🤍

CH14-18

▼ 14. Prototype

▼ Object-Oriented Programming based Prototype

Java, C++: 클래스 기반 객체지향 프로그래밍 언어

-> 객체 생성 이전에 클래스 정의

Javascript : 프로토타입 기반 객체지향 프로그래밍 언어

-> 클래스 없이도 객체 생성 가능

▼ Prototype vs Class

자바스크립트에는 class 존재 X, 대신 Prototype 존재

class X → 상속 기능 X

⇒ Prototype 기반으로 상속 흉내내도록 구현해 사용

Cf) ECMA6 표준에서 class 문법 추가됨

Quiz)

아래 Person 함수를 변수 kim, park에 상속하면 ?

```
function Person(){
    this.eyes = 2;
    this.nose = 1;
}
```

A)

```
function Person(){
    this.eyes = 2;
    this.nose = 1;
}
var kim = new Person();
var park = new Person();

console.log(kim.eyes);
console.log(kim.nose);

console.log(park.eyes);
console.log(park.nose);
```

kim, park 각자 eyes, nose 속성 가지고 있음

Prototype을 사용하면 ?)

```
function Person(){}

Person.prototype.eyes = 2;
Person.prototype.nose = 1;
```

```
var kim = new Person();
var park = new Person();

console.log(kim.eyes);
```

eyes, nose를 어딘가에 있는 빈 공간에 넣어놓고 kim, park이 공유해서 사용

⇒ 메모리 사용 효율적

▼ Prototype object & Prototype Link

: 이 둘을 통틀어 Prototype이라고 부름

▼ Prototype Object

Quiz)

객체는 언제나 함수(function)으로 생성?

```
function Person(){}
var personObject = new Person();
```

