* 예습
5/8
5/9, 5/11(1h40)
* 복습
5/12(~p.16)~5/13(p.16~끝)

22. this

TABLE OF CONTENTS

#1. this 키워드 존재 이유 : 객체의 프로퍼티에 접근하기 위해

"19.1. 객체지향 프로그래밍"에서 살펴보았듯이 객체는 상태(state)를 나타내는 프로퍼티와 동작(behavior)을 나타내는 메소드를 하나의 논리적인 단위로 묶은 복합적인 자료 구조이다.

동작을 나타내는 메소드는 자신이 속한 객체의 상태, 즉 프로퍼티를 참조하고 변경할 수 있어야 한다. 이때 메소드가 자신이 속한 객체의 프로퍼티를 참조하려면 먼저 **자신이 속한 객체를 가리키는 식별자를 참조할 수 있어야 한다.**

객체 리터럴 방식으로 생성한 객체의 경우, 메소드 내부에서 메소드 자신이 속한 객체를 가리키는 식별자를 재귀적으로 참조할 수 있다.

JAVASCRIPT

```
const circle = {
  // 프로퍼티: 객체 고유의 상태 데이터
  radius: 5,
  // 메소드: 상태 데이터를 참조하고 조작하는 동작
  getDiameter() {
    // 이 메소드가 자신이 속한 객체의 프로퍼티나 다른 메소드를 참조하려면
    // 자신이 속한 객체 circle 참조할 수 있어야 한다.
    return 2 * circle.radius;
  }
};

console.log(circle.getDiameter()); // 10
```

getDiameter가 호출되어서 getDiameter가 실행되면 circle이 이미 존재하는 상태여야 함.
메소드에서 메소드가 속한 객체인 circle을 볼 수 있음.

getDiameter 메소드 내에서 메소드 자신이 속한 객체를 가리키는 식별자 circle을 참조하고 있다. 이 참조 표현식이 평가되는 시점은 getDiameter 메소드가 호출되어 함수 몸체가 실행되는 시점이다.

객체 리터럴은 할당 단계에 평가된다. 따라서 getDiameter 메소드가 호출되는 시점에는 이미 객체 리터럴의 평가가 완료되어 객체가 생성되었고 식별자 circle에 생성된 객체가 할당된 이후이다. 따라서 메소드 내부에서 식별자 circle을 참조할 수 있다.

=런타임 이전

하지만 자기 자신이 속한 객체를 재귀적으로 참조하는 방식은 일반적이지 않으며 바람직하지도 않다. 생성자 함수 방식으로 인스턴스를 생성하는 경우를 생각해보자.

JAVASCRIPT

```
function Circle(radius) {
  // 이 시점에는 생성자 함수 자신이 생성할 인스턴스를 가리키는 식별자를 알 수 없다.
  ????.radius = radius;
}

Circle.prototype.getDiameter = function () {
  // 이 시점에는 생성자 함수 자신이 생성할 인스턴스를 가리키는 식별자를 알 수 없다.
  return 2 * ????.radius;
};

// 생성자 함수로 인스턴스를 생성하려면 먼저 생성자 함수를 정의해야 한다.
const circle = new Circle(5);
```

생성자 함수 내부에서는 프로퍼티 또는 메소드를 추가하기 위해 자신이 생성할 인스턴스를 참조할 수 있어야 한다. 하지만 생성자 함수에 의한 객체 생성 방식은 먼저 생성자 함수를 정의한 이후, new 연산자와 함께 생성자 함수를 호출하는 단계가 추가로 필요하다. 다시 말해, 생성자 함수로 인스턴스를 생성하려면 먼저 생성자 함수가 존재해야 한다.

따라서 생성자 함수를 정의하는 시점에는 아직 인스턴스를 생성하기 이전이므로 생성자 함수가 생성할 인스턴스를 가리키는 식별자를 알 수 없다. 따라서 자신이 속한 객체 또는 자신이 생성할 인스턴스를 가리키는 특수한 식별자가 필요하다. 이를 위해 자바스크립트는 this라는 특수한 식별자를 제공한다.

this**는 자신이 속한 객체 또는 자신이 생성할 인스턴스를 가리키는 자기 참조 변수**(Self-referencing variable)**이다.** this**를 통해 자신이 속한 객체 또는 자신이 생성할 인스턴스의 프로퍼티나 메소드를 참조할 수 있다.**

this는 자바스크립트 엔진에 의해 암묵적으로 생성되며 코드 어디에서든지 참조할 수 있다. 함수를 호출하면 arguments 객체와 this가 암묵적으로 함수 내부에 전달된다. 함수 내부에서 arguments 객체를 지역 변수처럼 사용할 수 있는 것처럼 this도 지역 변수처럼 사용할 수 있다. 단, this**가 가리키는 값**, 즉 this **바인딩은 함수 호출 방식에 의해 동적으로 결정된다.**

> **바인딩**
> 바인딩(binding)이란 식별자와 값을 연결하는 과정을 의미한다. 예를 들어 변수는 할당에 의해 값이 바인딩된다.

