

# 19. Prototype-2

2022.03.19

# 목차

1. 프로토타입 생성 및 결정
2. 오버라이딩과 프로퍼티 재도입
3. 프로토타입의 교체
4. instanceof 연산자
5. 직접 상속
6. 정적 프로퍼티/메서드
7. 프로퍼티 존재 확인
8. 프로퍼티 열거

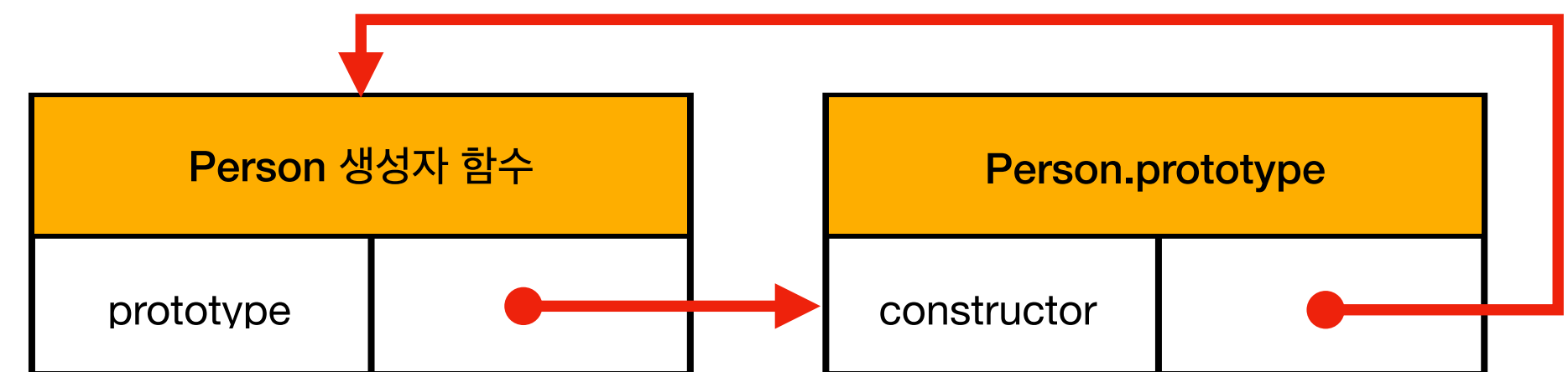
# 프로토타입 생성 및 결정



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

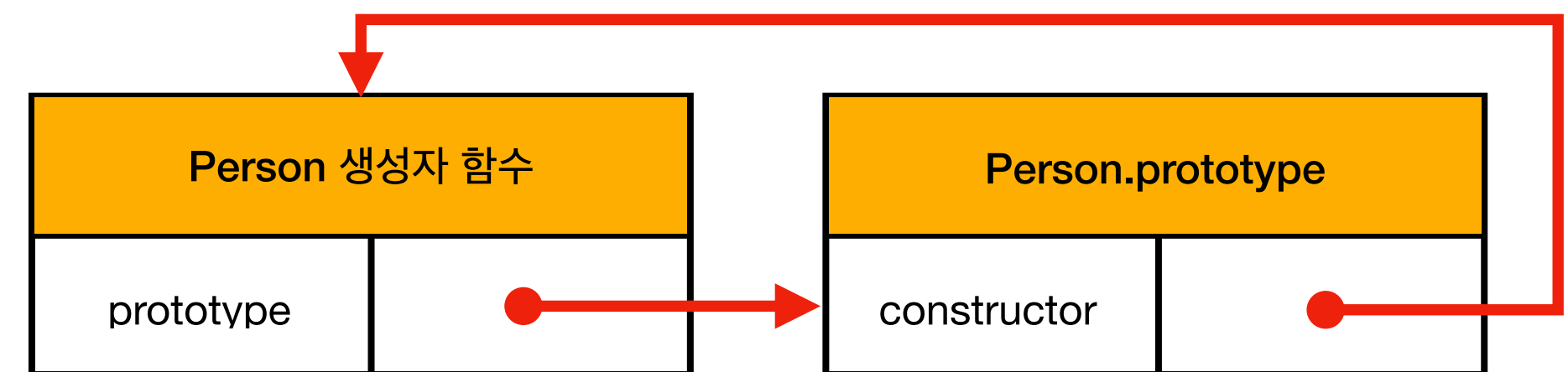
# 프로토타입 생성 및 결정

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



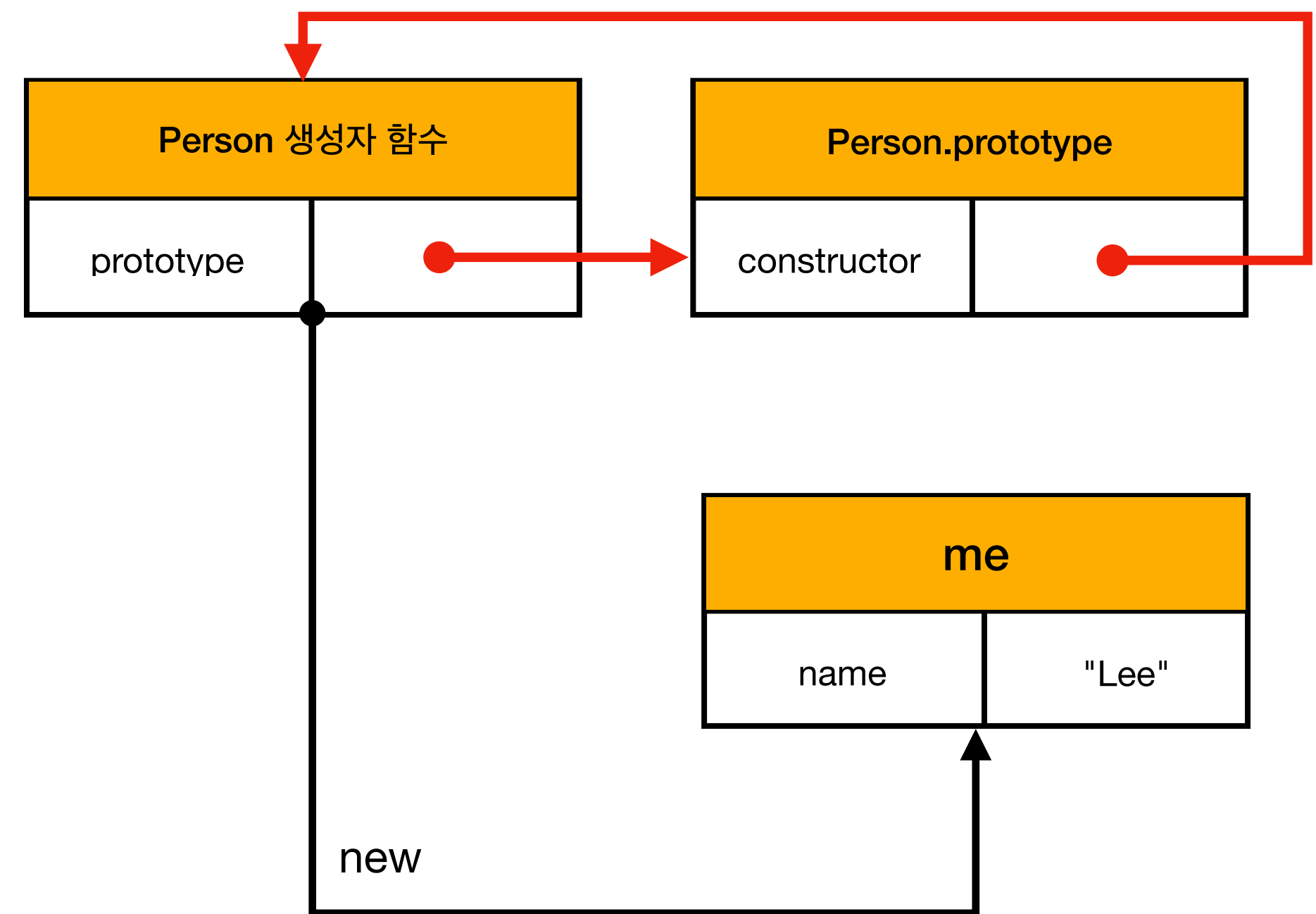
# 프로토타입 생성 및 결정

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



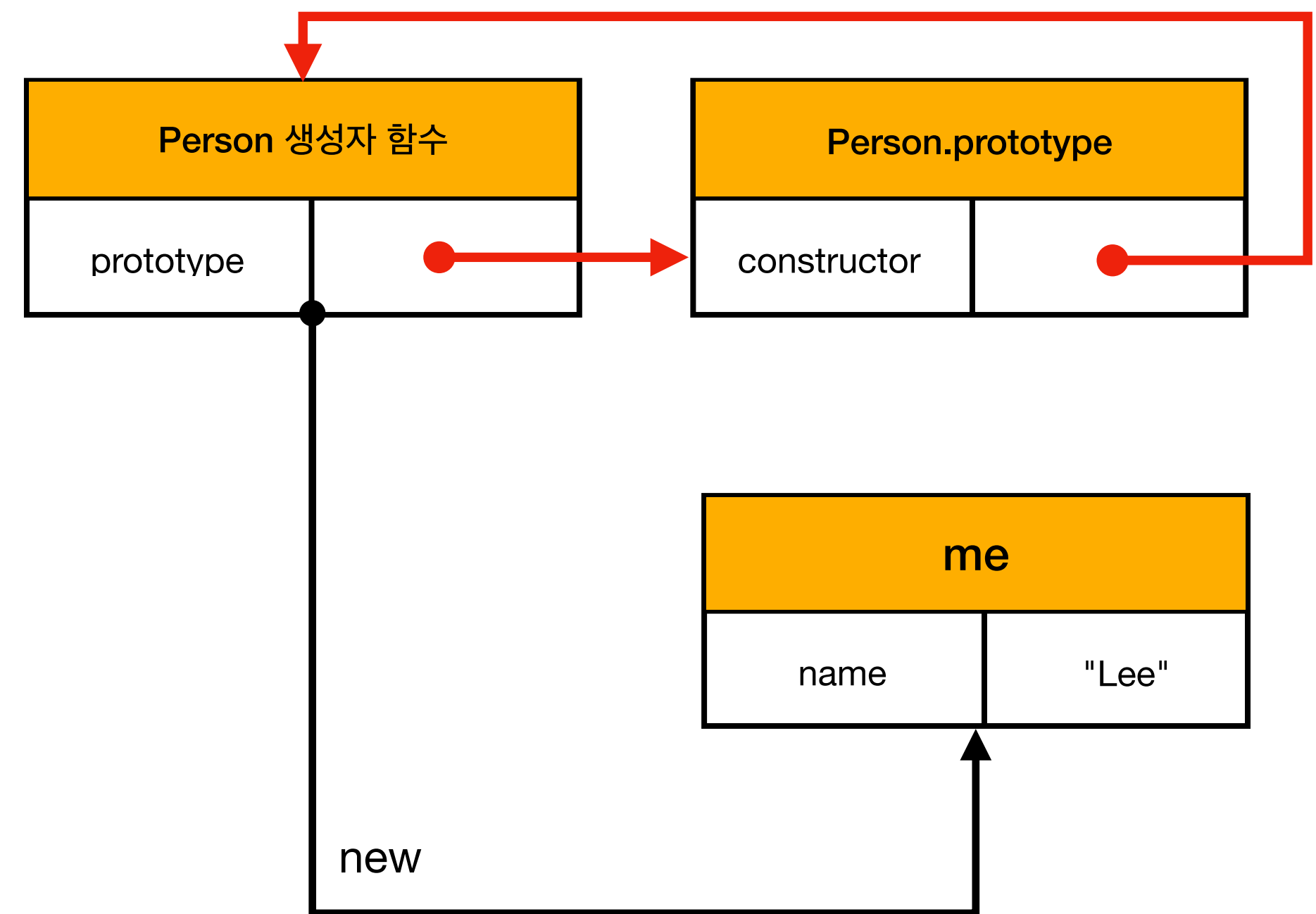
# 프로토타입 생성 및 결정

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



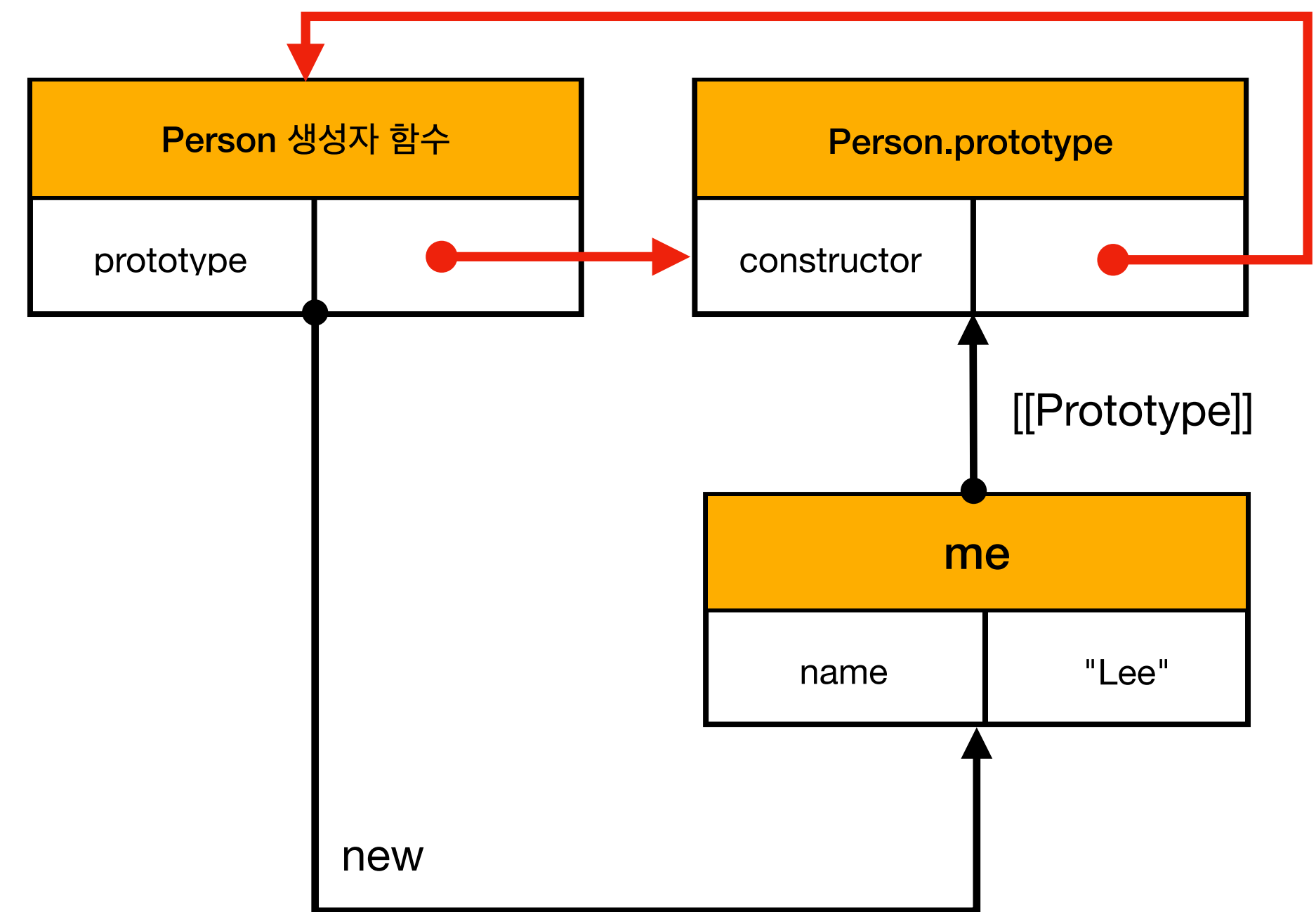
# 프로토타입 생성 및 결정

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



# 프로토타입 생성 및 결정

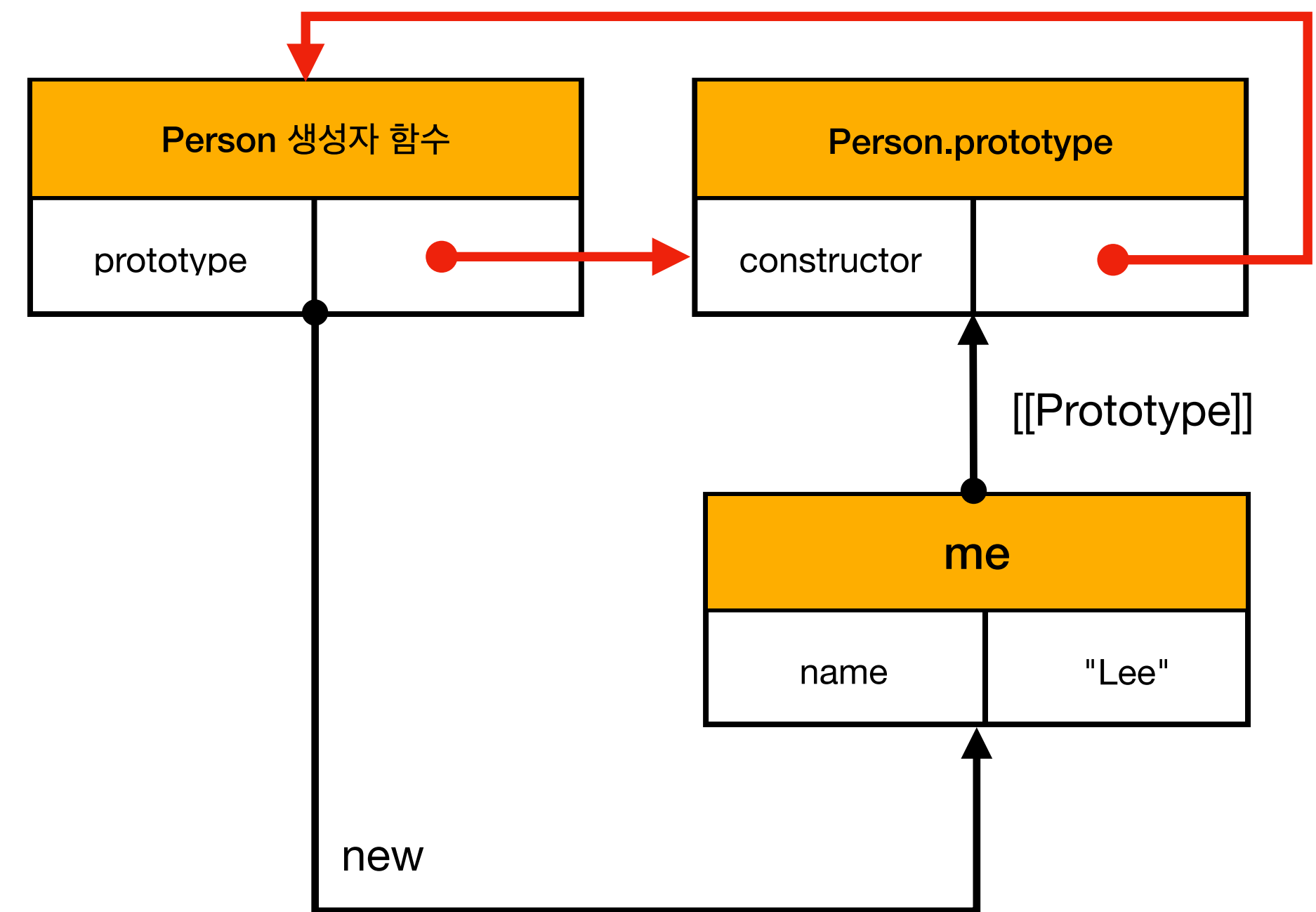
```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
console.log(Person.prototype === me.__proto__); // true
```