personObject 객체: Person이라는 함수로 생성된 객체

일반적인 객체 생성도 마찬가지,

```
var obj = {};
```

```
var obj = new Object();
```

Object, Function, Array: 자바스크립트에서 모두 함수로 정의되어 있음

함수가 정의될 때는,

1) 해당 함수에 constructor 자격 부여

constructor만 new 사용 가능

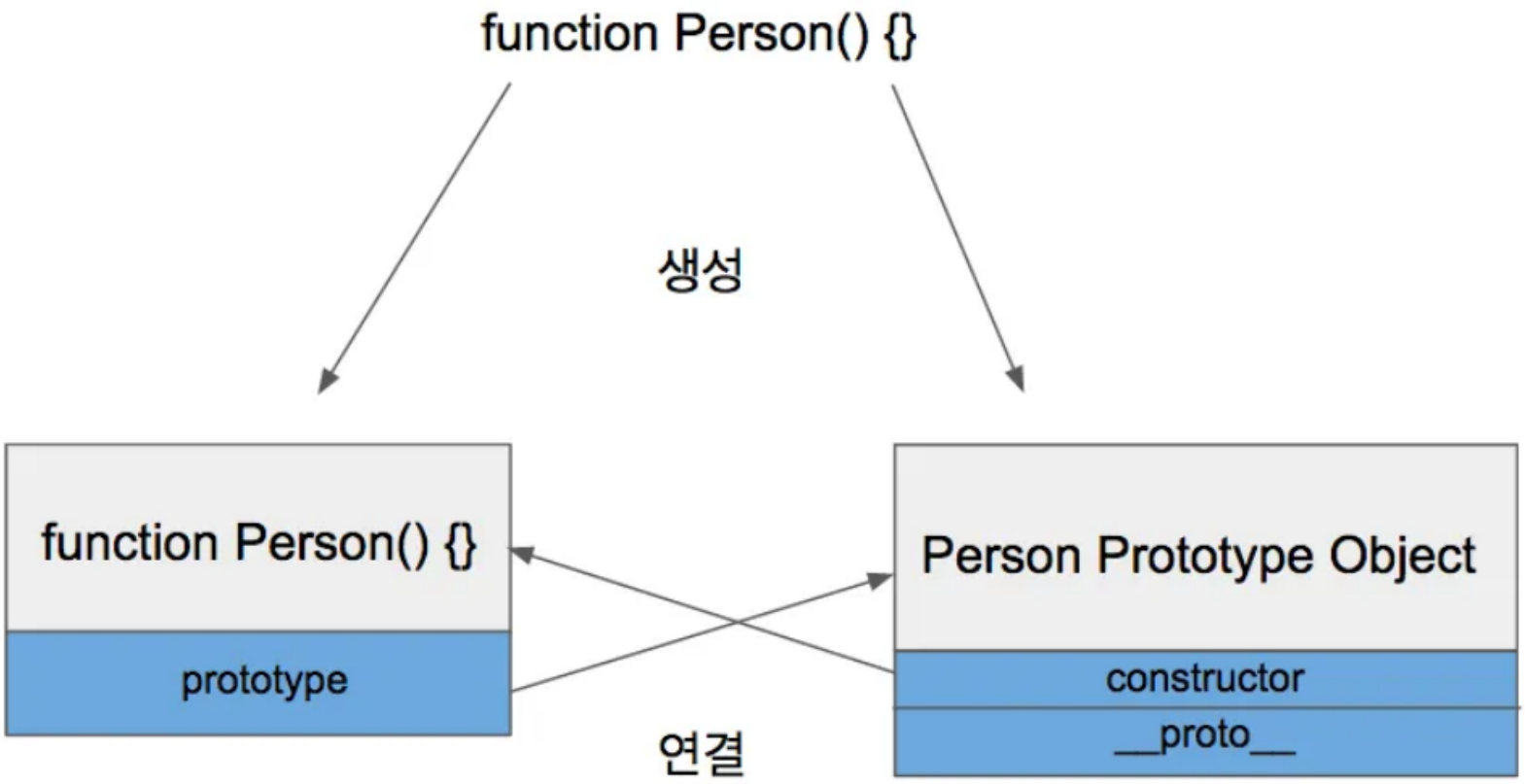
⇒ new를 통해 객체 만들어낼 수 있음

2) 해당 함수의 Prototype Object 생성 및 연결

함수 정의하면, 함수 + Prototype Object 같이 생성

함수

: prototype이라는 속성을 통해 Prototype Object에 접근 가능

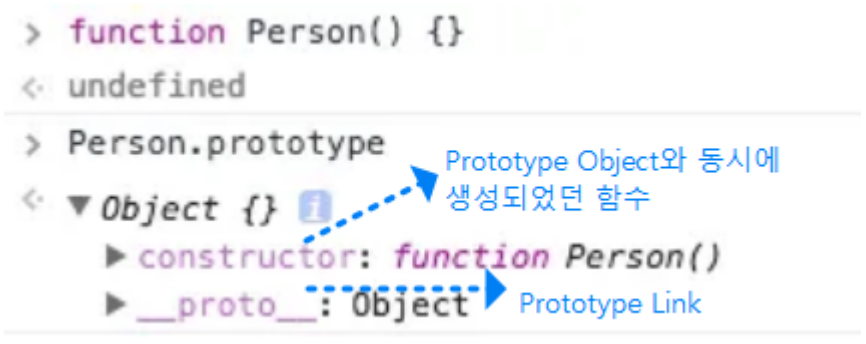


Prototype Object

: 일반적인 객체와 같음

: constructor, __proto__ 를 기본 속성으로 가지고 있음

- constructor = Prototype Object와 함께 생성되었던 함수
- __proto__ = Prototpye Link



Cf) console.dir: object의 속성 계층구조로 출력

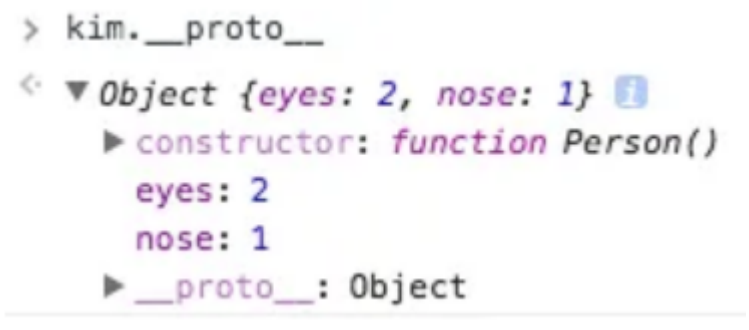
▼ Prototype Link

Quiz) prototype 속성은 함수만 가지고 있다. (O, X)

