 프로퍼티를 동적으로 추가한다.

## 스코프

참조 대상 식별자(identifier, 변수, 함수의 이름과 같이 어떤 대상을 다른 대상과 구분하여 식별할 수 있는 유일한 이름)를 찾아내기 위한 규칙이다. 자바스크립트는 이 규칙대로 식별자를 찾는다.

## 스코프의 구분

자바스크립트에서 스코프를 구분

### 전역 스코프 (Global scope)

코드 어디에서든지 참조할 수 있다.

### 지역 스코프 (Local scope or Function-level scope)

함수 코드 블록이 만든 스코프로 함수 자신과 하위 함수에서만 참조할 수 있다.

변수의 관점에서 스코프를 구분

### 전역 변수 (Global variable)

전역에서 선언된 변수이며 어디에든 참조할 수 있다.

### 지역 변수 (Local variable)

지역(함수) 내에서 선언된 변수이며 그 지역과 그 지역의 하부 지역에서만 참조할 수 있다.

변수는 선언위치(전역 또는 지역)에 의해 스코프를 가지게 된다. 즉 전역에서 선언된 변수는 전역스코프를 갖는 전역 변수이고, 지역(자바스크립트의 경우 함수 내부)에서 선언된 변수는 지역 스코프를 갖는 지역 변수가 된다. 전역 스코프를 갖는 전역 변수는 전역에서 참조할 수 있다. 지역에서 선언된 지역 변수는 그 지역과 그 지역의 하부 지역에서만 참조할 수 있다.

## 자바스크립트 스코프의 특징

자바스크립트는 함수 레벨 스코프를 따른다. 함수 레벨 스코프란 함수 코드 블록 내에서 선언된 변수는 함수 코드 블록 내에서만 유효하고 함수 외부에서는 유효하지 않다(참조할 수 없다)는 것이다. 단, `let` keyword를 사용하면 블록 레벨 스코프를 사용할 수 있다.

```
var x = 0;
{
  var x = 1;
  console.log(x); // 1
}
console.log(x); // 1

let y = 0;
{
  let y = 1;
  console.log(y); // 1
}
console.log(y); // 0
```

## 전역 스코프(Global scope)

전역에 변수를 선언하면 이 변수는 어디서든지 참조할 수 있는 전역 스코프를 갖는 전역 변수가 된다. `var` 키워드로 선언한 전역 변수는 전역 객체(Global Object) `window`의 프로퍼티이다.

```
var global = 'global';

function foo() {
  var local = 'local';
  console.log(global);
  console.log(local);
}
foo();
```

```
console.log(global);
console.log(local); // Uncaught ReferenceError: local is not defined
```

전역 변수의 사용은 변수 이름이 중복될 수 있고, 의도치 않은 재할당에 의한 상태 변화로 코드를 예측하기 어렵게 만드므로 사용을 억제하여야 한다.

## 비 블록 레벨 스코프(Non block-level scope)

자바스크립트는 블록 레벨 스코프를 사용하지 않으므로 함수 밖에서 선언된 변수는 코드 블록 내에서 선언되었다 할지라도 모두 전역 스코프를 갖게 된다.

## 함수 레벨 스코프(Function-level scope)

자바스크립트는 함수 레벨 스코프를 사용한다. 함수 내에서 선언된 매개변수와 변수는 함수 외부에서는 유효하지 않다.

전역변수와 지역변수가 중복 선언 될 경우 전역 영역에서는 전역 변수만이 참조 가능하고 함수 내 지역 영역에서는 전역과 지역 변수 모두 참조 가능하나 변수명이 중복된 경우, 지역변수를 우선하여 참조한다.

```
var x = 'global';

function foo() {
  var x = 'local';
  console.log(x);

  function bar() { // 내부함수
    console.log(x); // ? local
  }

  bar();
}
foo();
console.log(x); // ? global
```

내부함수는 자신을 포함하고 있는 외부함수의 변수에 접근할 수 있다. 함수 bar에서 참조하는 변수 x는 함수 foo에서 선언된 지역변수이다. 이는 실행 컨텍스트의 스코프 체인에 의해 참조 순위에서 전역변수 x가 뒤로 밀렸기 때문이다.

```
var x = 10;

function foo() {
  x = 100;
  console.log(x); //100
}
foo();
console.log(x); // ? 100
```

함수(지역) 영역에서 전역변수를 참조할 수 있으므로 전역변수의 값도 변경할 수 있다. 내부 함수의 경우, 전역변수는 물론 상위 함수에서 선언한 변수에 접근/변경이 가능하다.

```

var x = 10;

function foo(){
  var x = 100;
  console.log(x); //100

  function bar(){ // 내부함수
    x = 1000;
    console.log(x); // ? 1000
  }

  bar();
}
foo();
console.log(x); // ? 10

