코어 자바스크립트

6강 프로토 타입

참고문헌

- 코어자바스크립트
- 모던자바스크립트 DeepDive

프로토타입(prototype)

prototype [proʊtətaɪp]

명사

1. 원형(原型)(archetype); 견본, 전형; (후대 사물의) 선조, 원조(元祖)

   the **prototype** of a character
   (소설에서) 인물의 원형

2. 원형(原形)

자바스크립트는 명령형(imperative), 함수형(functional), 프로토타입 기반(prototype-based) 객체지향 프로그래밍(OOP)를 지원하는 멀티 패러타임 프로그래밍 언어!

*명령형 프로그래밍
컴퓨터가 수행할 명령들을 순서대로 써 놓은 것
*선언형 프로그래밍
프로그램이 무엇을 수행해야 하는지를 설명하고, 그 명령을 실행하는 방법은 컴퓨터가 알아서 결정하게 함
ex) HTML, jQuery

*객체지향 프로그래밍
객체를 프로그램의 요소를 사용하여 문제를 해결
*함수형 프로그래밍
자료 처리를 수학적 함수의 계산으로 취급하고 상태와 가변 데이터를 멀리함(ex 고차함수..)

프로토타입(prototype)

```
var instance = new Constructor();
```

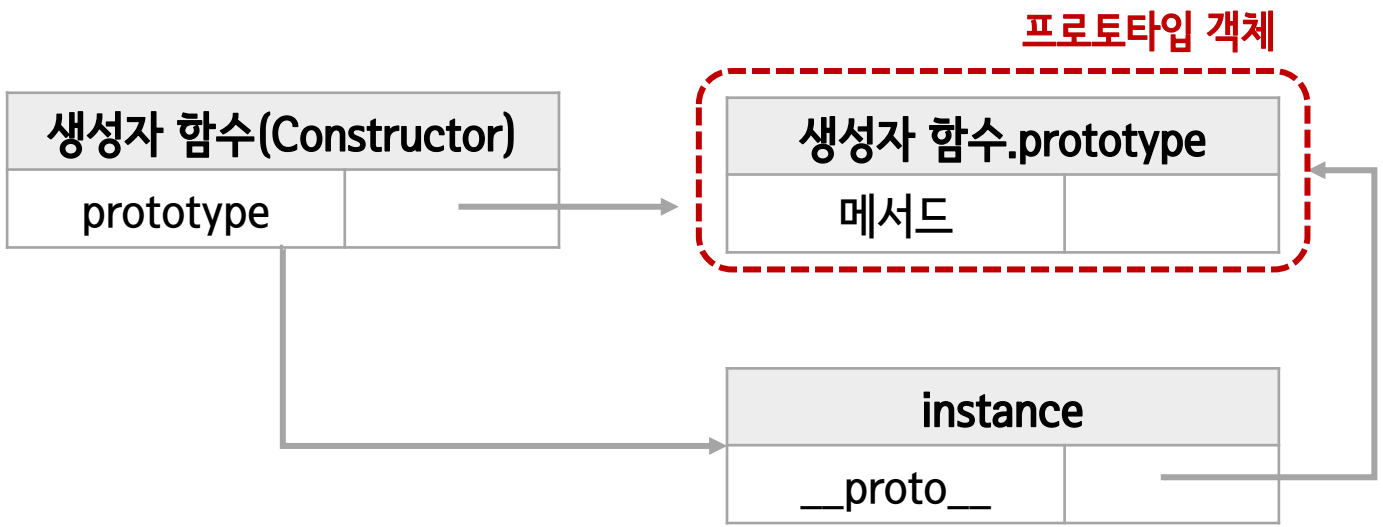
어떤 생성자 함수(Constructor)를 new연산자와 함께 호출하면,

Constructor에서 정의된 내용을 바탕으로 새로운 인스턴스(instance)가 생성

이때 instance에서는 __proto__라는 프로퍼티가 자동으로 부여되는데,

던더 프로토

이 프로퍼티는 Constructor의 prototype이라는 프로퍼티를 참조



프로토타입(prototype) prototype 프로퍼티

prototype 프로퍼티는 함수 객체만 소유. 생성자 함수가 생성할 인스턴스의 프로토타입을 가리킴

```
> (function () {}).hasOwnProperty('prototype');
< true
> console.dir(function(){})
  ▼ƒ anonymous() i
      arguments: null
      caller: null
      length: 0
      name: ""
    ▶prototype: {constructor: ƒ}
      [[FunctionLocation]]: VM273:1
    ▶[[Prototype]]: ƒ ()
    ▶[[Scopes]]: Scopes[1]
```

```
> ({}).hasOwnProperty('prototype');
< false
> console.dir({});
  ▼Object i
    ▶[[Prototype]]: Object
```

