

19장. 프로토타입 - 1

2022.03.12

서론

자바스크립트는 멀티 패러다임 프로그래밍 언어다.

- 명령형 프로그래밍
- 함수형 프로그래밍
- 객체지향 프로그래밍

서론

클래스 기반 객체지향 프로그래밍 언어

- C++
- Java
- Python
- ...

서론

자바스크립트는 **프로토타입 기반 객체지향** 프로그래밍 언어다

1. 프로토타입을 알아야하는 이유

1. 프로토타입을 알아야하는 이유

1. ES6에서 등장한 클래스는 프로토타입을 통해서 클래스를 흉내낸 문법이다.

1. 프로토타입을 알아야하는 이유

1. ES6에서 등장한 클래스는 프로토타입을 통해서 클래스를 흉내낸 문법이다.
2. 클래스를 지원하지만, 자바스크립트가 클래스 기반 언어가 된 것은 아니다.

1. 프로토타입을 알아야하는 이유

1. ES6에서 등장한 클래스는 프로토타입을 통해서 클래스를 흉내낸 문법이다.
2. 클래스를 지원하지만, 자바스크립트가 클래스 기반 언어가 된 것은 아니다.
3. ES5 이하의 레거시 프론트엔드 코드를 만져야할 경우가 있기 때문이다.

2. 객체지향 프로그래밍

객체?

세상에 존재하는 모든 것을 객체라고 할 수 있다.

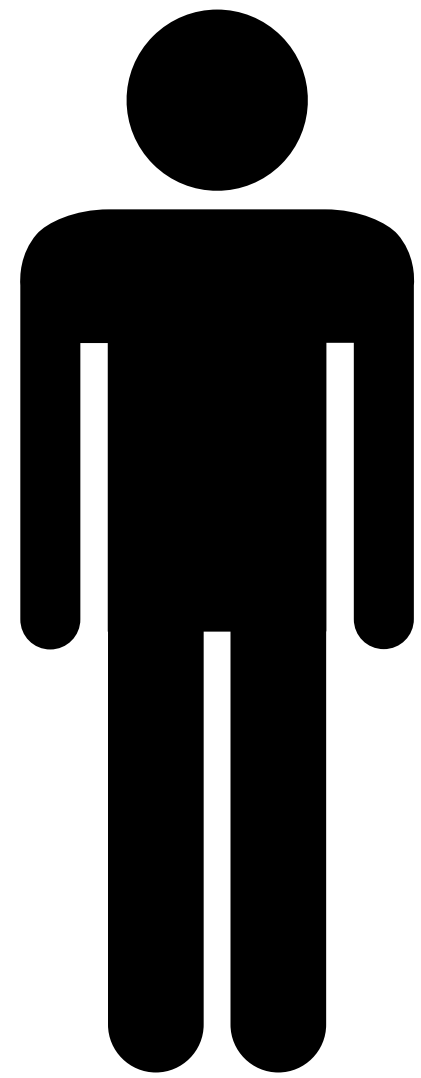
2. 객체지향 프로그래밍

객체?

2. 객체지향 프로그래밍

객체?

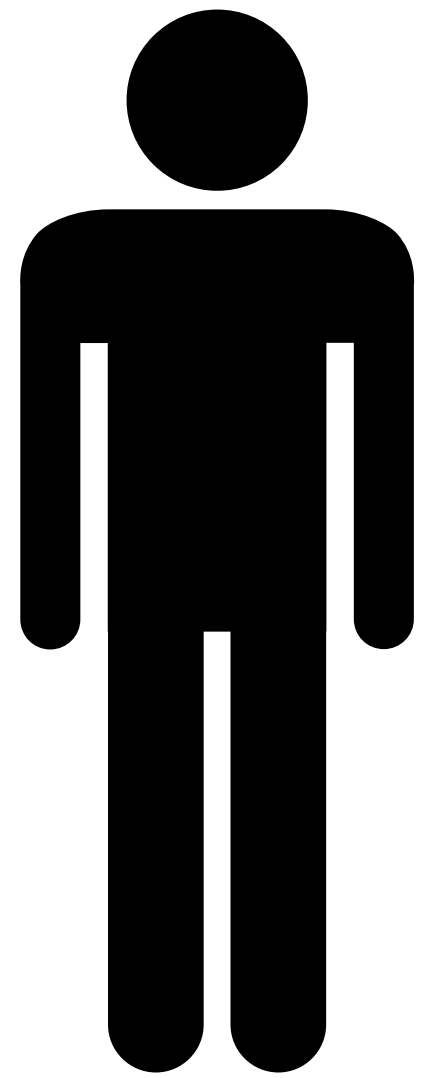
사람



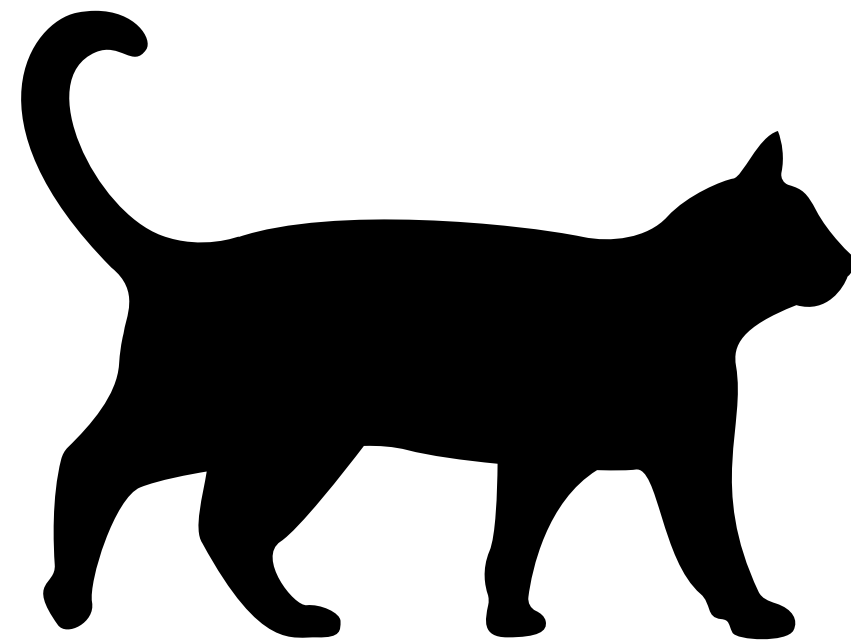
2. 객체지향 프로그래밍

객체?

사람



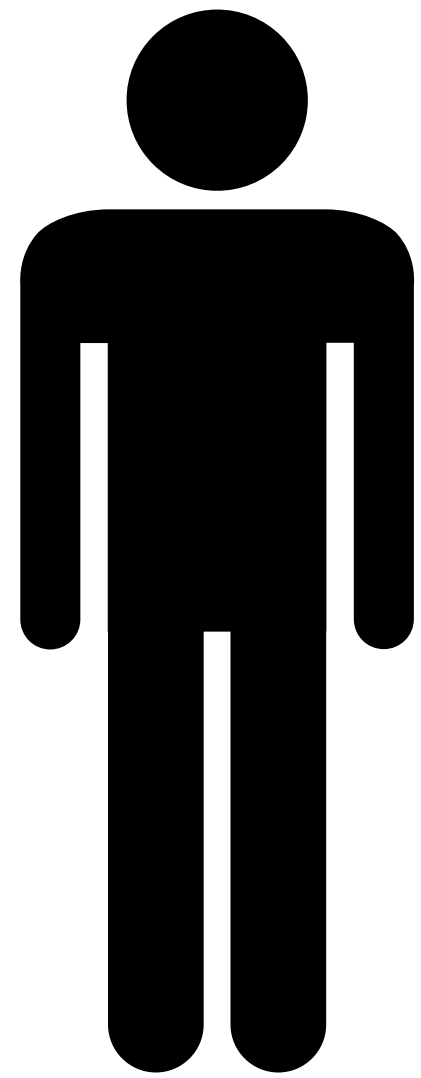
고양이



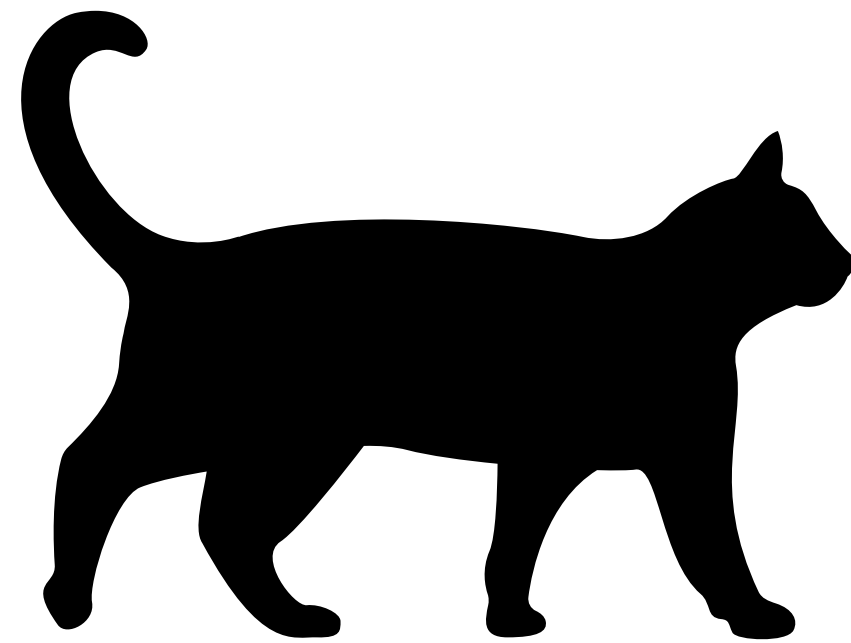
2. 객체지향 프로그래밍

객체?