___proto__ 속성

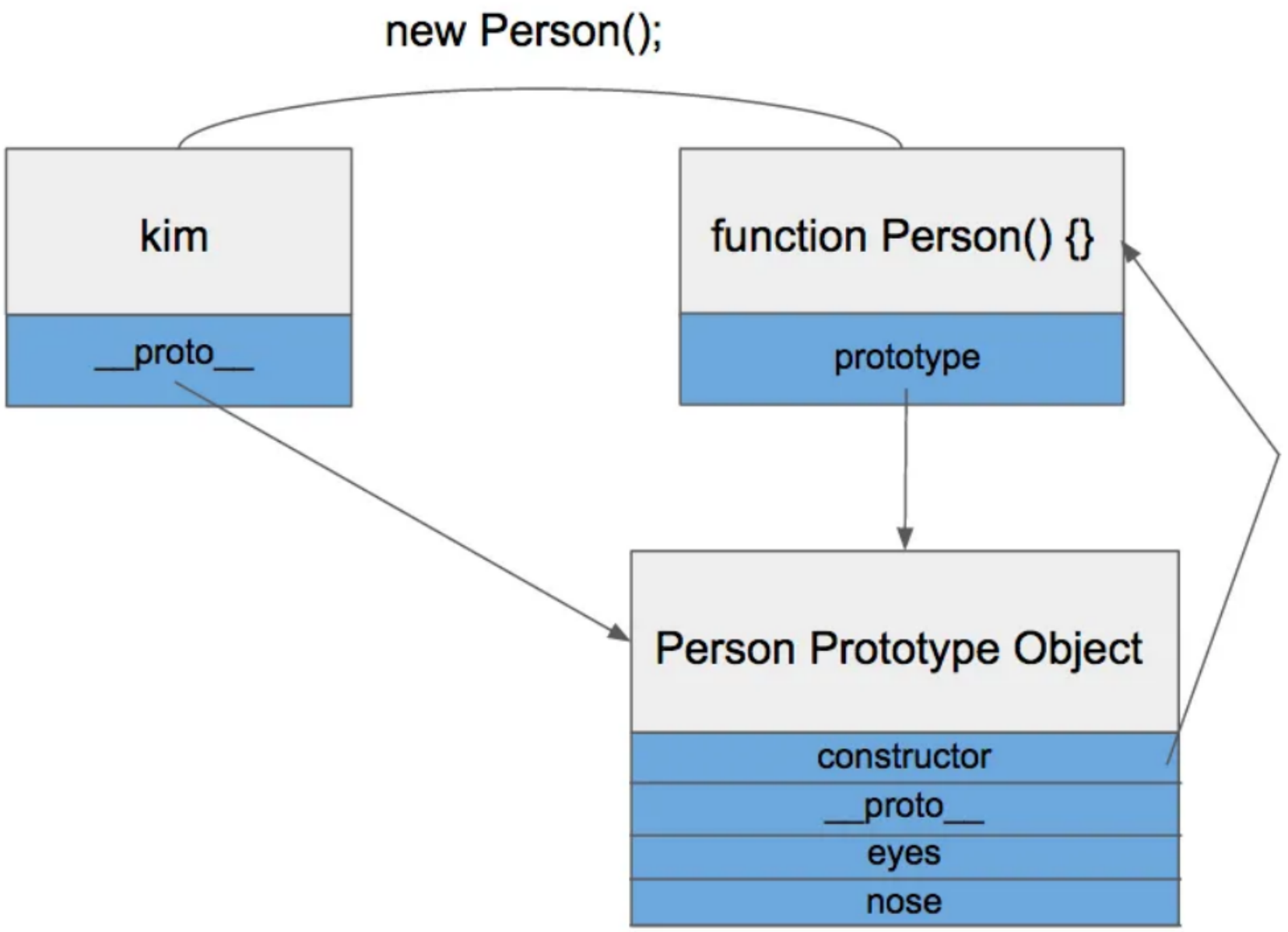
: 모든 객체가 빠짐없이 가지고 있는 속성

: 객체가 생성될 때 조상이었던 함수의 Prototype Object 가리킴



kim 객체: Person함수로부터 생성

Person 함수의 Prototype Object를 가리킴



▼ Prototype Chain

kim 객체가 eyes를 직접 가지고 있지 않음

→ eyes 속성 찾을 때까지 상위 prototype 검색, 최상위인 Object의 Prototype Object까지 도달했는데도 못찾을 경우, undefined 리턴

⇒ 이렇게 __proto__ 속성을 통해 상위 prototype과 연결된 형태

‘Prototype Chain’