프로토타입(prototype) prototype 프로퍼티

non-constructor인 화살표 함수와 ES6 메서드 축약 표현으로 정의한 메서드는 prototype 메서드를 소유하지 않고,
프로토타입을 생성하지 않음!

```
const Person = name => {
  this.name = name;
};
console.log(Person.hasOwnProperty('prototype')); //false
console.log(Person.prototype); //undefined

const obj = {
  foo() {}
};
console.log(obj.foo.hasOwnProperty('prototype')); //false
console.log(obj.foo.prototype); //undefined
```

프로토타입(prototype)

```
var Person = function (name) {
  this._name = name;
};
Person.prototype.getName = function() {
  return this._name;
}
var suzi = new Person('Suzi');
console.log(Person.prototype === suzi.__proto__); //true ·····(1)
console.log(suzi.__proto__.getName()); //undefined ·····(2)
console.log(suzi.__proto__.getAge()); //TypeError ·····(3)
```

(1) instance의 __proto__는 Constructor의 프로퍼티를 참조하기 때문에 true

(2) __proto__에 getName()이라는 함수는 실행할 수 있으나, this가 suzi.__proto__라는 객체를 가리켜서 해당 객체인 { getName: f }에는 _name 프로퍼티가 없기 때문에 undefined라는 결과를 출력 * 함수를 메서드로서 호출할 때 바로 앞의 객체가 this

(3) __proto__에 getAge()라는 함수가 없어 실행할 수 없으므로 TypeError를 출력

프로토타입(prototype)

```
var Person = function (name) {
  this._name = name;
};
Person.prototype.getName = function() {
  return this._name;
}
var suzi = new Person('Suzi');
suzi.__proto__._name = 'SUZI__proto__';
console.log(suzi.__proto__.getName()); // SUZI__proto__
```

__proto__ 에 _name 프로퍼티 추가

BUT getName()을 호출해서 출력하고 싶은 결과는 인스턴스의 _name 프로퍼티임.

```
var suzi = new Person('Suzi', 28);
console.log(suzi.getName()); // Suzi
var iu = new Person('Jieun', 28);
console.log(iu.getName()); // Jieun
```

__proto__없이 인스턴스에서 메서드 사용

프로토타입(prototype)