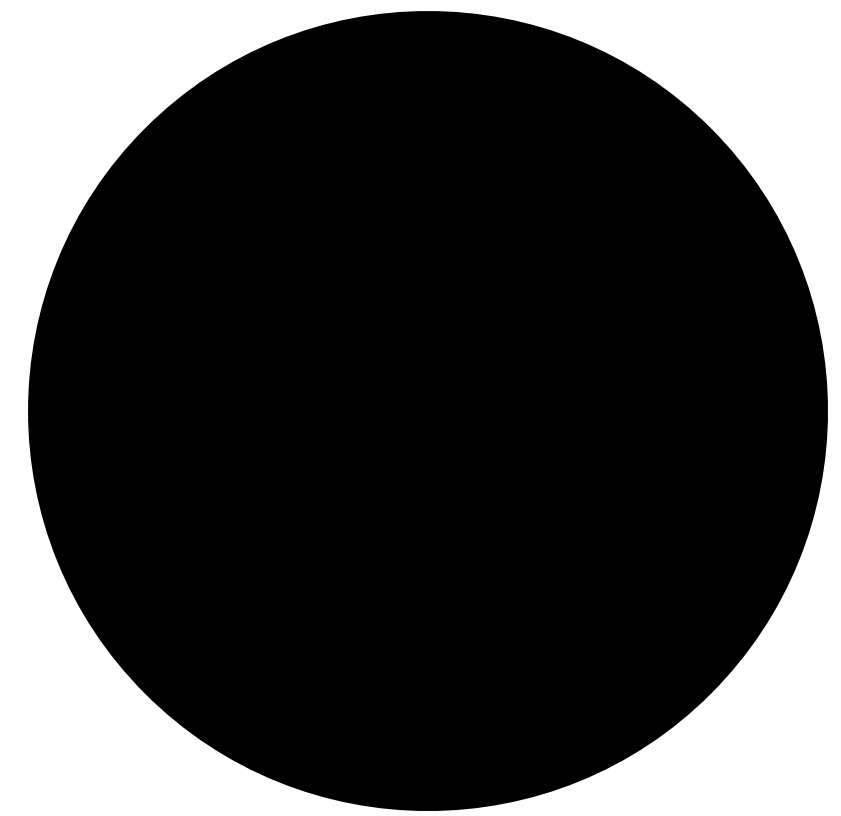
사람



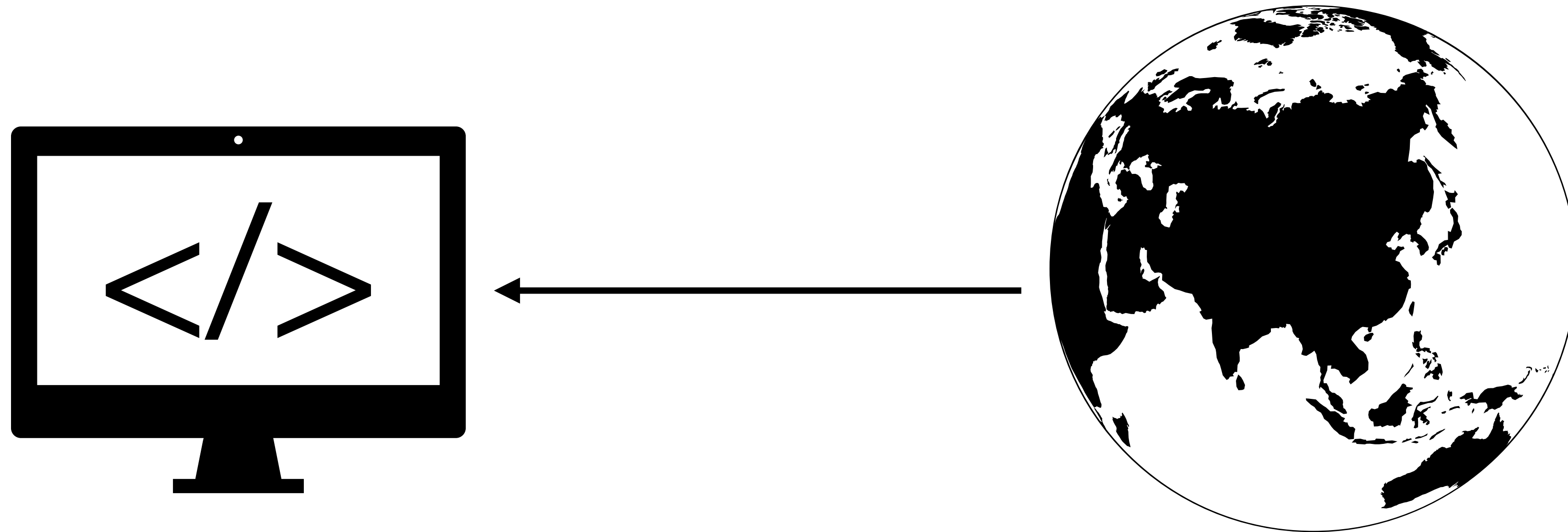
고양이



원

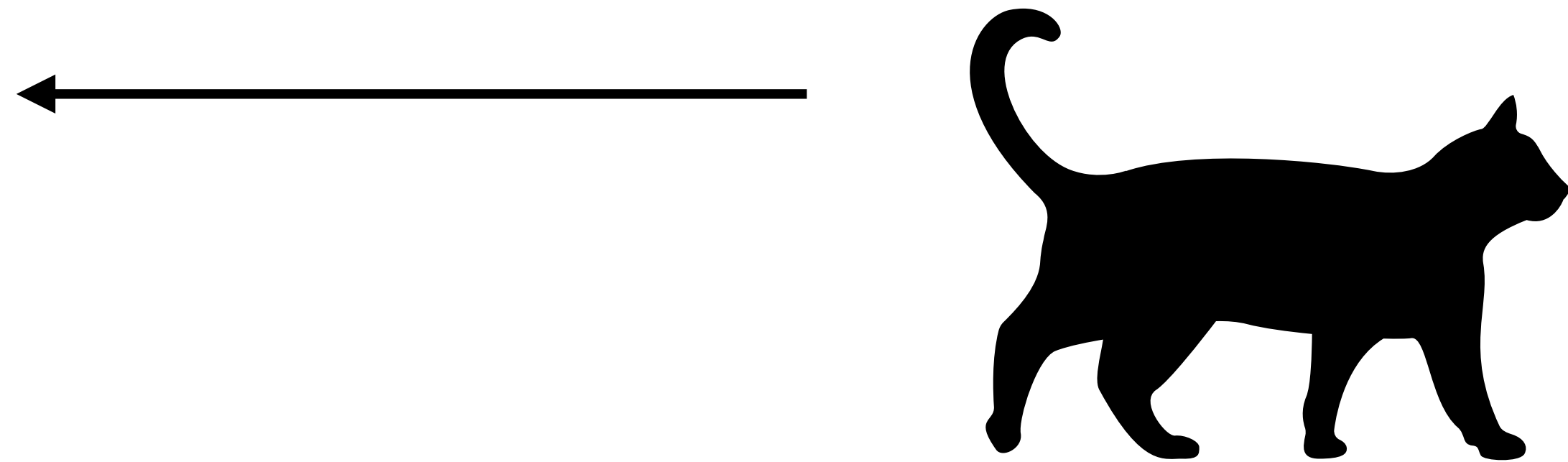


2. 객체지향 프로그래밍



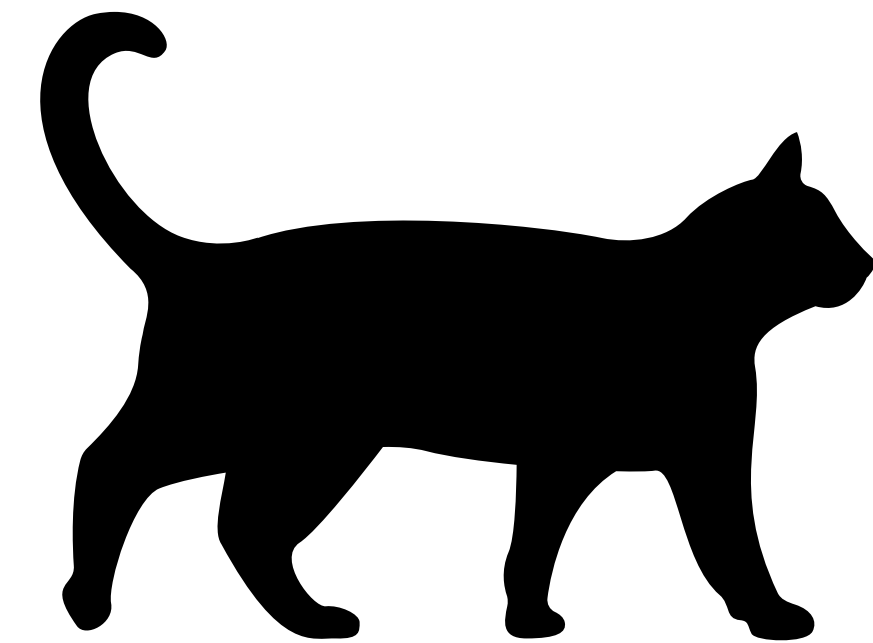
2. 객체지향 프로그래밍

고양이



2. 객체지향 프로그래밍

고양이

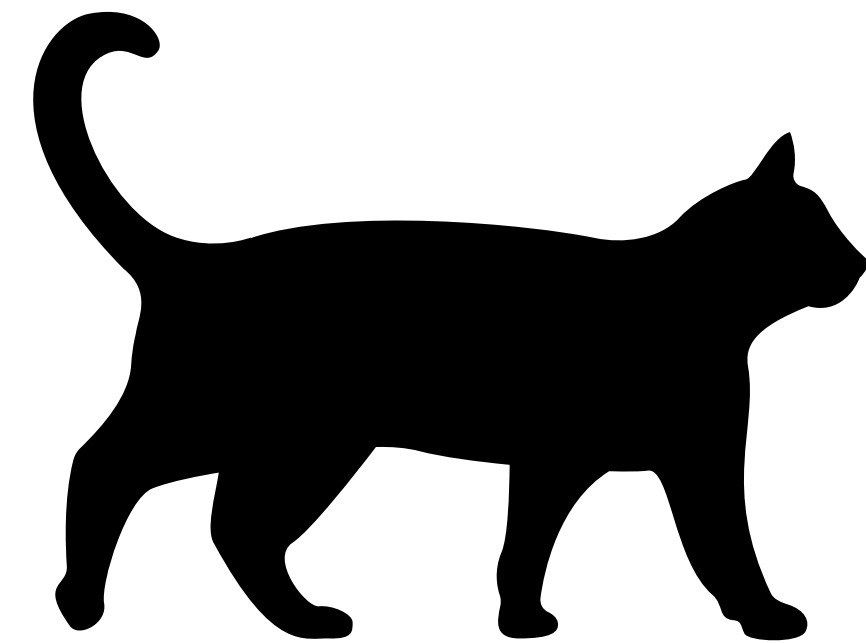


다리가 4개 있고,
털이 있고,
야옹야옹거리고,
높이 점프한다.

2. 객체지향 프로그래밍

```
const cat = {  
  lags: 4,  
  fur: true,  
  meow() {  
    console.log("meow");  
  },  
  jump() {  
    console.log("jump");  
  }  
}
```

고양이



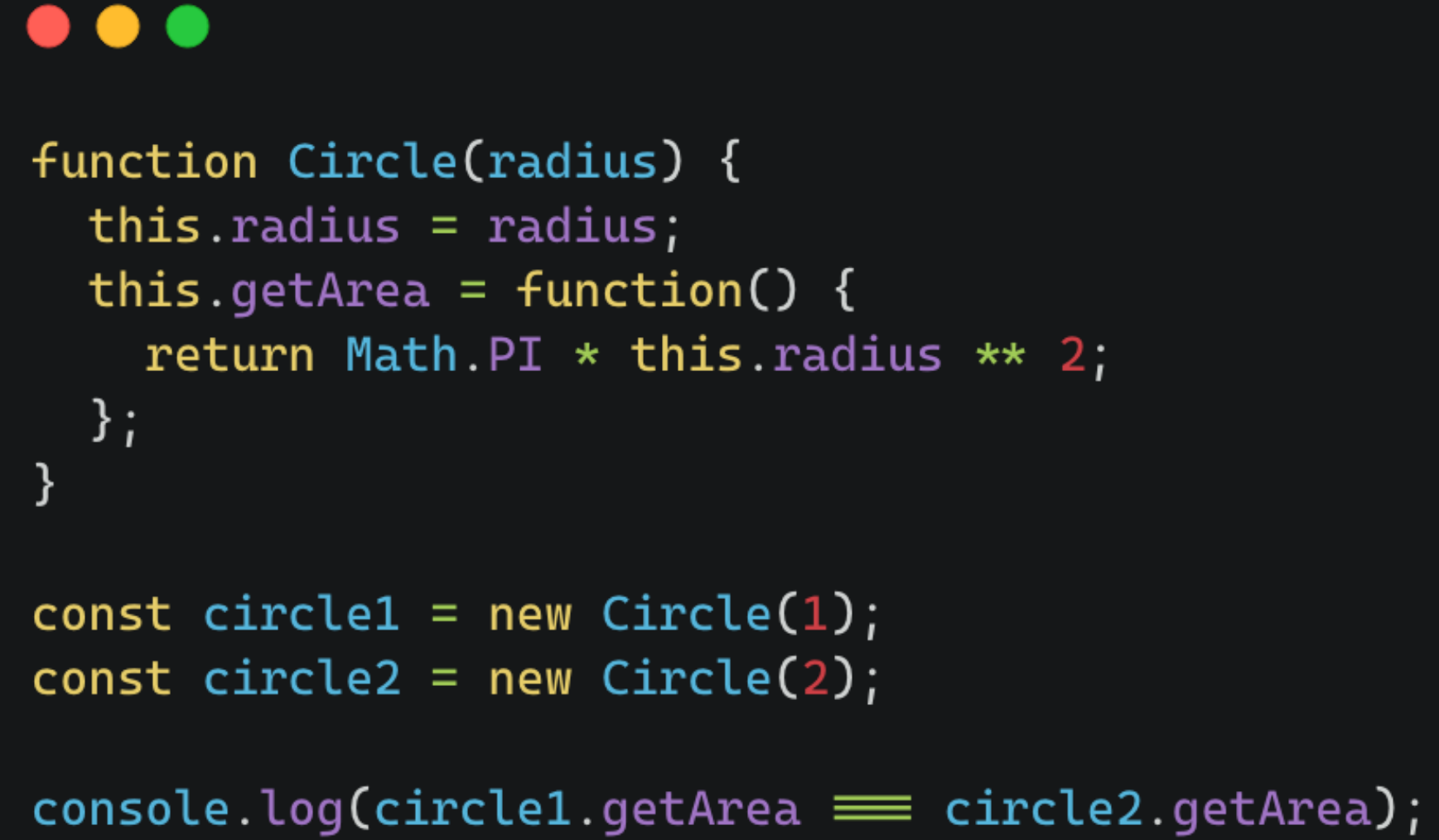
다리가 4개 있고,
털이 있고,
야옹야옹거리고,
높이 점프한다.

3. 상속과 프로토타입

상속이란?

어떤 객체의 프로퍼티 또는 메서드를 다른 객체가 그대로 사용할 수 있도록 하는 것

3. 상속과 프로토타입



```
function Circle(radius) {  
  this.radius = radius;  
  this.getArea = function() {  
    return Math.PI * this.radius ** 2;  
  };  
}  
  
const circle1 = new Circle(1);  
const circle2 = new Circle(2);  
  
console.log(circle1.getArea === circle2.getArea);
```

3. 상속과 프로토타입

같은 기능을 하는 메서드가 중복 생성되는 문제가 있다.

```
function Circle(radius) {  
  this.radius = radius;  
  this.getArea = function() {  
    return Math.PI * this.radius ** 2;  
  };  
}  
  
const circle1 = new Circle(1);  
const circle2 = new Circle(2);  
  
console.log(circle1.getArea === circle2.getArea);
```

3. 상속과 프로토타입



```
function Circle(radius) {  
  this.radius = radius;  
  this.getArea = function() {  
    return Math.PI * this.radius ** 2;  
  };  
}  
  
const circle1 = new Circle(1);  
const circle2 = new Circle(2);  
  
console.log(circle1.getArea === circle2.getArea);
```

같은 기능을 하는 메서드가 중복 생성되는 문제가 있다.