위에서 살펴본 객체 리터럴과 생성자 함수의 예제를 this를 사용하여 수정해 보자.

this는 식별자 비스무리한 것으로 바라봐야 함. 즉, 어떤 값을 가리키고 있다는!

JAVASCRIPT

```
// 객체 리터럴
const circle = {
  radius: 5,
  getDiameter() {
    // this는 메소드를 호출한 객체를 가리킨다.
    return 2 * this.radius;
  }
};

console.log(circle.getDiameter()); // 10
```

[주의] 객체 리터럴 외부에서 호출할 때 this가 아닌 circle객체 사용

객체 리터럴의 메소드 내부에서의 this는 메소드를 호출한 객체, 즉 circle을 가리킨다.

JAVASCRIPT

```
// 생성자 함수
function Circle(radius) {
  // this는 생성자 함수가 생성할 인스턴스를 가리킨다.
  this.radius = radius;
}

Circle.prototype.getDiameter = function () {
  // this는 생성자 함수가 생성할 인스턴스를 가리킨다.
  return 2 * this.radius;
};

// 인스턴스 생성
const circle = new Circle(5);
console.log(circle.getDiameter()); // 10
```

생성자 함수 내부의 this는 생성자 함수가 생성할 인스턴스를 가리킨다. 이처럼 this는 상황에 따라 가리키는 대상이 다르다.

Java, C++와 같은 클래스 기반 언어에서 this는 언제나 클래스가 생성하는 인스턴스를 가리킨다. 하지만 **자바스크립트의 this는 함수가 호출되는 방식에 따라 this에 바인딩될 값, 즉 this 바인딩이 동적으로 결정된다**. 또한 엄격 모드(strict mode) 역시 this 바인딩에 영향을 준다.(“19.6.1. 일반 함수의 this” 참고)

this는 코드 어디든지 참조가능하다. 전역에서도 함수 내부에서도 참조할 수 있다.

JAVASCRIPT

```
// this는 어디서든지 참조 가능하다.
// 전역에서 this는 전역 객체 window를 가리킨다.
console.log(this); // window

function square(number) {
  // 일반 함수 내부에서 this는 전역 객체 window를 가리킨다.
  console.log(this); // window
  return number * number;
}
square(2);

const person = {
  name: 'Lee',
```

->이건 호출되야 정해지는 거임. 호출되기 전까지는 모름.
즉 함수 호출 방식에 의해 this가 정해진다.
this : 함수 호출 시 동적으로
함수 스코프 : 함수 정의 시 정적으로

```
  getName() { 메소드 축약 표현->따라서 non-construtor->prototype프로퍼티가 없음
    // 메소드 내부에서 this는 메소드를 호출한 객체를 가리킨다.
    console.log(this); // {name: "Lee", getName: f}
    return this.name;
  }
};
console.log(person.getName()); // Lee

function Person(name) {
  this.name = name;       //name 프로퍼티 동적추가
  // 생성자 함수 내부에서 this는 생성자 함수가 생성할 인스턴스를 가리킨다.
  console.log(this); // Person {name: "Lee"}
}

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

여기서의 this는 빈 객체임

하지만 this는 객체의 프로퍼티나 메소드를 참조하기 위한 자기 참조 변수이므로 일반적으로 객체의 메소드 내부 또는 생성자 함수 내부에서만 의미가 있다. 따라서 strict mode가 적용된 일반 함수 내부의 this에는 undefined가 바인딩된다. 일반 함수 내부에서 this를 사용할 필요가 없기 때문이다.

2. 함수 호출 방식과 this 바인딩

this가 가리키는 값, 즉 this 바인딩은 함수의 호출 방식, 즉 함수가 어떻게 호출되었는지에 따라 동적으로 결정된다.

렉시컬 스코프와 this 바인딩은 결정 시기가 다르다.
함수의 상위 스코프를 결정하는 방식인 렉시컬 스코프(Lexical scope)는 함수 정의가 평가되어 함수 객체가 생성되는 시점에 상위 스코프를 결정한다. this에 바인딩될 객체는 함수 호출 시점에 결정된다.

= 정적 = 동적

함수를 호출하는 방식은 아래와 같이 다양하다.

1. 일반 함수 호출
2. 메소드 호출
3. 생성자 함수 호출
4. Function.prototype.apply/call/bind 메소드에 의한 간접 호출

주의할 것은 동일한 함수도 다양한 방식으로 호출할 수 있다는 것이다. 아래 예제를 살펴보자.

JAVASCRIPT

```
Q. 메소드/ 생성자 함수로 호출하려면 각각 어떻게?
// this에 바인딩될 객체는 함수 호출 방식에 따라 동적으로 결정된다.
const foo = function () {
  console.dir(this);
};

// 동일한 함수도 다양한 방식으로 호출할 수 있다.

