객체

객체(Object)란?

자바스크립트는 객체(object) 기반의 스크립트 언어이며 자바스크립트를 이루고 있는 거의 "모든 것"이 객체이다. 원시 타입(Primitives)을 제외한 나머지 값들(함수, 배열, 정규표현식 등)은 모두 객체이다.

자바스크립트의 객체는 키(key)과 값(value)으로 구성된 프로퍼티(Property)들의 집합이다. 프로퍼티의 값으로 자바스크립트에서 사용할 수 있는 모든 값을 사용할 수 있다. 자바스크립트의 함수는 일급 객체이므로 값으로 취급할 수 있다. 따라서 프로퍼티 값으로 함수를 사용할 수도 있으며 프로퍼티 값이 함수일 경우, 일반 함수와 구분하기 위해 메소드라 부른다.

이와 같이 객체는 데이터를 의미하는 프로퍼티(property)와 데이터를 참조하고 조작할 수 있는 동작(behavior)을 의미하는 메소드(method)로 구성된 집합이다. 객체는 데이터(프로퍼티)와 그 데이터에 관련되는 동작(메소드)을 모두 포함할 수 있기 때문에 데이터와 동작을 하나의 단위로 구조화할 수 있어 유용하다.

자바스크립트의 객체는 객체지향의 상속을 구현하기 위해 "프로토타입(prototype)"이라고 불리는 객체의 프로퍼티와 메소드를 상속받을 수 있다. 이 프로토타입은 타 언어와 구별되는 중요한 개념이다.

프로퍼티

프로퍼티는 프로퍼티 키(이름)와 프로퍼티 값으로 구성된다. 프로퍼티는 프로퍼티 키로 유일하게 식별할 수 있다. 즉, 프로퍼티 키는 프로퍼티를 식별하기 위한 식별자(identifier)다. 프로퍼티 키의 명명 규칙과 프로퍼티 값으로 사용할 수 있는 값은 아래와 같다.

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

프로퍼티 키에 문자열이나 symbol 값 이외의 값을 지정하면 암묵적으로 타입이 변환되어 문자열이 된다. 이미 존재하는 프로퍼티 키를 중복 선언하면 나중에 선언한 프로퍼티가 먼저 선언한 프로퍼티를 덮어쓴다. 배열과는 달리 객체는 프로퍼티를 열거할 때 순서를 보장하지 않는다.

메소드

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

객체 생성 방법

자바와 같은 클래스 기반 객체 지향 언어는 클래스를 사전에 정의하고 필요한 시점에 new 연산자를 사용하여 인스턴스를 생성하는 방식으로 객체를 생성한다. 하지만 자바스크립트는 프로토타입 기반 객체 지향 언어로서 클래스라는 개념이 없고 별도의 객체 생성 방법이 존재한다.

> ECMAScript 6에서 새롭게 클래스가 도입되었다. 프로토타입 기반 프로그래밍은 클래스가 존재하지 않는 객체지향 프로그래밍 스타일이다. 클래스없이 프로토타입 체인과 클로저 등으로 객체 지향 언어의 상속, 캡슐화(정보 은닉) 등의 개념을 구현한다. 하지만 클래스 기반 언어에 익숙한 프로그래머들은 혼란을 일으킬 수 있으며 자바스크립트를 어렵게 느끼게하는 하나의 장벽처럼 인식되었다. ES6의 클래스는 기존 프로토타입 기반 객체지향 프로그래밍보다 클래스 기반 언어에 익숙한 프로그래머가 보다 빠르게 학습할 수 있는 단순하고 깨끗한 새로운 문법을 제시하고 있다. ES6의 클래스가 새로운 객체지향 모델을 제공하는 것이 아니며 클래스도 사실 함수이고 기존 프로토타입 기반 패턴의 문법적 설탕(Syntactic sugar)이다.

객체 리터럴

가장 일반적인 자바스크립트의 객체 생성 방식이다. 클래스 기반 객체 지향 언어와 비교할 때 매우 간편하게 객체를 생성할 수 있다. 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", sayHell

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.