불필요한 메모리를 낭비하는 문제가 발생한다.

3. 상속과 프로토타입



```
function Circle(radius) {  
  this.radius = radius;  
  this.getArea = function() {  
    return Math.PI * this.radius ** 2;  
  };  
}  
  
const circle1 = new Circle(1);  
const circle2 = new Circle(2);  
  
console.log(circle1.getArea === circle2.getArea);
```

같은 기능을 하는 메서드가 중복 생성되는 문제가 있다.

불필요한 메모리를 낭비하는 문제가 발생한다.

인스턴스를 생성할 때마다 메서드를 생성하므로 성능에도 악영향을 준다.

3. 상속과 프로토타입

```
function Circle(radius) {  
  this.radius = radius;  
  this.getArea = function() {  
    return Math.PI * this.radius ** 2;  
  };  
}  
  
const circle1 = new Circle(1);  
const circle2 = new Circle(2);  
  
console.log(circle1.getArea === circle2.getArea);
```

같은 기능을 하는 메서드가 중복 생성되는 문제가 있다.

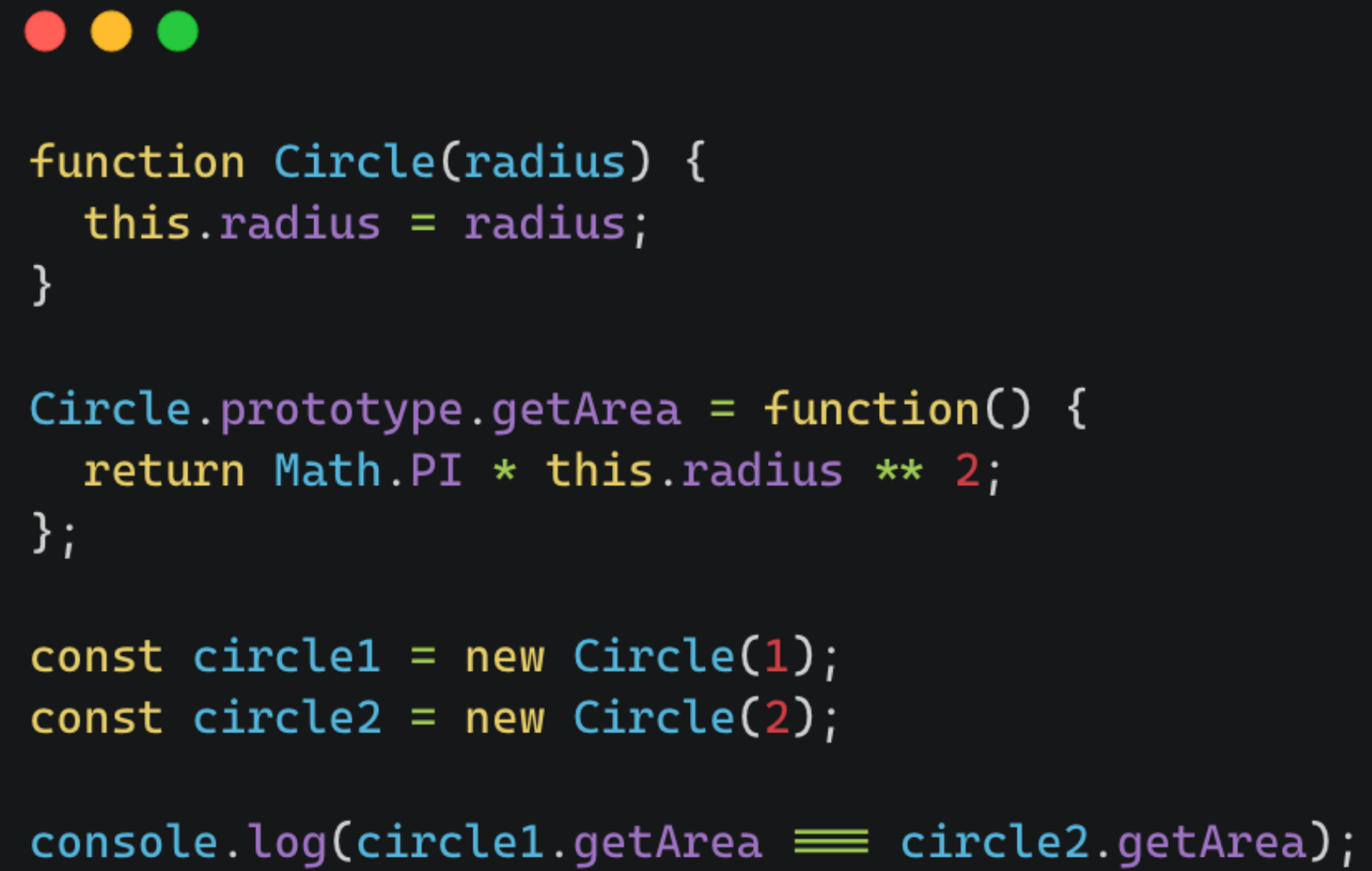
불필요한 메모리를 낭비하는 문제가 발생한다.

인스턴스를 생성할 때마다 메서드를 생성하므로 성능에도 악영향을 준다.



프로토타입 기반의 상속으로 중복을 제거할 수 있다.

3. 상속과 프로토타입



```
function Circle(radius) {  
  this.radius = radius;  
}  
  
Circle.prototype.getArea = function() {  
  return Math.PI * this.radius ** 2;  
};  
  
const circle1 = new Circle(1);  
const circle2 = new Circle(2);  
  
console.log(circle1.getArea === circle2.getArea);
```


3. 상속과 프로토타입

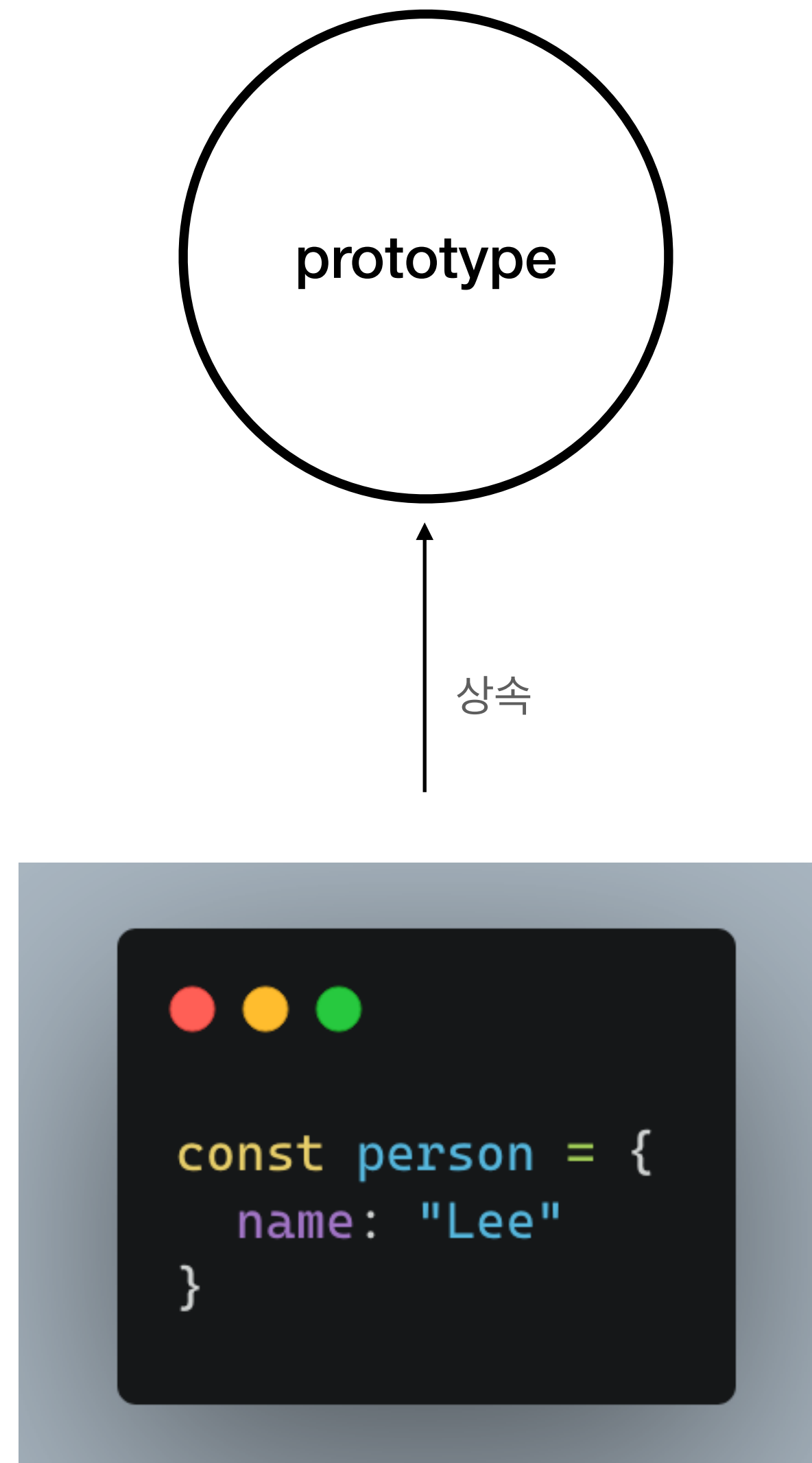
```
function Circle(radius) {  
  this.radius = radius;  
}  
  
Circle.prototype.getArea = function() {  
  return Math.PI * this.radius ** 2;  
};  
  
const circle1 = new Circle(1);  
const circle2 = new Circle(2);  
  
console.log(circle1.getArea === circle2.getArea);
```

상속은 코드의 재사용이란 관점에서 매우 유용하다.