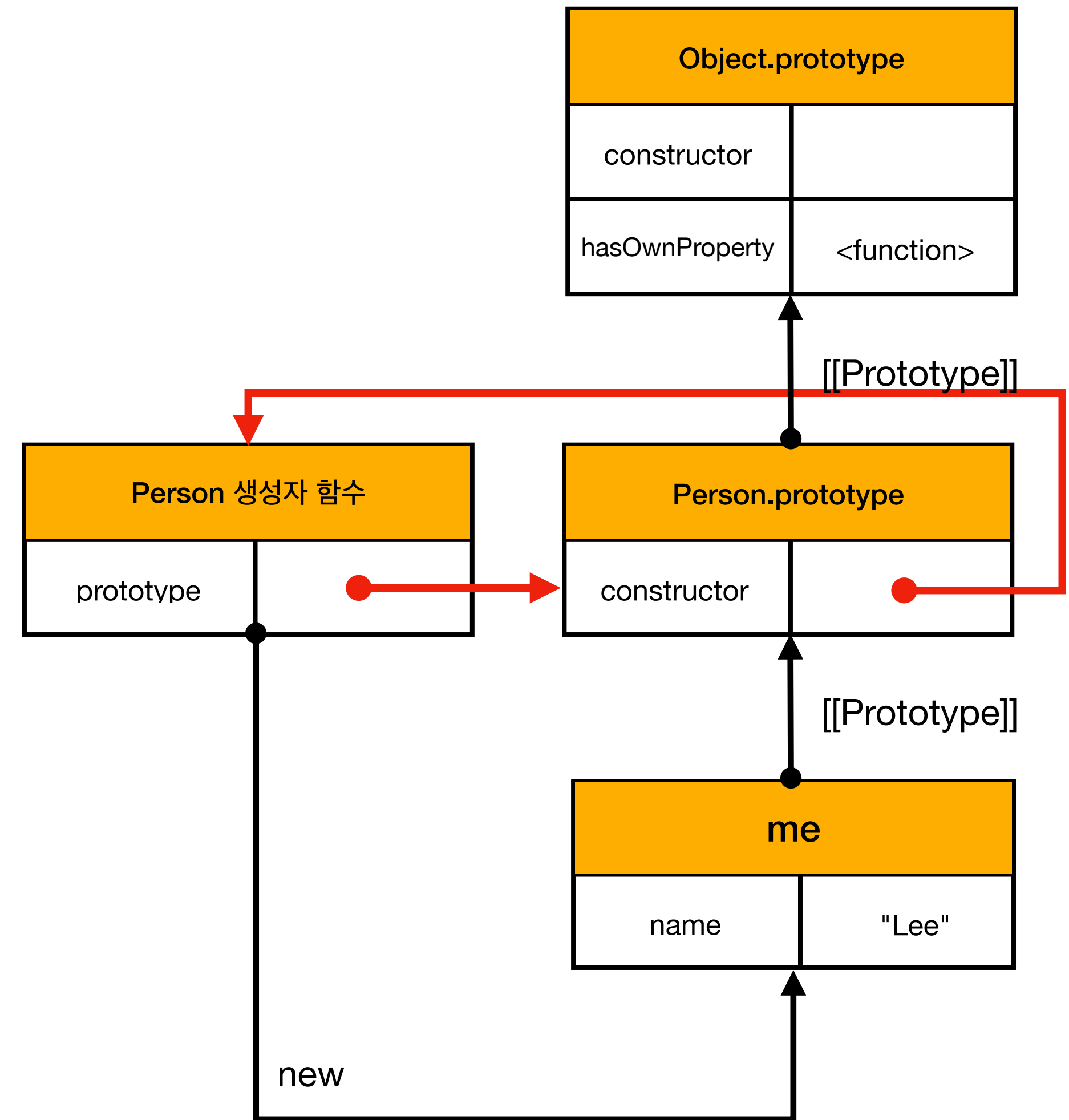
# 프로토타입 생성 및 결정

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



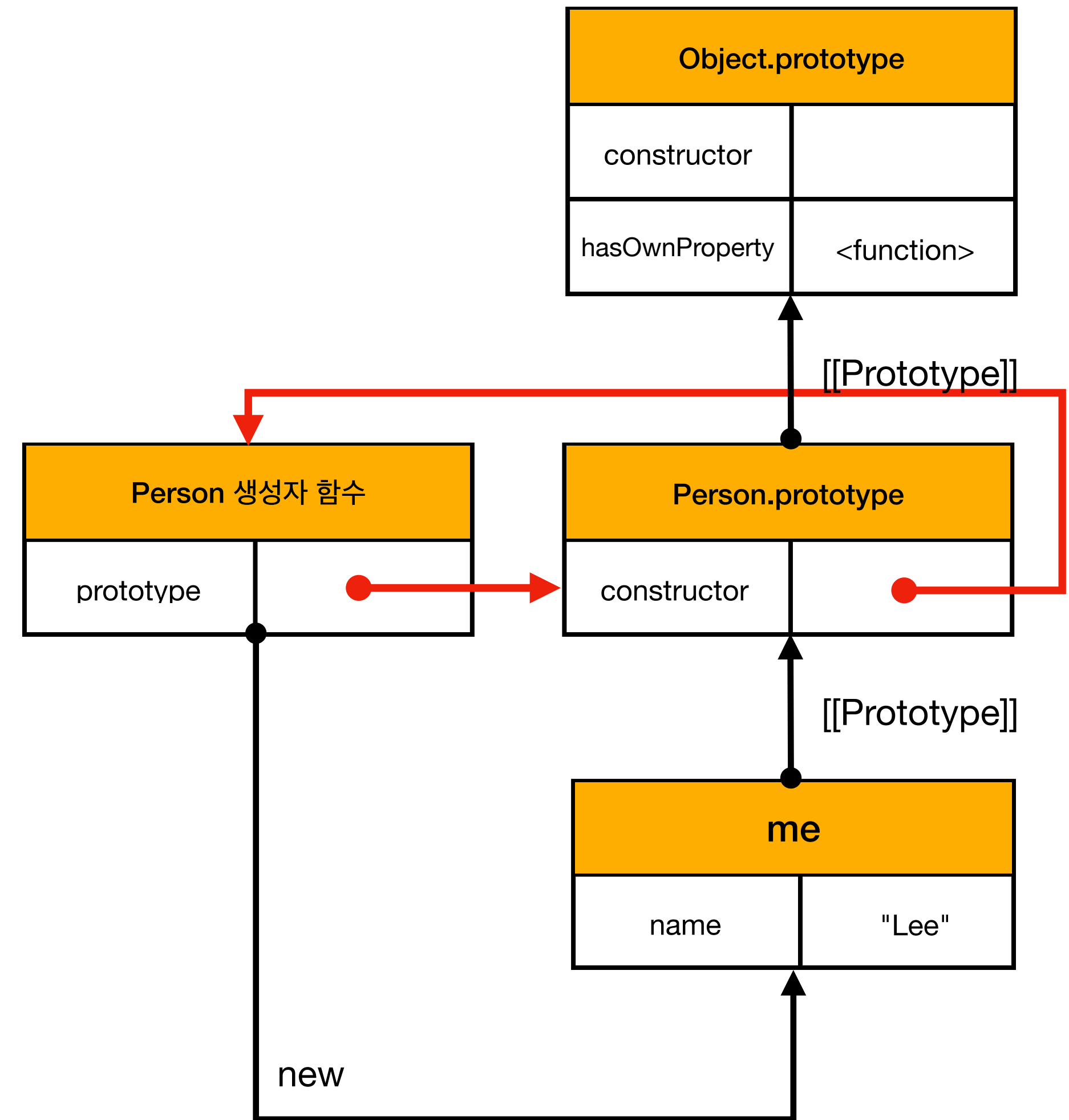
# 프로토타입 생성 및 결정

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



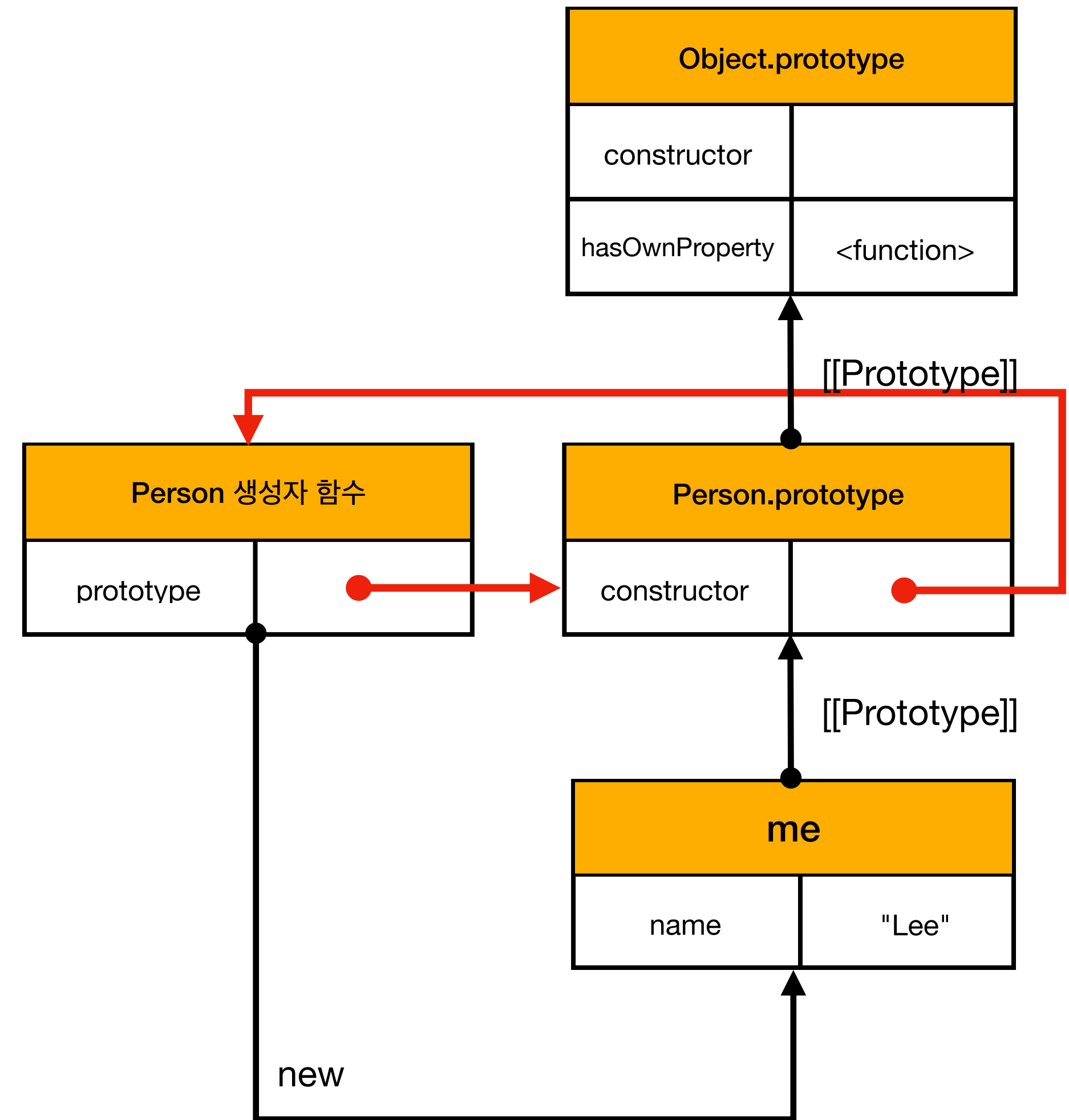
# 프로토타입 생성 및 결정

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



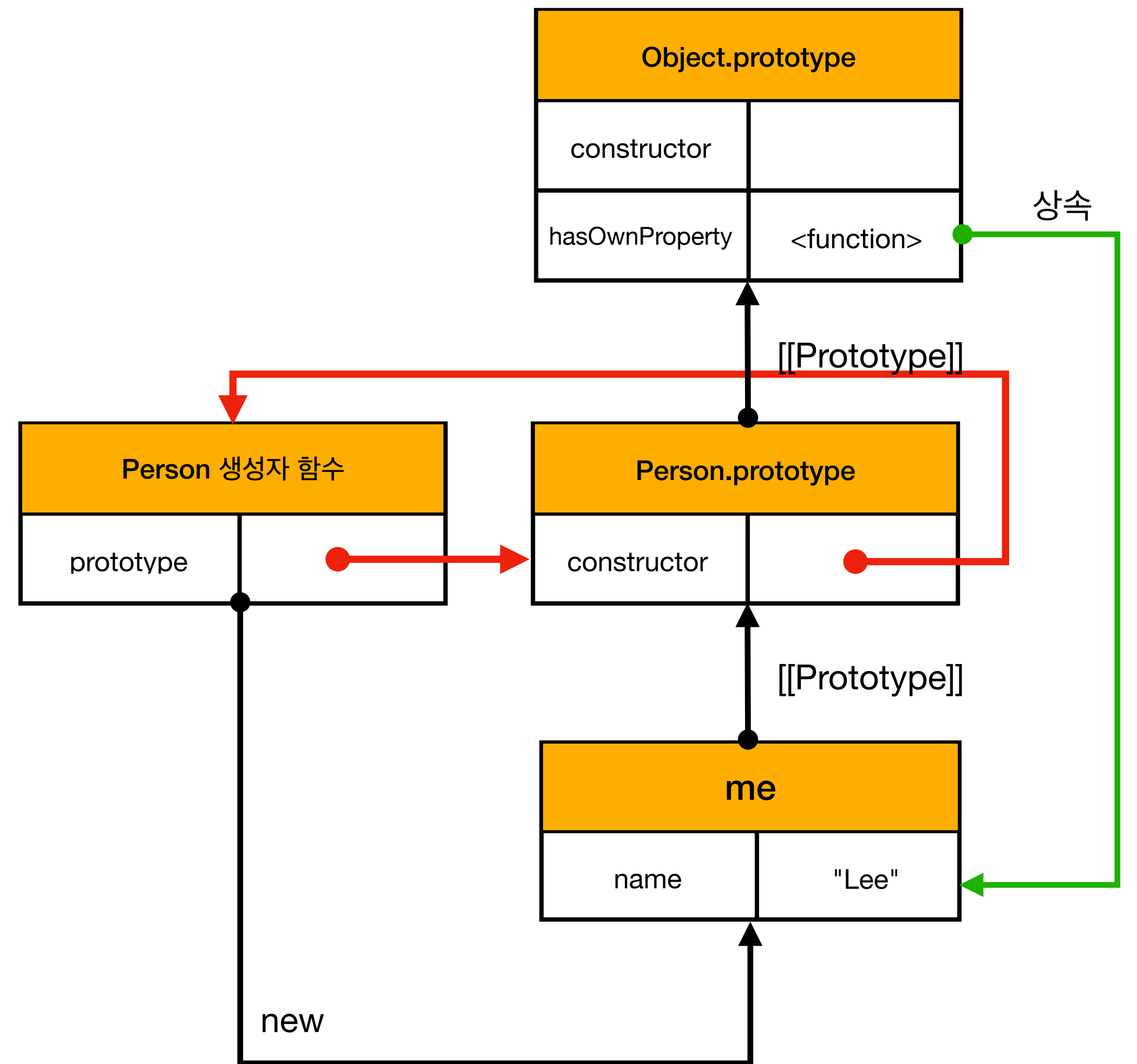
# 프로토타입 생성 및 결정

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



# 프로토타입 생성 및 결정

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



# 오버라이딩과 프로퍼티 재도잉

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

  // 프로토타입 메서드
  Person.prototype.sayHello = function () {
    console.log(`Hi! My name is ${this.name}`);
  };

  return Person;
})();

const me = new Person("Lee");