객체가 소유하고 있지 않은 프로퍼티 키에 값을 할당하면 해당 객체에 프로퍼티를 추가하고 값을 할당한다. 이미 존재하는 프로퍼티 키에 새로운 값을 할당하면 프로퍼티 값은 할당한 값으로 변경된다.

```
// 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", sayHell

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

반드시 Object 생성자 함수를 사용해 빈 객체를 생성해야 하는 것은 아니다. 객체를 생성하는 방법은 객체 리터럴을 사용하는 것이 더 간편하다. Object 생성자 함수 방식은 특별한 이유가 없다면 그다지 유용해 보이지 않는다.

사실 **객체 리터럴 방식으로 생성된 객체는 결국 빌트인(Built-in) 함수인 Object 생성자 함수로 객체를 생성하는 것을 단순화시킨 축약 표현(short-hand)이다.** 다시 말해, 자바스크립트 엔진은 객체 리터럴로 객체를 생성하는 코드를 만나면 내부적으로 Object 생성자 함수를 사용하여 객체를 생성한다. 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.

```
var person1 = {
  name: 'Lee',
  gender: 'male',
```

```
  sayHello: function () {
    console.log('Hi! My name is ' + this.name);
  }
};

var person2 = {
  name: 'Kim',
  gender: 'female',
  sayHello: function () {
    console.log('Hi! My name is ' + this.name);
  }
};
```

생성자 함수를 사용하면 마치 객체를 생성하기 위한 템플릿(클래스)처럼 사용하여 프로퍼티가 동일한 객체 여러 개를 간편하게 생성할 수 있다.

```
// 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.

- this에 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',

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

자바스크립트의 생성자 함수는 이름 그대로 객체를 생성하는 함수이다. 하지만 자바와 같은 클래스 기반 객체지향 언어의 생성자(constructor)와는 다르게 그 형식이 정해져 있는 것이 아니라 기존 함수와 동일한 방법으로 생성자 함수를 선언하고 new 연산자를 붙여서 호출하면 해당 함수는 생성자 함수로 동작한다.

이는 생성자 함수가 아닌 일반 함수에 new 연산자를 붙여 호출하면 생성자 함수처럼 동작할 수 있다는 것을 의미한다. 따라서 일반적으로 생성자 함수명은 첫문자를 대문자로 기술하여 혼란을 방지하려는 노력을 한다.

new 연산자와 함께 함수를 호출하면 `this` 바인딩이 다르게 동작한다. 자세한 내용은 생성자 호출 패턴을 참조하기 바란다.

객체 프로퍼티 접근

프로퍼티 키

프로퍼티 키는 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.

```
var person = {
  'first-name': 'Ung-mo',
  'last-name': 'Lee',
  gender: 'male',
  1: 10,
  function: 1 // OK. 하지만 예약어는 사용하지 말아야 한다.
};

console.log(person);
```

프로퍼티 키 first-name에는 반드시 따옴표를 사용해야 하지만 first_name에는 생략 가능하다. first-name은 자바스크립트에서 사용 가능한 유효한 이름이 아니라 '-' 연산자가 있는 표현식이기 때문이다.

```
var person = {
  first-name: 'Ung-mo', // SyntaxError: Unexpected token -
};
```

표현식을 프로퍼티 키로 사용하려면 키로 사용할 표현식을 대괄호로 묶어야 한다. 이때 자바스크립트 엔진은 표현식을 평가하기 위해 식별자 first를 찾을 것이고 이때 ReferenceError가 발생한다.

```
var person = {
  [first-name]: 'Ung-mo', // ReferenceError: first is not defined
};
```

예약어를 프로퍼티 키로 사용하여도 에러가 발생하지는 않는다. 하지만 예상치 못한 에러가 발생할 수 있으므로 예약어를 프로퍼티 키로 사용해서는 않된다. 자바스크립트 예약어는 아래와 같다.

```
abstract  arguments boolean break byte
case  catch char  class*  const
continue  debugger  default delete  do
double  else  enum* eval  export*
extends*  false final finally float
for function  goto  if  implements
import* in  instanceof  int interface
let long  native  new null
package private protected public  return
short static  super*  switch  synchronized
this  throw throws  transient true
try typeof  var void  volatile
while with  yield
// *는 ES6에서 추가된 예약어
```