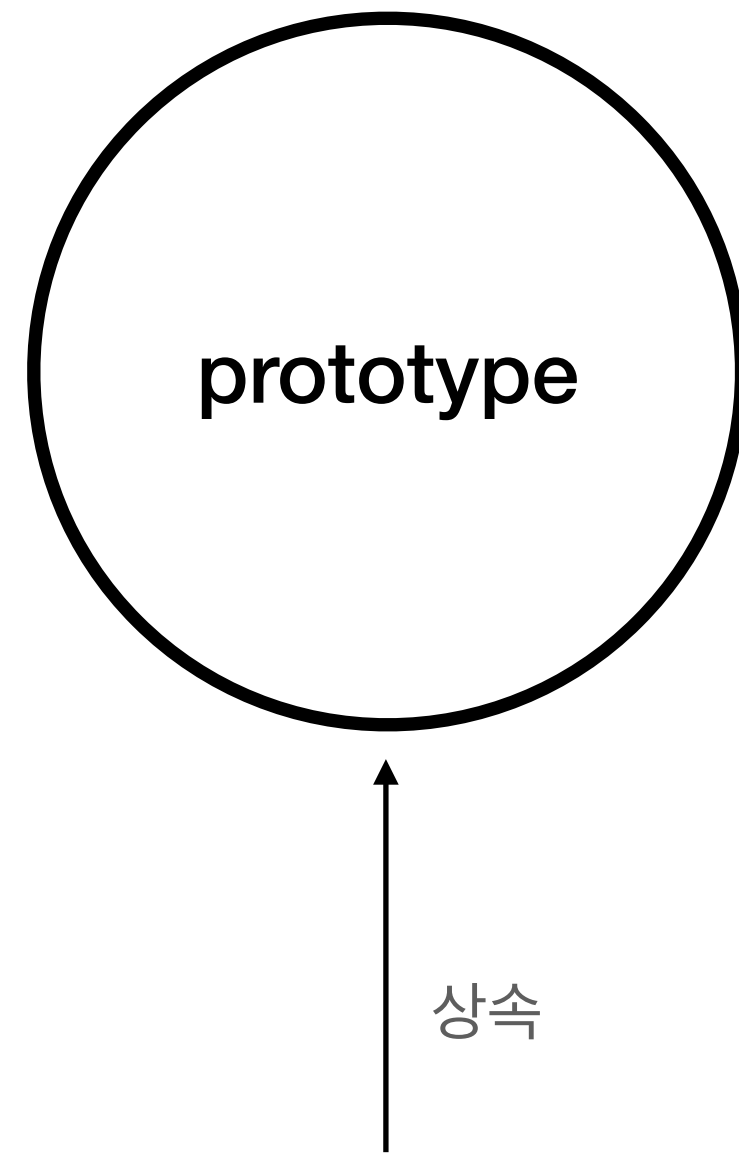
4. 프로토타입 객체



4. 프로토타입 객체

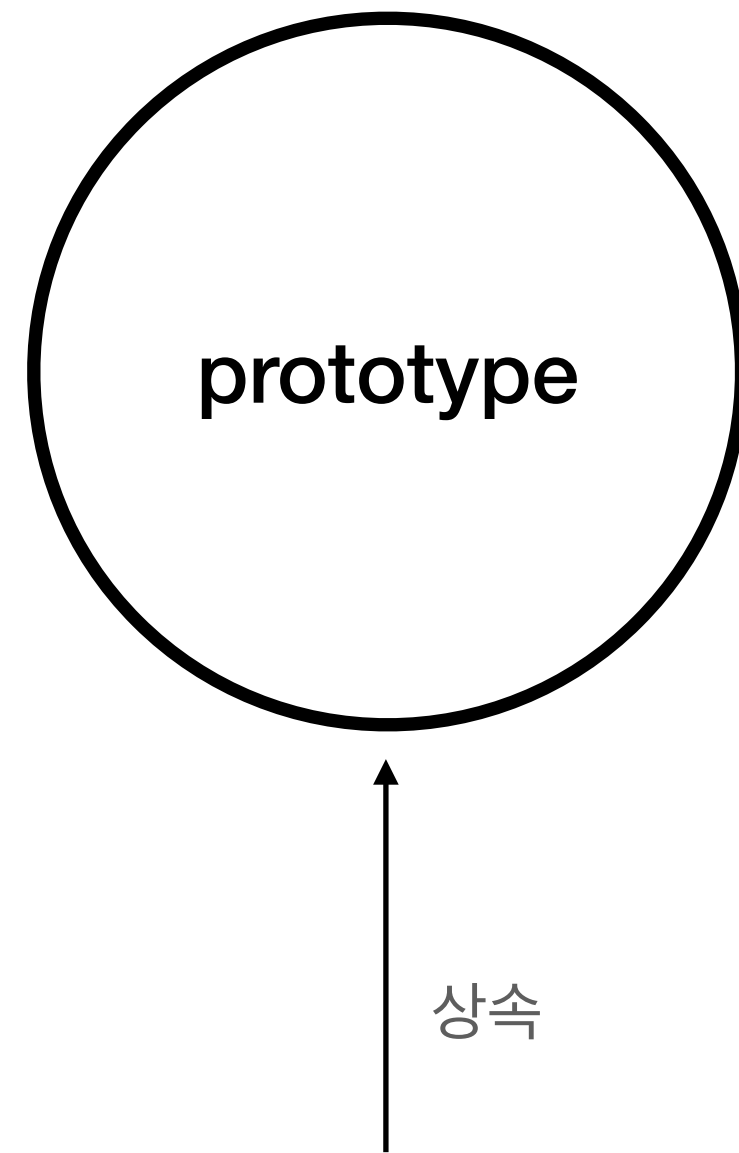


4. 프로토타입 객체



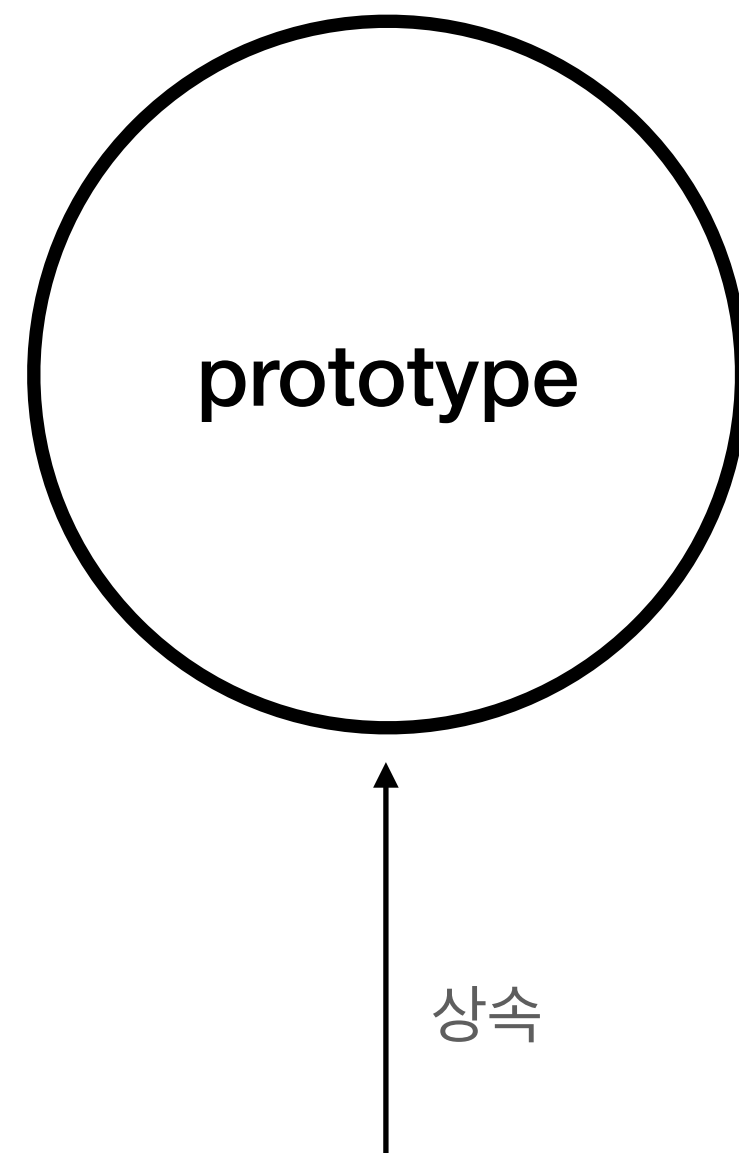
```
const person = {  
  name: "Lee"  
}
```

4. 프로토타입 객체



모든 객체는 `[[Prototype]]` 이라는 내부 슬롯을 가진다.

4. 프로토타입 객체



모든 객체는 `[[Prototype]]` 이라는 내부 슬롯을 가진다.

이 내부 슬롯의 값은 프로토타입의 참조다. (null인 경우도 있음)

```
const person = {  
  name: "Lee"  
}
```

4. 프로토타입 객체

모든 객체는 `[[Prototype]]` 이라는 내부 슬롯을 가진다.

4. 프로토타입 객체

모든 객체는 `[[Prototype]]` 이라는 내부 슬롯을 가진다.

하지만 내부슬롯에는 직접 접근할 수 없다.

4. 프로토타입 객체

모든 객체는 `[[Prototype]]` 이라는 내부 슬롯을 가진다.

하지만 내부슬롯에는 직접 접근할 수 없다.



`__proto__` 접근자 프로퍼티로 `[[Prototype]]` 내부 슬롯에 간접적으로 접근 가능

4. 프로토타입 객체

```
> const person = { name: "Lee" };
< undefined

> person.__proto__
< {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...} ⓘ
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ __proto__: (...)
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()
```

__proto__ 가능

4. 프로토타입 객체

__proto__ 접근자 프로퍼티에 접근 가능한 이유



4. 프로토타입 객체

__proto__ 접근자 프로퍼티에 접근 가능한 이유



상속을 통해서 Object.prototype.__proto__ 접근자 프로퍼티를 사용하는 것이다.

4. 프로토타입 객체



```
const person = {  
  name: "Lee"  
};
```

```
console.log(Object.hasOwnProperty(person, "__proto__"));
```

```
console.log(Object.getOwnPropertyDescriptor(Object.prototype, "__proto__"));
```

4. 프로토타입 객체



```
const person = {  
  name: "Lee"  
};
```

```
console.log(Object.hasOwnProperty(person, "__proto__")); -> false
```

```
console.log(Object.getPrototypeOf(Object.prototype, "__proto__"));
```

4. 프로토타입 객체



```
const person = {  
  name: "Lee"  
};
```

```
console.log(Object.hasOwnProperty(person, "__proto__")); -> false
```

```
console.log(Object.getOwnPropertyDescriptor(Object.prototype, "__proto__"));  
-> {get: f, set: f, enumerable: false, configurable: true}
```

4. 프로토타입 객체

__proto__ 접근자 프로퍼티를 통해 프로토타입에 접근하는 이유



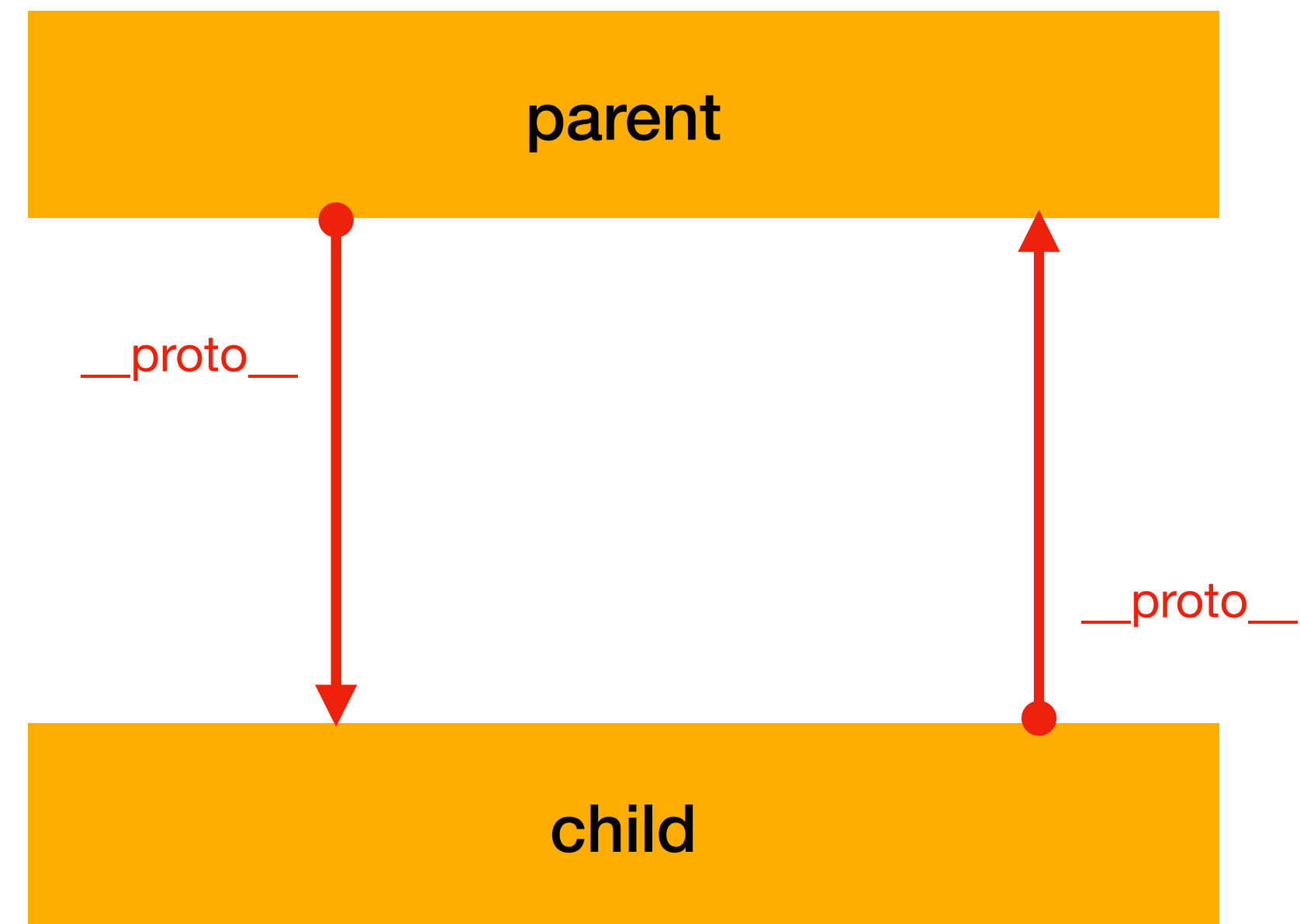
```
const parent = {};  
const child = {};  
  
child.__proto__ = parent;  
parent.__proto__ = child;
```