→ 이런 Chain 구조 때문에, 모든 객체 = Object 의 자식

⇒ Object Prototype Object에 있는 모든 속성 사용 가능

```
> Object.prototype
◂ ▼ Object {} ℹ
    ▶ __defineGetter__: function __defineGetter__()
    ▶ __defineSetter__: function __defineSetter__()
    ▶ __lookupGetter__: function __lookupGetter__()
    ▶ __lookupSetter__: function __lookupSetter__()
    ▶ constructor: function Object()
    ▶ hasOwnProperty: function hasOwnProperty()
    ▶ isPrototypeOf: function isPrototypeOf()
    ▶ propertyIsEnumerable: function propertyIsEnumerable()
    ▶ toLocaleString: function toLocaleString()
    ▶ toString: function toString()
    ▶ valueOf: function valueOf()
    ▶ get __proto__: function get __proto__()
    ▶ set __proto__: function set __proto__()
> kim.toString();
◂ "[object Object]"
```

▼ 15. Scope

: 유효범위 = 변수에 접근하거나 참조할 수 있는 위치

▼ Block-level scope

Recap) 블록이 뭐였지 ?

⇒ 중괄호 {} 안에 있는 코드

Recap) var, let, const의 scope는 ?

```c
int main(void){
    if (1){
        int x = 5;
        printf("x = %d\n, x);
    }
    printf("x = %d\n, x);
    //use of undeclared identifier 'x'
    return 0;
}
```

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

if (true){
    var x = 5;
}
console.log(x); //5

//let, const 사용시,
//Block-level scope 사용 가능
let y = 0;
{
    let y = 1;
    console.log(y); //1
}
console.log(y); //0
```

▼ Scope 구분

변수는 선언 위치(Global, Local)에 의해 scope 가짐

▼ Global scope

: 코드 어디에서든 참조 가능

```c
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  // local variable declaration
  int a, b;

  // actual initialization
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and
```

```javascript
var global = 'global';

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

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

```
g = %d\n", a, b, g);

  return 0;
}
```

☝ **C**: main함수가 시작점 → main 함수 밖에 변수를 선언해야함

Javascript: 특별한 시작점이 없어 전역변수를 선언하기 쉬움

전역 변수의 단점

: 변수 이름 중복 가능

: 의도치 않은 재할당에 의한 상태 변화 → 코드 예측 어려움

⇒ 사용 억제해야함

▼ Local Scope(= Block-level scope)

▼ Function-level scope

: 함수 코드 블록이 만든 scope, 함수 자신과 하위 함수에서만 참조 가능

```
var a = 10; //전역변수

(function(){
        var b = 20; //지역변수
    })();

console.log(a); //10
console.log(b); //'b' is not defined
```

```
var x = 'global';

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

foo(); //local
console.log(x); //global
```

```
var x = 'global'

function foo(){
    var x = 'local';
    console.log(x); // 1)

    function bar(){
        console.log(x); // 2)
    }
    bar();
}
foo();
console.log(x); // 3)
```

⇒ scope chian에 의해 참조 순위 전역변수 x가 뒤로 밀림

```
var x = 10;

function foo(){
    var x = 100;
    console.log(x); // 1)

    function bar(){
        x = 1000;
        console.log(x); // 2)
    }
```

```
    bar();
}
foo();
console.log(x); // 2)
```

▼ Lexical Scope

: scope 결정방식 중 하나

```
var x = 1;

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

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

foo() // 1) ?
bar() // 2) ?
```

1. **Dynamic scope**

   : 함수를 어디서 호출했는지

   → bar()의 상위 스코프 = foo(), 전역 스코프

2. **Lexical scope(Static scope)**

   : 어디서 선언했는지

   → bar()의 상위 스코프 = 전역 스코프

   ⇒ 함수를 선언한 시점에 상위 스코프가 결정됨

▼ 암묵적 전역

```
var x = 10; // 전역 변수

function foo(){
    y = 20;
    console.log(x+y);
}

foo(); // 결과는 ?
```

변수를 선언하지 않은 상태에서 값을 할당하면 자동으로 전역 변수로 취급됨

⇒ y : 전역변수

```
console.log(x);
console.log(y);

var x = 10;

function foo(){
    y = 20;
    console.log(x+y);
}
foo();
```

```
var x = 10;

function foo(){
    y = 20;
    console.log(x+y);
}

foo();

console.log(window.x);
console.log(window.y);
```

```
delete x;
delete y;

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

▼ 전역변수 사용 최소화

1. 전역변수 객체 하나