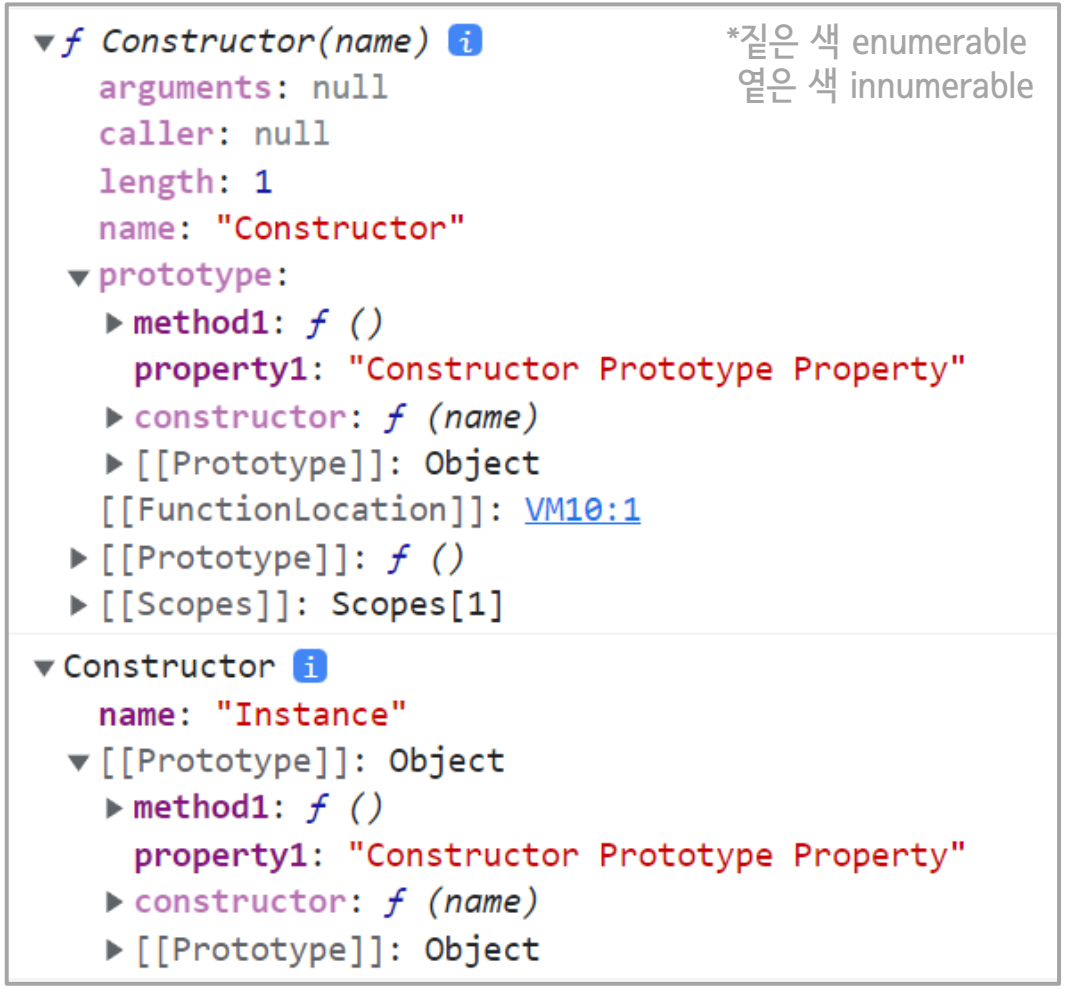
__proto__ 없이 메서드를 바로 쓰면 this는 instance인 건 맞지만 어떻게 메서드를 호출할 수 있는거지?

__proto__는 생략 가능한 프로퍼티.
생성자 함수의 prototype에 어떤 메서드나 프로퍼티가 있다면 인스턴스에서도 마치 자신의 것처럼 해당 메서드나 프로퍼티에 접근할 수 있게 됨

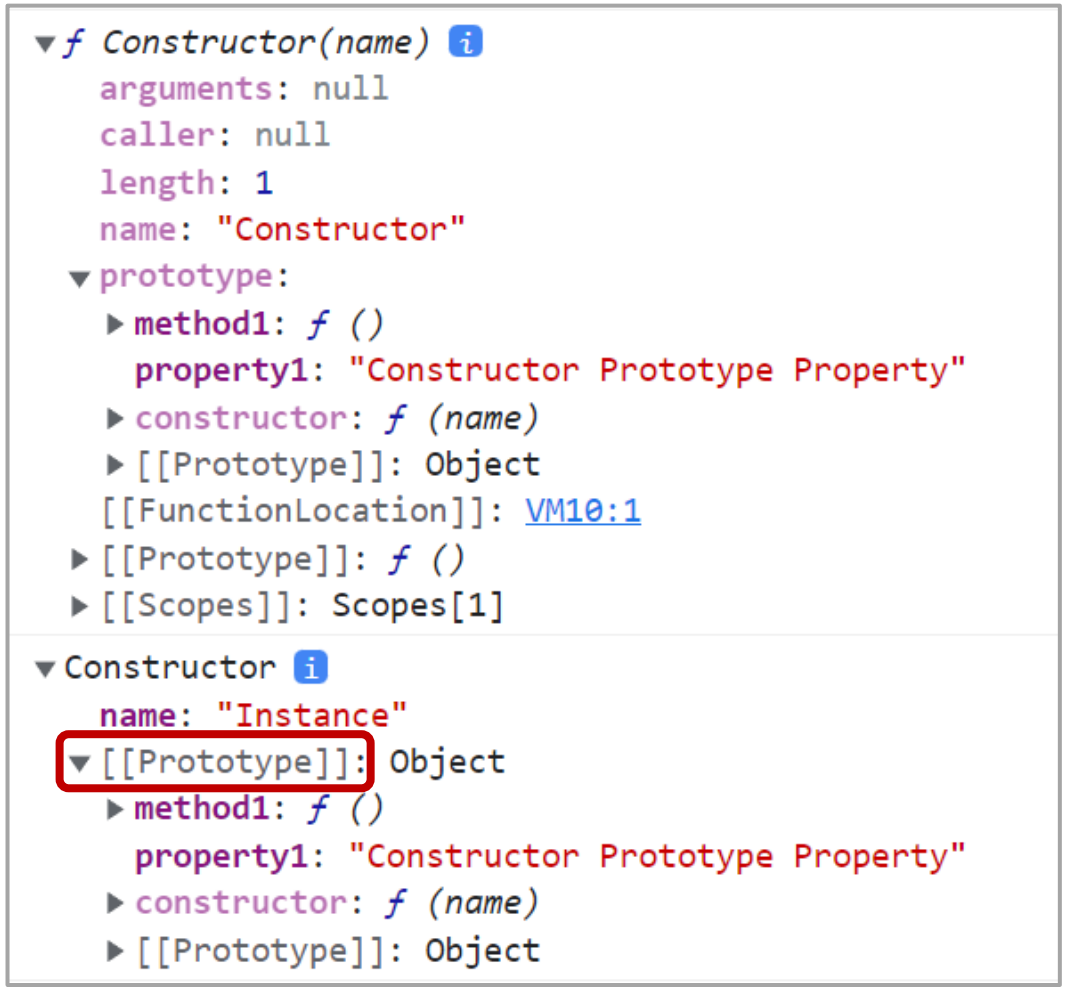
* __proto__를 생략하지 않았을 때의 this는 인스턴스.__proto__지만, __proto__를 생략하게 되면 this는 인스턴스가 된다!