4. 프로토타입 객체

__proto__ 접근자 프로퍼티를 통해 프로토타입에 접근하는 이유



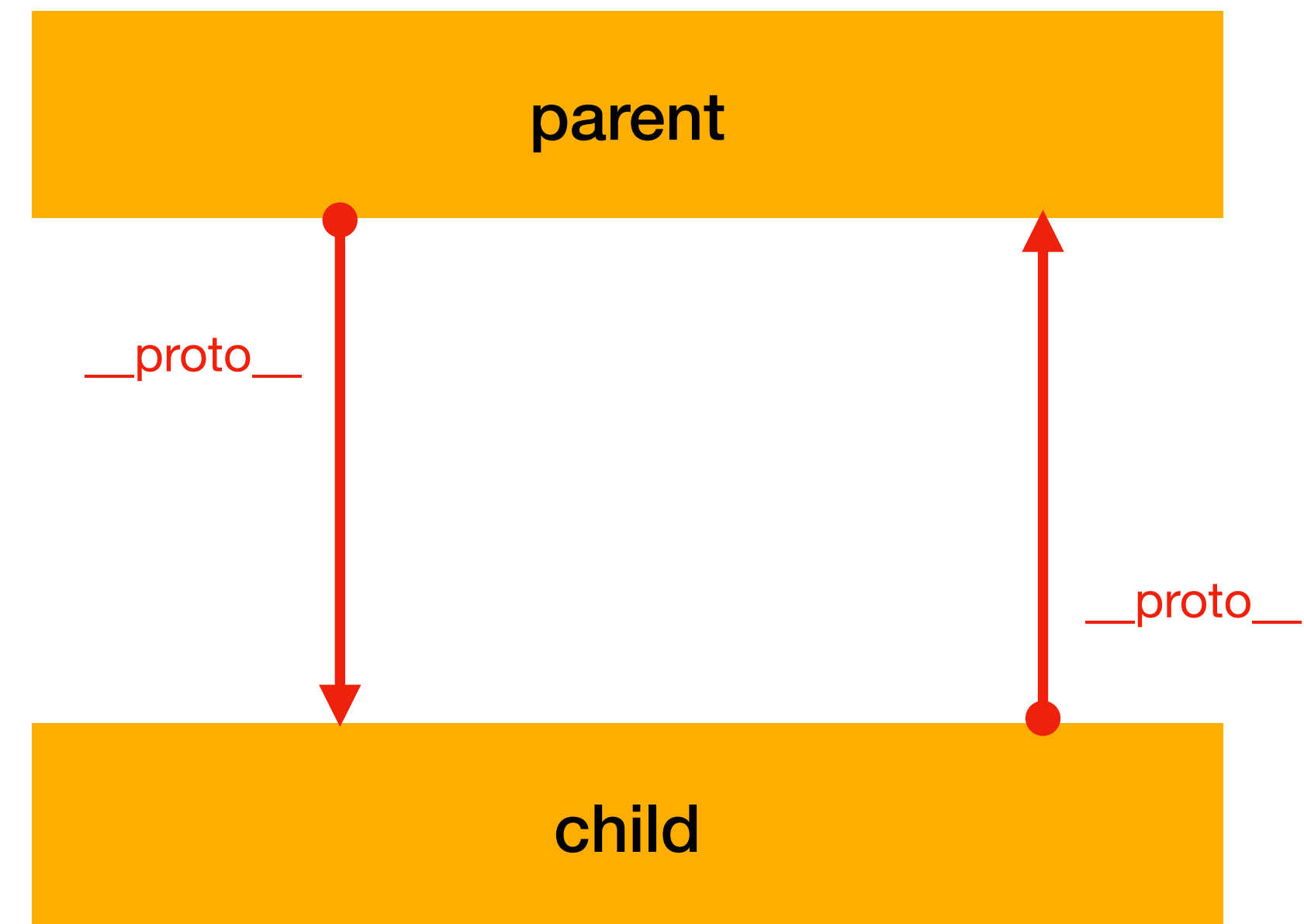
```
const parent = {};  
const child = {};  
  
child.__proto__ = parent;  
parent.__proto__ = child;
```



4. 프로토타입 객체

__proto__ 접근자 프로퍼티를 통해 프로토타입에 접근하는 이유

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



위 상황은 서로가 자신의 프로토타입이 되는 비정상적인 상황이다.
프로토타입 체인의 종점이 존재하지 않기 때문에 프로퍼티 검색시 무한루프에 빠진다.

4. 프로토타입 객체

`__proto__` 접근자 프로퍼티를 코드 내에서 직접 사용하는 것을 권장하지 않는 이유

모든 객체가 `__proto__` 접근자 프로퍼티를 사용할 수 있는 것은 아니기 때문이다.

4. 프로토타입 객체

__proto__ 접근자 프로퍼티를 코드 내에서 직접 사용하는 것을 권장하지 않는 이유

모든 객체가 __proto__ 접근자 프로퍼티를 사용할 수 있는 것은 아니기 때문이다.

-> 직접 상속을 통해 Object.prototype을 상속받지 않는 객체를 생성할 수도 있기 때문에 __proto__ 접근자 프로퍼티를 사용할 수 없는 경우가 있다.

4. 프로토타입 객체

`__proto__` 접근자 프로퍼티를 코드 내에서 직접 사용하는 것을 권장하지 않는 이유

모든 객체가 `__proto__` 접근자 프로퍼티를 사용할 수 있는 것은 아니기 때문이다.

-> 직접 상속을 통해 `Object.prototype`을 상속받지 않는 객체를 생성할 수도 있기 때문에 `__proto__` 접근자 프로퍼티를 사용할 수 없는 경우가 있다.



```
const obj = {};  
const parent = { x: 1 };  
  
Object.getPrototypeOf(obj); // obj.__proto__  
  
Object.setPrototypeOf(obj, parent); // obj.__proto__ = parent
```

4. 프로토타입 객체

함수 객체의 prototype 프로퍼티

함수 객체만이 prototype 프로퍼티를 가진다.



```
(function () {}).hasOwnProperty("prototype");  
({}).hasOwnProperty("prototype");
```

4. 프로토타입 객체

함수 객체의 prototype 프로퍼티

함수 객체의 prototype 프로퍼티는 생성자 함수가 생성할 인스턴스의 프로토타입을 가리킨다.



```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
console.log(Person.prototype === me.__proto__);
```

4. 프로토타입 객체

함수 객체의 prototype 프로퍼티

함수 객체의 prototype 프로퍼티는 생성자 함수가 생성할 인스턴스의 프로토타입을 가리킨다.

-> non-constructor인 화살표 함수와 ES6 메서드 축약 표현으로 정의한 메서드는 prototype 프로퍼티를 소유하지 않으며 프로토타입도 생성하지 않는다.



```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
console.log(Person.prototype === me.__proto__);
```


4. 프로토타입 객체

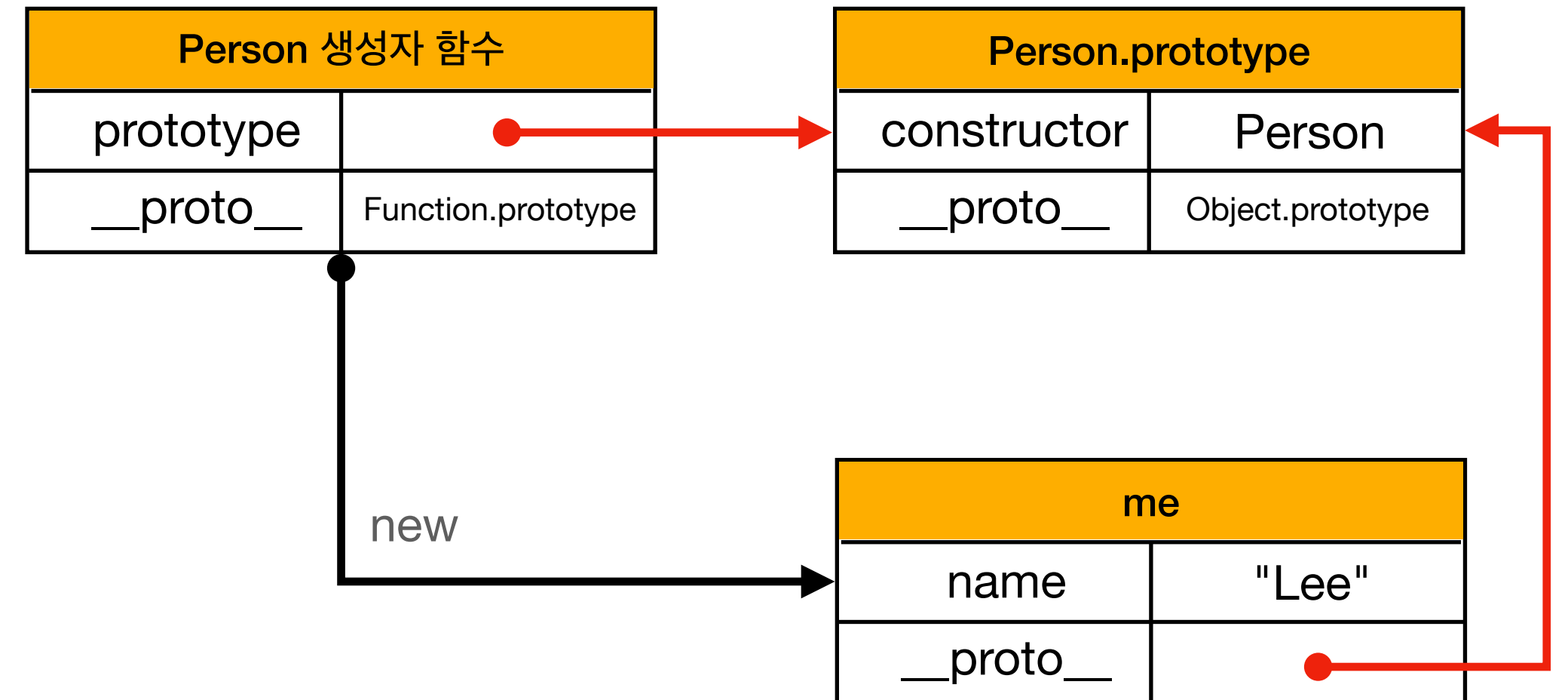
함수 객체의 prototype 프로퍼티

함수 객체의 prototype 프로퍼티는 생성자 함수가 생성할 인스턴스의 프로토타입을 가리킨다.

-> non-constructor인 화살표 함수와 ES6 메서드 축약 표현으로 정의한 메서드는 prototype 프로퍼티를 소유하지 않으며 프로토타입도 생성하지 않는다.




```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
console.log(Person.prototype === me.__proto__);
```



4. 프로토타입 객체

프로토타입의 constructor 프로퍼티와 생성자 함수

모든 프로토타입은 constructor 프로퍼티를 갖는다. 이 프로퍼티는 prototype 프로퍼티를 통해서 자신을 참조하고 있는 생성자 함수를 가리킨다.



```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
console.log(me.constructor === Person);
```

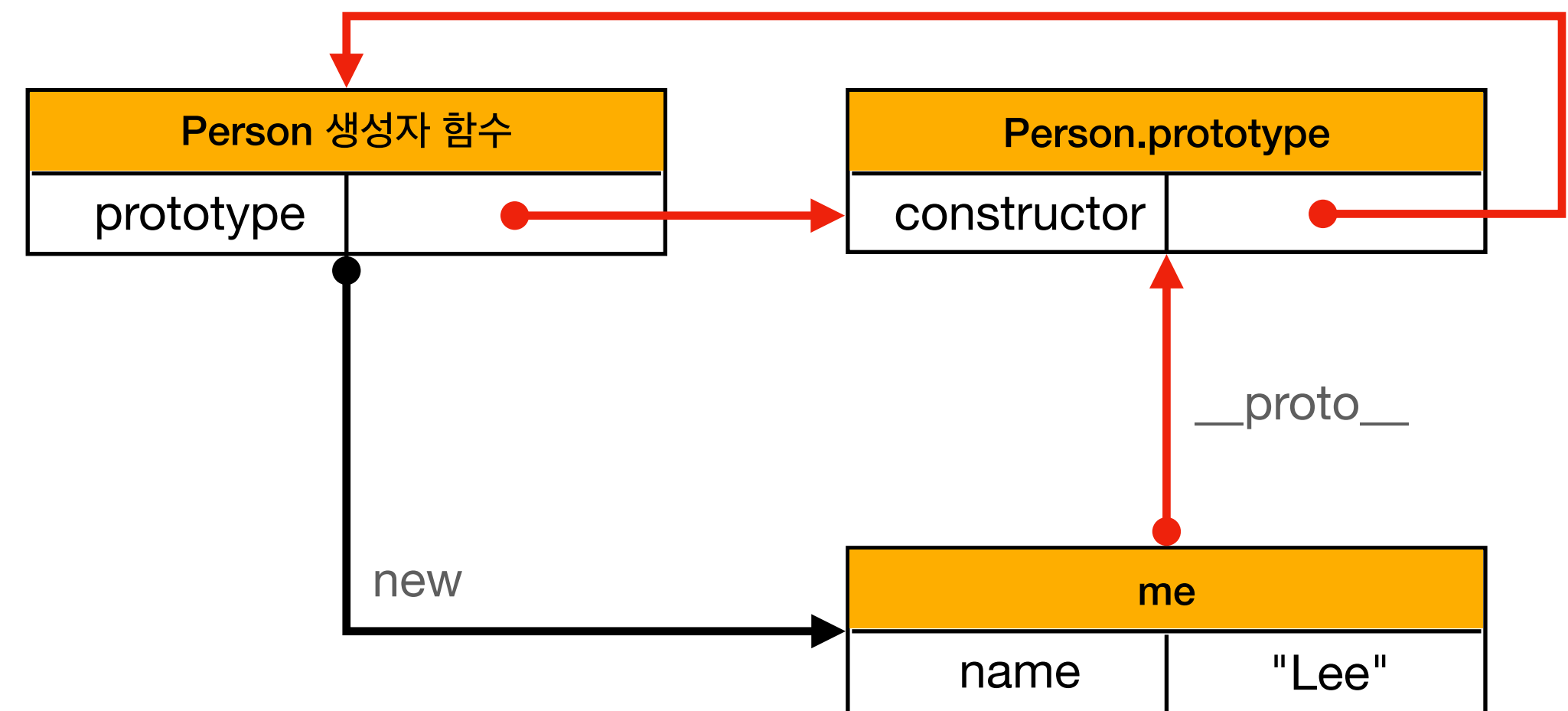
4. 프로토타입 객체

프로토타입의 constructor 프로퍼티와 생성자 함수

모든 프로토타입은 constructor 프로퍼티를 갖는다. 이 프로퍼티는 prototype 프로퍼티를 통해서 자신을 참조하고 있는 생성자 함수를 가리킨다.



```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
console.log(me.constructor === Person);
```



5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

생성자 함수로 객체를 생성하는 경우

프로토타입의 constructor 프로퍼티는 생성자 함수를 가리킵니다.