// 1. 일반 함수 호출
// foo 함수를 일반적인 방식으로 호출
// this는 전역 객체 window를 가리킨다.
foo(); // window

// 2. 메소드 호출 ->호출하려면 객체를 하나 생성하면 됨.
// foo 함수를 프로퍼티의 값으로 할당하여 호출
// this는 메소드를 호출한 객체 obj를 가리킨다.
const obj = { foo };
obj.foo(); // obj

// 3. 생성자 함수 호출
// foo 함수를 new 연산자와 함께 생성자 함수로 호출
// this는 생성자 함수가 생성한 인스턴스를 가리킨다.
new foo(); // foo {}

// 4. Function.prototype.apply/call/bind 메소드에 의한 간접 호출
// this는 인수에 의해 결정된다.
const bar = { name: 'bar' };

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

다양한 함수의 호출 방식에 따라 어떻게 this에 바인딩될 객체가 결정되는지 알아보자.

2.1. 일반 함수 호출

기본적으로 this에는 전역 객체(Global object)가 바인딩된다.

JAVASCRIPT

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

<실제 출력값>

```
foo's this:  ▶ Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}
bar's this:  ▶ Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}
```

위 예제처럼 전역 함수는 물론이고 중첩 함수를 **일반 함수로 호출하면 함수 내부의 this에는 전역 객체가 바인딩된다.** 다만, this는 객체의 프로퍼티나 메소드를 참조하기 위한 자기 참조 변수이므로 객체를 생성하지 않는 일반 함수에서 this는 의미가 없다. 따라서 아래 예제처럼 strict mode가 적용된 일반 함수 내부의 this에는 undefined가 바인딩된다.

JAVASCRIPT

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

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

주의!!

메소드 내에서 정의한 중첩 함수도 일반 함수로 호출되면 중첩 함수 내부의 this에는 전역 객체가 바인딩된다. -> 그냥 메소드 내의 중첩 함수가 아닌 메소드 내에서 '정의한'!!

JAVASCRIPT

```
// var 키워드로 선언한 변수 value는 전역 객체의 프로퍼티이다.
var value = 1;
// const 키워드로 선언한 변수 value는 전역 객체의 프로퍼티가 아니다.
// const value = 1;
```

-> 전역 변수는 맞으나 전역 객체의 프로퍼티는 아님

```
const obj = {
  value: 100,
  foo() {                                메소드를 호출한 객체 obj
    console.log("foo's this: ", this);  // {value: 100, foo: f}
    console.log("foo's this.value: ", this.value); // 100

    // 메소드 내에서 정의한 중첩 함수
    function bar() {
      console.log("bar's this: ", this); // window
      console.log("bar's this.value: ", this.value); // 1
    }

    // 메소드 내에서 정의한 중첩 함수도 일반 함수로 호출되면
    // 중첩 함수 내부의 this에는 전역 객체가 바인딩된다.
    bar();
  }
};

obj.foo();
```

콜백 함수 내부의 this에도 전역 객체가 바인딩된다. 어떠한 함수라도 일반 함수로 호출되면 this에 전역 객체가 바인딩된다. 즉, 함수가 어디에 있느냐, 어떤 함수냐가 중요한 것이 아니라, 일반 함수로 호출 되기만 하면 this는 전역 객체가 바인딩됨.

JAVASCRIPT

```
var value = 1;

const obj = {
  value: 100,
  foo() {
    console.log("foo's this: ", this); // {value: 100, foo: f}
    // 콜백 함수 내부의 this에는 전역 객체가 바인딩된다.
    setTimeout(function () {  => 콜백함수(setTimeout함수의 첫번째 매개변수에 전달된 함수)
      console.log("callback's this: ", this); // window
      console.log("callback's this.value: ", this.value); // 1
    }, 100);
  }
};
```

```
obj.foo();
```

setTimeout 함수
setTimeout 함수는 두번째 매개변수에 전달한 시간(ms)만큼 대기한 다음, 첫번째 매개변수에 전달한 콜백 함수를 호출하는 타이머 함수이다. 위 예제의 경우, 100ms을 대기한 다음, 콜백 함수를 호출한다.
->호출은 브라우저가 함

이처럼 일반 함수로 호출된 모든 함수(중첩 함수, 콜백 함수 포함) 내부의 this에는 전역 객체가 바인딩된다.

하지만 메소드 내에서 정의한 중첩 함수 또는 메소드에게 전달한 콜백 함수(보조 함수)의 this가 전역 객체를 바인딩하는 것은 문제가 있다. 중첩 함수 또는 콜백 함수(보조 함수)는 외부 함수를 돕는 헬퍼 함수로서 역할하므로 외부 함수의 일부 로직을 대신하는 경우가 대부분이다. 하지만 외부 함수인 메소드와 중첩 함수의 this가 일치하지 않는다는 것은 중첩 함수 또는 콜백 함수(보조 함수)를 헬퍼 함수로 동작하기 어렵게 만든다.