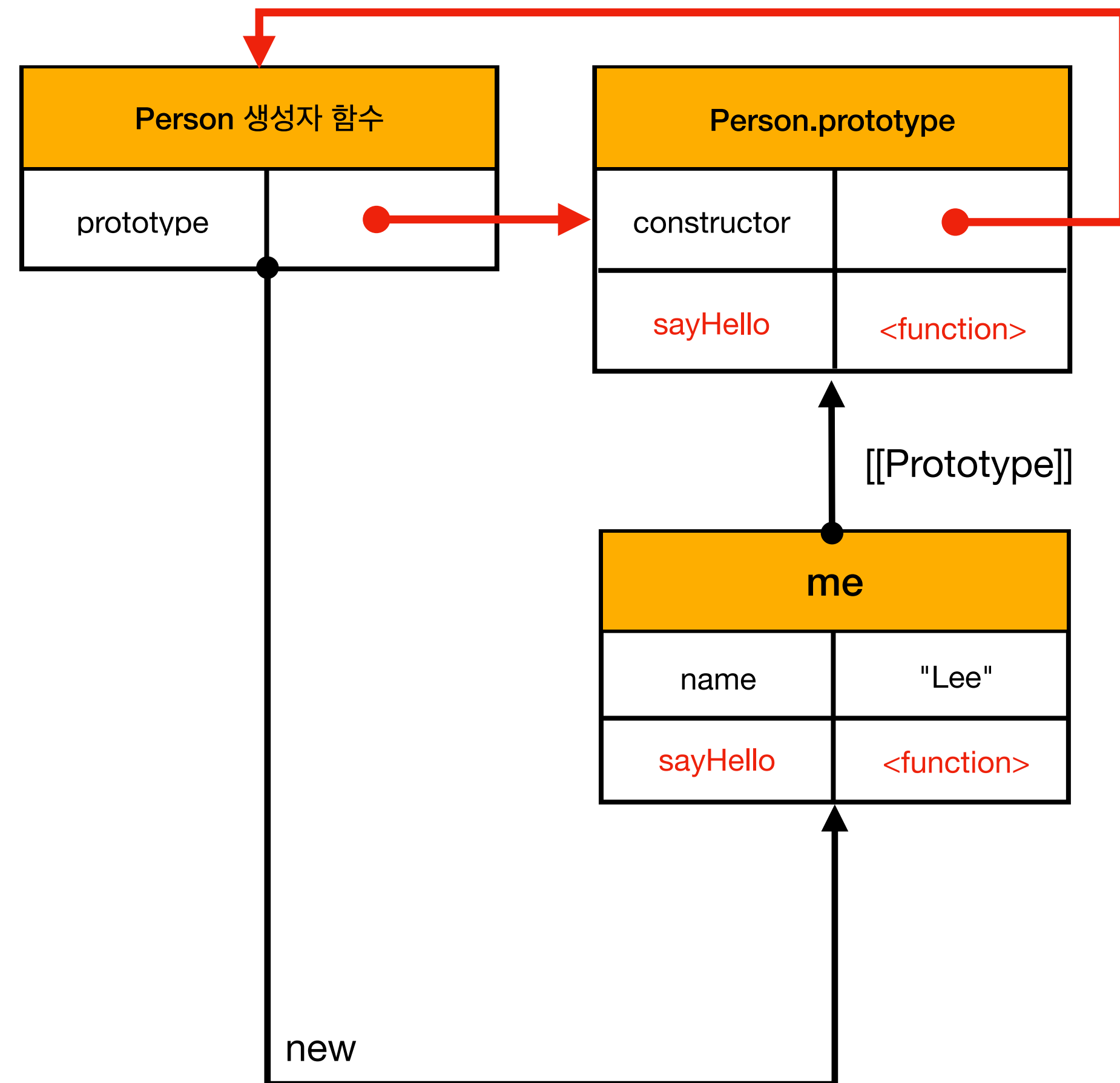
// 인스턴스 메서드
me.sayHello = function () {
  console.log(`Hey! My name is ${this.name}`);
};

// 인스턴스 메서드가 호출된다.
me.sayHello() // Hey! My name is Lee
```

# 오버라이딩과 프로퍼티 재도잉



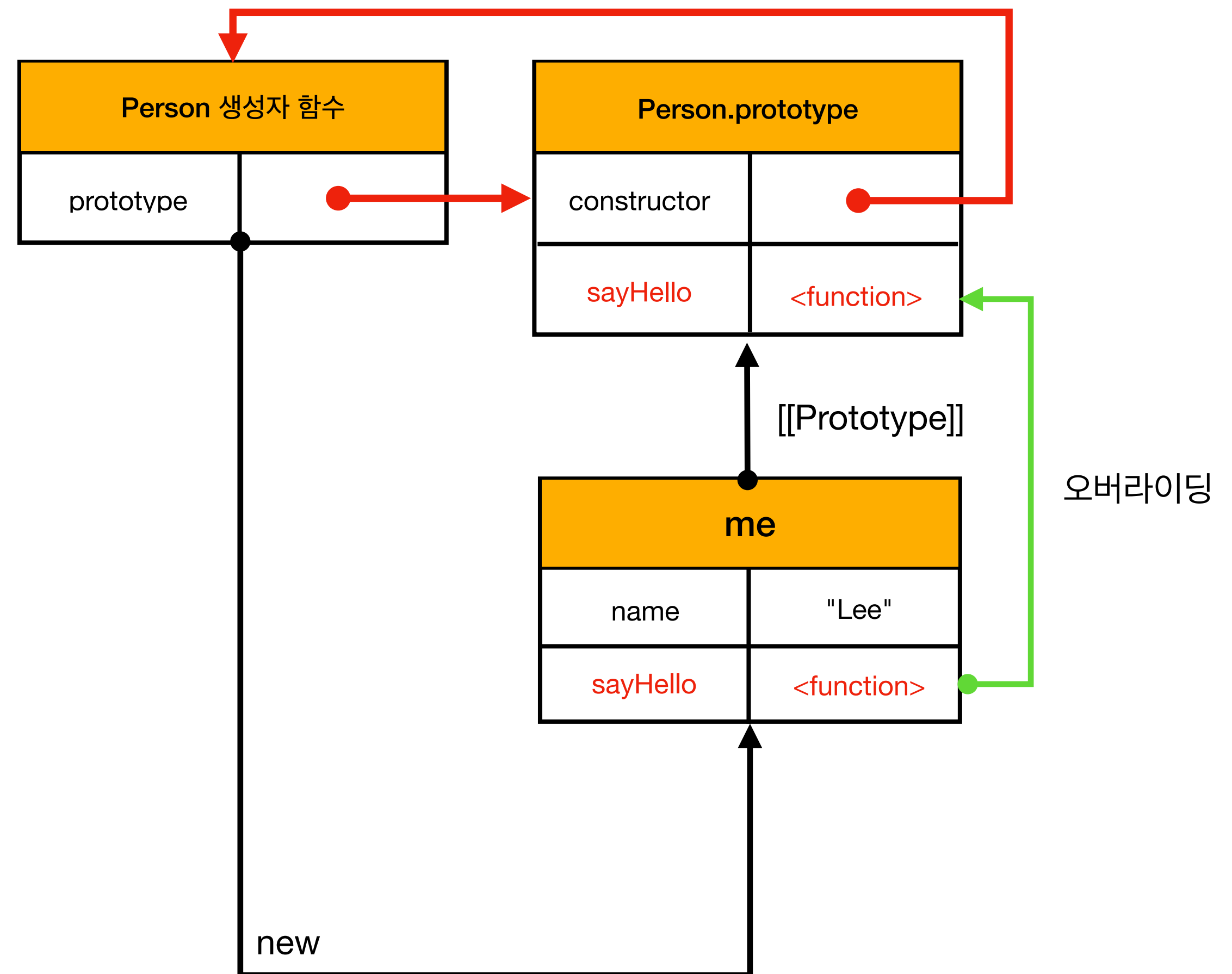
```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  Person.prototype.sayHello = function () {  
    console.log(`Hi! My name is ${this.name}`);  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");  
  
// 인스턴스 메서드  
me.sayHello = function () {  
  console.log(`Hey! My name is ${this.name}`);  
};  
  
// 인스턴스 메서드가 호출된다.  
me.sayHello() // Hey! My name is Lee
```



# 오버라이딩과 프로퍼티 재도잉



```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  Person.prototype.sayHello = function () {  
    console.log(`Hi! My name is ${this.name}`);  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");  
  
// 인스턴스 메서드  
me.sayHello = function () {  
  console.log(`Hey! My name is ${this.name}`);  
};  
  
// 인스턴스 메서드가 호출된다.  
me.sayHello() // Hey! My name is Lee
```

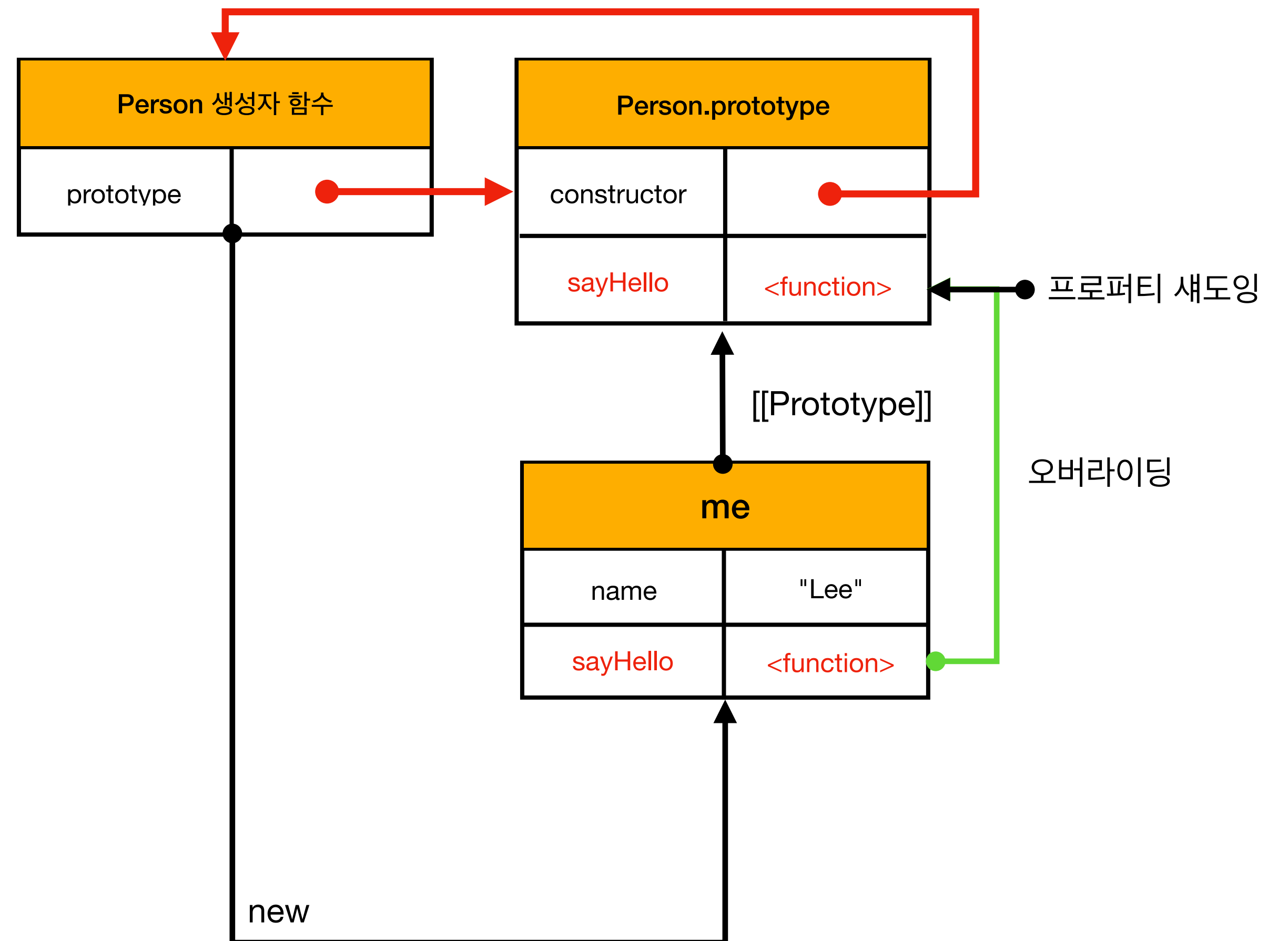




# 오버라이딩과 프로퍼티 재도잉



```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  Person.prototype.sayHello = function () {  
    console.log(`Hi! My name is ${this.name}`);  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");  
  
// 인스턴스 메서드  
me.sayHello = function () {  
  console.log(`Hey! My name is ${this.name}`);  
};  
  
// 인스턴스 메서드가 호출된다.  
me.sayHello() // Hey! My name is Lee
```

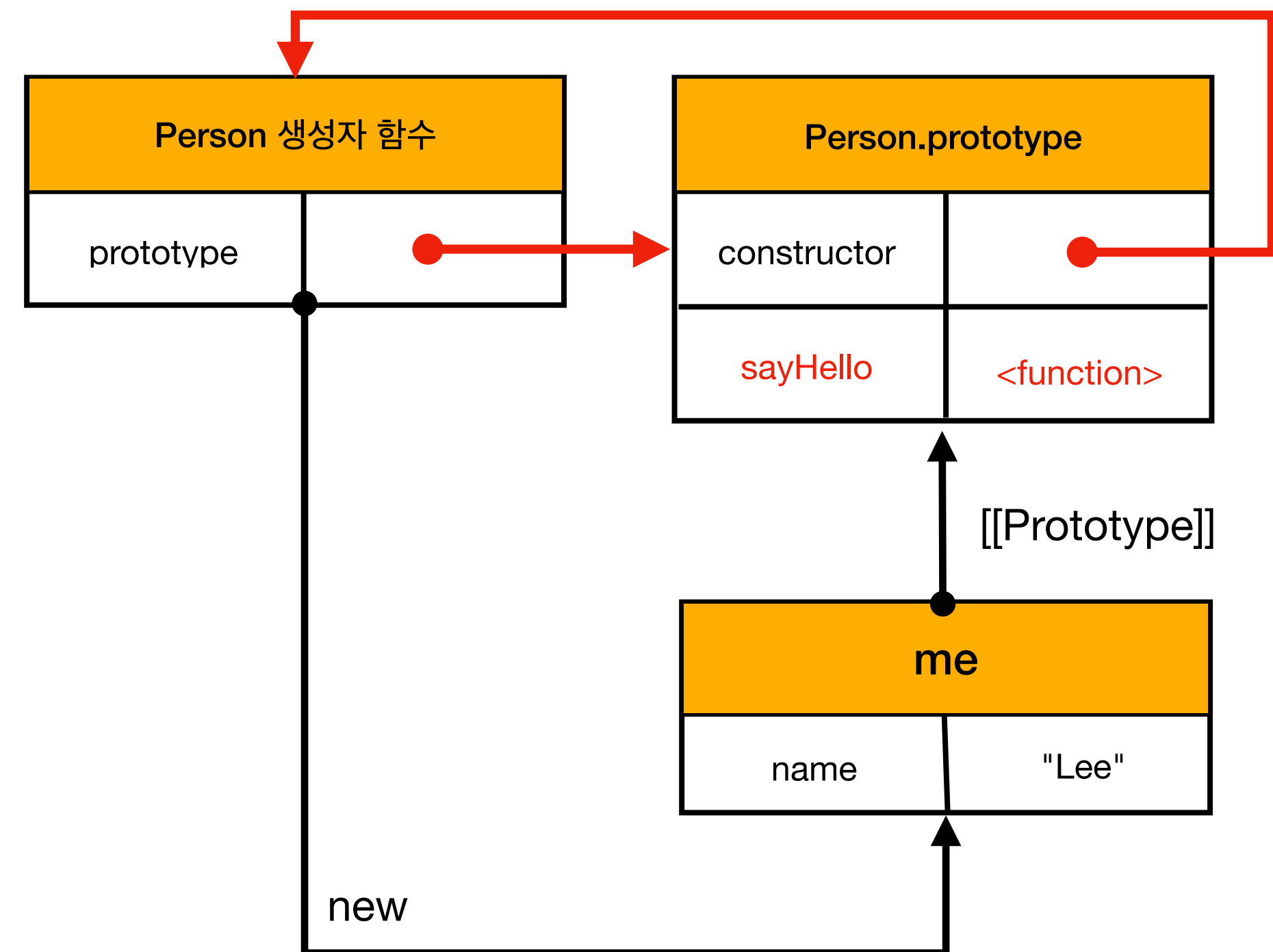


# 오버라이딩과 프로퍼티 재도잉

```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  Person.prototype.sayHello = function () {  
    console.log(`Hi! My name is ${this.name}`);  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");  
  
// 인스턴스 메서드  
me.sayHello = function () {  
  console.log(`Hey! My name is ${this.name}`);  
};  
  
// 인스턴스 메서드가 호출된다.  
me.sayHello() // Hey! My name is Lee  
  
delete me.sayHello  
  
// 프로토타입 메서드가 호출된다.  
me.sayHello() // Hi! My name is Lee
```

# 오버라이딩과 프로퍼티 재도잉

```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  Person.prototype.sayHello = function () {  
    console.log(`Hi! My name is ${this.name}`);  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");  
  
// 인스턴스 메서드  
me.sayHello = function () {  
  console.log(`Hey! My name is ${this.name}`);  
};  
  