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

생성자 함수로 객체를 생성하는 경우

프로토타입의 constructor 프로퍼티는 생성자 함수를 가리킵니다.



```
const obj = new Object();  
console.log(obj.constructor === Object); // true
```

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

생성자 함수로 객체를 생성하는 경우

프로토타입의 constructor 프로퍼티는 생성자 함수를 가리킵니다.



```
const obj = new Object();  
console.log(obj.constructor === Object); // true
```



```
const add = new Function("a", "b", 'return a + b');  
console.log(add.constructor === Function); // true
```

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

생성자 함수로 객체를 생성하는 경우

프로토타입의 constructor 프로퍼티는 생성자 함수를 가리킵니다.



```
const obj = new Object();  
console.log(obj.constructor === Object); // true
```



```
const add = new Function("a", "b", 'return a + b');  
console.log(add.constructor === Function); // true
```



```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // true
```

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법으로 객체를 생성하는 경우

프로토타입의 constructor 프로퍼티가 반드시 객체를 생성한 생성자 함수를 가리킨다고 단정할 순 없습니다.



```
const obj = {}; // 객체 리터럴

const add = function(a, b) { return a + b }; // 함수 리터럴

const arr = [1, 2, 3]; // 배열 리터럴

const regexp = /is/ig; // 정규 표현식 리터럴
```


5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법으로 객체를 생성하는 경우 - 객체 리터럴



```
const obj = {};
```

```
console.log(obj.constructor === Object); // true
```

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법으로 객체를 생성하는 경우 - 객체 리터럴



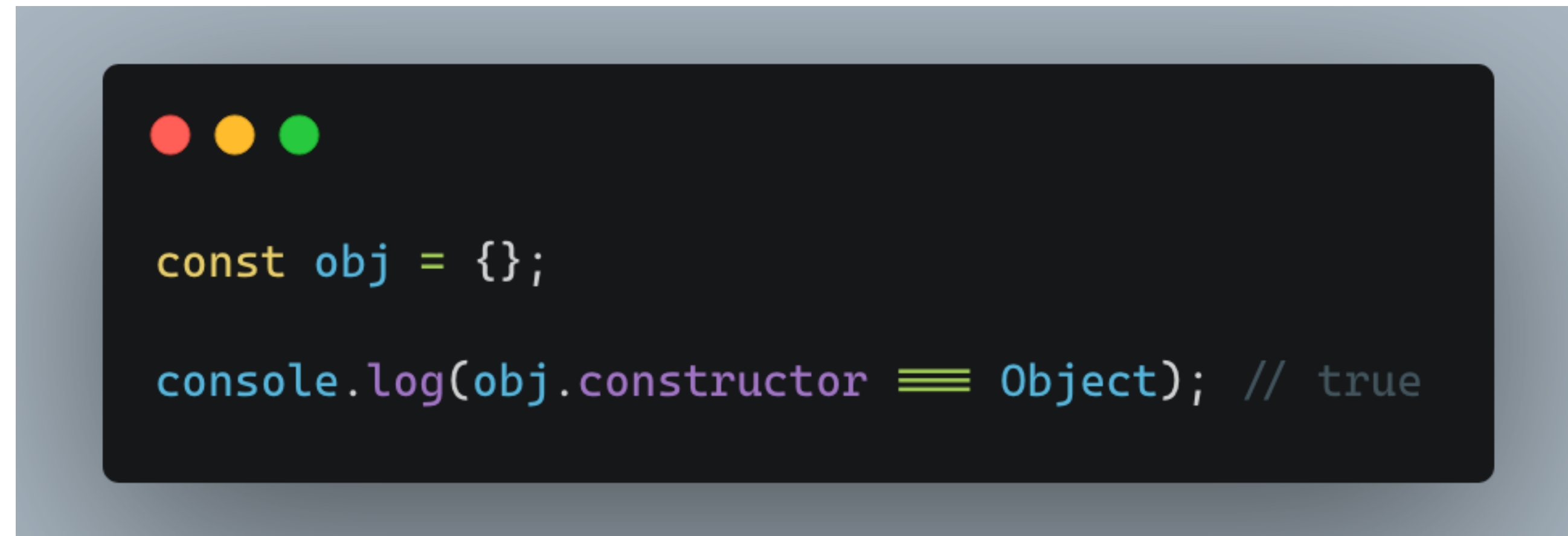
```
const obj = {};
```

```
console.log(obj.constructor === Object); // true
```

ECMAScript 사양에 따르면 추상연산 OrdinaryObjectCreate를 호출하여 빈 객체를 생성하는 점에서는 동일하나 세부 내용은 다르다.

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법으로 객체를 생성하는 경우 - 객체 리터럴



```
const obj = {};  
console.log(obj.constructor === Object); // true
```

ECMAScript 사양에 따르면 추상연산 OrdinaryObjectCreate를 호출하여 빈 객체를 생성하는 점에서는 동일하나 세부 내용은 다르다.

따라서 객체 리터럴에 의해 생성된 객체는 Object 생성자 함수가 생성한 객체가 아니다.

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법으로 객체를 생성하는 경우 - 함수 리터럴

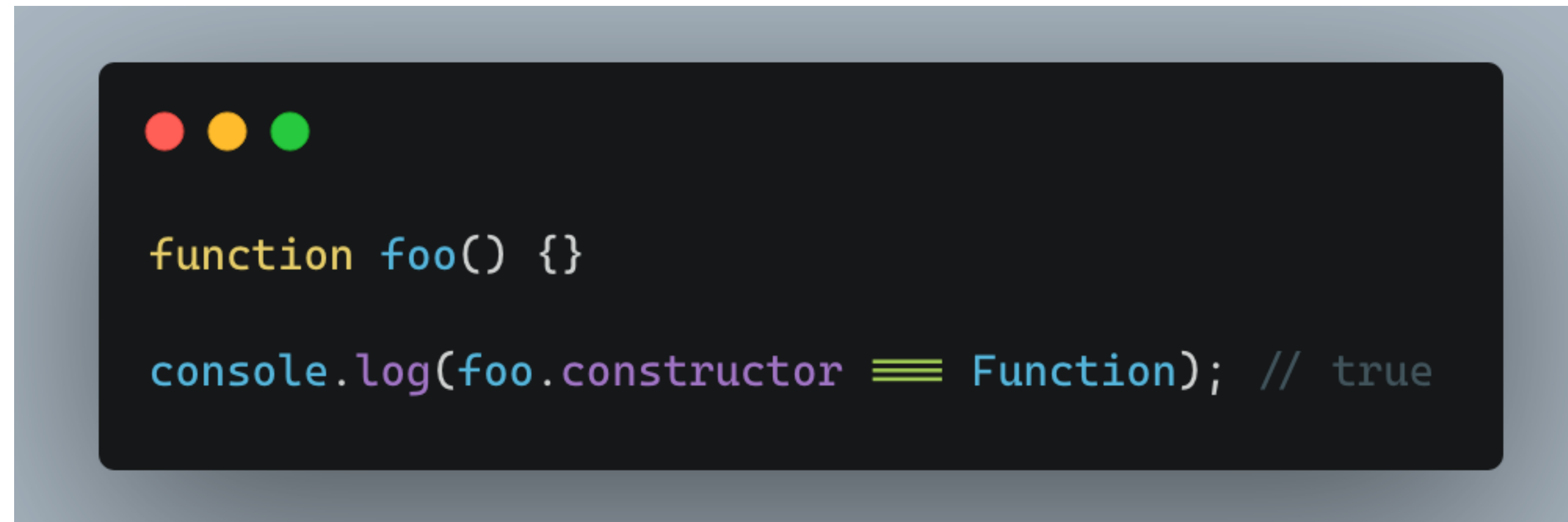


```
function foo() {}
```

```
console.log(foo.constructor === Function); // true
```

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법으로 객체를 생성하는 경우 - 함수 리터럴



```
function foo() {}  
  
console.log(foo.constructor === Function); // true
```

Function 생성자 함수를 호출하여 생성한 함수와 함수 선언문, 함수 표현식을 이용하여 생성한 함수는 특성이 다르다.

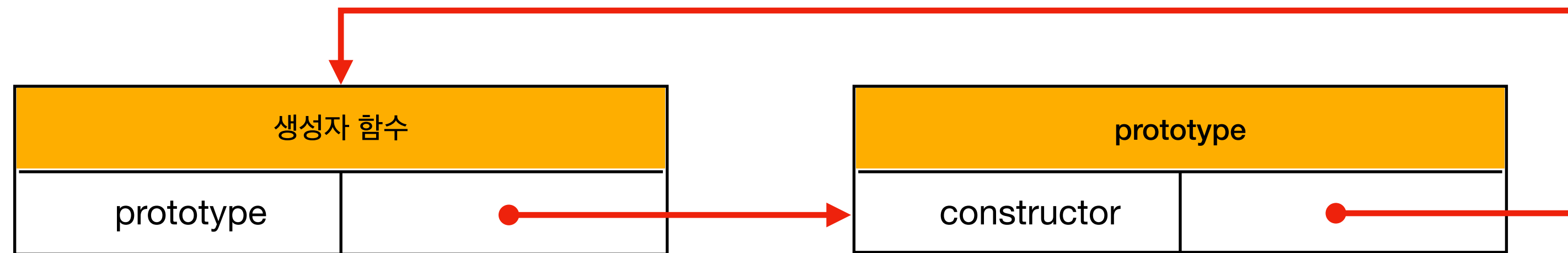
하지만 위의 foo 함수의 constructor 프로퍼티를 보면 Function이다.

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입 결론

리터럴 표기법에 의해 생성된 객체도 상속을 위해 프로토타입이 필요하기 때문에 가상의 생성자 함수를 갖는다.

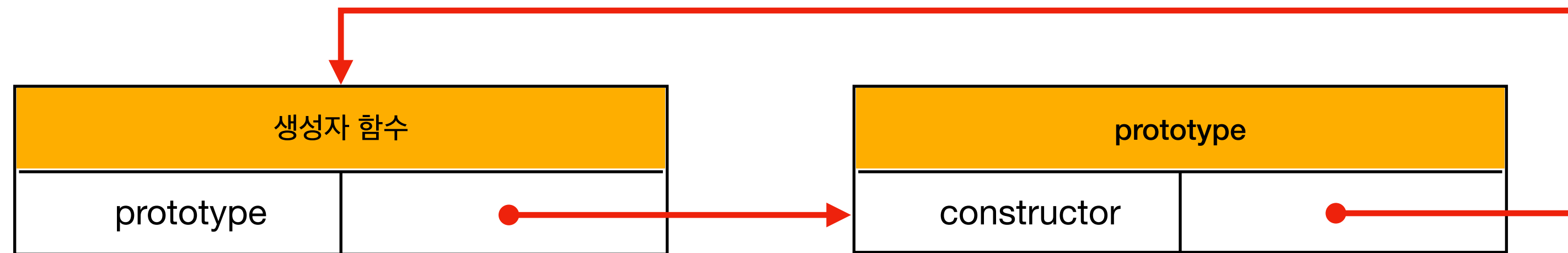
5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입 결론

리터럴 표기법에 의해 생성된 객체도 상속을 위해 프로토타입이 필요하기 때문에 가상의 생성자 함수를 갖는다.



5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입 결론

리터럴 표기법에 의해 생성된 객체도 상속을 위해 프로토타입이 필요하기 때문에 가상의 생성자 함수를 갖는다.



프로토타입과 생성자 함수는 단독으로 존재할 수 없고 언제나 쌍으로 존재한다.

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입 결론

리터럴 표기법으로 생성한 객체도 생성자 함수로 생성한 객체와 본질적인 면에서 큰 차이는 없다.

5. 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

결론

리터럴 표기법으로 생성한 객체도 생성자 함수로 생성한 객체와 본질적인 면에서 큰 차이는 없다.

리터럴 표기법	생성자 함수	프로토타입
객체 리터럴	Object	Object.prototype
함수 리터럴	Function	Function.prototype
배열 리터럴	Array	Array.prototype
정규 표현식 리터럴	RegExp	RegExp.prototype

6. 프로토타입의 생성 시점

프로토타입은 생성자 함수가 생성되는 시점에 더불어 생성된다.

6. 프로토타입의 생성 시점

프로토타입은 생성자 함수가 생성되는 시점에 더불어 생성된다.