위 예제의 경우, 메소드 내부에서 setTimeout 함수에 전달된 콜백 함수의 this에는 전역 객체가 바인딩된다. 따라서 this.value는 객체 obj의 value 프로퍼티가 아닌 전역 객체의 value 프로퍼티, 즉 window. value를 참조한다. var 키워드로 선언한 전역 변수는 전역 객체의 프로퍼티가 되므로 window. value은 1이다.

메소드 내부의 중첩 함수나 콜백 함수의 this 바인딩을 메소드의 this 바인딩과 일치시키기 위한 방법은 아래와 같다. -> 가장 원시적이면서 심플한 방법

JAVASCRIPT

```
var value = 1;

const obj = {
  value: 100,
  foo() {
    // this 바인딩(obj)를 변수 that에 할당한다.
    const that = this;

    // 콜백 함수 내부에서 this 대신 that을 참조한다.
    setTimeout(function () {
      console.log(that.value); // 100
    }, 100);
  }
```

```
};

obj.foo();
```

위 방법 이외에도 자바스크립트는 this를 명시적으로 바인딩할 수 있는 Function.prototype.apply, Function.prototype.call, Function.prototype.bind 메소드를 제공한다.

JAVASCRIPT

```
var value = 1;

const obj = {
  value: 100,
  foo() {
    // 콜백 함수에 명시적으로 this를 바인딩한다.
    setTimeout(function () {
      console.log(this.value); // 100
    }.bind(this), 100);
  }
};

obj.foo();
```

이 foo함수 내부의 this는 obj 왜냐면 아래 호출할 때 obj.foo()로 호출했으므로

전역 객체

여기서의 this는 obj
이건 함수 외부에 있는 것임.
이 this를 위에 있는 this로
갈아껴서 this를 obj로 맞추는 것임.

Function.prototype.apply, Function.prototype.call, Function.prototype.bind 메소드에 대해서는 “21.2.4 Function.prototype.apply/call/bind 메소드에 의한 간접 호출”에서 살펴보도록 하자.

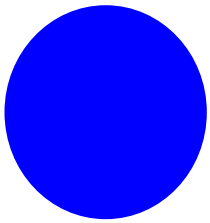
2.2. 메소드 호출

메소드 내부의 this는 메소드를 호출한 객체, 즉 메소드 이름 앞의 마침표(.) 연산자 앞에 기술한 객체에 바인딩된다.

JAVASCRIPT

```
const person = {
  name: 'Lee',
  getName() {
    // 메소드의 this는 메소드를 호출한 객체에 바인딩된다.
```

-> 이 객체 리터럴이 평가되면 객체가 몇 개 만들어질까?
2개임.
왜냐면 이 객체 만들어지고, getName의 객체가 또 하나 더 만들어짐.
getName은 소속된 게 아니라 참조



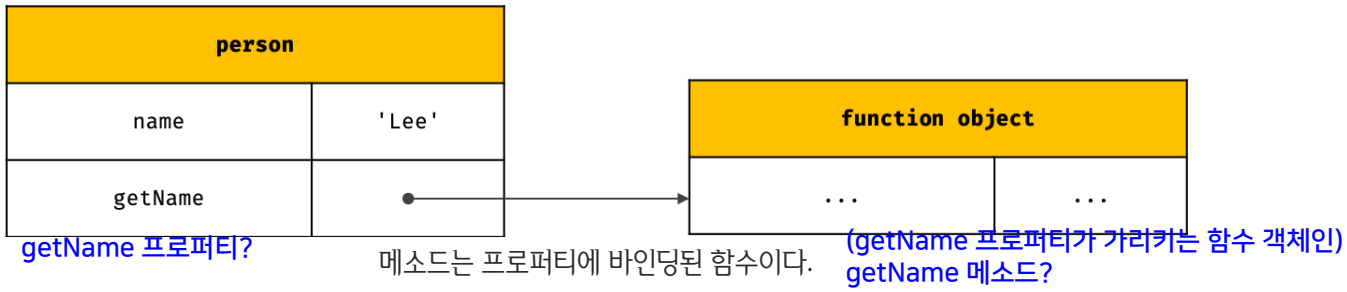
```
    return this.name;
  }
};

// 메소드 getName을 호출한 객체는 person이다.
console.log(person.getName());
```

주의할 것은 메소드 내부의 this는 메소드를 소유한 객체가 아닌 메소드를 호출한 객체에 바인딩된다는 것이다. 아래 예제를 살펴보자.

JAVASCRIPT

```
const person = {
  name: 'Lee',
  getName() {
    // 메소드의 this는 메소드를 호출한 객체에 바인딩된다.
    return this.name;
  }
};

// 메소드 getName을 호출한 객체는 person이다.
console.log(person.getName()); // Lee
```

위 예제의 getName 메소드는 person 객체의 메소드로 정의되었다. 메소드는 프로퍼티에 바인딩된 함수이다. 즉, person 객체와 getName 프로퍼티가 가리키는 함수 객체는 별도의 객체이다. getName 프로퍼티가 생성된 함수 객체를 가리키고 있을 뿐이다.