```
var MYAPP = {}; // 전역 변수 객체

MYAPP.student = { // 변수들이 MYAPP 의 프로퍼티로 정의됨
    name: 'Lee',
    gender: 'male'
};

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

2. 즉시 실행 함수 사용

```
(function(){ // 즉시 실행 함수
    var MYAPP = {};

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

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

    console.log(MYAPP.student.name); // MYAPP 변수 정의 X,  참조 에러 발생
)
```

▼ 16. Strict mode

```
function foo(){
    x = 10;
}

console.log(x); // ?
```

암묵적 전역 변수 → 오류를 발생시키는 원인이 될 수 O

: 이런 잠재적인 오류를 발생시키기 어려운 개발 환경을 만드는 것

: 린트(ex. ESLint) 도구 사용해서 유사한 효과 얻을 수 있음

▼ Strict mode의 적용

‘use strict’; 사용

1. 전역에 적용
2. 함수 단위에 사용

```
'use strict';

function foo(){
    x = 10; // ReferenceError: x is not d
efined
}

foo();3
```

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

    x = 10; // ReferenceError: x is not d
efined
}
foo();
```

```
function foo(){
    x = 10; // 에러 발생 X
    'use strict';
}
```

strict mode: 스크립트 단위로 적용

```
<!DOCTYPE html>
<html>
<body>
    <script>
        'use strict';
    </script>
    <script>
        x = 1; // 에러 발생 X
        console.log(x);
    </script>
    <script>
        'use strict';
        y = 1; // ReferenctError: y is no
t defined
        console.log(y);
    </script>
```

```
(function(){
    var let = 10; // non-strict mode

    function foo(){
        'use strict';
        let = 20; //SyntaxError: Unexpect
ed strict mode reserved word
    }
    foo();
}());
```

strict mode, non-strict mode 혼용하는 것 → 오류 발생 가능
즉시 실행 함수로 감싼 스크립트 단위로 적용하는 것이 바람직함

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

▼ Strict mode가 발생시키는 에러

▼ 암묵적 전역 변수

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

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

▼ 변수, 함수, 매개변수의 삭제

delete: 변수의 삭제 X, object의 프로퍼티 삭제하려는 용도

```
(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; // 함수 삭제 X
    // SyntaxError: Delete of an unqualified identifier in strict mode.
}());
```

매개변수 이름의 중복

▼ with 문 사용

▼ with문

```
with(객체){
    // 객체의 속성에 직접 접근 가능
    // 객체의 속성이나 메소드를 사용할 때 객체 이름을 반복하지 않아도 됨
}
```

ECMAScript 표준에서 ‘with’문의 사용 금지함

EX.

```
var person = {
    name: 'John',
    age: 30,
    city: 'New York'
};

with (person){
    console.log(name);
    console.log(age);
    console.log(city);
}
```

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

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

▼ 일반 함수의 this

```
(function(){
    'use strict';
    function foo(){
        console.log(this); // undefiend
```

```
    }
    foo(); // 일반 함수로 호출 -> this: undefiend
    function Foo(){
        console.log(this); // Foo
    }
    new Foo() // 생성자 함수 호출 -> this: 생성된 객체
}());
```

▼ 17. this

함수가 호출될 때, arguments 객체, this 암묵적으로 전달 받음

```
function square(numeber){

    console.log(arguments);
    console.log(this);

    return number*number;
}
square(2);
```

In Java,

this: 인스턴스 자신(self)를 가리키는 참조변수

: 객체 자신에 대한 참조값

⇒ 매개변수와 객체 자신이 가지고 있는 멤버변수명이 같을 때 구분하기 위함

```
public Class Person {

    private String name;
    public Person(String name){
        this.name = name;
    }
}
```

this.name: 멤버변수

name: 생성자 함수가 전달받은 매개변수

▼ 함수 호출 방식과 this바인딩

함수 호출 방식에 의해 $this$에 바인딩할 object가 동적으로 결정

⇒ '누가 어떻게' 호출

cf) Lexical scope(함수의 상위 스코프 결정)

: 함수를 선언할 때 결정

cf) 함수 & 메소드

함수: function func( ){ } 형태로 정의, func( ) 형태로 호출할 수 있는 함수

메소드: obj.func( ) 형태로 호출할 수 있도록, 객체 내에 정의된 함수

▼ 함수 호출 방식