프로토타입과 생성자 함수는 단독으로 존재할 수 없고 언제나 쌍으로 존재하기 때문

6. 프로토타입의 생성 시점

사용자 정의 생성자 함수와 프로토타입 생성 시점

생성자 함수로서 호출할 수 있는 함수, 즉 constructor는 함수 객체를 생성하는 시점에 프로토타입도 더불어 생성된다.



```
console.log(Person.prototype); // {constructor: f}
```

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

6. 프로토타입의 생성 시점

사용자 정의 생성자 함수와 프로토타입 생성 시점

생성자 함수로서 호출할 수 있는 함수, 즉 constructor는 함수 객체를 생성하는 시점에 프로토타입도 더불어 생성된다.



```
console.log(Person.prototype); // {constructor: f}
```

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

화살표 함수, 메서드 축약 표현 등 생성자 함수로서 호출할 수 없는 함수는 프로토타입이 생성되지 않는다.

6. 프로토타입의 생성 시점

사용자 정의 생성자 함수와 프로토타입 생성 시점

생성된 프로토타입은 오직 constructor 프로퍼티만을 갖는 객체다.

```
> function Person(name) {  
    this.name = name;  
}  
< undefined  
  
> console.log(Person.prototype)  
▼ {constructor: f} ⓘ VM4901:1  
  ► constructor: f Person(name)  
  ► [[Prototype]]: Object
```

6. 프로토타입의 생성 시점

빌트인 생성자 함수와 프로토타입 생성 시점

- Object
- String
- Number
- Function
- Array
- RegExp
- Date
- Promise

6. 프로토타입의 생성 시점

빌트인 생성자 함수와 프로토타입 생성 시점

- Object
- String
- Number
- Function
- Array
- RegExp
- Date
- Promise

모든 빌트인 생성자 함수는 전역 객체가 생성되는 시점에 생성된다!

7. 객체 생성 방식과 프로토타입의 결정

- 객체 리터럴
- Object 생성자 함수
- 생성자 함수
- Object.create 메서드
- 클래스 (ES6)

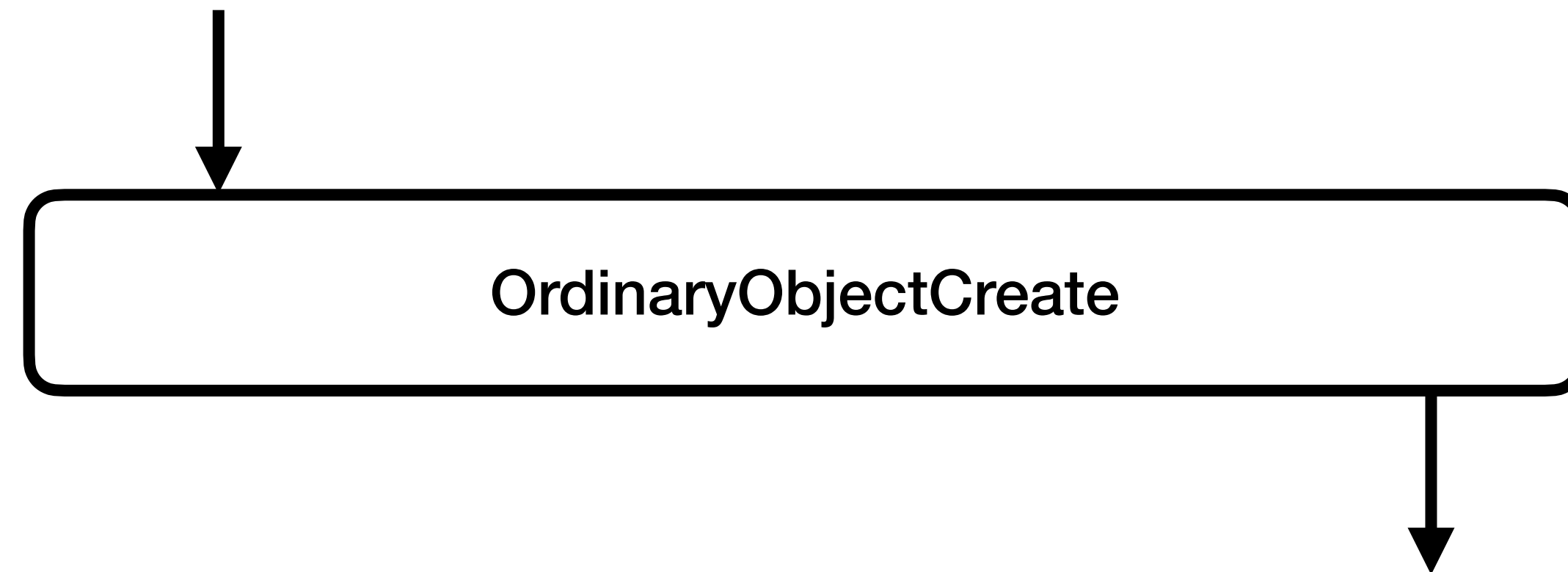
7. 객체 생성 방식과 프로토타입의 결정

- 객체 리터럴
- Object 생성자 함수
- 생성자 함수
- Object.create 메서드
- 클래스 (ES6)

세부적인 방식의 차이는 있으나 추상연산 OrdinaryObjectCreate에 의해 생성된다.

7. 객체 생성 방식과 프로토타입의 결정

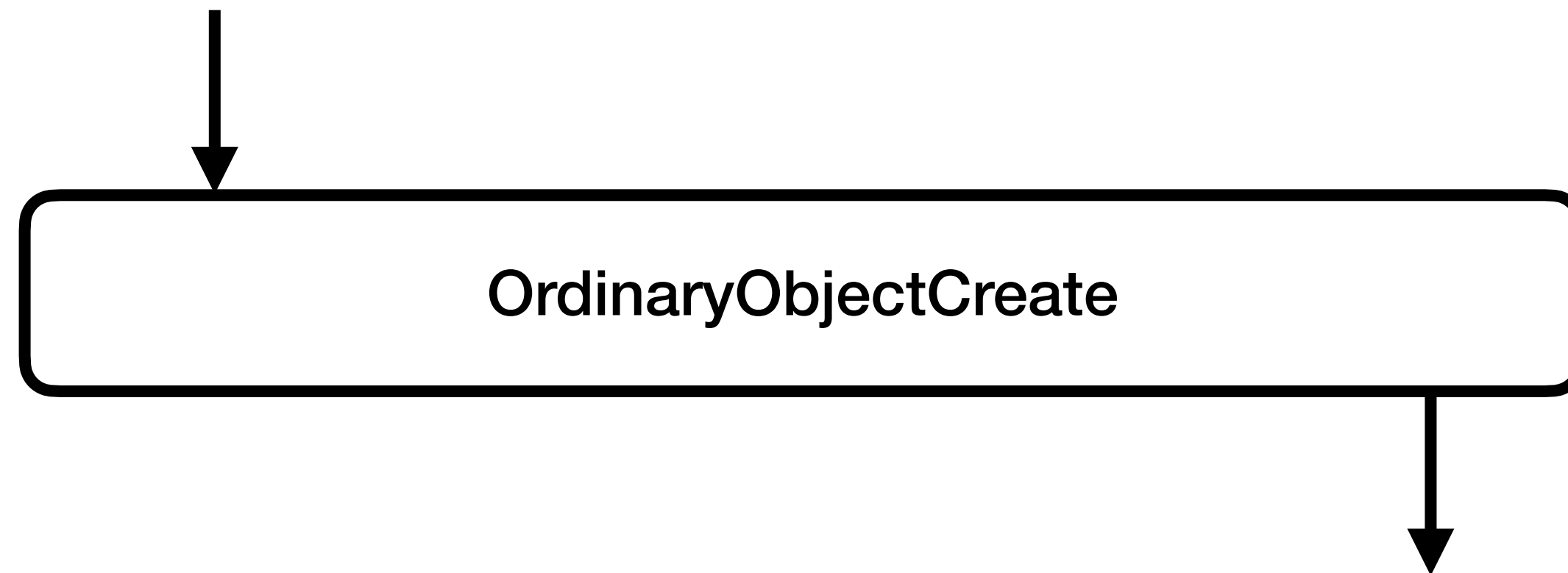
추상연산 OrdinaryObjectCreate



7. 객체 생성 방식과 프로토타입의 결정

추상연산 OrdinaryObjectCreate

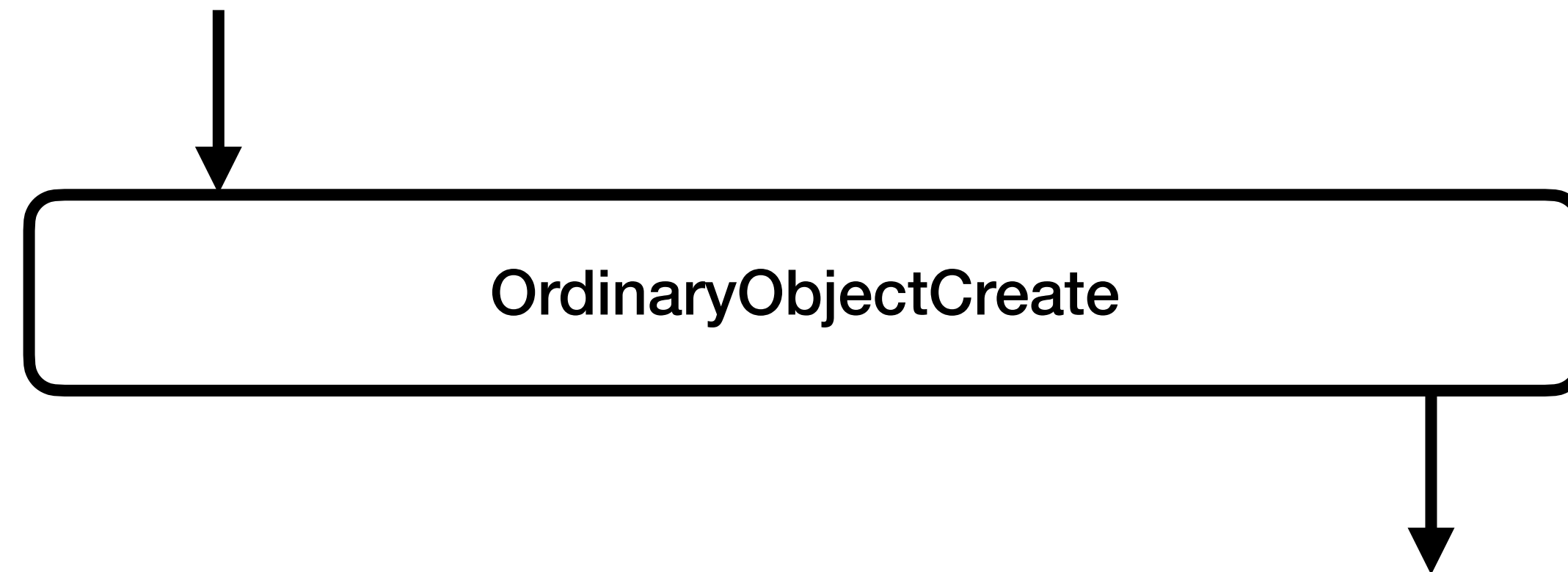
연산을 통해 생성될 객체의 프로토타입



7. 객체 생성 방식과 프로토타입의 결정

추상연산 OrdinaryObjectCreate

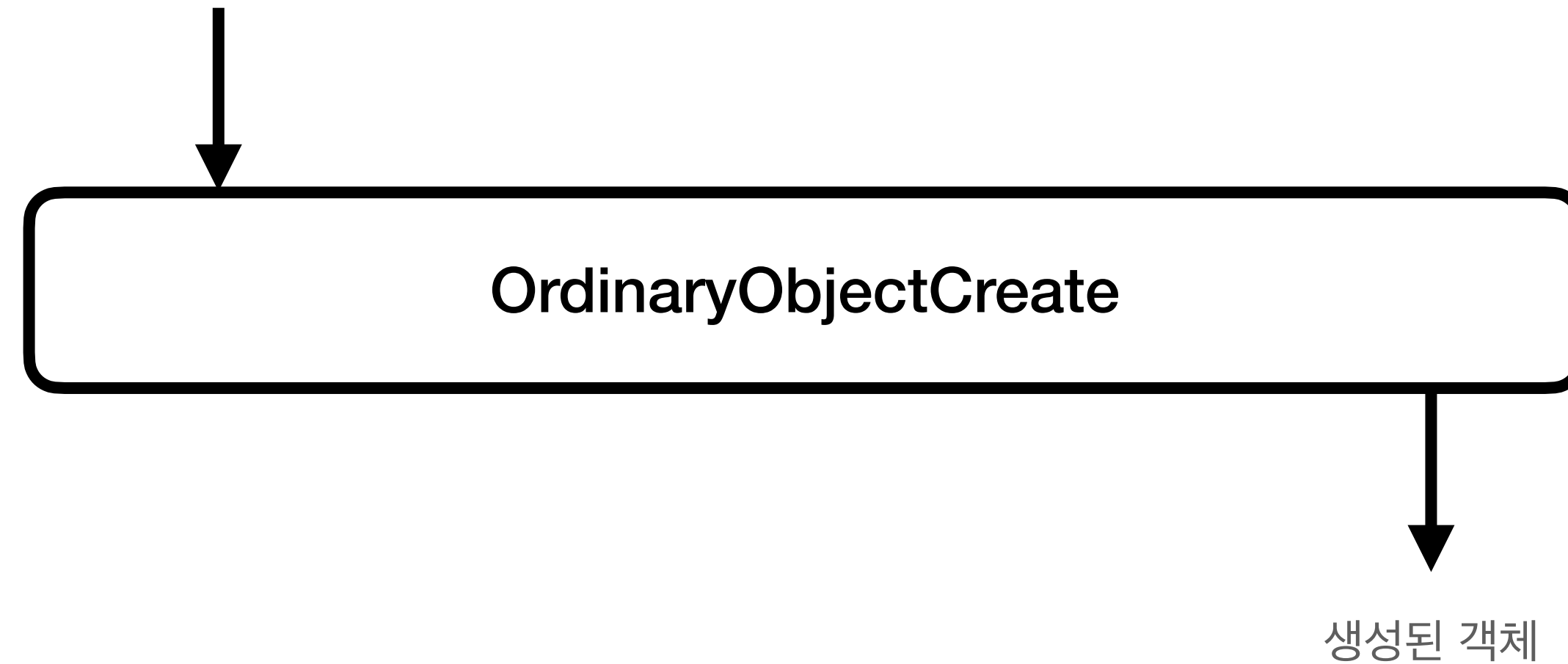
연산을 통해 생성될 객체의 프로토타입
연산을 통해 생성될 객체의 프로퍼티 목록 (optional)