프로토타입(prototype)

```
var Constructor = function (name) {
  this.name = name;
};
Constructor.prototype.method1 = function() {};
Constructor.prototype.property1 = 'Constructor
Prototype Property';
var instance = new Constructor('Instance');
console.dir(Constructor);
console.dir(instance);
```



프로토타입(prototype)



이해가 쉽게 하기 위해 __proto__를 사용했지만 사실은 [[Prototype]]으로 정의되어 있음.

instance.__proto__로 사용하는 것은 성능에 좋지 않고 권장되지 않는 방법임*

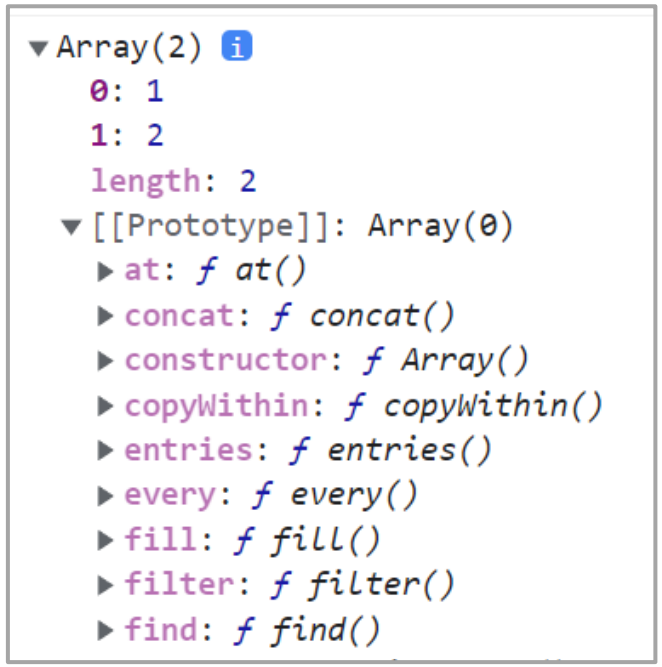
객체의 [[Prototype]]을 설정하는 대신 Object.create()를 사용하여 원하는 [[Prototype]]으로 새 객체를 만들거나, Object.getPrototypeOf()/ Object.setPrototypeOf()를 사용하자.