```
var foo = function(){
    console.dir(this);
};
```

▼ 1. 함수 호출

```
foo(); //window.foo();
```

▼ 전역 객체

: 모든 객체의 유일한 최상위 객체

Browser-side: **window**

Sever-side(Node.js): **global**

: 전역변수를 프로퍼티로 소유

: 글로벌 영역에 선언한 함수 = 전역객체의 프로퍼티로 접근 가능한 전역 변수의 메소드

```
var ga = 'Global variable';

console.log(ga);
console.log(window.ga);

function foo(){ // 글로벌 영역에 선언한 함수
    console.log('invoked!');
}
window.foo(); //전역객체(window)의 프로퍼티
```

▼ this

전역함수, 내부함수 모두 전역 객체에 바인딩

```
function foo(){
    console.log("foo's this: ", this); //window
    function bar(){
        console.log("bar's this: ", this); //window
    }
    bar();
}
foo();

//foo(): 전역함수
//bar(): 내부함수
```

메소드의 내부함수일 경우에도

```
var value = 1;

var obj = {
    value: 100,
    foo: function(){
        console.log("foo's this: ", this); //obj(foo는 메소드니까)
        console.log("foo's this.vale: ", this.value); //100
        function bar(){
            console.log("bar's this: ", this); // window
            console.log("bar's this.value: ", this.value); // 1
        }
        bar();
    }
};

obj.foo();
```

콜백함수의 경우에도

```
var value = 1;

var obj = {
    value: 100,
    foo: function(){
        setTimeout(function(){
            console.log("callback's this: ", this);
            console.log("callback's this.value: ", this.value);
        }, 100);
    }
};

obj.foo();
```

⇒ 내부함수: 어디에서 선언되었든 관계없이 this는 전역객체를 바인딩

▼ 2. 메소드 호출

```
var obj = {foo:foo};
obj.foo();
```

▼ this

메소드 내부의 this

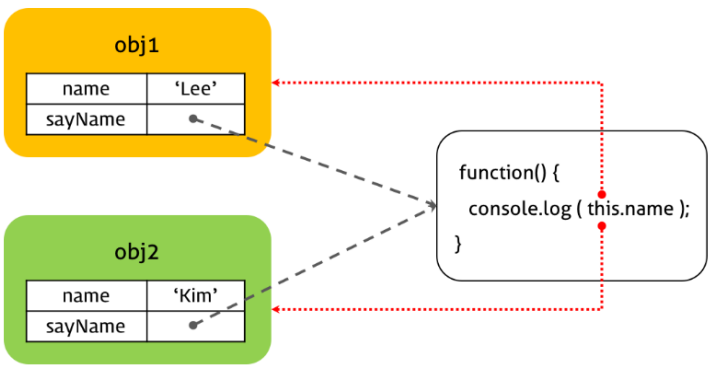
: 해당 메소드를 호출한 객체

```
var obj1 = {
    name: 'Lee',
    sayName: function(){
        console.log(this.name);
    }
}

var obj2 = {
    name: 'Kim'
}

obj2.sayName = obj1.sayName;

obj1.sayName();
obj2.sayName();
```

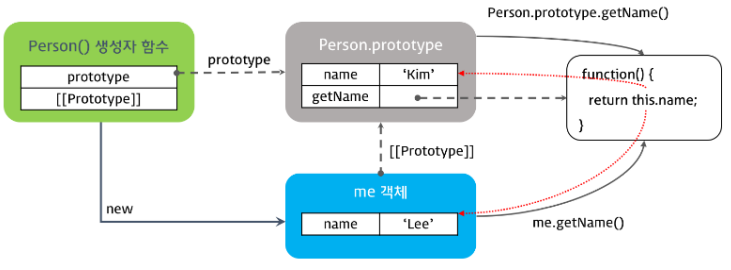


프로토타입 객체 메소드 내부의 this에서도

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

Person.prototype.getName = function
(){
    return this.name;
}

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



```
console.log(me.getName());