```

가장 인접한 지역을 우선하여 참조한다.

```

var foo = function ( ) {

  var a = 3, b = 5;

  var bar = function ( ) {
    var b = 7, c = 11;

    // 이 시점에서 a는 3, b는 7, c는 11

    a += b + c;

    // 이 시점에서 a는 21, b는 7, c는 11

  };

  // 이 시점에서 a는 3, b는 5, c는 not defined

  bar( );

  // 이 시점에서 a는 21, b는 5

};

```

## 렉시컬 스코프

```

var x = 1;

function foo() {
  var x = 10;
  bar();
}

function bar() {
  console.log(x);
}

foo(); // ? 1
bar(); // ? 1

```

함수 bar의 상위 스코프가 무엇인지에 따라 결정

두가지 패턴 1. 함수를 어디서 호출하였는지에 따라 상위 스코프를 결정하는 것 (동적 스코프)

2. 함수를 어디서 선언하였는지에 따라 상위 스코프를 결정하는 것 (렉시컬 스코프 또는 정적 스코프)

자바스크립트를 비롯한 대부분의 프로그래밍 언어는 렉시컬 스코프를 따른다. 렉시컬 스코프는 함수를 어디서 호출하는지가 아니라 어디에 선언하였는지에 따라 결정된다.

## 암묵적 전역

```
var x = 10; // 전역 변수

function foo () {
  // 선언하지 않은 식별자
  y = 20;
  console.log(x + y);
}

foo(); // 30
```

y는 선언하지 않은 식별자이다. 식별자 y는 마치 선언된 변수처럼 동작한다. 이는 선언하지 않은 실백자에 값을 할당하면 전역 객체의 프로퍼티가 되기 때문이다. 자바스크립트 엔진은 y=20을 window.y = 20으로 해석하여 프로퍼티를 동적 생성한다. 결국 y는 전역 객체의 프로퍼티가 되어 마치 전역 변수처럼 동작한다. 이러한 현상을 **암묵적 전역(implicit global)**이라 한다. 하지만 변수 선언없이 전역객체의 프로퍼티로 추가되었을 분 변수가 아니다. 따라서 **변수 호이스팅이 발생하지 않는다**.

```
// 전역 변수 x는 호이스팅이 발생한다.
console.log(x); // undefined
// 전역 변수가 아니라 단지 전역 프로퍼티인 y는 호이스팅이 발생하지 않는다.
console.log(y); // ReferenceError: y is not defined

var x = 10; // 전역 변수

function foo () {
  // 선언하지 않은 변수
  y = 20;
  console.log(x + y);
}

foo(); // 30
```

변수가 아니라 단지 프로퍼티인 y는 delete 연산자로 삭제할 수 있다. 전역 변수는 프로퍼티이지만 delete 연산자로 삭제할 수 없다.

```
var x = 10; // 전역 변수

function foo () {
  // 선언하지 않은 변수
  y = 20;
```

```

    console.log(x + y);
  }

  foo(); // 30

  console.log(window.x); // 10
  console.log(window.y); // 20

  delete x; // 전역 변수는 삭제되지 않는다.
  delete y; // 프로퍼티는 삭제된다.

  console.log(window.x); // 10
  console.log(window.y); // undefined

```

## 최소한의 전역변수 사용

전역변수 사용을 최소화하는 방법 중 하나는 애플리케이션에서 전역변수 사용을 위해 전역변수 객체 하나를 만들어 사용하는 것

```

var MYAPP = {};

MYAPP.student = {
  name: 'Lee',
  gender: 'male'
};

console.log(MYAPP.student.name);

```

## 즉시실행함수를 이용한 전역변수 사용 억제

전역변수 사용을 억제하기 위해, 즉시 실행 함수를 사용할 수 있다. 이 방법을 사용하면 전역변수를 만들지 않으므로 라이브러리 등에 자주 사용된다. 즉시 실행 함수는 즉시 실행되고 그 후 전역에서 바로 사라진다.

```

(function () {
  var MYAPP = {};

  MYAPP.student = {
    name: 'Lee',
    gender: 'male'
  };

  console.log(MYAPP.student.name);
})();

console.log(MYAPP.student.name);

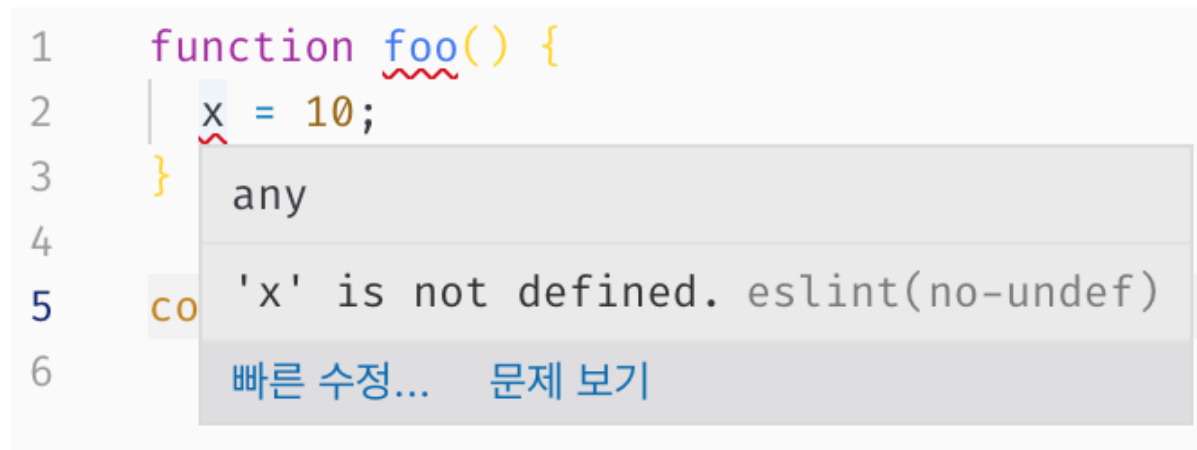
```