프로퍼티 값 읽기

객체의 프로퍼티 값에 접근하는 방법은 `마침표(.) 표기법`과 `대괄호([]) 표기법`이 있다. 예제를 통해 이 두 방법의 차이를 살펴보자.

```
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 키가 유효한 자바스크립트 이름이고 예약어가 아닌 경우 프로퍼티 값은 마침표 표기법, 대괄호 표기법 모두 사용할 수 있다.

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

퍼티 이름은 반드시 문자열이어야 한다.

객체에 존재하지 않는 프로퍼티를 참조하면 `undefined`를 반환한다.

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

console.log(person.age); // undefined
```

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-in 문

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.

```
// 배열 요소들만을 순회하지 않는다.
var array = ['one', 'two'];
array.name = 'my array';

for (var index in array) {
  console.log(index + ': ' + array[index]);
}

/*
0: one
1: two
name: my array
*/
```

이와 같은 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

object type을 객체 타입 또는 참조 타입이라 한다. 참조 타입이란 객체의 모든 연산이 실제 값이 아닌 참조값으로 처리됨을 의미한다. 원시 타입은 값이 한번 정해지면 변경할 수 없지만(immutable), 객체는 프로퍼티를 변경, 추가, 삭제가 가능하므로 변경 가능(mutable)한 값이라 할 수 있다.

따라서 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.