Person.prototype.name = 'Kim';
console.log(Person.prototype.getName
());
```

▼ 3. 생성자 함수 호출

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

▼ 생성자 함수

: 기존 함수에 new 연산자 붙여서 호출, 해당 함수가 생성자 함수로 동작

동작 방식

1️⃣ 빈 객체 생성 및 this 바인딩

: 생성자 함수의 코드 실행 전, 빈 객체 생성(생성자 함수가 새로 생성할 객체)

생성자 함수 내의 this = 이 빈 객체

2️⃣ this를 통한 프로퍼티 생성

: this를 사용해서 동적으로 프로퍼티, 메소드 생성 가능

3️⃣ 생성된 객체 반환

1) 반환문이 없을 때,

this에 바인딩된 새로 생성한 객체 반환

2) 반환문(this 아닌 다른 객체)이 있을때,

해당 객체 반환

```
function Person(name){
    // 생성자 함수 코드 실행 전 -- 1
    this.name = name; // ------- 2
    // 생성된 함수 반환 ---------- 3
}

var me = new Person('Lee');
console.log(me.name);
```

▼ 4. apply/call/bind 호출

: 특정 객체에 **명시적**으로 바인딩하는 방법(1~3은 자바스크립트가 암묵적으로 해줌)

⇒ function.prototype.apply, function.prototype.call 메소드로 가능

(function.prototype 객체의 메소드)

```
var bar = { name: 'bar'};
foo.call(bar):
```

```
foo.apply(bar);
foo.bind(bar);
```

```
func.apply(thisArg, [argsArray])

// thisArg: 함수 내부의 this에 바인딩할 객체
// argsArray: 함수에 전달할 argument의 배열
```

⚠️ **apply() 메소드**를 **호출**하는 **주체 = 함수**

apply() 메소드

: this를 특정 객체에 바인딩할 뿐, **본질적인 기능 = 함수 호출**

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

var foo = {};

// apply 메소드는 생성자함수 Person을 호출한다. 이때 this에 객체 foo를 바인딩한다.
Person.apply(foo, ['name']);

console.log(foo); // { name: 'name' }
```

```
Person.apply(foo, [1, 2, 3]);

Person.call(foo, 1, 2, 3);
// call() 메소드: apply()와 기능 같음
//                but, 배열 형태 X 각각 하나의 인자로 넘김
```

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

Person.prototype.doSomething = function (callback) {
  if (typeof callback == 'function') {
    // callback.call(this);
    // this가 바인딩된 새로운 함수를 호출
    callback.bind(this)();
  }
};

function foo() {
  console.log('#', this.name);
}

var p = new Person('Lee');
p.doSomething(foo);  // 'Lee'
```

▼ 18. Closure

: 함수를 일급 객체로 취급하는 함수형 프로그래밍 언어에서 사용되는 중요한 특성

: 반환된 내부함수가 자신이 선언됐을 때의 환경(Lexical enviroment)인 스코프를 기억해, 만일 자신이 선언됐을 때의 환경(스코프) 밖에서 호출되어도 스코프에 접근할 수 있는 함수

```
function outerFunc() {
  var x = 10;
  var innerFunc = function () { console.log(x); };
  innerFunc();
}

outerFunc(); // 10
```

```
function outerFunc() {
  var x = 10;
  var innerFunc = function () { console.log(x); };
  return innerFunc;
}

/**
 *  함수 outerFunc를 호출하면 내부 함수 innerFunc가 반환된다.
 *  그리고 함수 outerFunc의 실행 컨텍스트는 소멸한다.
 */
var inner = outerFunc();
// outerFunc() : innerFunc() 반환 후 생을 마감
inner(); // 10
```

⇒ 자신을 포함하고 있는 **외부함수**보다 **내부함수가 더 오래 유지**되는 경우, **외부 함수밖에서 내부함수**가 호출되더라도 **외부함수의 지역 변수에 접근 가능**

~> 이런 함수를 Closure라고 함

▼ Closure의 활용

▼ 상태 유지

: 현재 상태 기억, 변경된 최신 상태 유지

```
<!DOCTYPE html>
<html>
<body>
  <button class="toggle">toggle</button>
  <div class="box" style="width: 100px; height: 100px; background: red;"></div>

  <script>
    var box = document.querySelector('.box');
    var toggleBtn = document.querySelector('.toggle');

    var toggle = (function () {
      var isShow = false;

      // ① 클로저를 반환
      return function () {
        box.style.display = isShow ? 'block' : 'none';
        // ③ 상태 변경
        isShow = !isShow;
      };
    })();

    // ② 이벤트 프로퍼티에 클로저를 할당
    toggleBtn.onclick = toggle;
  </script>
```

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

▼ 전역 변수 사용 억제

▼ 정보 은닉