프로토타입(prototype)

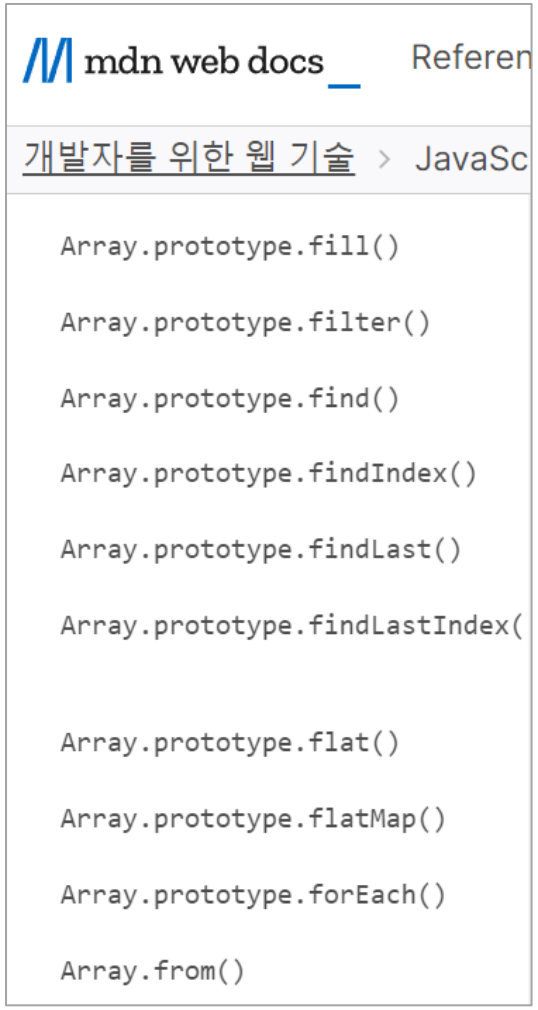
```
var arr = [1,2];
console.dir(arr);
console.dir(Array);
```

```
arr.isArray(); // TypeError
```

* console.dir(arr) 출력결과



* console.dir(Array) 출력결과



프로토타입(prototype) constructor 프로퍼티

```
var arr = [1, 2];
console.log(Array.prototype.constructor); //[Function: Array]
console.log(arr.__proto__.constructor); //[Function: Array]
console.log(arr.constructor); //[Function: Array]

var arr2 = arr.constructor(3,4);
console.log(arr2); // [3,4];
```

prototype 객체 내부에서는 constructor라는 프로퍼티가 있고, constructor 프로퍼티는 원래의 생성자 함수를 참조.

constructor 프로퍼티는 읽기 전용 속성이 부여된 number, string, boolean을 제외하고 값을 바꿀 수 있으나,

construct의 값을 바꾸는 것은 참조 대상이 바뀔 뿐 원래의 인스턴스 원형이 바뀌거나 데이터 타입이 바뀌는 것은 아님!

```
var newConstructor = function(){
  console.log('this is new constructor!');
};
dataTypes = [1, 'test', new Number()];
dataTypes.forEach(function (d) {
  d.constructor = newConstructor;
  console.log(d.constructor.name, d instanceof newConstructor);
})
```

* object instance of constructor

생성자의 프로토타입 속성이 객체의 프로토타입 체인에 나타나는지 여부를 테스트하는 인스턴스

프로토타입(prototype) constructor 프로퍼티

```
var Person = function (name) {
  this.name = name;
};
var p1 = new Person('사람1'); // {name: '사람1'} true
var p1Proto = Object.getPrototypeOf(p1);
var p2 = new Person.prototype.constructor('사람2'); // {name: '사람2'} true
var p3 = new p1Proto.constructor('사람3'); // {name: '사람3'} true
var p4 = new p1.__proto__.constructor('사람4'); // {name: '사람4'} true
var p5 = new p1.constructor('사람5'); // {name: '사람5'} true

[p1, p2, p3, p4, p5].forEach(function (p) {
  console.log(p, p instanceof Person);
});
```

```
[Constructor]
[instance].__proto__.constructor
[instance].constructor
Object.getPrototypeOf([instance]).constructor
[Constructor].prototype.constructor
```

모두 프로토타입 객체의 constructor프로퍼티를 가리킴!

```
[Constructor].prototype
[instance].__proto__
[instance]
Object.getPrototypeOf([instance])
```

프로토타입 객체의 메서드나 프로퍼티에 접근할 수 있음!

프로토타입 체인 메서드 오버라이드

```
var Person = function (name) {
  this.name = name;
};
Person.prototype.getName = function (){
  return this.name;
};
var iu = new Person('지금');
iu.getName = function () {
  return '바로 ' + this.name;
};
console.log(iu.getName()); // 바로 지금
```

프로토타입에 이미 정의된 메서드와 같은 이름의 프로퍼티나 메서드를 인스턴스가 가지고 있는 상황이라면, 메서드 위에 메서드를 덮어씌우는 **메서드 오버라이드** 현상이 일어난다.