// 인스턴스 메서드가 호출된다.  
me.sayHello() // Hey! My name is Lee  
  
delete me.sayHello  
  
// 프로토타입 메서드가 호출된다.  
me.sayHello() // Hi! My name is Lee
```



# 프로토타입의 교체

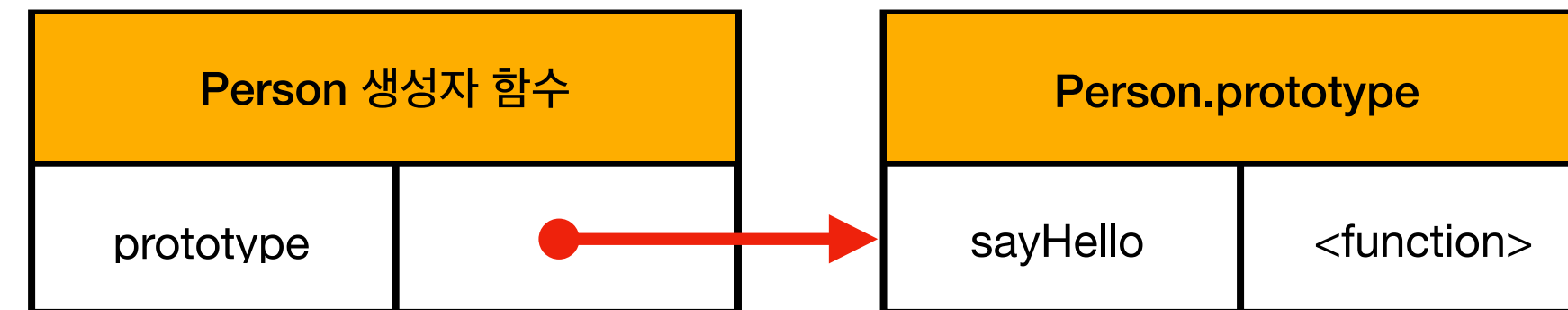
## 생성자 함수에 의한 프로토타입의 교체

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

# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체

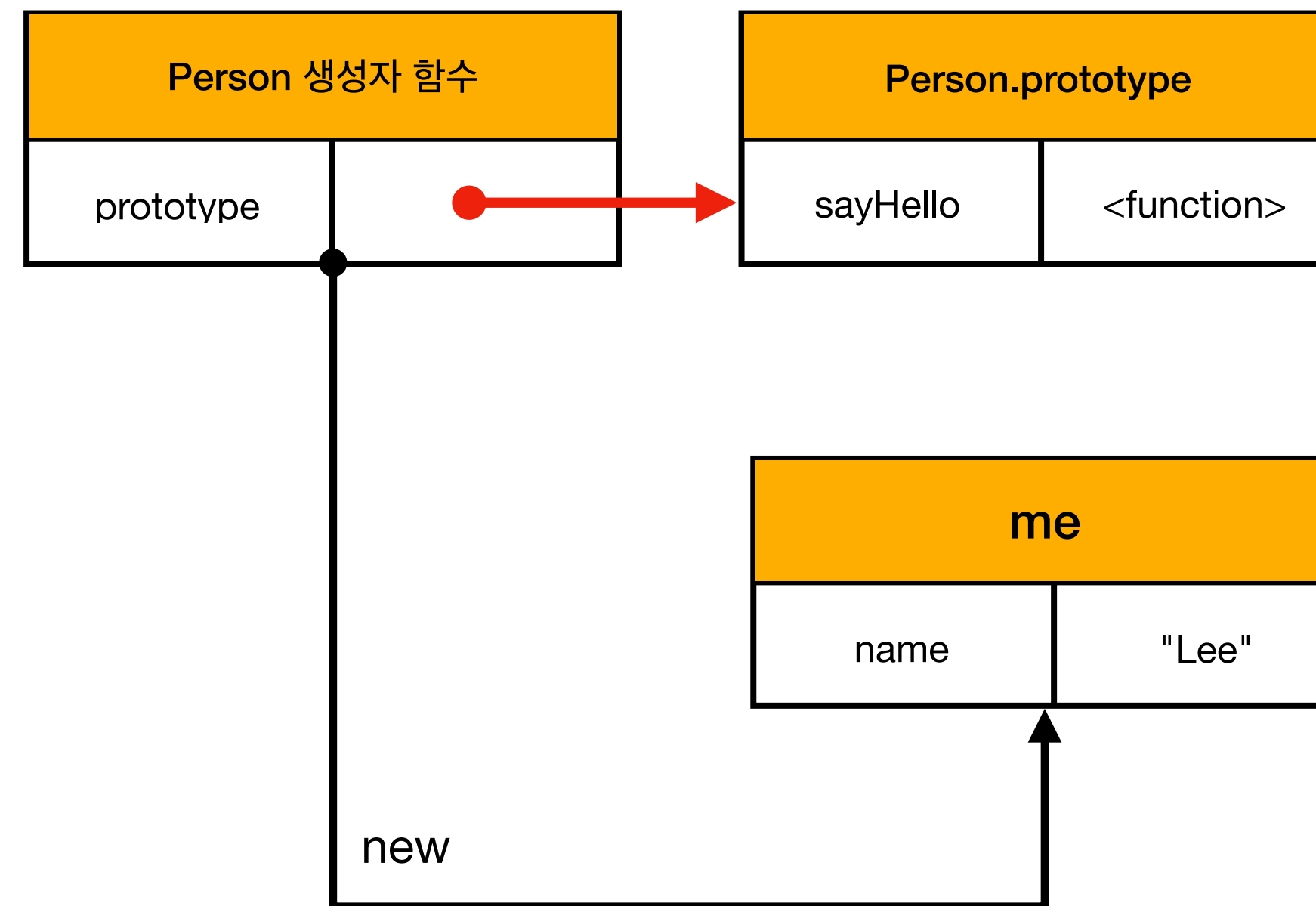
```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  Person.prototype = {  
    sayHello() {  
      console.log(`Hi! My name is ${this.name}`);  
    },  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");
```



# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체

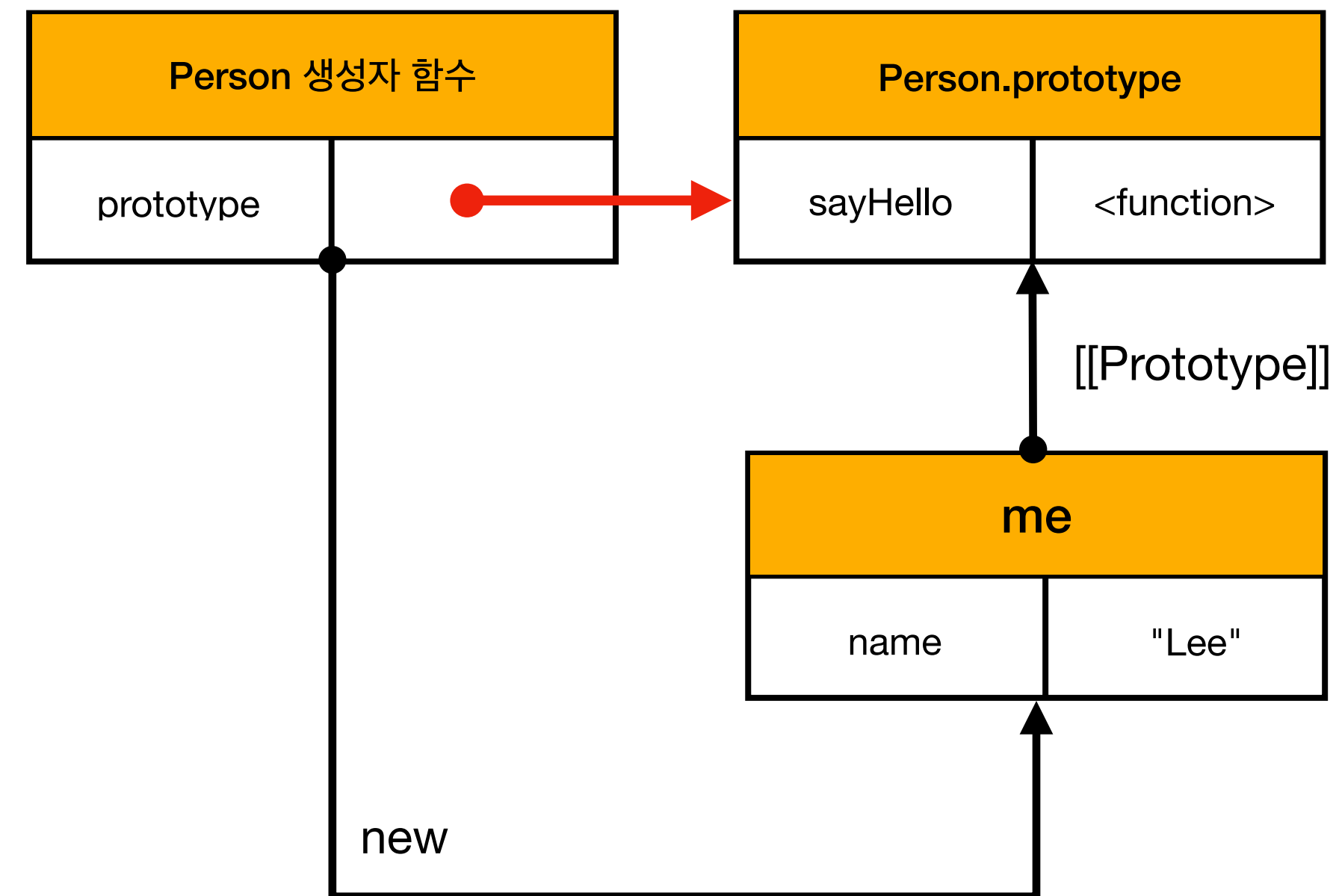
```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  Person.prototype = {  
    sayHello() {  
      console.log(`Hi! My name is ${this.name}`);  
    },  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");
```



# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체

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

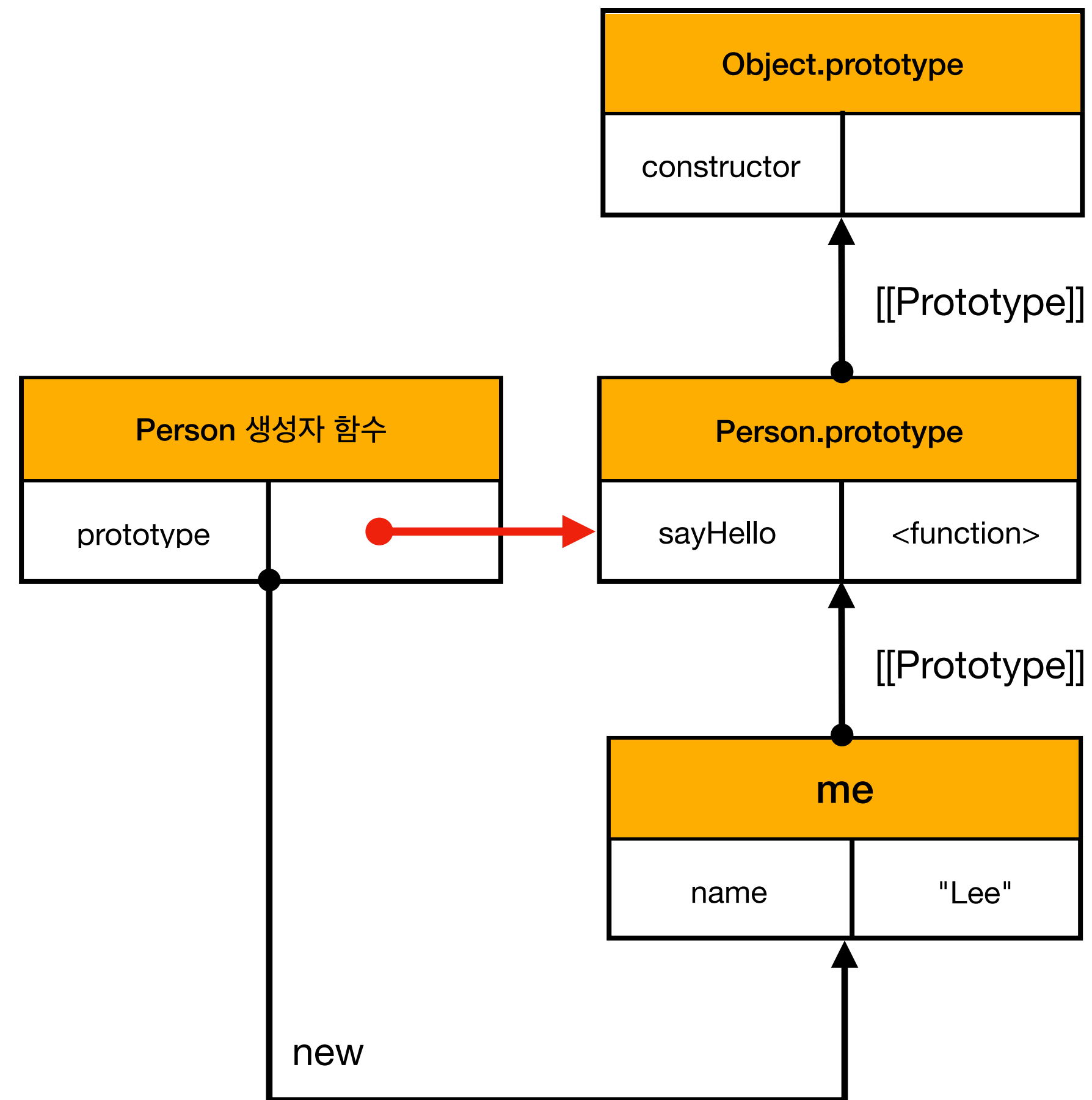


# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체



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



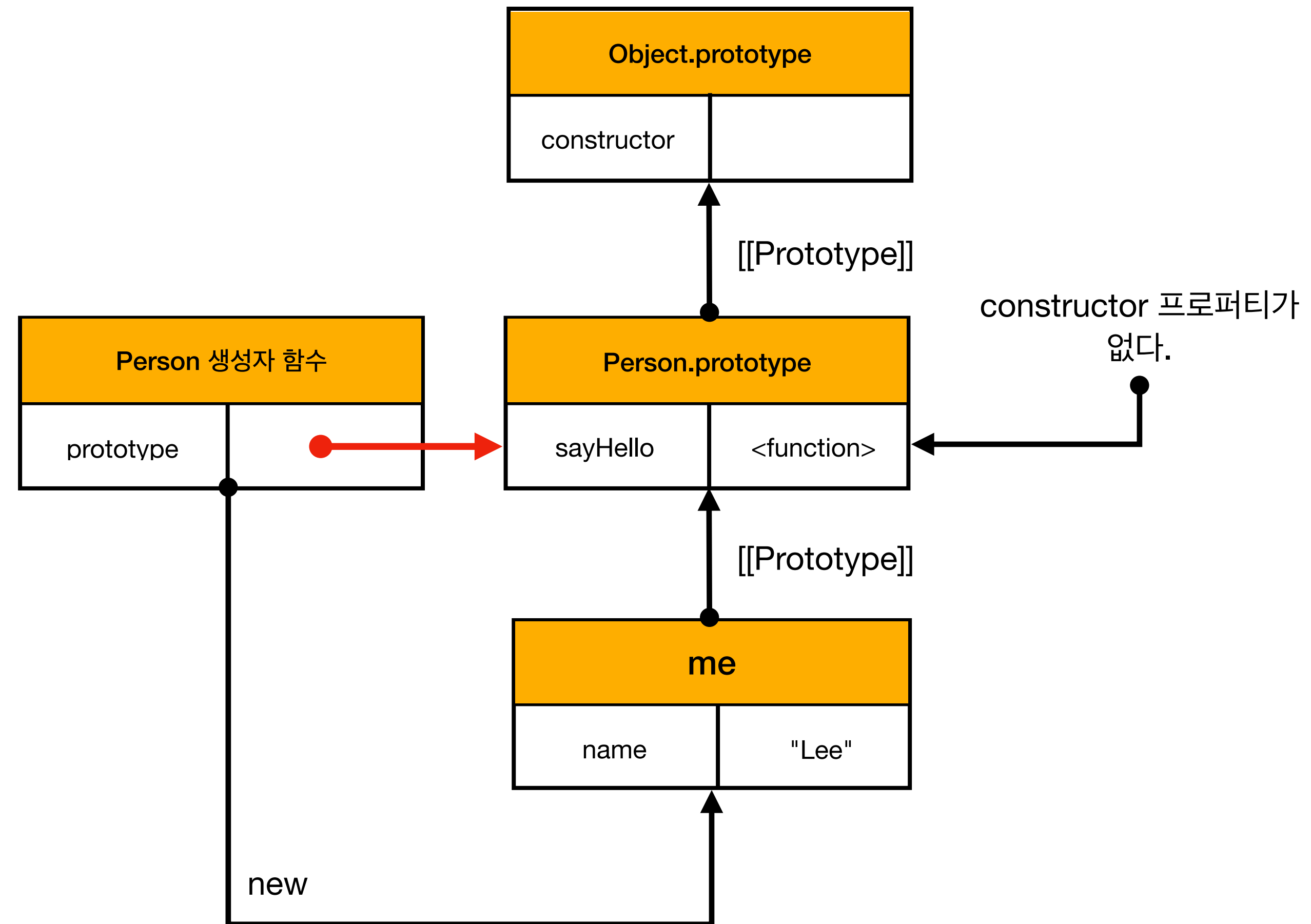


# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체



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



# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체



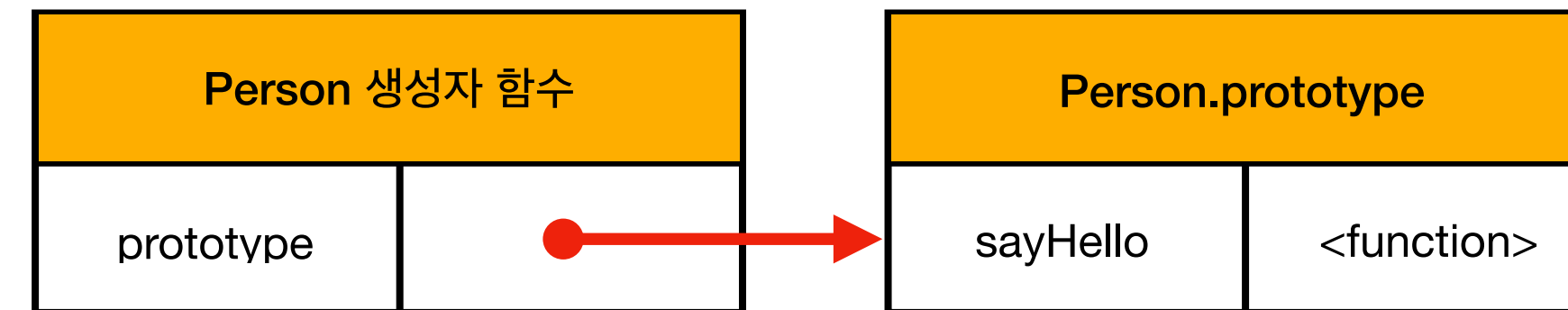
```
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // false  
console.log(me.constructor === Object); // true
```

# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체



