JavaScript

Object 확장 – 김근형 강사

Object

- Object
  - 기본 타입이 아닌 레퍼런스 타입들을 생성할 때 반드시 __proto__ 타입의 최상에 위치하는 객체
  - 모든 레퍼런스 타입들은 Object를 상속받으며 __proto__를 추적해보면 모든 레퍼런스 타입들은 Object를 가진다.
  - 레퍼런스 타입들은 Object들의 속성을 쓸 수가 있으며 해당 속성을 통해 다양한 기능을 활용할 수 있다.
  - Object()는 __proto__가 존재하지 않으며 때로는 다른 객체를 생성할 때도 Object를 통한 생성이 가능하다.
  - Object의 생성자는 window객체에 저장되어 있다.

Object Operation

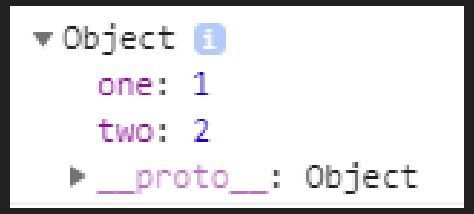
- Object 같은 key 사용
  - (var obj = { key : value }) 형태에서 key 값이 같은 프로퍼티를 두 개 작성했을 때 자바스크립트 버전 별로 차이가 있다.
  - ES3에서는 key 값이 같더라도 추가되고 ES5 strict 모드 에서는 에러가 발생한다.
  - 하지만 ES6에서는 strict 모드에 관계없이 에러가 발생하지 않는다.

```
let sameKey = {one: 1, one: 2};
console.log(sameKey);
```

Object Operation

- 변수 이름으로 값 설정
  - 변수 이름을 사용하여 Object 프로퍼티 값을 설정할 수 있다.

```
let one = 1, two = 2;
let values = {one, two};
console.log(values);
```



Object Operation

- Object에 function 작성
  - ES5 에서 Object에 함수를 다음 형태로 작성이 가능하다.

```
//ES5
/*
let obj = {
    getTotal : function(param){
        return param = 123;
    }
}
*/
//ES6
let obj = {
  getTotal(param){
    return param + 123;
  }
};
console.log(obj.getTotal(400));
```

523

Object 주요 기능들

- Object 주요 기능들

기능	설명
객체.hasOwnProperty(속성명)	객체의 속성이 상속받지 않은 속성인지 알려준다.
객체.isPrototypeOf(대상)	객체가 대상의 조상인지 알려준다.
Object.getPrototypeOf(객체), Object.setPrototypeOf(객체, prototype)	객체의 prototype을 조회하거나 설정할 수 있다.
instanceof	객체가 특정 생성자의 자식인지 조회할 수 있다.
객체.propertyIsEnumerable(속성)	해당 속성이 열거 가능한 속성인지 알려준다.
객체.toString	객체의 종류를 알려준다.
객체.valueOf	객체의 기본 값을 출력한다.

Object 주요 기능들

- Object 주요 기능들

기능	설명
Object.create(prototype, 속성들)	객체를 생성한다.
Object.defineProperties(객체, 속성들), Object.defineProperty(객체, 속성, 설명)	객체의 속성을 자세하게 정의한다.
Object.getOwnPropertyDescriptor(객체, 속성)	속성의 설명 값을 불러온다.
Object.freeze	객체 전체를 바꾸지 못하게 고정한다,,
Object.seal	속성의 추가, 제거를 막는다.
Object.preventExtensions	속성의 추가를 막는다.
Object.keys	객체의 속성명을 모두 가져와 배열로 만든다.
Object.isFrozen, Object.isSealed, Object.isExtensible	객체가 freeze 되었는지, sealed 되었는지 또는 preventExtension 상태인지 알려준다.
typeof	타입을 알려준다

hasOwnProperty

- hasOwnProperty
  - 객체의 속성이 상속받지 않은 속성인지 알려준다.
  - 자신의 속성이면 true, 부모의 속성이거나 아예 속성이 아니면 false를 반환한다.

```
var obj = {
  example: 'yes',
};
console.log(obj.example); // yes
console.log(obj.hasOwnProperty('example')); // true
console.log(obj.toString); // function toString() { [native code] }
console.log(obj.hasOwnProperty('toString')); // false
```