자바스크립트 엔진이 메서드를 찾을 때, 가장 가까운 자신의 프로퍼티를 검색하고 없으면 __proto__를 검색하는 순서로 진행되기 때문!

* 지금처럼 메서드 오버라이드가 일어나서 상위의 프로퍼티가 가려지는 현상을 프로퍼티 섀도잉(property shadowing)이라고 함

```
console.log(iu.__proto__.getName()); // undefined
Person.prototype.name = '이지금';
console.log(iu.__proto__.getName()); // 이지금
```

메서드 오버라이드가 일어났을 때, __proto__로 접근하면 프로토타입 객체의 메서드에 접근할 수 있음.

```
console.log(iu.__proto__.getName.call(iu)); // 지금
```

메서드 오버라이드가 일어났을 때, 프로토타입 객체의 메서드를 사용하고 싶다면, call이나 apply 메서드로 protype을 바라보고 있는 this를 인스턴스를 바라보도록 변경하면 됨!

프로토타입 체인

프로토타입 체인(prototype chain)은 어떤 데이터의 __proto__ 프로퍼티 내부에 다시 __proto__ 프로퍼티가 연쇄적으로 이어진 것.
이 체인을 따라가며 검색하는 것을 **프로토타입 체이닝(prototype chaining)**이라고 함

* 어떤 메서드를 호출하면 자바스크립트 엔진에서 자신의 프로퍼티들을
검색해서 원하는 메서드가 있으면 그 메서드를 실행하고, 없으면
__proto__를 검색해서 있으면 그 메서드를 실행하고, 없으면 다시
__proto__를 검색해서 실행하는 식으로 진행

프로토타입 체인

```
console.dir(arr);
▼ Array(2) i
    0: 1
    1: 2
    length: 2
  ▼ [[Prototype]]: Array(0)
    ▶ map: ƒ map()
    ▶ pop: ƒ pop()
    ▶ push: ƒ push()
    ▶ toString: ƒ toString()
    ▶ unshift: ƒ unshift()
    ▼ [[Prototype]]: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
```

배열이 Array.prototype 내부의 메서드와 Obejct.prototype 내부의 메서드를 실행할 수 있음!

```
var arr = [1, 2];
arr(.__proto__).push(3);
arr(.__proto__)(.__proto__).hasOwnProperty(2);
```

프로토 타입 체이닝 예시

```
var arr = [1, 2];
Array.prototype.toString.call(arr); // 1, 2
Object.prototype.toString.call(arr); // [object
array]
arr.toString(); // 1,2

arr.toString = function () {
  return this.join('_');
};
arr.toString(); //1_2
```

프로토타입 체인 객체 전용 메서드의 예외사항

protype은 반드시 객체이기 때문에 Object.prototype은 항상 프로토타입 체인의 최상단. 객체에서 사용할 메서드는 다른 데이터 타입과 달리 프로토타입 객체에 정의 불가능.

```
Object.prototype.getEntries = function() {
  var res = [];
  for (var prop in this){
    console.dir(this);
    if(this.hasOwnProperty(prop)){
      res.push([prop, this[prop]]);
    }
  }
  return res;
};
var data = [
  ['object', {a:1, b:2, c:3}], //['a','1'], ['b','2'], ['c','3']
  ['number', 345], //[]
  ['string', 'abc'] //['0','a'],['1','b'],['2','c']
];
data.forEach(function (d) {
  console.log(d[1].getEntries());
});
```

* Object.prototype.getEntries를 객체에서만 사용하기 위해 정의했으나, 프로토타입 체인에 의 다른 데이터타입에서 항상 Object.prototype객체에 접근할 수 있어서 모든 데이터타입에서 Object.prototype.getEntries를 사용할 수 있음!

프로토타입 체인 객체 전용 메서드의 예외사항

객체 전용 메서드는 Object.prototype.method(인스턴스 메서드)가 아닌 Object.method로 정의되어 있음(스태틱 메서드) *

다른 데이터 타입과 달리 Object와 인스턴스 사이에 this를 통한 연결이 불가능하기 때문에 this를 사용하지 못하고, Object.method(instance)와 같이 대상 인스턴스를 인자로 직접 주입하는 방법으로 구현되어 있음!

```
Object.freeze(obj);

//다른 데이터 타입처럼 인스턴스 객체와 this를 통한 연결이 가능했다면,
//생성자 함수의 prototype에 freeze라는 메서드가 있고,
//아래와 같이 사용 가능했었을지도! but 불가능
instance.freeze();
instance.__proto__.freeze();
```

인스턴스 메서드에 정의된 toString, hasOwnProperty, valueOf, isProtypeOf는 모든 데이터 타입의 변수에서 호출 가능!