```
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // false  
console.log(me.constructor === Object); // true
```

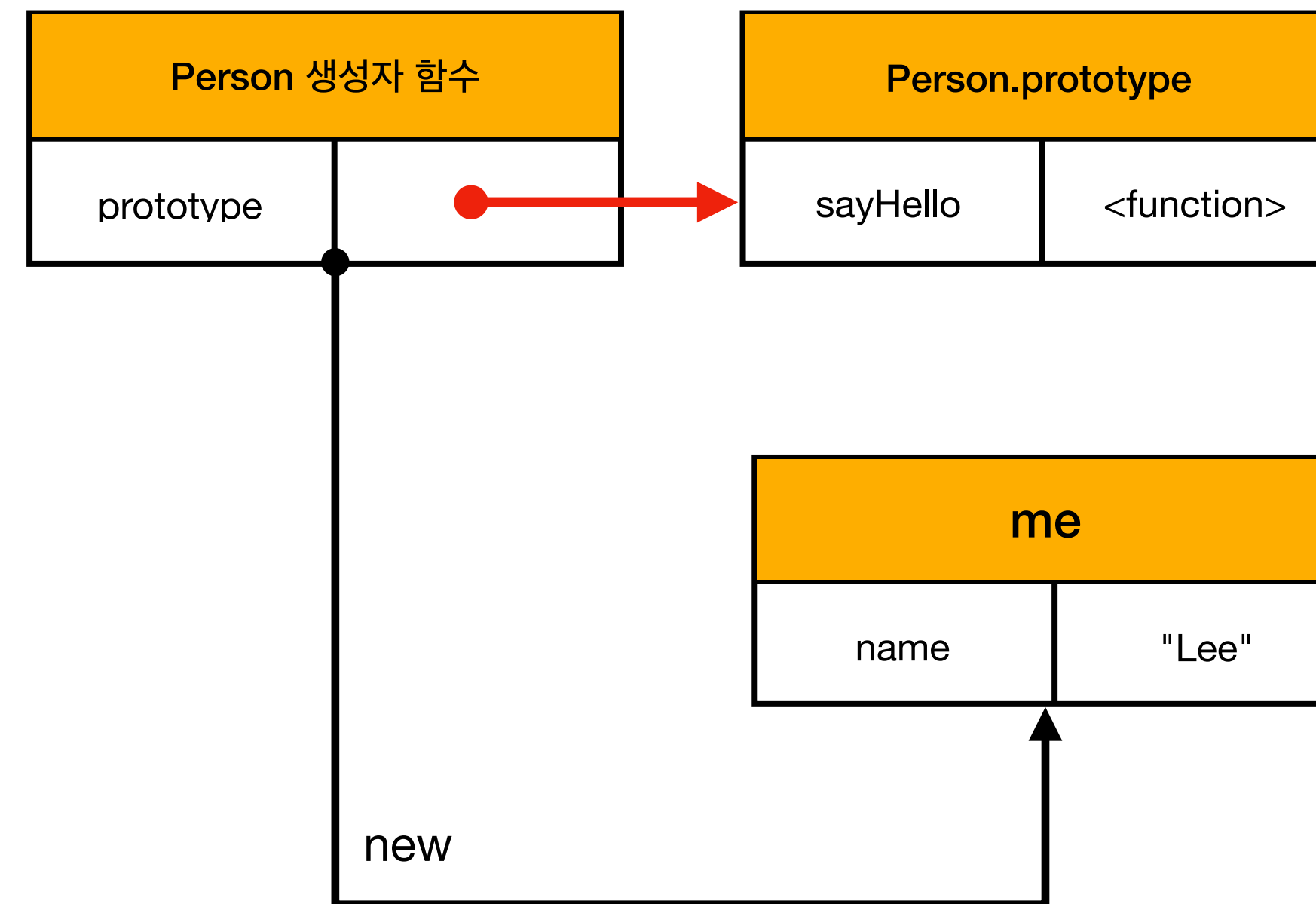


# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체



```
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // false  
console.log(me.constructor === Object); // true
```

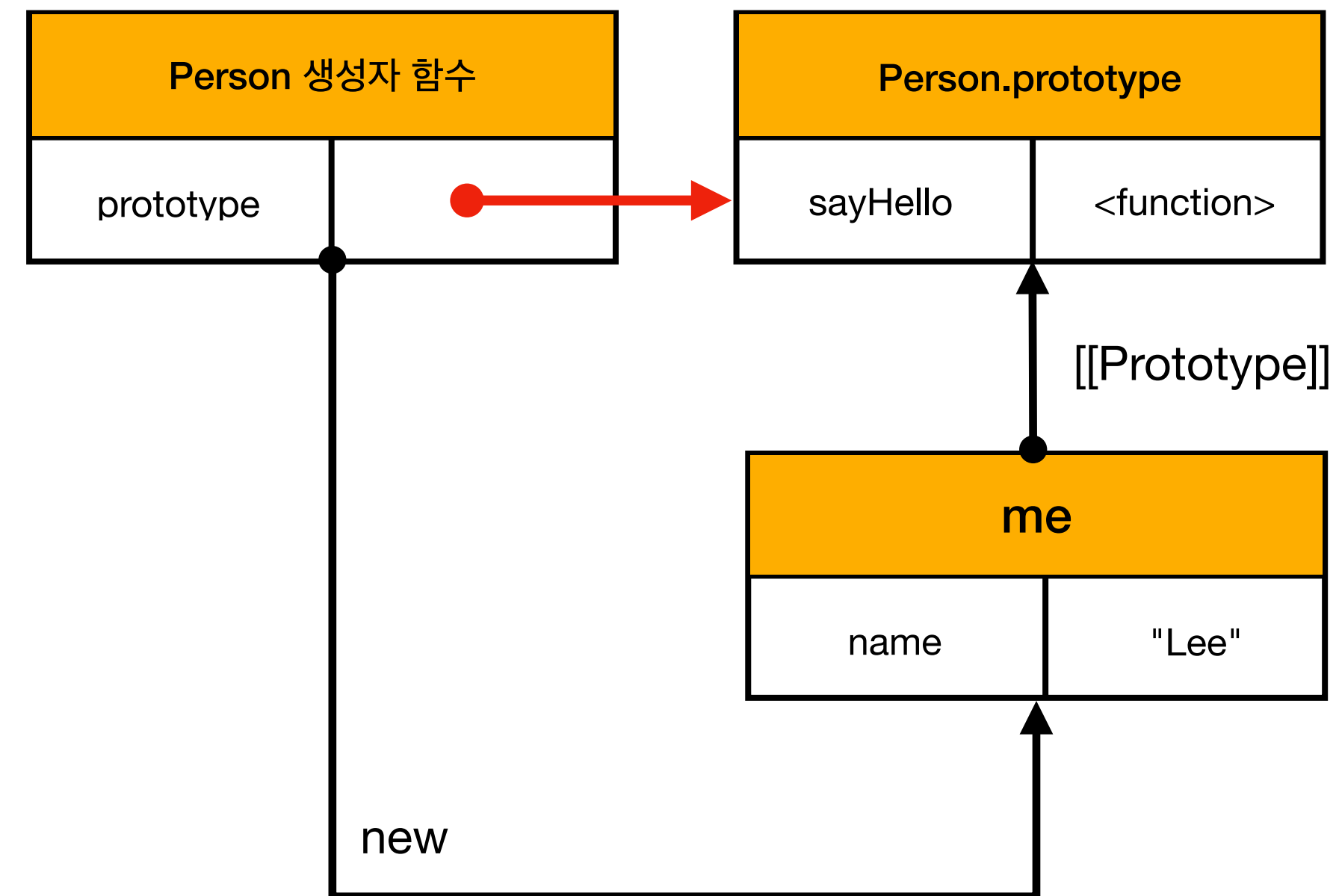


# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체



```
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // false  
console.log(me.constructor === Object); // true
```

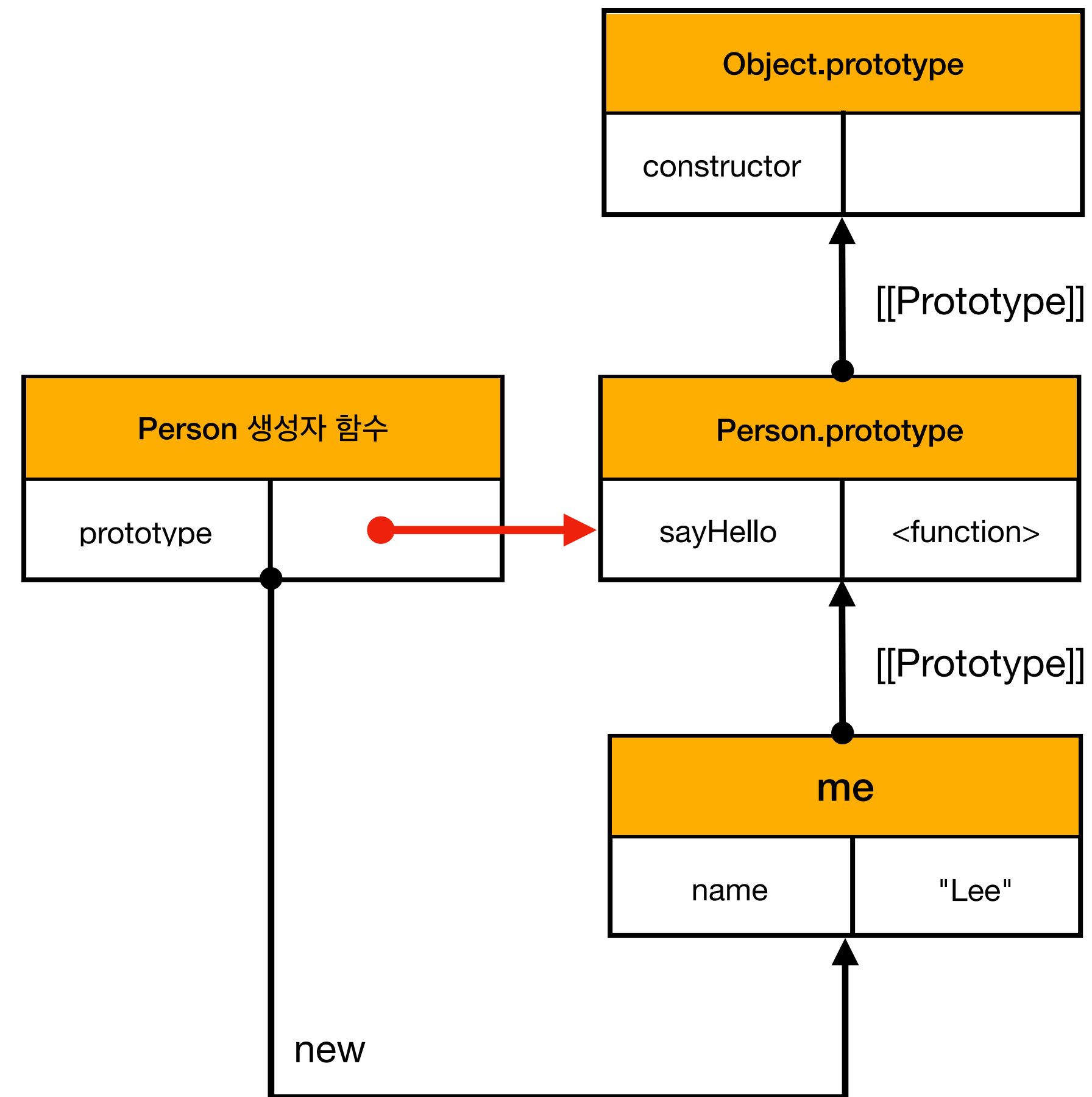


# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체



```
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // false  
console.log(me.constructor === Object); // true
```



# 프로토타입의 교체

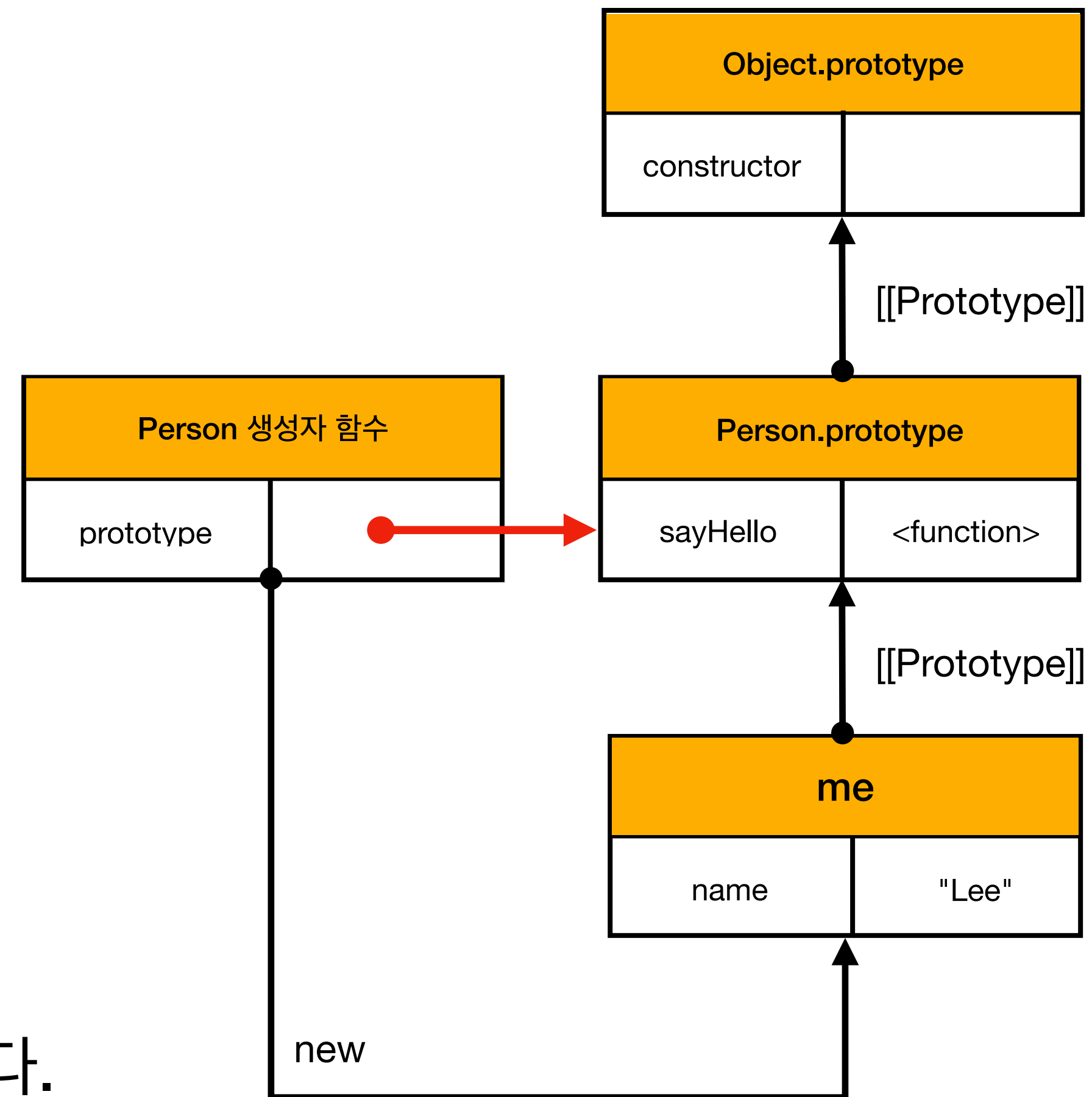
## 생성자 함수에 의한 프로토타입의 교체



```
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // false  
console.log(me.constructor === Object); // true
```

프로토타입 체인을 따라

Object.prototype의 constructor 프로퍼티가 검색된다.



# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체

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

  Person.prototype = {
    constructor: Person,
    sayHello() {
      console.log(`Hi! My name is ${this.name}`);
    },
  };

  return Person;
})();

const me = new Person("Lee");

console.log(me.constructor === Person); // true
console.log(me.constructor === Object); // false
```