isPropertyOf

- isPropertyOf
  - 객체가 대상의 조상인지 알려준다.

```
var GrandParent = function() { };

var Parent = function() { };
Parent.prototype = new GrandParent();
Parent.prototype.constructor = Parent;

var Child = function() { };
Child.prototype = new Parent();
Child.prototype.constructor = Child;

var child = new Child();
console.log(Parent.prototype.isPrototypeOf(child)); // true
console.log(GrandParent.prototype.isPrototypeOf(child)); // true
```

getPrototypeOf

- getPrototypeOf
  - 객체의 prototype을 조회하거나 설정할 수 있다

```
console.log(Object.getPrototypeOf(child)); // Parent
var parent = new Parent();
console.log(Object.getPrototypeOf(parent)); // GrandParent
var grandParent = new GrandParent();
console.log(Object.getPrototypeOf(grandParent)); // {constructor: f}
console.log(Object.getPrototypeOf(new GrandParent())); // {constructor: f}

Object.setPrototypeOf(child, new GrandParent());
console.log(Object.getPrototypeOf(child)); // GrandParent
```

instanceof

- instanceof
  - 객체가 특정 생성자의 자식인지 조회할 수 있다.

```
var GrandParent = function() { };

var Parent = function() { };
Parent.prototype = new GrandParent();
Parent.prototype.constructor = Parent;

var Child = function() { };
Child.prototype = new Parent();
Child.prototype.constructor = Child;

var child = new Child();
console.log(child instanceof Parent); // true
console.log(child instanceof GrandParent); // true
```

propertyIsEnumerable

- propertyIsEnumerable
  - 해당 속성이 열거 가능한 속성인지 알려준다.
  - 상속받은 속성과 해당 객체의 속성이 아닌 것은 기본적으로 제외된다.

```
var a = [false, 1, '2'];
console.log(a.propertyIsEnumerable(0)); // true
console.log(a.propertyIsEnumerable('length')); // false
for (var value in a) {
  console.log(value); // 0, 1, 2
}
```

toString

- toString
  - 객체를 출력 시 출력하려는 값을 지정한다.
  - 문자열끼리 더할 때 주로 호출되며, 기본적으로는 객체의 종류를 알려주고, 사용자가 임의로 바꿀 수 있다.

```
var obj = { a: 'hi', b: 'kim' };
console.log(obj.toString()); // [object Object]
console.log(Math.toString()); // [object Math]
obj.toString = function() {
  return this.a + ' ' + this.b;
}; // 임의로 바꿈
console.log(obj.toString()); // 'hi kim';
console.log(obj + ' harven'); // 'hi kim harven'
```

valueOf

- valueOf
  - 객체의 기본 값을 의미한다.
  - 숫자 계산을 할 때 내부적으로 호출된다.
  - toString처럼 내부적으로 호출되기 때문에 관리하기 어렵습니다.

```
var obj = { a: 'hi', b: 'kim' };
console.log(obj.valueOf()); // { a: 'hi', b: 'kim' }
console.log(obj + 5); // '[object Object]5' <-- 내부적으로 toString이 호출됨
obj.valueOf = function() {
  return 3;
}
console.log(obj + 5); // 8 <-- 내부적으로 valueOf가 호출됨
```

create

- create
  - 객체를 생성하는 또 다른 방법

```
var obj = {}; // Object.create(Object.prototype); 과 같음
var obj2 = Object.create(null, {
  a: {
    writable: true,
    configurable: false,
    value: 5,
  }
});
console.log(obj2.a); // 5
```

defineProperty

- defineProperty
  - defineProperty 메소드는 객체에 직접 새로운 속성을 정의하거나 이미 존재하는 객체를 수정한 뒤 그 객체를 반환한다.

```
Object.defineProperty(obj, prop, descriptor)
```

  - Obj : 속성을 정의하고자 하는 객체.
  - prop :새로 정의하거나 수정하려는 속성의 이름.
  - Descriptor : 새로 정의하거나 수정하려는 속성에 대해 기술하는 객체.

Property Descriptor

- Property descriptor
  - 디스크립터(Descriptor)는 ES5에서 제시되었으며 ES6에서 이를 바탕으로 여러 기능이 추가되었다

타입	속성 이름	속성 값 형태	디폴트 값	개요
데이터	value	JavaScript 데이터 타입	undefined	프로퍼티 값으로 사용
	writable	true, false	false	속성 값의 변경 가능 여부
액세스	get	function, undefined	undefined	속성의 값을 가져올 때 쓰는 함수
	set	function, undefined	undefined	속성의 값을 설정할 때 쓰는 함수
공용	enumerable	true, false	false	for ~ in 으로 열거 가능 여부
	configurable	true, false	false	프로퍼티 삭제 가능 여부