프로토타입 체인 Object.create(null)

Object.create(null)을 사용하면 __proto__가 없는 객체를 생성하고, 객체에 반드시 존재하던 내장 메서드 및 프로퍼티들이 제거됨으로써 기본 기능에는 제약이 생기지만, 객체 자체의 무게가 가벼워져 성능상 이점을 갖음!

* 이 때는 __proto__접근자 프로퍼티를 사용할 수 없음!

```
var _proto = Object.create(null);
console.dir(_proto);
```

```
▼Object
    속성 없음
```

```
var _proto = Object.create(null);
_proto.getValue = function(key) {
  return this[key];
};
var obj = Object.create(_proto);
obj.a = 1;
console.log(obj.getValue('a'));
console.dir(obj);
```

```
1
▼Object
    a: 1
  ▼[[Prototype]]: Object
    ▶getValue: ƒ (key)
```

프로토타입 체인 다중 프로토타입 체인

__proto__를 연결해 나가면 무한대로 체인 관계를 이어나갈 수 있음!

```
var Grade = function() {
  var args = Array.prototype.slice.call(arguments);
  for(var i = 0; i < args.length; i++){
    this[i] = args[i];
  }
  this.length = args.length;
};
var g = new Grade(100, 80);
g.push(90); // TypeError
```

유사배열객체에서 배열 메서드를 사용하기 위해
call/apply메서드를 사용
인스턴스에서 배열 메서드를 직접 사용하면 TypeError!

```
Grade.prototype = [];
var g = new Grade(100, 80);
g.push(90); // Grade(3) [100, 80, 90]
```

Grade.prototype이 배열의 인스턴스를 바라보게 하면, Grade의 인스턴스인 g에서 직접 배열의 메서드를 사용할 수 있음!

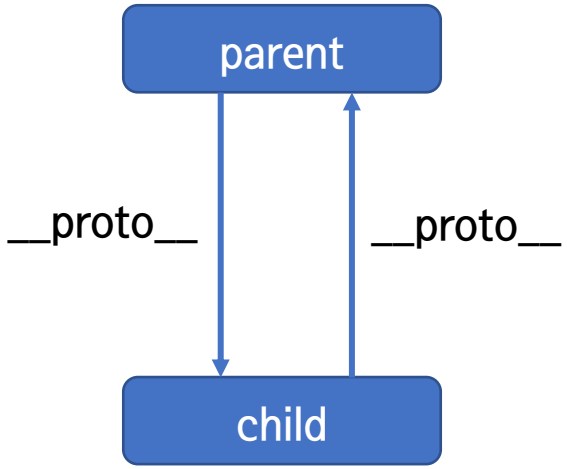
```
console.dir(g);
▼Grade(2) i
    0: 100
    1: 80
    length: 2
  ▼[[Prototype]]: Array(0)
      length: 0
    ▼[[Prototype]]: Array(0)
      ▶at: ƒ at()
      ▶concat: ƒ concat()
      ▶constructor: ƒ Array()
      ▶copyWithin: ƒ copyWithin()
      ▶entries: ƒ entries()
      ▶every: ƒ every()
      ▶fill: ƒ fill()
```

프로토타입 체인 단방향

프로토타입 체인은 단방향 링크드 리스트로 구현되어야 함!

```
const parent = {};
const child = {};

child.__proto__ = parent;
parent.__proto__ = child; // TypeError: Cyclic __proto__ value
```



불가능!!

QnA 지수님 스터디 참고

- 자바스크립트는 클래스 기반 언어와 어떠한 차이가 있나요?
- 프로토타입의 흐름에 대해서 설명해보세요.
- 메서드 오버라이드에 대해서 설명해주세요.
- 프로토타입 체인이란 무엇인가요?
- 프로토타입 체인의 최상위 단계는 무엇인가요?
- 객체에만 적용시킬 수 있는 객체 전용 메서드를 생성하기 위해서는 어떻게 해야하나요?
- constructor의 값을 바꿀 수 있나요?
- typeof Array가 object로 나오는 이유를 프로토타입과 연관지어 설명해주세요