# 프로토타입의 교체

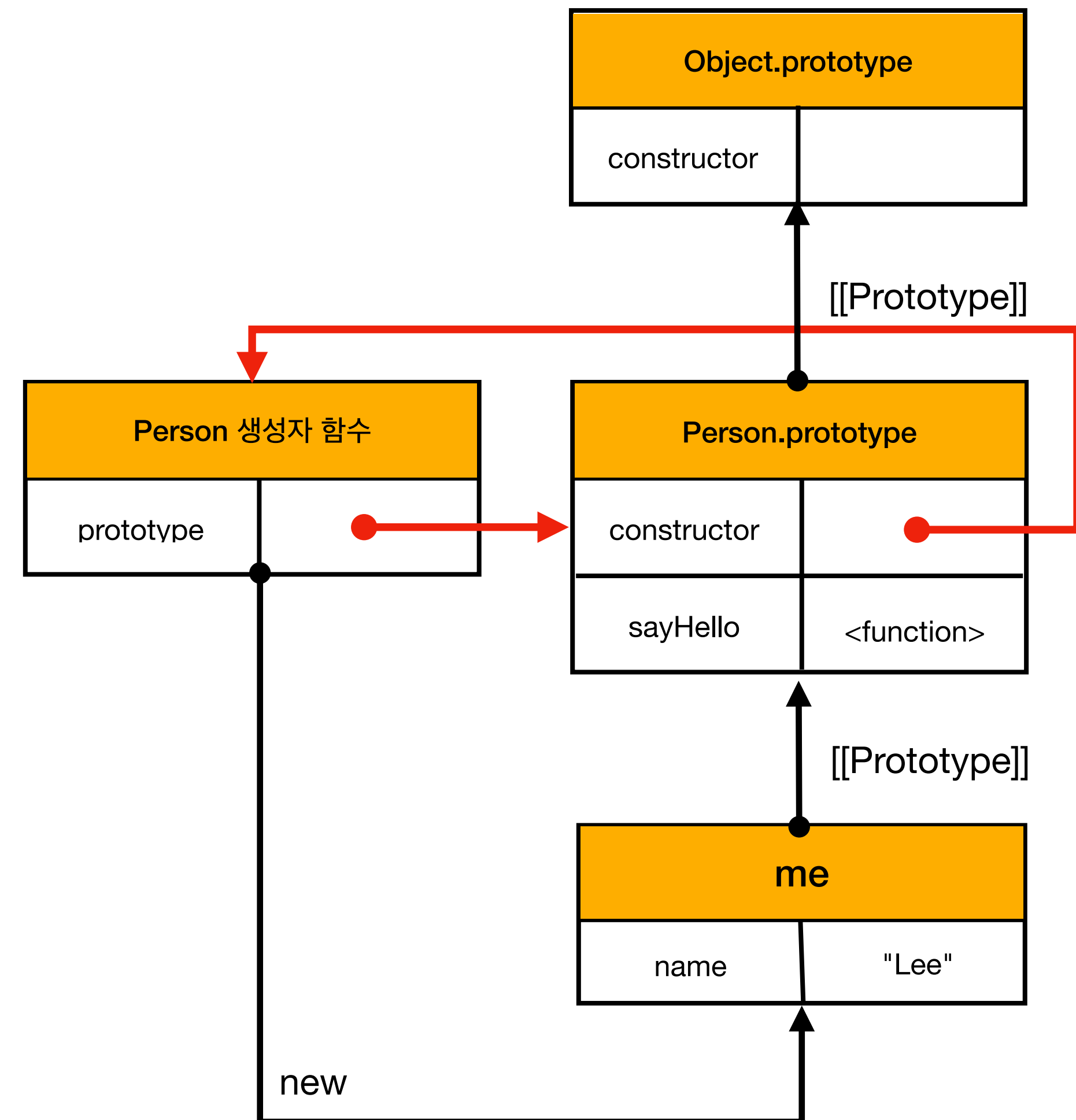
## 생성자 함수에 의한 프로토타입의 교체

```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  Person.prototype = {  
    constructor: Person,  
    sayHello() {  
      console.log(`Hi! My name is ${this.name}`);  
    },  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // true  
console.log(me.constructor === Object); // false
```

# 프로토타입의 교체

## 생성자 함수에 의한 프로토타입의 교체

```
const Person = (function () {  
  function Person(name) {  
    this.name = name;  
  }  
  
  Person.prototype = {  
    constructor: Person,  
    sayHello() {  
      console.log(`Hi! My name is ${this.name}`);  
    },  
  };  
  
  return Person;  
})();  
  
const me = new Person("Lee");  
  
console.log(me.constructor === Person); // true  
console.log(me.constructor === Object); // false
```



# 프로토타입의 교체

## 인스턴스에 의한 프로토타입의 교체



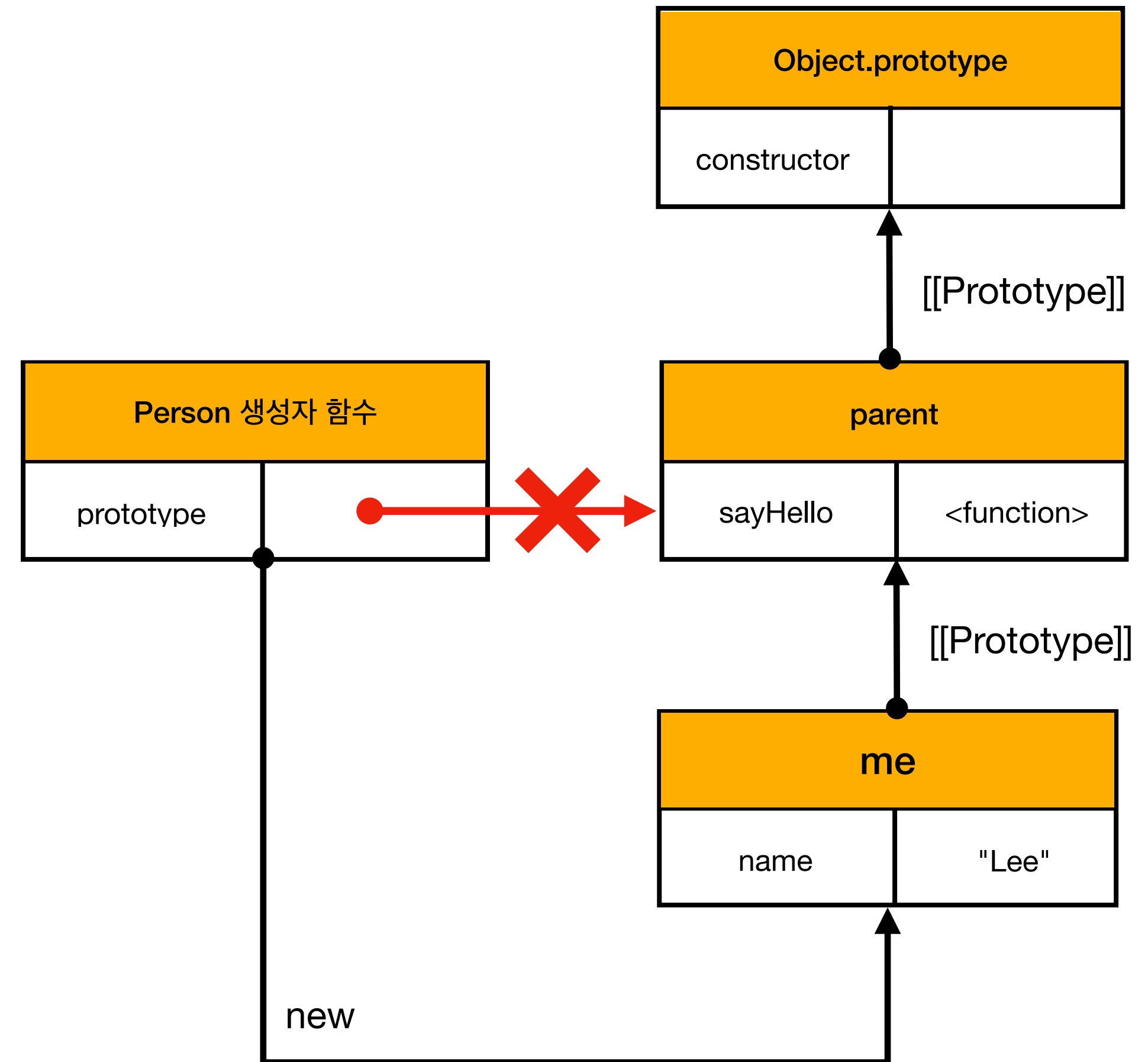
```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
const parent = {  
  sayHello() {  
    console.log(`Hi! My name is ${this.name}`);  
  },  
};  
  
Object.setPrototypeOf(me, parent);  
  
me.sayHello();
```

# 프로토타입의 교체

## 인스턴스에 의한 프로토타입의 교체



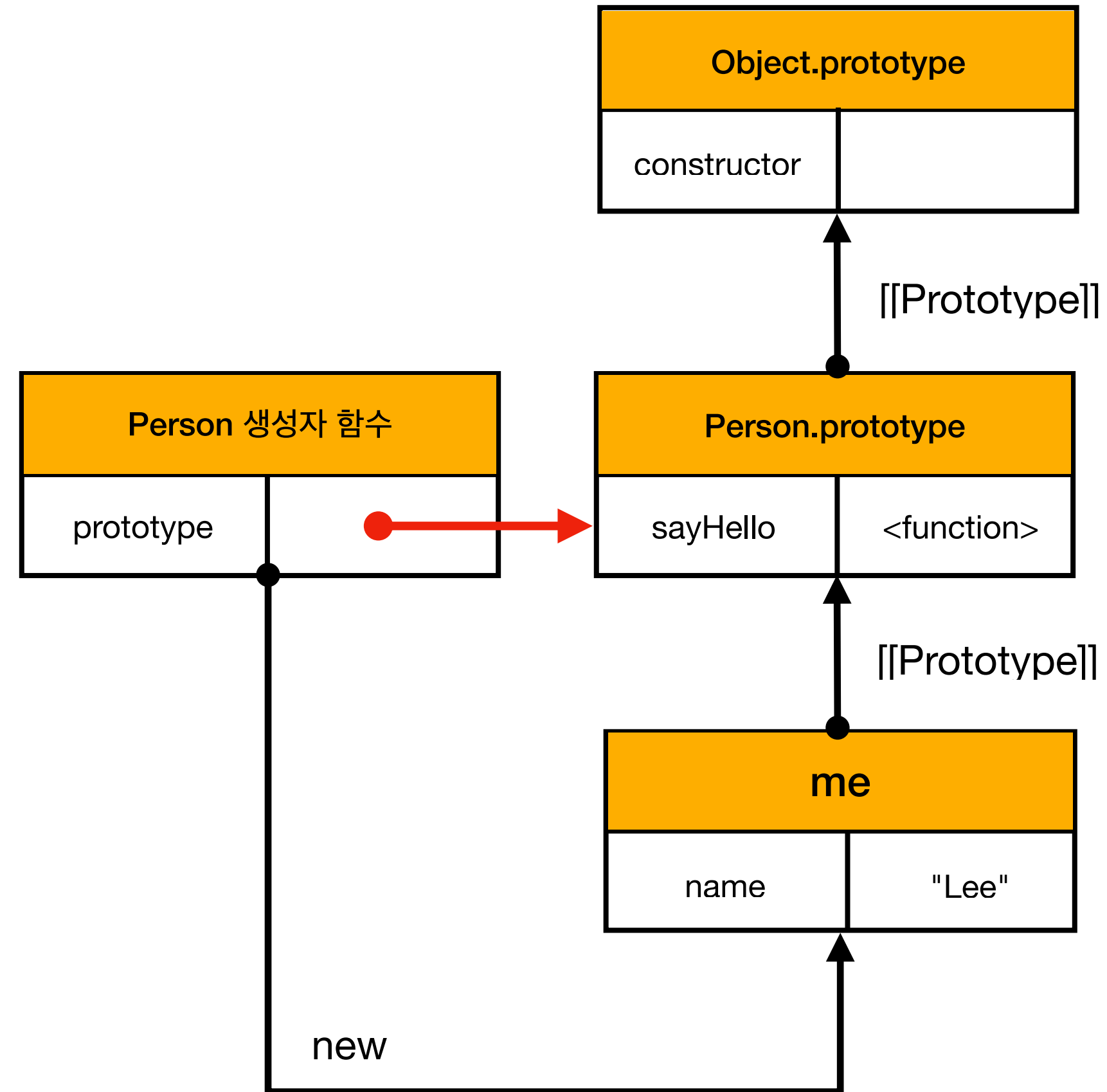
```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
const parent = {  
  sayHello() {  
    console.log(`Hi! My name is ${this.name}`);  
  },  
};  
  
Object.setPrototypeOf(me, parent);  
  
me.sayHello();
```