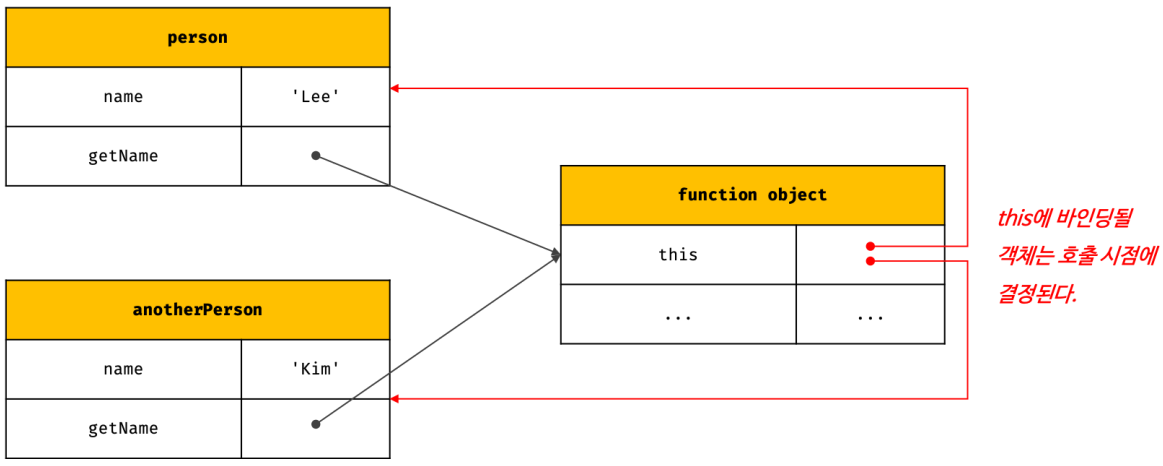


따라서 getName 프로퍼티가 가리키는 함수 객체, 즉 getName 메소드는 다른 객체의 프로퍼티에 할당하는 것으로 다른 객체의 메소드가 될 수도 있고 일반 변수에 할당하여 일반 함수로 호출될 수도 있다.

JAVASCRIPT

```
const person = {
  name: 'Lee',
  getName() {
    // 메소드의 this는 메소드를 호출한 객체에 바인딩된다.
    return this.name;
  }
};

const anotherPerson = {
  name: 'Kim'
};
// 메소드 getName을 anotherPerson 객체의 메소드로 할당
anotherPerson.getName = person.getName;

// 메소드 getName을 호출한 객체는 anotherPerson이다.
console.log(anotherPerson.getName()); // kim

// 메소드 getName을 변수에 할당
const getName = person.getName;

// 메소드 getName을 일반 함수로 호출
console.log(getName()); // ''  -> 일반 함수 내부에서 this : 전역 객체 가리킴
// ⇒ getName 함수 내부에서 참조한 this.name은 window.name과 같다
// window.name은 브라우저 창의 이름을 나타내는 빌트인 프로퍼티이다. window.name의 값은 ''이다.
// 만약 Node.js 환경에서 실행하면 undefined가 출력된다.
```

따라서 메소드 내부의 this는 메소드를 소유한 객체와는 관계가 없고 메소드를 호출한 객체에 바인딩된다.



메소드 내부의 this는 자신을 호출한 객체를 가리킨다.

프로토타입 메소드 내부에서 사용된 this도 일반 메소드와 마찬가지로 해당 메소드를 호출한 객체에 바인딩된다.

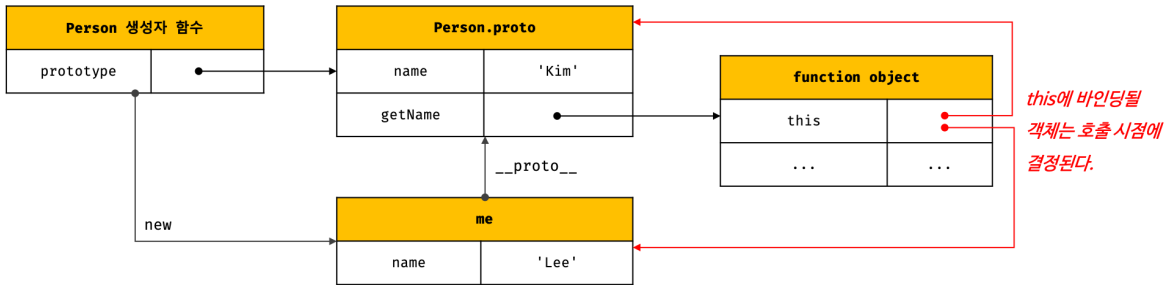
JAVASCRIPT

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

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

const me = new Person('Lee');
// getName 메소드를 호출한 객체는 me이다.
console.log(me.getName()); // ① Lee