7. 객체 생성 방식과 프로토타입의 결정

추상연산 OrdinaryObjectCreate

연산을 통해 생성될 객체의 프로토타입
연산을 통해 생성될 객체의 프로퍼티 목록 (optional)



7. 객체 생성 방식과 프로토타입의 결정

객체 리터럴

객체 리터럴에 의해 생성된 객체는 Object.prototype을 프로토타입으로 갖게된다



```
const obj = { x: 1 };
```

```
console.log(obj.constructor === Object); // true
```

```
console.log(obj.hasOwnProperty("x")); // true
```


7. 객체 생성 방식과 프로토타입의 결정

Object 생성자 함수

Object생성자 함수를 호출하면 추상연산이 호출되고, Object.prototype이 프로토타입으로 전달된다.



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

```
obj.x = 1;
```

```
console.log(obj.constructor === Object); // true
```

```
console.log(obj.hasOwnProperty("x")); // true
```

7. 객체 생성 방식과 프로토타입의 결정

Object 생성자 함수

Object생성자 함수를 호출하면 추상연산이 호출되고, Object.prototype이 프로토타입으로 전달된다.



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

```
obj.x = 1;
```

```
console.log(obj.constructor === Object); // true
```

```
console.log(obj.hasOwnProperty("x")); // true
```

프로토타입의 연결 구조는 객체 리터럴에 의해 생성된 객체와 동일한 구조다.

7. 객체 생성 방식과 프로토타입의 결정

생성자 함수

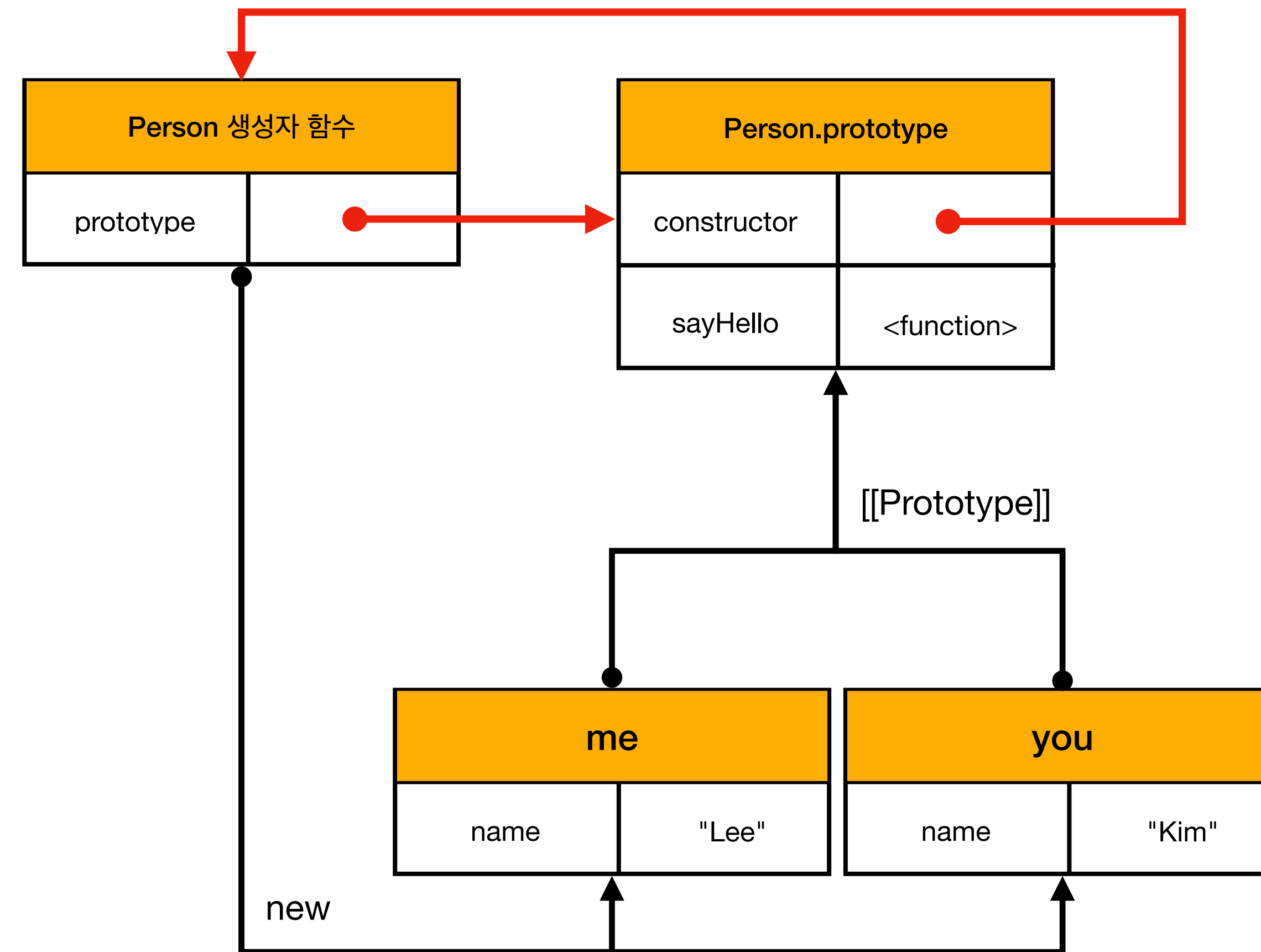
생성자 함수를 호출하면 추상연산이 호출되고, 생성자 함수의 prototype 프로퍼티에 바인딩되어있는 객체가 프로토타입으로 전달된다.



```
function Person(name) {  
  this.name = name;  
}  
  
Person.prototype.sayHello = function() {  
  console.log(`Hi! My name is ${this.name}`);  
};  
  
const me = new Person("Lee");  
const you = new Person("Kim");  
  
me.sayHello() // Hi! My name is Lee  
you.sayHello() // Hi! My name is Kim
```

7. 객체 생성 방식과 프로토타입의 결정

생성자 함수



8. 프로토타입 체인



```
function Person(name) {  
  this.name = name;  
}  
  
Person.prototype.sayHello = function() {  
  console.log(`Hi! My name is ${this.name}`);  
};  
  
const me = new Person("Lee");  
  
console.log(me.hasOwnProperty("name"));
```

8. 프로토타입 체인



```
function Person(name) {  
  this.name = name;  
}  
  
Person.prototype.sayHello = function() {  
  console.log(`Hi! My name is ${this.name}`);  
};  
  
const me = new Person("Lee");  
  
console.log(me.hasOwnProperty("name"));
```

hasOwnProperty는 어떻게 호출할 수 있을까?

8. 프로토타입 체인



```
function Person(name) {  
  this.name = name;  
}  
  
Person.prototype.sayHello = function() {  
  console.log(`Hi! My name is ${this.name}`);  
};  
  
const me = new Person("Lee");  
  
console.log(me.hasOwnProperty("name"));
```

hasOwnProperty는 어떻게 호출할 수 있을까?

me 객체가 Person.prototype뿐만 아니라 Object.prototype을 상속받았다는 것을 의미한다.

8. 프로토타입 체인

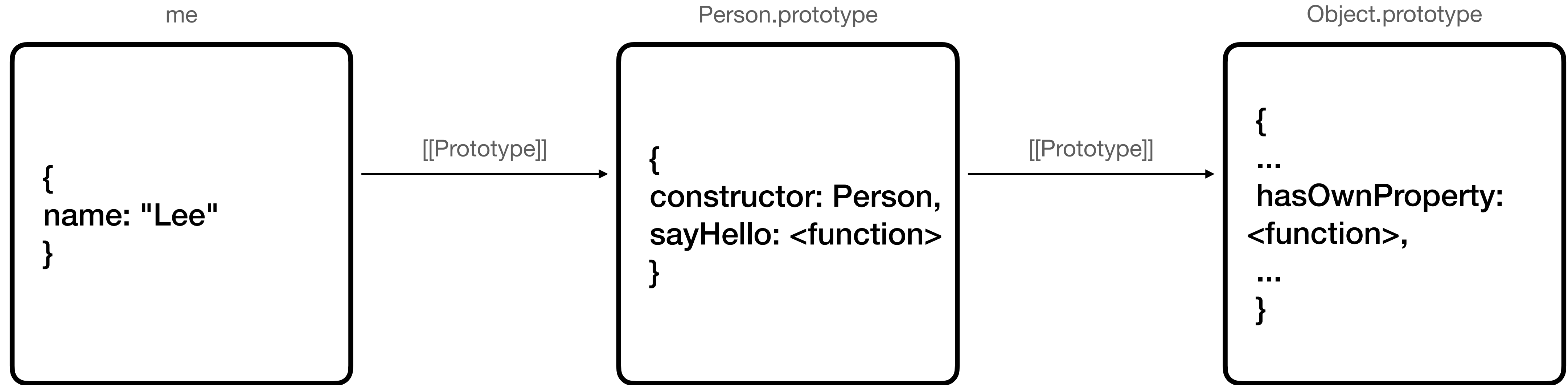


```
Object.getPrototypeOf(me) === Person.prototype // true
```



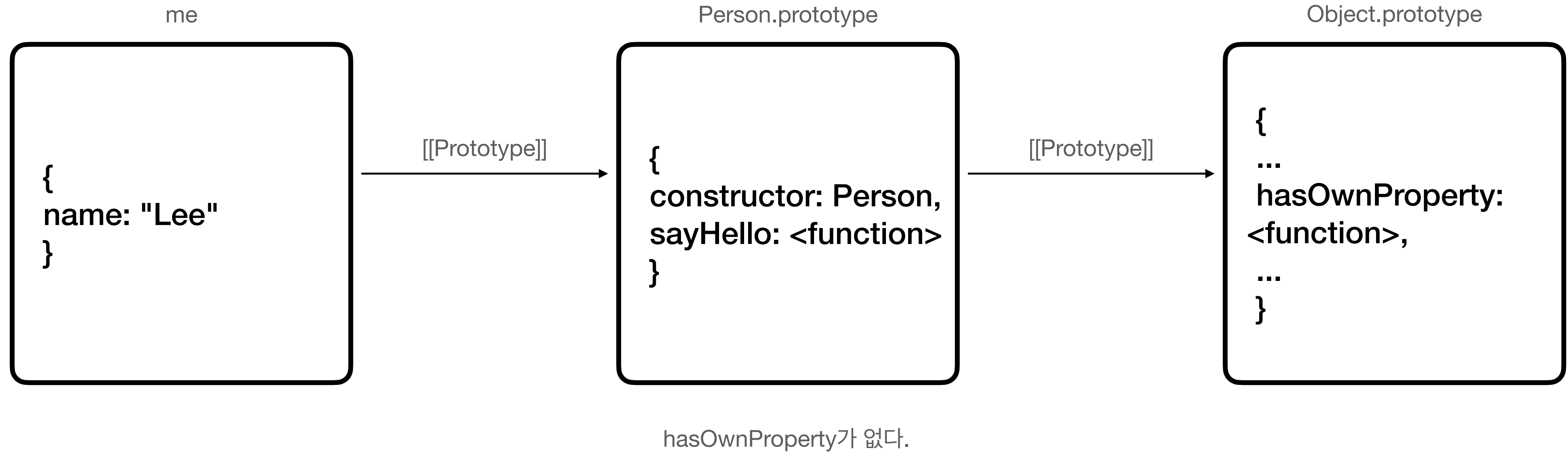
```
Object.getPrototypeOf(Person.prototype) === Object.prototype // true
```


8. 프로토타입 체인



hasOwnProperty가 없다.

8. 프로토타입 체인



8. 프로토타입 체인