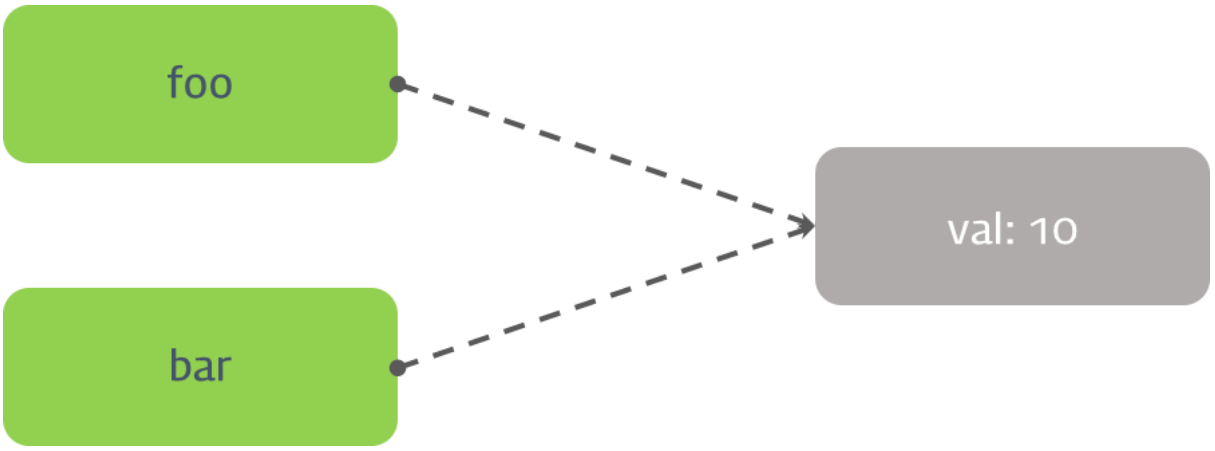
이에 반해 원시 타입은 값(value)으로 전달된다. 즉, 복사되어 전달된다. 이를 pass-by-value라 한다.

```
// 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)를 저장하고 있다.

변수 bar에 변수 foo의 값을 할당하였다. 변수 foo의 값은 생성된 객체를 가리키는 참조값이므로 변수 bar에도 같은 참조값이 저장된다. 즉, 변수 foo, bar 모두 동일한 객체를 참조하고 있다. 따라서 참조하고 있는 객체의 val 값이 변경되면 변수 foo, bar 모두 동일한 객체를 참조하고 있으므로 두 변수 모두 변경된 객체의 프로퍼티 값을 참조하게 된다. 객체는 참조(Reference) 방식으로 전달된다. 결코 복사되지 않는다.



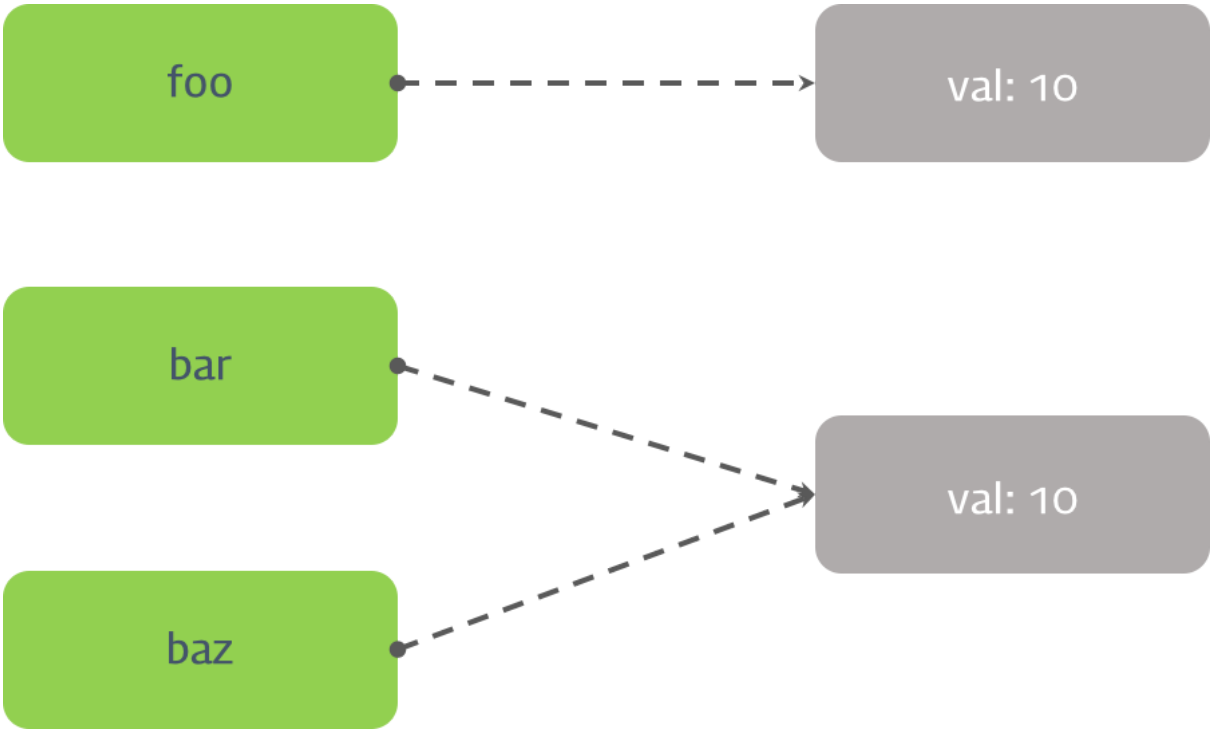
아래의 경우는 위의 경우와 약간 차이가 있다.

```
var foo = { val: 10 };
var bar = { val: 10 };

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

var baz = bar;
```

```
console.log(baz.val, bar.val); // 10 10
console.log(baz === bar);      // true
```

변수 foo와 변수 bar는 비록 내용은 같지만 별개의 객체를 생성하여 참조값을 할당하였다. 따라서 변수 foo와 변수 bar의 참조값 즉 어드레스는 동일하지 않다.

변수 baz에는 변수 bar의 값을 할당하였다. 결국 변수 baz와 변수 bar는 동일한 객체를 가리키는 참조값을 저장하고 있다. 따라서 변수 baz와 변수 bar의 참조값은 동일하다.




```
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.