Property Descriptor

- Property descriptor
  - 데이터 타입의 value, writable 속성과 액세스 타입의 get, set 속성을 같이 작성할 수 없다.
  - 공용 타입은 데이터 타입과 액세스 타입에서 공용으로 사용할 수 있다.
  - 프로퍼티 디스크립터에 타입을 별도로 작성하지 않고 속성으로 데이터 타입과 액세스 타입을 구분한다.(예를들면 value 또는 writable 속성의 작성 여부를 체크하고 이를 작성하면 데이터 프로퍼티 디스크립터, 아닐경우 액세스 프로퍼티 디스크립터로 체크한다.)
  - 보통 여기서 부르는 get, set 속성을 우리는 getter, setter라고 한다.

defineProperty

- value, writable
  - value, writable을 이용한 속성 설정 방법

```
var obj = {};
Object.defineProperty(obj, 'c', {
  value: 5,
  writable: false,
  enumerable: true,
  configurable : false
});
console.log(obj.c); // 5
obj.c = 7;
console.log(obj.c); // writable이 false이므로 5
for (var x in obj) {
    console.log(x); // c의 enumerable이 true이니까 c는 반환
}
delete obj.c
console.log(obj.c); // configurable이 false이므로 c는 지워지지 않고 5 출력
```

defineProperty

- get, set 속성
  - get은 getter 기능을 제공하고 set 속성은 setter 기능을 제공한다.

```
var obj = {};
Object.defineProperty(obj, "book", {
  get: function(){
    return "책";
  }
});

console.log(obj.book);
```

책

```
var obj = {};
Object.defineProperty(obj, "item", {
  set: function(param){
    this.sports = param;
  }
});

obj.item = "야구";
console.log(obj.sports);
```

야구

defineProperty

- get, set 속성
  - getter, setter 한꺼번에 사용.

```
var obj = {a : 10};
Object.defineProperty(obj, "roll", {
  get: function(){
    return this.a;
  },
  set: function(a){
    this.a = a+3
  }
});


console.log(obj.roll); // 10
obj.roll = 3;
console.log(obj.roll); // 6
```

defineProperty

- get, set 속성

```
var obj = {};
Object.defineProperties(obj, {
  a: {
    value: 5,
    writable: false,
    enumerable: true,
  },
  b: {
    get: function() {
      return 'kim';
    },
    set: function(value) {
      console.log(this, value);
      this.a = value;
    },
    enumerable: false,
    configurable: false,
  },
});
```

```
console.log(obj.a); // 5
console.log(obj.b); // 'kim'
obj.a = 10;
console.log(obj.a); // writable이 false라 그대로 5
for (var key in obj) {
  console.log(key); // b의 enumerable이 false이니까 a만 log됨
}
obj.b = 15; // 15로 설정되는 대신 set의 내용이 실행됨. set의 value는 15
console.log(obj.a); // this.a = value로 인해 15로 바뀌어야 하나 writable이 false라 무시됨
console.log(obj.b); // 그대로 'kim'
Object.defineProperty(obj, 'b', {
  value: 5
}); // Uncaught TypeError: Cannot redefine property: b // configuration이 false라 갱신 불가
```

defineProperty

- get, set 속성
  - writable은 속성 값을 바꾸는 것을 막지만 만약 속성의 값이 객체인 경우에는 그 객체 안의 속성을 바꾸는 것은 막지 못한다.
  - 바꾸는 것을 전체적으로 막기 위해서 Object.freeze 메소드가 있다.

```
Object.defineProperty(obj, 'c', {
  value: { x: 3, y: 4 },
  writable: false,
  enumerable: true,
});
console.log(obj.c); // { x: 3, y: 4 }
obj.c = 'kim';
console.log(obj.c); // writable이 false라 그대로 { x: 3, y: 4 }
obj.c.x = 5; // 값이 객체인 경우 그 객체의 속성을 바꿈
console.log(obj.c); // { x: 5, y: 4 }로 바뀜
```

defineProperty

- get, set
  - ES6로 들어오면서 getter와 setter 설정방식이 달라진다.

```
let obj = {
  value: 123,
  get getValue(){
    return this.value;
  }
};
console.log(obj.getValue);
```

```
let obj = {
  set setValue(value){
    this.value = value;
  }
};
obj.setValue = 123;
console.log(obj.value);
```

defineProperty

- get, set

```
var obj = {
  a: 5,
  get getA(){
    return this.a;
  },
  set setA(a){
    this.a = a;
  }
};
console.log(obj.getA); // 5
obj.setA = 3;
console.log(obj.getA); // 3
```

getOwnPropertyDescriptor

- getOwnPropertyDescriptor
  - 속성의 설명 값을 불러온다.

```
// { enumerable: false, configurable: false, get: function() {}, set: function(value) {} }
Object.getOwnPropertyDescriptor(obj, 'b');
```