Person.prototype.name = 'Kim';
// getName 메소드를 호출한 객체는 Person.prototype이다.
console.log(Person.prototype.getName()); // ② Kim
```

둘 다 호출?

①의 경우, getName 메소드를 호출한 객체는 me이다. 따라서 getName 메소드 내부의 this는 me를 가리키며 this.name은 'Lee'이다.

②의 경우, getName 메소드를 호출한 객체는 Person.prototype이다. Person.prototype도 객체이므로 직접 메소드를 호출할 수 있다. 따라서 getName 메소드 내부의 this는 Person.prototype를 가리키며 this.name은 'Kim'이다.



프로토타입 메소드와 this

2.3. 생성자 함수 호출

생성자 함수 내부의 this에는 생성자 함수가 (미래에) 생성할 인스턴스가 바인딩된다.

JAVASCRIPT

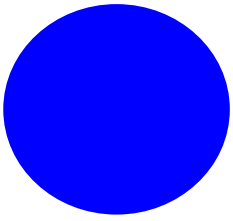
```
// 생성자 함수
function Circle(radius) {
  // 생성자 함수 내부의 this는 생성자 함수가 생성할 인스턴스를 가리킨다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}

// 인스턴스의 생성
// 반지름이 5인 Circle 객체를 생성
const circle1 = new Circle(5);
// 반지름이 10인 Circle 객체를 생성
const circle2 = new Circle(10);

console.log(circle1.getDiameter()); // 10
console.log(circle2.getDiameter()); // 20
```

"17.2. 생성자 함수"에서 살펴보았듯이 생성자 함수는 이름 그대로 객체(인스턴스)를 생성하는 함수이다. 일반 함수와 동일한 방법으로 생성자 함수를 정의하고 new 연산자와 함께 호출하면 해당 함수는 생성자 함수로 동작한다. 만약 new 연산자와 함께 생성자 함수를 호출하지 않으면 생성자 함수가 아니라 일반 함수로 동작한다.

JAVASCRIPT

```
// new 연산자와 함께 호출하지 않으면 생성자 함수로 동작하지 않는다.
// 즉, 일반적인 함수의 호출이다.
const circle3 = Circle(15);

// 일반 함수 Circle은 반환문이 없으므로 암묵적으로 undefined를 반환한다.
console.log(circle3); // undefined

// 일반 함수 Circle내의 this는 전역 객체를 가리킨다.
console.log(radius); // 15
```

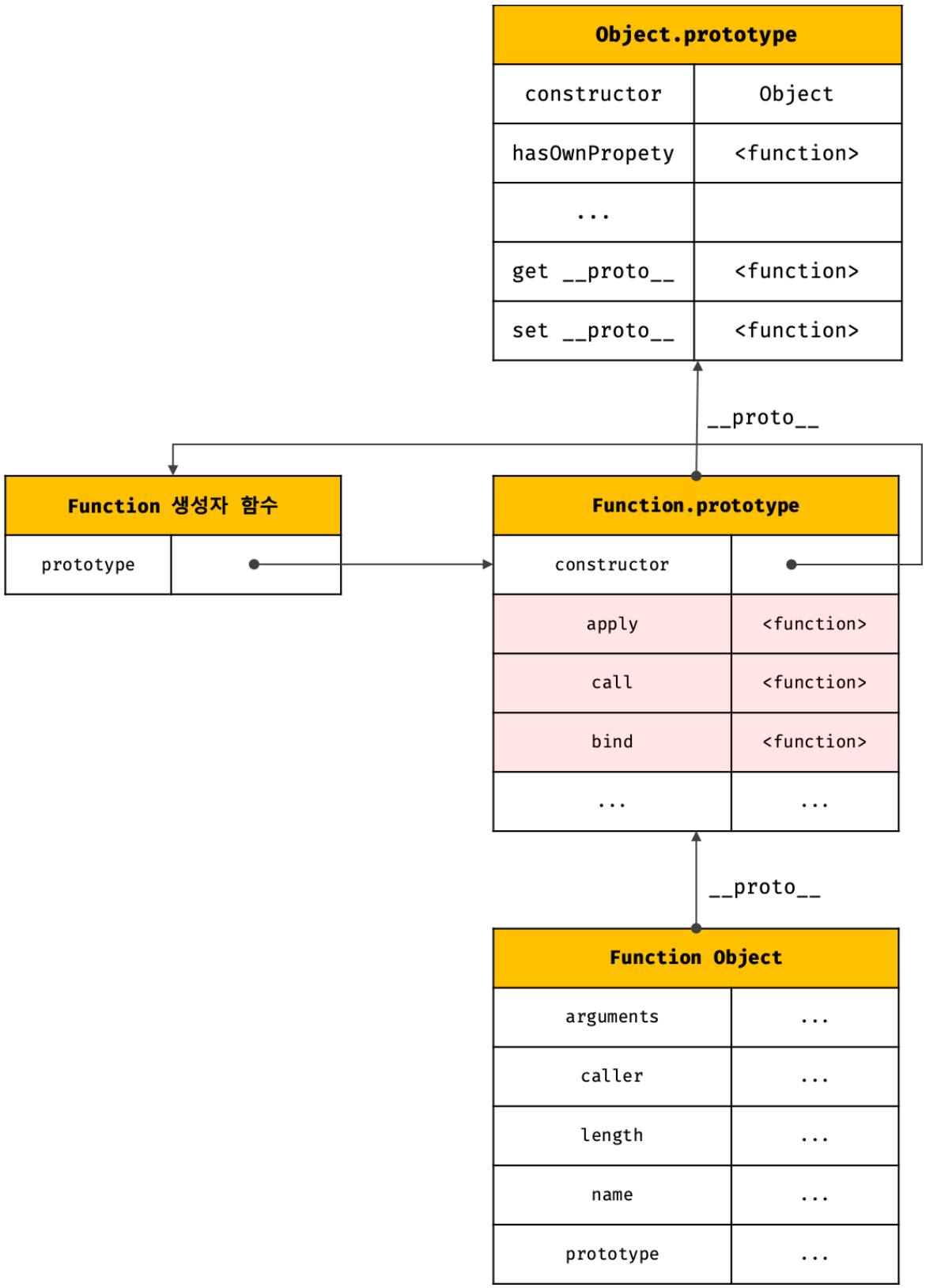
왜 15? window.radius를



2.4. Function.prototype.apply/call/bind 메소드에 의한 간접 호출

-> 이름에서 알 수 있듯이 이 메소드 앞에는 함수가 와야 함.
즉 이 3개의 메소드들은 함수로 호출해야 함.

Function.prototype.apply, Function.prototype.call 메소드는 인수로 this와 인수 리스트를 전달받아 함수를 호출한다. apply와 call 메소드는 Function.prototype의 메소드이다. 즉, apply와 call 메소드는 Function 생성자 함수를 constructor 프로퍼티로 가리키는 모든 함수가 상속받아 사용할 수 있다.



apply, call 메소드는 모든 함수가 상속받아 사용할 수 있다.

apply와 call 메소드의 사용 방법은 아래와 같다.

JAVASCRIPT