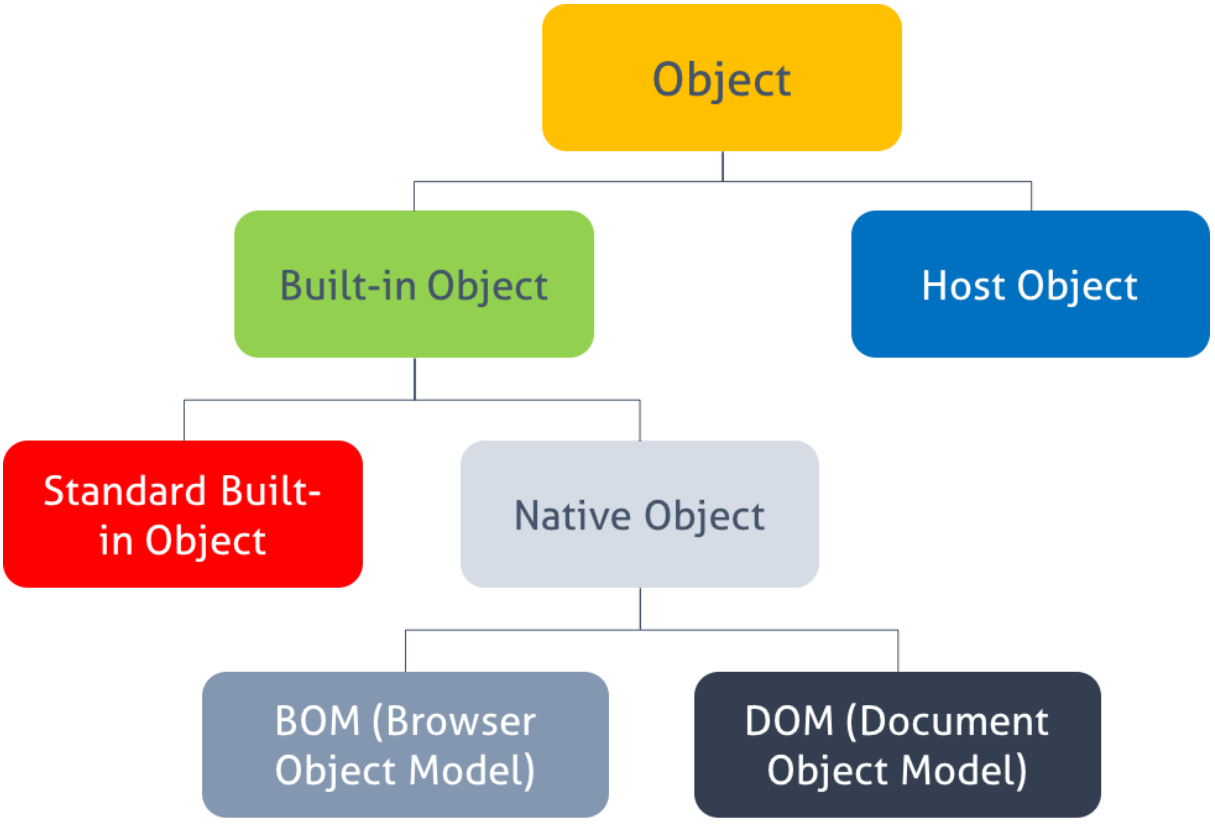
immutable에 대한 상세한 내용은 객체와 변경불가성(Immutability)을 참조하기 바란다.

```
// 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
```

변수 a는 원시 타입인 숫자 타입 1을 저장하고 있다. 원시 타입의 경우 값이 복사되어 변수에 저장된다. 즉, 참조 타입으로 저장되는 것이 아니라 값 자체가 저장되게 된다. 변수 b에 변수 a를 할당할 경우, 변수 a의 값 1은 복사되어 변수 b에 저장된다.

객체의 분류

객체는 아래와 같이 분류할 수 있다.



Built-in Object(내장 객체)는 웹페이지 등을 표현하기 위한 공통의 기능을 제공한다. 웹페이지가 브라우저에 의해 로드되자마자 별다른 행위없이 바로 사용이 가능하다. Built-in Object는 아래와 같이 구분할 수 있다.

- Standard Built-in Objects (or Global Objects)
- BOM (Browser Object Model)
- DOM (Document Object Model)

Standard Built-in Objects(표준 빌트인 객체)를 제외한 BOM과 DOM을 **Native Object**라고 분류하기도 한다. 또한 사용자가 생성한 객체를 **Host Object(사용자 정의 객체)**라 한다.

Host Object(사용자 정의 객체)

사용자가 생성한 객체 들이다. constructor 혹은 객체리터럴을 통해 사용자가 객체를 정의하고 확장시킨 것들이기 때문에 Built-in Object 와 Native Object가 구성된 이후에 구성된다.