Object.freeze, Object.seal, Object.preventExtensions

- Object.freeze, Object.seal, Object.preventExtensions
  - 위의 예시에서 writable을 false로 해도, value가 객체인 경우에는 객체의 속성을 바꾸는 것을 막지 못한다.
  - Object.freeze를 사용하면 객체 전체를 바꾸지 못하게 고정할 수 있다. 값도 못 바꿀뿐더러, 속성을 추가 또는 제거할 수도 없고, 속성의 설명을 바꿀 수도 없다.

```
var frozenObj = Object.freeze(obj);
frozenObj.a = 10;
console.log(frozenObj.a); // 그대로 5
console.log(delete frozenObj.c); // false
obj.c.x = 5;
console.log(obj.c.x);
Object.freeze(obj.c); // 이것까지 해야 내부 객체까지 완전히 얼려짐
obj.c.x = 7;
console.log(obj.c.x);
```

Object.freeze, Object.seal, Object.preventExtensions

- Object.freeze, Object.seal, Object.preventExtensions
  - Object.seal의 경우는 속성의 추가, 제거를 막고, configurable을 false로 바꾼다.
  - 대신 속성의 값은 writable이 true이기만 하면 바꿀 수 있다.

```
var sealedObj = Object.seal(obj);
sealedObj.a = 10;
console.log(sealedObj.a); // 5로 변경이 안 되지만 writable이 true면 변경 가능
console.log(delete sealedObj.c); // false
```

Object.freeze, Object.seal, Object.preventExtensions

- Object.freeze, Object.seal, Object.preventExtensions
  - 속성의 추가만 막고 싶다면 Object.preventExtensions가 있다.
  - 그 외의 속성 제거, 값 변경, 설정 변경은 가능하다.

```
var nonExtensible = Object.preventExtensions(obj);
nonExtensible.d = 'new';
console.log(nonExtensible.d); // undefined
```

keys

- keys
  - 객체의 속성명을 모두 가져와 배열로 만든다.
  - enumerable이 false인 것은 빠진다.

```
console.log(Object.keys(obj)); // ['a', 'c']
```

Object.isFrozen, Object.isSealed, Object.isExtensible

- Object.isFrozen, Object.isSealed, Object.isExtensible
  - 객체가 freeze 되었는지, sealed 되었는지 또는 preventExtension 상태인지 알려준다.

```
console.log(Object.isFrozen(frozenObj)); // true
console.log(Object.isSealed(sealedObj)); // true
console.log(Object.isExtensible(nonExtensible)); // false
```

Object Descripter

- is()
  - 두 개의 파라미터 값의 값 타입을 비교하여 같으면 true를, 아니면 false를 반환한다.

구분	타입	데이터(값)
형태		Object.is()
파라미터	Any	비교 대상 값
	Any	비교 대상 값
반환	Boolean	값과 값 타입이 같으면 true 아니면 false

  - 값과 값 타입을 비교하는 것이지 오브젝트를 비교하는 것이 아니다.

Object Descripter

- is()
  - 비교하는 방법에는 3가지가 있다.
    - === : 값과 값 타입을 모두 비교한다
    - == : 값 타입은 비교하지 않고 값만 비교한다.
    - Object.is() : 값과 값 타입을 모두 비교한다.

Object Descripter

- is()

```
console.log("1:", Object.is(1, "1"));
console.log("2:", Object.is(NaN, NaN), NaN === NaN);

console.log("3:", Object.is(0, -0), 0 === -0);
console.log("4:", Object.is(-0, 0), -0 === 0);

console.log("5:", Object.is(-0, -0), -0 === -0);
console.log("6:", Object.is(NaN, 0/0), NaN === 0/0);

console.log("7:", Object.is(null, null), null === null);
console.log("8:", Object.is(undefined, null), undefined === null);
```

```
1: false
2: true false
3: false true
4: false true
5: true true
6: true false
7: true true
8: false false
```

Assign

- assign()
  - 객체를 병합할 때 사용한다.
  - object.assign 메서드의 첫 번째 인자는 타겟이며, 두 번째 인자부터 마지막 인자까지는 소스 오브젝트이다.
  - 소스 오브젝트는 타겟 오브젝트에 병합되며 리턴 값으로 타겟 오브젝트를 반환한다.