```
/**
 * 주어진 this 바인딩과 인수 리스트 배열을 사용하여 함수를 호출한다.
 * @param thisArg - this로 사용될 객체
 * @param argsArray - 함수에게 전달할 인수 리스트의 배열 또는 유사 배열 객체
```

```
 * @returns 호출된 함수의 반환값
 */
Function.prototype.apply(thisArg[, argsArray])
```

JAVASCRIPT

```
/**
 * 주어진 this 바인딩과 인수 리스트를 사용하여 함수를 호출한다.
 * @param thisArg - this로 사용될 객체
 * @param arg1, arg2, ... - 함수에게 전달할 인수 리스트
 * @returns 호출된 함수의 반환값
 */
Function.prototype.call (thisArg[, arg1[, arg2[, ...]]])
```

아래 예제를 살펴보자.

JAVASCRIPT

```
function getThisBinding() {
  return this;
}

// this로 사용할 객체
const thisArg = { a: 1 };

console.log(getThisBinding()); // window
```

-> 일반함수로 호출했으므로 저 위의 return this의 this는 전역객체

```
// 함수(getThisBinding)를 호출하면서 인수로 전달한 객체를 호출한 함수의 this에 바인딩한다.
console.log(getThisBinding.apply(thisArg)); // { a: 1 }
console.log(getThisBinding.call(thisArg)); // { a: 1 }
```

1) apply, call을 호출하면 2) getThisBinding을 호출함 3) getThisBinding의 this를 인수 thisArg로 교체함

-> getThisBinding을 호출하려면 원래 뒤에 ()를 해야 하는데,
이건 getThisBinding을 apply, call 함수를 써서 간접 호출하는 것임.

apply와 call **메소드의 본질적인 기능은 함수를 호출하는 것이다**. apply와 call 메소드는 함수를 호출하면서 첫번째 인수로 전달한 특정 객체를 호출한 함수의 this에 바인딩한다.

apply와 call 메소드는 호출할 함수에 인수를 전달하는 방식만 다를 뿐 동일하게 동작한다. 위 예제는 호출할 함수, 즉 getThisBinding 함수에 인수를 전달하지 않는다. apply와 call 메소드를 통해 getThisBinding 함수를 호출하면서 인수를 전달해 보자.

JAVASCRIPT

```
function getThisBinding() {
  console.log(arguments);
  return this;
}

// this로 사용할 객체
const thisArg = { a: 1 };

// 함수(getThisBinding)를 호출하면서 인수로 전달한 객체를 호출한 함수의 this에 바인딩한다.
// apply 메소드는 호출할 함수의 인수를 배열로 묶어 전달한다.
console.log(getThisBinding.apply(thisArg, [1, 2, 3]));
// Arguments(3) [1, 2, 3, callee: ƒ, Symbol(Symbol.iterator): ƒ]
// { a: 1 }