# 프로토타입의 교체 차이점

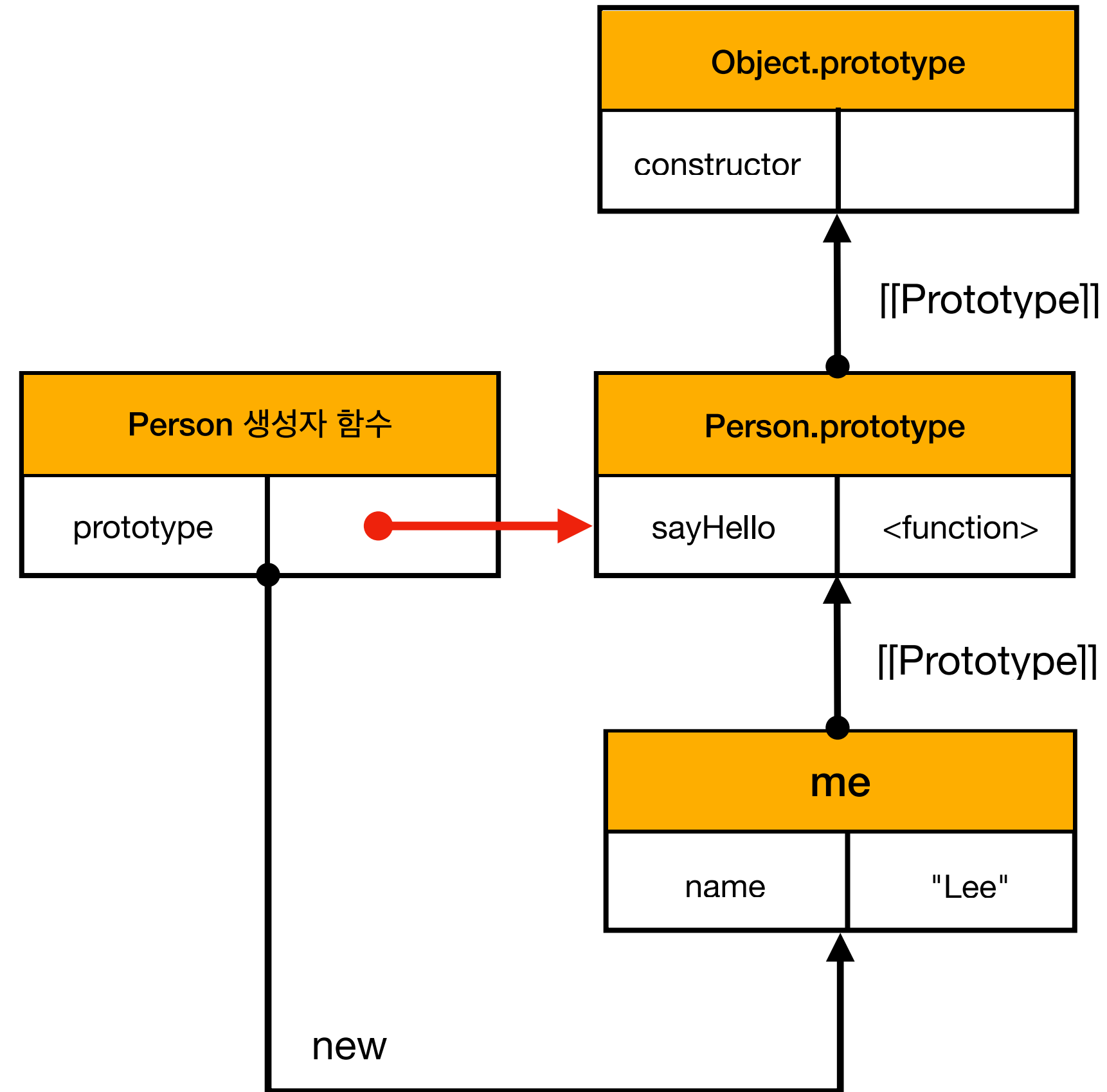
# 프로토타입의 교체

## 차이점

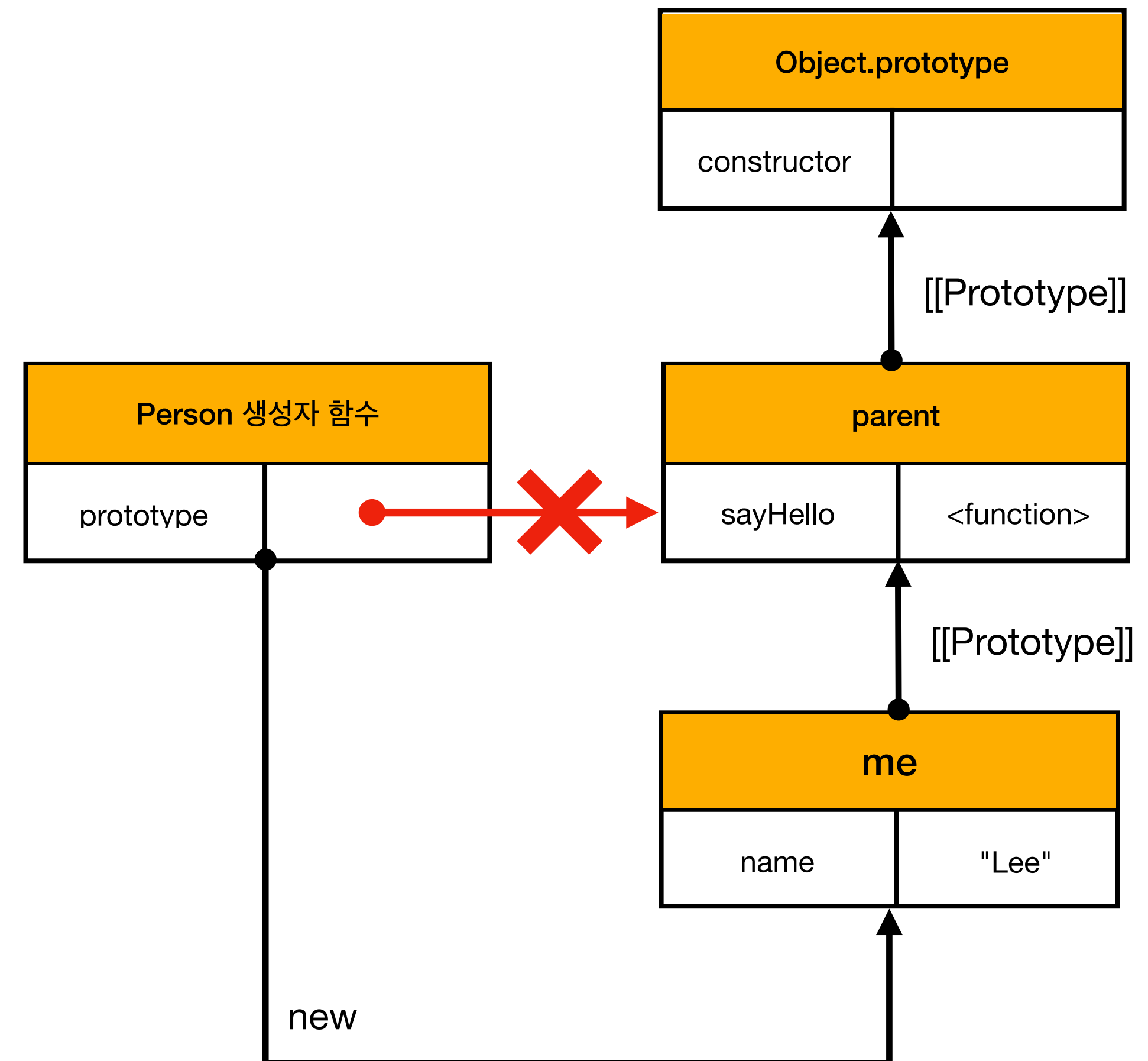


생성자 함수에 의한 프로토타입의 교체

# 프로토타입의 교체 차이점



생성자 함수에 의한 프로토타입의 교체



인스턴스에 의한 프로토타입의 교체

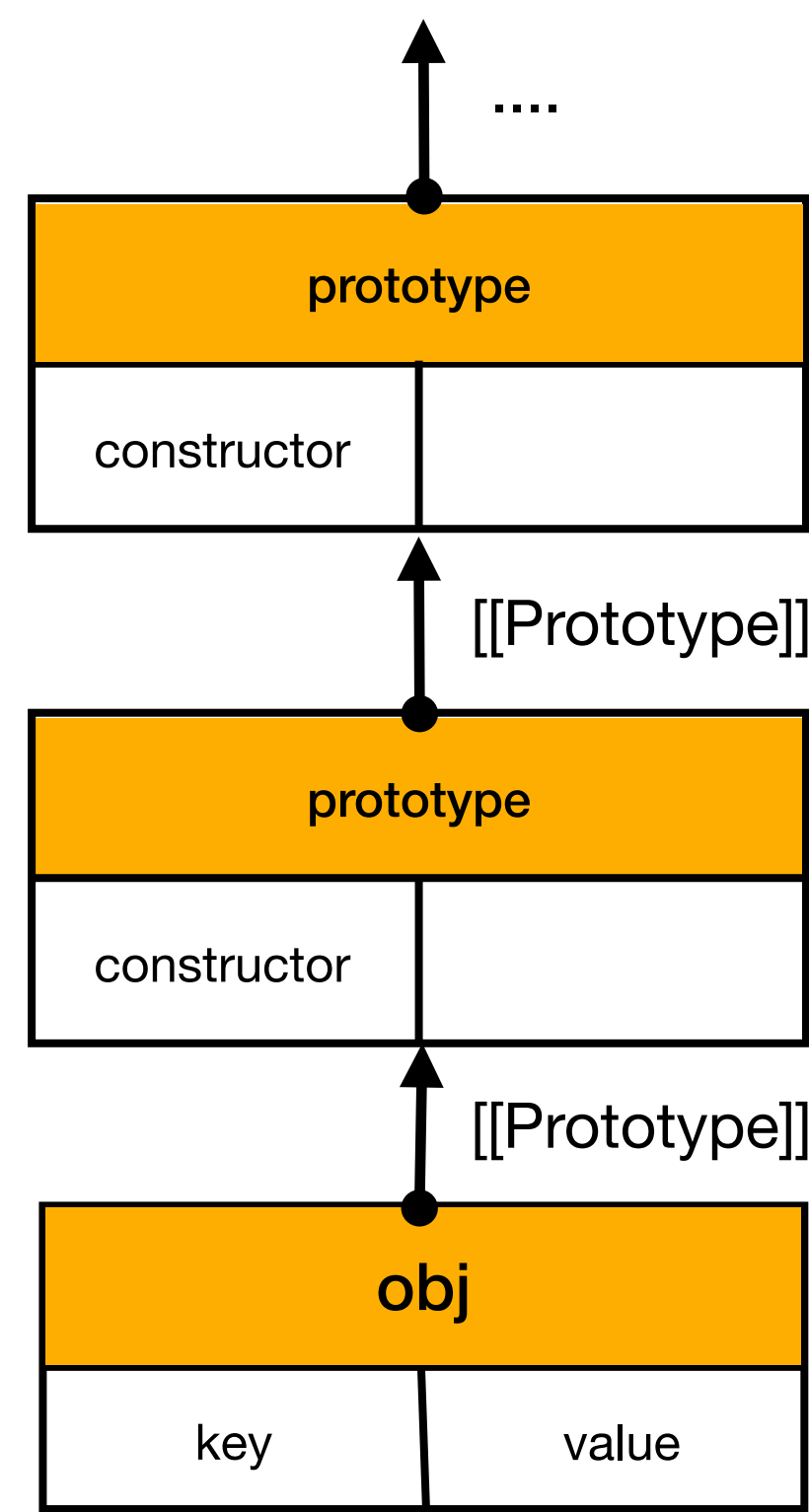
# instanceof 연산자

객체 instanceof 생성자 함수



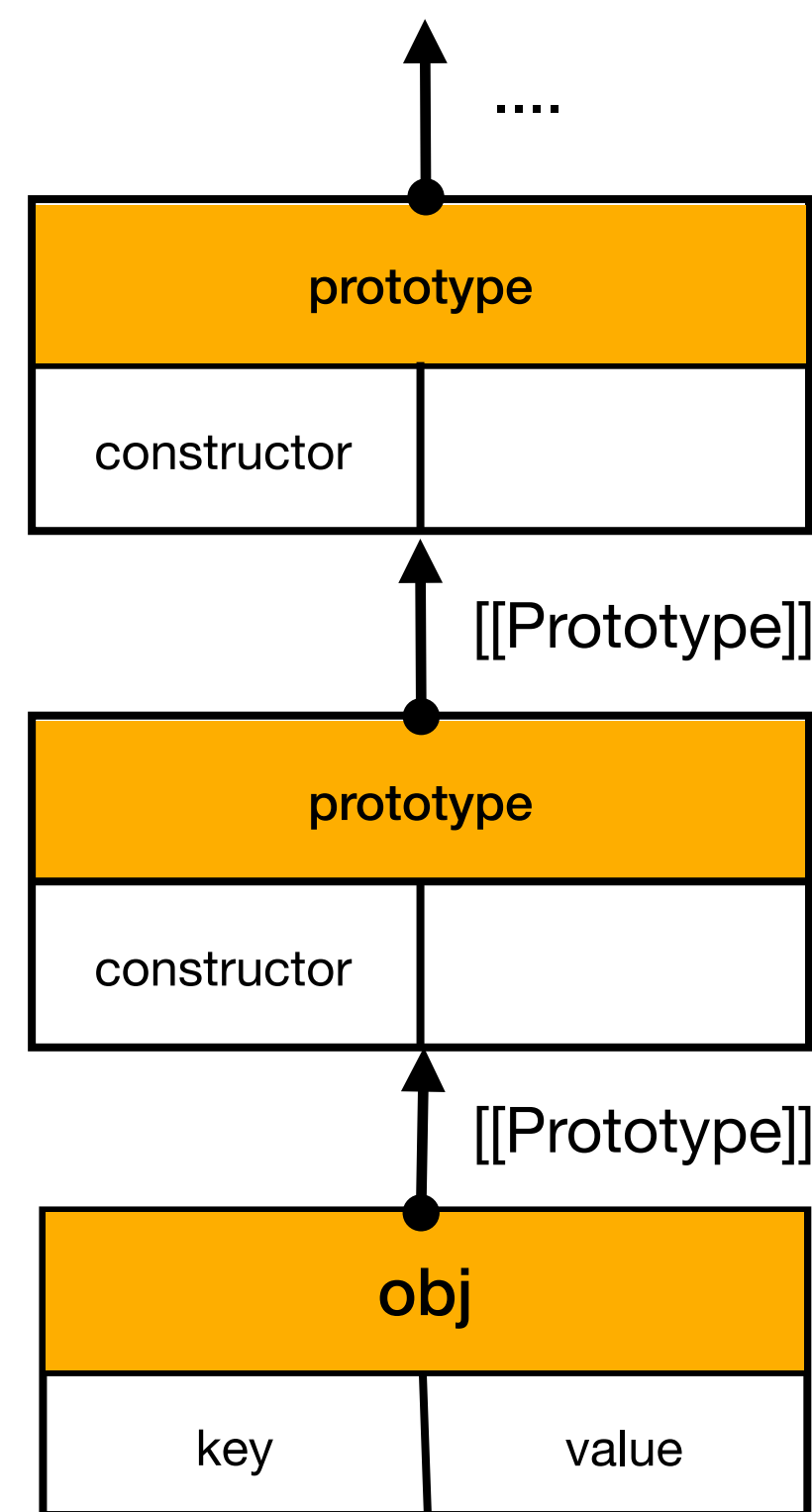
# instanceof 연산자

객체 instanceof 생성자 함수



# instanceof 연산자

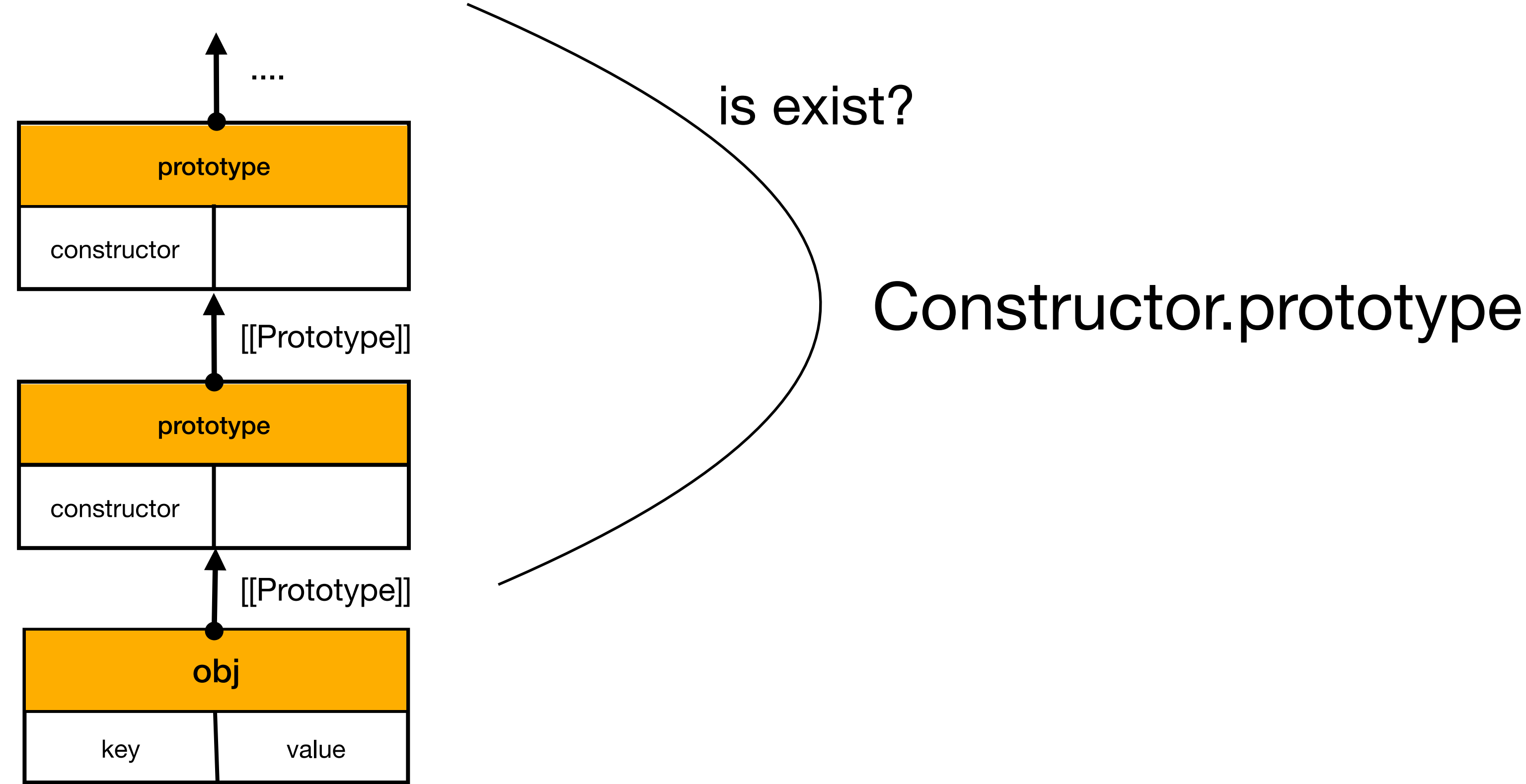
객체 instanceof 생성자 함수



Constructor.prototype

# instanceof 연산자

객체 instanceof 생성자 함수



# instanceof 연산자

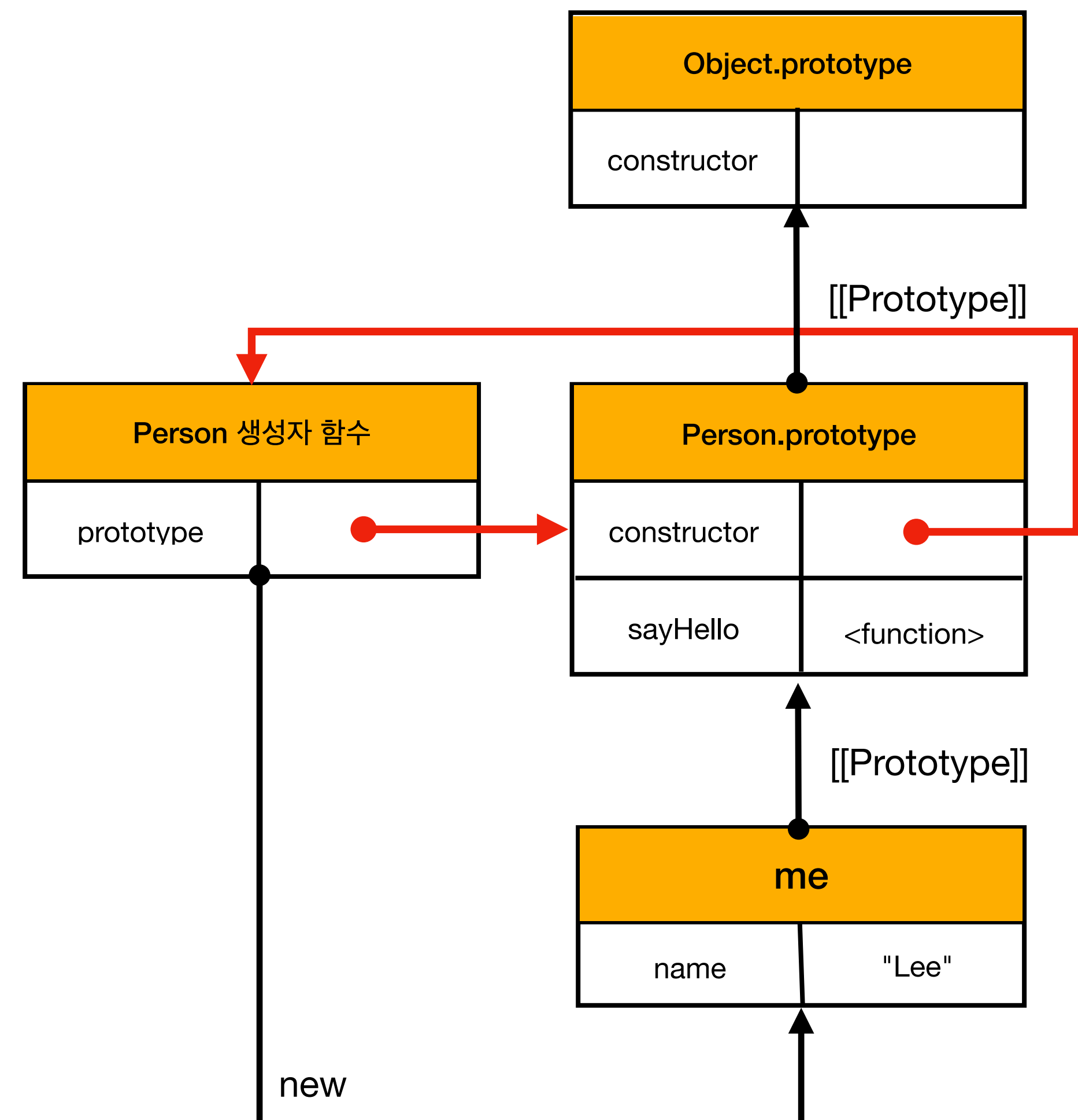
예시 1.