구분	타입	데이터(값)
형태		Object.assign()
파라미터	Any	Target, 열거 가능 오브젝트
	Any	(선택)sources, 열거 가능 오브젝트, 다수 지정 가능
반환	Any	

Assign

- assign()

```
// 첫번째 파라미터가 null 또는 undefined를 지정하면 TypeError 발생
try {
  let obj = Object.assign(null, {x: 1});
} catch (e) {
  console.log("null 지정 불가");
}
/*
 * - Object.assign() 의 첫번째 파라미터에 123과 같이 Number, Boolean, String, Symbol 같은
 *   값을 지정하면 값 타입의 오브젝트를 생성하고 파라미터 값을 생성한 오브젝트의 [[Primitive:Value]]
 *   에 설정한다.
 * - [[Primitive:Value]]에 설정된 값은 valueOf() 또는 Symbol.toPrimitive로 구할 수 있다.
 * */
console.log(Object.assign(123));
/* - Object.assign()의 두번째 파라미터가 열거 가능한 오브젝트가 아니면 복사하지 않는다.
 * */
console.log(Object.assign(456, 70));
```

```
null 지정 불가
▼ Number
  ▶ __proto__: Number
    [[PrimitiveValue]]: 123
▼ Number
  ▶ __proto__: Number
    [[PrimitiveValue]]: 456
```

Assign

- assign()
  - 객체에 객체를 합쳐서 새로운 객체를 출력한다.
  - 맨 앞의 객체는 실제 합쳐진 객체로 변환되므로 주의.

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

```
var obj1 = {a:1};
var obj2 = {b:2};
var obj3 = {c:3};
var newObj = Object.assign({}, obj1, obj2, obj3);
console.log(newObj); // {a: 1, b: 2, c: 3}
```

```
var obj1 = {a:1};
var obj2 = {b:2};
var obj3 = {c:3};
var newObj = Object.assign(obj1, obj2, obj3);
console.log(newObj); // {a: 1, b: 2, c: 3}
console.log(obj1); // {a: 1, b: 2, c: 3}
```

Assign

- assign()
  - 문자열 과 객체 혹은 Symbol과 객체와의 병합은 가능하지만 문자열과 문자열 간의 병합은 되지 않음

```
console.log(Object.assign("ABC", {one: 1}));

console.log(Object.assign(Symbol("ABC"), {one: 1}));

try {
  let obj = Object.assign("ABC", "ONE");
} catch (e) {
  console.log("파라미터 모두 문자열 사용 불가")
};
```

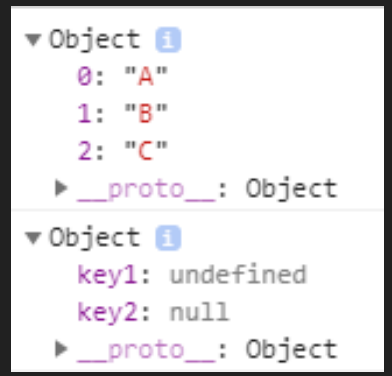
```
▼String
    0: "A"
    1: "B"
    2: "C"
    one: 1
    length: 3
  ▶ __proto__: String
    [[PrimitiveValue]]: "ABC"
▼Symbol
    one: 1
  ▶ __proto__: Symbol
    [[PrimitiveValue]]: Symbol(ABC)
파라미터 모두 문자열 사용 불가
```

Assign

- assign()
  - assign을 통해 undefined와 null을 나열형태로 설정할 경우 해당 속성은 객체 안에서 무시가 된다.
  - 하지만 객체 형태로 집어넣을 경우에는 해당 키에 대한 값으로 들어간다.

```
let oneObj = {};
Object.assign(oneObj, "ABC", undefined, null);
console.log(oneObj);

let twoObj = {};
Object.assign(twoObj, {key1: undefined, key2: null});
console.log(twoObj);
```



Assign

- Assign() 함수의 필요성
  - Object를 변수에 할당하면 프로퍼티가 연동되어 한 쪽의 프로퍼티 값을 바꾸면 다른 한쪽의 프로퍼티 값이 자동으로 바뀐다.
  - 연동을 불허할 경우 Object.assign()으로 복사하면 값이 연동되지 않는다.

```
let sports = {
  event: "축구",
  player: 11
}
let dup = sports;

sports.player = 55;
console.log(dup.player);

dup.event = "농구";
console.log(sports.event);
```

```
55
농구
```