// call 메소드는 호출할 함수의 인수를 쉼표로 구분한 리스트 형식으로 전달한다.
console.log(getThisBinding.call(thisArg, 1, 2, 3));
// Arguments(3) [1, 2, 3, callee: ƒ, Symbol(Symbol.iterator): ƒ]
// { a: 1 }
```

apply, call : 이 함수를 호출할 때 this를 바꾼다. 이 때 씀.
인수 없으면 일반적으로 call을 씀.
bind : 호출해주지는 않고 인수로 전달받은 객체를 this로 묶어주는 역할만 함.
호출하고 싶으면 별도로 따로 호출해줘야 함.

apply 메소드는 호출할 함수의 인수를 배열로 묶어 전달한다. call 메소드는 호출할 함수의 인수를 쉼표로 구분한 리스트 형식으로 전달한다. 이처럼 apply와 call 메소드는 호출할 함수에 인수를 전달하는 방식만 다를 뿐 this로 사용할 객체를 전달하면서 함수를 호출하는 것은 동일하다.

apply와 call 메소드의 대표적인 용도는 arguments 객체와 같은 유사 배열 객체에 배열 메소드를 사용하는 경우이다. arguments 객체는 배열이 아니기 때문에 Array.prototype.slice와 같은 배열의 메소드를 사용할 수 없으나 apply와 call 메소드를 이용하면 가능하다.

JAVASCRIPT

```
function convertArgsToArray() {
  console.log(arguments);

  // arguments 객체를 배열로 변환
  // slice: 배열의 특정 부분에 대한 복사본을 생성한다.
  const arr = Array.prototype.slice.apply(arguments);
  // const arr = Array.prototype.slice.call(arguments);
  console.log(arr);
```

```
  return arr;
}

convertArgsToArray(1, 2, 3); // [ 1, 2, 3 ]
```

아직 배열에 대해 살펴보지 않았기 때문에 지금은 위 예제를 이해하지 못해도 좋다. “27. 배열”에서 다시 자세히 살펴보도록 하자. 참고로 ES6의 “31. 디스트럭처링 할당”에서도 이에 대해 언급하도록 할 것이다.

Function.prototype.bind 메소드는 apply와 call 메소드와는 달리 함수를 호출하지 않고 this로 사용할 객체만을 전달한다.

JAVASCRIPT

```
function getThisBinding() {
  return this;
}

// this로 사용할 객체
const thisArg = { a: 1 };

// bind 메소드는 함수에 this로 사용할 객체를 전달한다.
// bind 메소드는 함수를 호출하지는 않는다.
console.log(getThisBinding.bind(thisArg)); // getThisBinding
// bind 메소드는 함수를 호출하지는 않으므로 명시적으로 호출해야 한다.
console.log(getThisBinding.bind(thisArg)()); // {a: 1}
```

bind 메소드는 메소드의 this와 메소드 내부의 중첩 함수 또는 콜백 함수의 this가 불일치하는 문제를 해결하기 위해 유용하게 사용된다. 아래 예제를 살펴보자.

JAVASCRIPT

```
const person = {
  name: 'Lee',
  foo(callback) {
    // ①
    setTimeout(callback, 100);
  }
};

person.foo(function () {
  console.log(`Hi! my name is ${this.name}.`); // ② Hi! my name is .
  // window.name은 브라우저 창의 이름을 나타내는 프로퍼티이며 기본값은 ''이다
  // 만약 Node.js 환경에서 실행하면 undefined가 출력된다.
});
```

person.foo의 콜백 함수가 호출되기 이전인 ①의 시점에서 this는 foo 메소드를 호출한 객체, 즉 person 객체를 가리킨다. 그러나 person.foo의 콜백 함수가 일반 함수로서 호출된 ②의 시점에서 this는 전역 객체 window를 가리킨다. 따라서 person.foo의 콜백 함수 내부에서 this.name은 window.name과 같다.

이때 위 예제에서 person.foo의 콜백 함수는 외부 함수 person.foo를 돕는 헬퍼 함수(보조 함수)의 역할을 하기 때문에 외부 함수 person.foo 내부의 this와 콜백 함수 내부의 this가 상이하면 문맥상 문제가 발생한다.

따라서 콜백 함수 내부의 this를 외부 함수 내부의 this와 일치시켜 주어야 한다. 이때 bind 메소드를 사용하여 this를 일치시킬 수 있다.

this로 사용할 객체?

JAVASCRIPT

```
const person = {
  name: 'Lee',
  foo(callback) {
    // bind 메소드로 callback 함수 내부의 this 바인딩을 전달
    setTimeout(callback.bind(this), 100);
  }
};

person.foo(function () {
  console.log(`Hi! my name is ${this.name}.`); // Hi! my name is Lee.
});
```

=person 객체임

지금까지 함수 호출 방식에 따라 this 바인딩이 동적으로 결정되는 것에 대해 살펴보았다. 이를 정리해 보면 아래와 같다.

함수 호출 방식	this 바인딩
일반 함수 호출	전역 객체
메소드 호출	메소드를 호출한 객체
생성자 함수 호출	생성자 함수가 (미래에) 생성할 인스턴스
Function.prototype.apply/call/bind 메소드에 의한 간접 호출	Function.prototype.apply/call/bind 메소드에 인자로 전달한 객체