## strict mode

자바스크립트 언어의 문법을 보다 엄격히 적용하여 기존에는 무시되던 오류를 발생시킬 가능성이 높거나 자바스크립트 엔진의 최적화 작업에 문제를 일으킬 수 있는 코드에 대해 명시적이 에러를 발생시킨

다.

ESLint와 같은 린트 도구를 사용하여도 strict mode와 유사한 효과를 얻을 수 있다. 린트 도구는 정적 분석(static analysis) 기능을 통해 소스 코드를 실행하기 전에 소스 코드를 스캔하여 문법적 오류만이 아니라 잠재적 오류까지 찾아내고 오류의 이유를 리포팅해주는 유용한 도구이다.



ESLint의 오류 리포팅

또한 strict mode가 제한하는 오류는 물론 코딩 컨벤션을 설정 파일 형태로 정의하고 강제할 수 있기 때문에 보다 강력한 효과를 얻을 수 있다.

## strict mode의 적용

전역의 선두 또는 함수 몸체의 선두에 `'use strict';` 를 추가한다. 전역의 선두에 추가하면 스크립트 전체에 strict mode가 적용된다.

## 전역에 strict mode를 적용하는 것은 피하자

strict mode 스크립트와 non-strict mode 스크립트를 혼용하는 것은 오류를 발생시킬 수 있다. 특히 외부 서드 파티 라이브러리를 사용하는 경우, 라이브러리가 non-strict mode일 경우도 있기 때문에 **전역에 strict mode를 적용하는 것은 바람직하지 않다**. 이러한 경우, 즉시 실행 함수로 스크립트 전체를 감싸서 스코프를 구분하고 즉시 실행 함수의 선두에 strict mode를 적용한다.

## 함수 단위로 strict mode를 적용하는 것도 피하자

strict mode는 즉시실행함수로 감싼 스크립트 단위로 적용하는 것이 바람직하다.

## strict mode가 발생시키는 에러

### 암묵적 전역 변수

선언하지 않은 변수를 참조하면 ReferenceError가 발생한다.

```
(function () {  
    'use strict';  
})
```

```

    x = 1;
    console.log(x); // ReferenceError: x is not defined
  }());

```

## 변수, 함수, 매개변수의 삭제

```

(function () {
  'use strict';

  var x = 1;
  delete x;
  // SyntaxError: Delete of an unqualified identifier in strict mode.

  function foo(a) {
    delete a;
    // SyntaxError: Delete of an unqualified identifier in strict mode.
  }
  delete foo;
  // SyntaxError: Delete of an unqualified identifier in strict mode.
}());

```

## 매개변수 이름의 중복

중복된 함수 파라미터 이름을 사용하면 `SyntaxError`가 발생한다.

```

(function () {
  'use strict';

  //SyntaxError: Duplicate parameter name not allowed in this context
  function foo(x, x) {
    return x + x;
  }
  console.log(foo(1, 2));
}());

```

## with 문의 사용

with문을 사용하면 `SyntaxError`가 발생한다.

```

(function () {
  'use strict';

  // SyntaxError: Strict mode code may not include a with statement
  with({ x: 1 }) {
    console.log(x);
  }
}());

```

## 일반 함수의 this



strict mode 에서 함수를 일반 함수로서 호출하면 this에 undefined가 바인딩 된다. 생성자 함수가 아닌 일반 함수 내부에서는 this를 사용할 필요가 없기 때문이다. 이때 에러는 발생하지 않는다.

```
(function () {  
  'use strict';  
  
  function foo() {  
    console.log(this); // undefined  
  }  
  foo();  
  
  function Foo() {  
    console.log(this); // Foo  
  }  
  new Foo();  
})();
```

## 브라우저 호환성

IE 9 이하는 지원하지 않는다.