Assign

- Assign() 함수의 필요성
  - 이전에는 프로퍼티 값이 연동되지 않게 하기 위해 키와 값을 따로 복사해서 놓는 방법 말고는 없었다.

```
let sports = {
  event: "축구",
  player: 11
}

let dup = {};
for (var key in sports){
  dup[key] = sports[key];
}
sports.player = 33;
console.log(dup.player);
```

11

Assign

- Assign() 함수의 필요성
  - Assign 함수를 쓰게 되면 위와 같은 프로퍼티 복사의 폐혜를 막을 수 있다.

```
let sports = {
  event: "축구",
  player: 11
};
let dup = Object.assign({}, sports);
console.log(dup.player);

dup.player = 33;
console.log(sports.player);

sports.event = "수영";
console.log(dup.event);
```

```
11
11
축구
```

Assign

- Assign() 주의 사항
  - Assign()에서의 첫번째 오브젝트와 출력하는 오브젝트는 동일 오브젝트이다.

```
let oneObj = {one: 1},
    twoObj = {two: 2};
let mergeObj = Object.assign(oneObj, twoObj);

console.log(Object.is(oneObj, mergeObj));

mergeObj.one = 456;
console.log(Object.is(oneObj, mergeObj));
```

```
true
true
```

Assign

- Assign() 주의 사항
  - 복사하려는 오브젝트의 키 값이 같다면 오른쪽의 프로퍼티 값으로 대체된다.

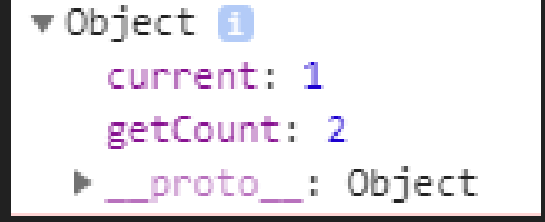
```
let obj = {one: 1};
Object.assign(obj, {two: 2}, {two: 3}, {four: 4});
for (var pty in obj){
  console.log(pty, obj[pty]);
};
```

```
one 1
two 3
four 4
```

Assign

- Assign() 주의 사항
  - 오브젝트의 프로퍼티를 복사할 때 getter이면 함수를 복사하지 않고 함수를 호출하여 반환된 값을 복사한다.
  - return 문을 작성하지 않으면 undefined를 반환한다.

```
let count = {
  current: 1,
  get getCount() {
    return ++this.current;
  }
};
let mergeObj = {};
Object.assign(mergeObj, count);
console.log(mergeObj);
```



setPrototypeOf

- setPrototypeOf() : __proto__
  - prototype에 해당 속성을 추가한다.

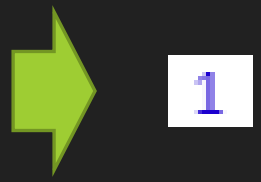
구분	타입	데이터(값)
형태		Object.setPrototypeOf()
파라미터	Object	Object, 오브젝트 또는 인스턴스
	Object	Prototype, 오브젝트 또는 null
반환	Object	첫 번째 파라미터

setPrototypeOf

- setPrototypeOf() : __proto__

```
let protoObj = Object.setPrototypeOf({},{count:1});

console.log(protoObj.count);
```

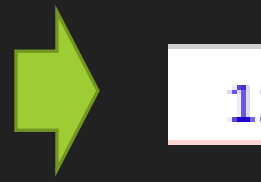


1

```
let Sports = function(){};
Sports.prototype.getCount = function(){
  return 123;
};

let protoObj = Object.setPrototypeOf({}, Sports.prototype);

console.log(protoObj.getCount());
```



123

typeof

- typeof
  - 타입을 알려준다.
  - 배열과 null도 object로 표시되기 때문에 배열을 구분하려면 Array.isArray 메소드를 사용하고, null을 구분하려면 따로 처리해야한다.

```
var a = 1;
var b = 'kim';
var c = true;
var d = {};
var e = [];
var f = function() {};
var g;
var h = null;

console.log(typeof a); // 'number'
console.log(typeof b); // 'string';
console.log(typeof c); // 'boolean';
console.log(typeof d); // 'object';
console.log(typeof e); // 'object';
console.log(typeof f); // 'function';
console.log(typeof g); // 'undefined'
console.log(typeof h); // 'object';
```