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

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

```
console.log(me instanceof Person); // true  
console.log(me instanceof Object); // true
```



# instanceof 연산자

## 예시 2.

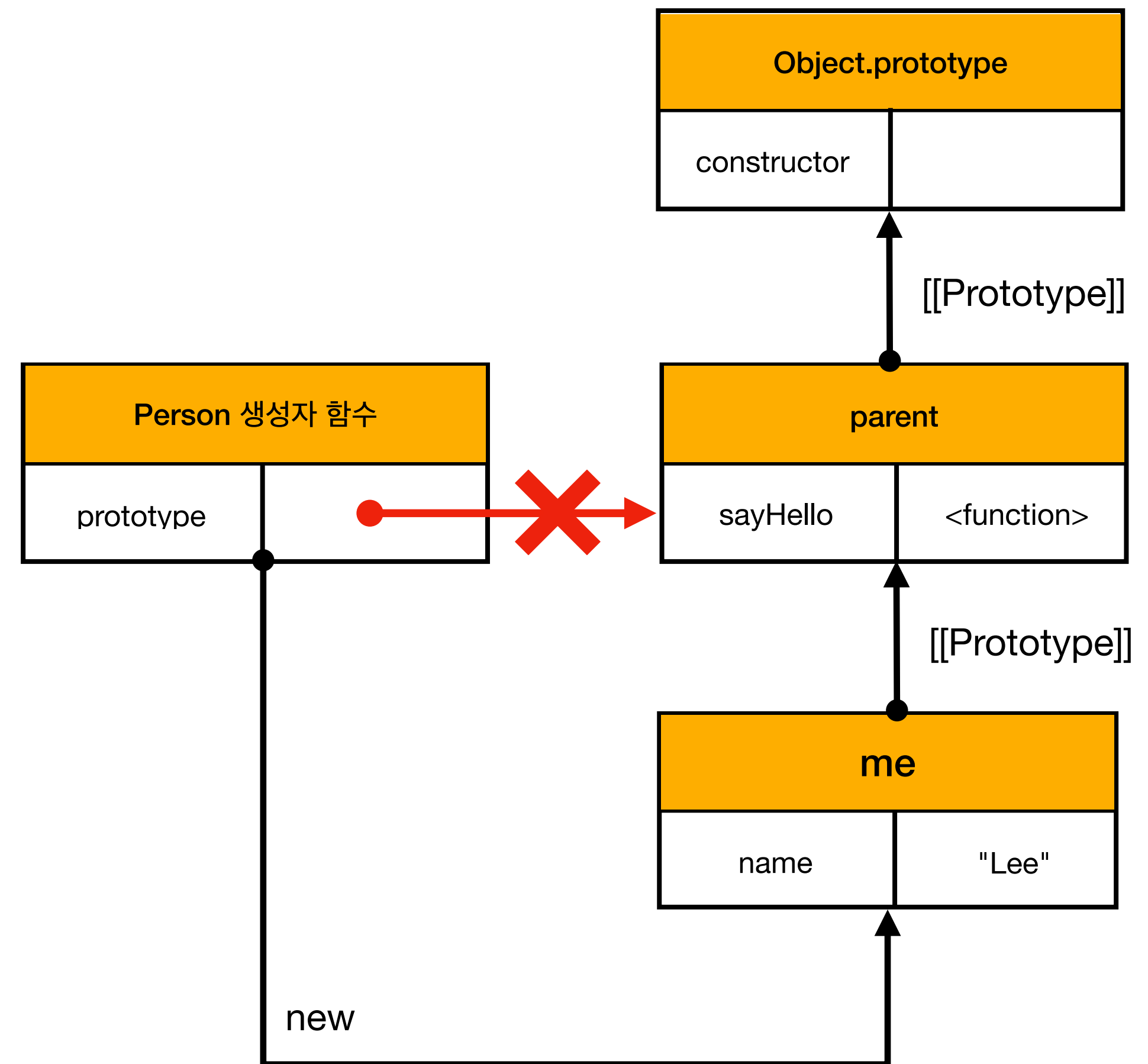


```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
const parent = {};  
  
Object.setPrototypeOf(me, parent);  
  
console.log(Person.prototype === parent); // false  
  
console.log(me instanceof Person); // false  
  
console.log(me instanceof Object); // true
```

# instanceof 연산자

## 예시 2.

```
function Person(name) {  
  this.name = name;  
}  
  
const me = new Person("Lee");  
  
const parent = {};  
  
Object.setPrototypeOf(me, parent);  
  
console.log(Person.prototype === parent); // false  
console.log(me instanceof Person); // false  
console.log(me instanceof Object); // true
```



# 직접 상속

## Object.create에 의한 직접 상속

Object.create는  
명시적으로 프로토타입을 지정하여 새로운 객체를 생성한다.

# 직접 상속

## Object.create에 의한 직접 상속



```
const obj = Object.create(null);

console.log(typeof obj); // "object"

console.log(Object.getPrototypeOf(obj) === null); // true

// TypeError: obj.toString is not a function
console.log(obj.toString());
```



# 직접 상속

## Object.create에 의한 직접 상속



```
const obj = Object.create(Object.prototype);  
  
console.log(Object.getPrototypeOf(obj) === Object.prototype); // true  
  
console.log(obj.toString()); // "[object Object]"
```

# 직접 상속

## Object.create에 의한 직접 상속



```
const obj = Object.create(Object.prototype);  
  
console.log(Object.getPrototypeOf(obj) === Object.prototype); // true  
  
console.log(obj.toString()); // "[object Object]"
```

const obj = {}; 와 동일하다.

# 직접 상속

## Object.create에 의한 직접 상속



```
const obj = Object.create(Object.prototype, {  
  x: { value: 1, writable: true, enumerable: true, configurable: true },  
});
```

```
console.log(obj.x); // 1
```

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

# 직접 상속

## Object.create에 의한 직접 상속



```
const obj = Object.create(Object.prototype, {  
  x: { value: 1, writable: true, enumerable: true, configurable: true },  
});  
  
console.log(obj.x); // 1  
  
console.log(Object.getPrototypeOf(obj) === Object.prototype); // true
```

const obj = { x: 1 }; 와 동일하다.

# 직접 상속

## Object.create에 의한 직접 상속



```
const myProto = { x: 10 };
```

```
const obj = Object.create(myProto);
```

```
console.log(obj.x); // 10
```

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

# 직접 상속

## Object.create에 의한 직접 상속



```
function Person(name) {  
  this.name = name;  
}  
  
const obj = Object.create(Person.prototype);  
  
obj.name = "Lee";  
  
console.log(obj.name); // Lee  
  
console.log(Object.getPrototypeOf(obj) === Person.prototype); // true
```

# 직접 상속

## Object.create에 의한 직접 상속



```
function Person(name) {  
  this.name = name;  
}  
  
const obj = Object.create(Person.prototype);  
  
obj.name = "Lee";  
  
console.log(obj.name); // Lee  
  
console.log(Object.getPrototypeOf(obj) === Person.prototype); // true
```

const obj = new Person("Lee"); 와 동일하다.

# 직접 상속

객체 리터럴 내부에서 `__proto__`에 의한 직접 상속



```
const myProto = { x: 10 };
```

```
const obj = {  
  y: 20,  
  __proto__: myProto  
}
```

```
console.log(obj.x, obj.y); // 10 20
```

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



# 정적 프로퍼티/메서드

정적 프로퍼티/메서드란?

생성자 함수로 인스턴스를 생성하지 않아도  
참조/호출할 수 있는 프로퍼티/메서드

# 정적 프로퍼티/메서드

## 예시

```
function Person(name) {  
  this.name = name;  
}  
  
// 프로토타입 메서드  
Person.prototype.sayHello = function () {  
  console.log(`Hi! My name is ${this.name}`);  
};  
  
// 정적 프로퍼티  
Person.staticProp = "static prop";  
  
// 정적 메서드  
Person.staticMethod = function () {  
  console.log("Static Method");  
};  
  
const me = new Person("Lee");  
  
Person.staticMethod(); // Static Method  
  
me.staticMethod(); // TypeError: me.staticMethod is not a function
```

# 정적 프로퍼티/메서드 예시

```
function Person(name) {  
  this.name = name;  
}  
  
// 프로토타입 메서드  
Person.prototype.sayHello = function () {  
  console.log(`Hi! My name is ${this.name}`);  
};  
  
// 정적 프로퍼티  
Person.staticProp = "static prop";  
  
// 정적 메서드  
Person.staticMethod = function () {  
  console.log("Static Method");  
};  
  
const me = new Person("Lee");  
  
Person.staticMethod(); // Static Method  
  
me.staticMethod(); // TypeError: me.staticMethod is not a function
```

# 정적 프로퍼티/메서드 예시

```
function Person(name) {  
  this.name = name;  
}  
  
// 프로토타입 메서드  
Person.prototype.sayHello = function () {  
  console.log(`Hi! My name is ${this.name}`);  
};  
  
// 정적 프로퍼티  
Person.staticProp = "static prop";  
  
// 정적 메서드  
Person.staticMethod = function () {  
  console.log("Static Method");  
};  
  
const me = new Person("Lee");  
  
Person.staticMethod(); // Static Method  
  
me.staticMethod(); // TypeError: me.staticMethod is not a function
```

# 정적 프로퍼티/메서드

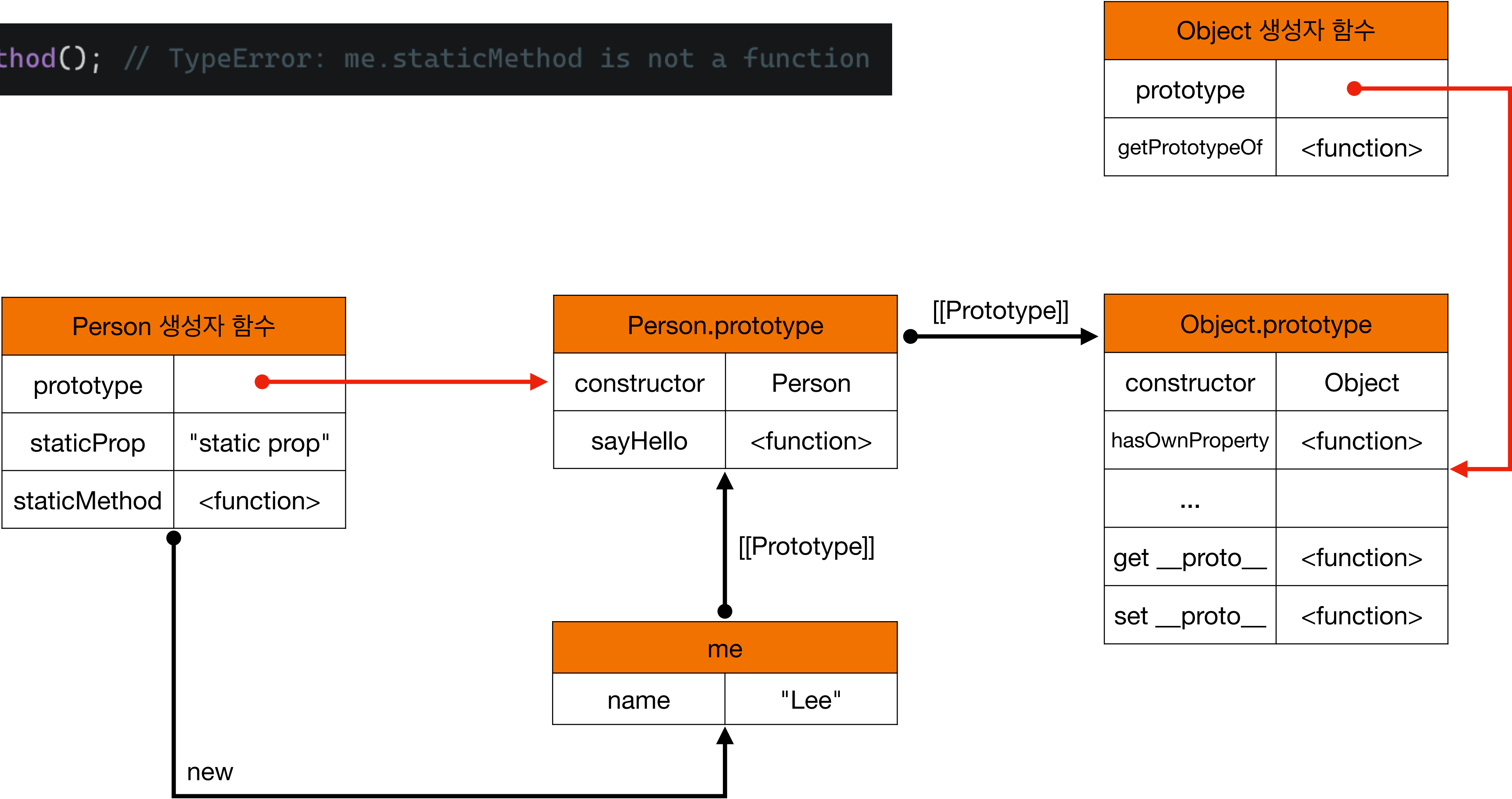
## 예시

```
function Person(name) {  
  this.name = name;  
}  
  
// 프로토타입 메서드  
Person.prototype.sayHello = function () {  
  console.log(`Hi! My name is ${this.name}`);  
};  
  
// 정적 프로퍼티  
Person.staticProp = "static prop";  
  
// 정적 메서드  
Person.staticMethod = function () {  
  console.log("Static Method");  
};  
  
const me = new Person("Lee");  
  
Person.staticMethod(); // Static Method  
  
me.staticMethod(); // TypeError: me.staticMethod is not a function
```

# 정적 프로퍼티/메서드

인스턴스에서는 정적 프로퍼티/메서드를 사용할 수 없는 이유

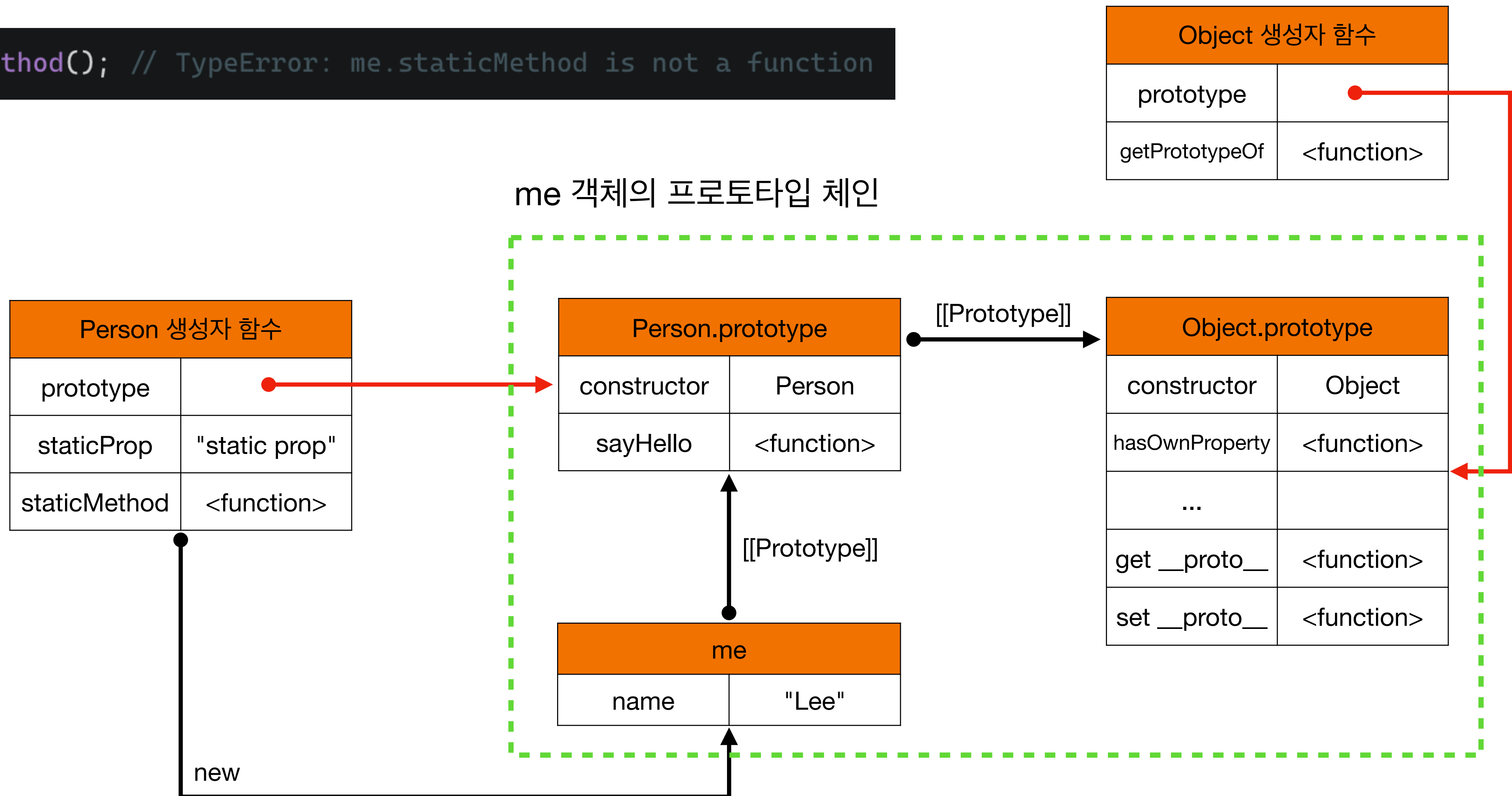
```
me.staticMethod(); // TypeError: me.staticMethod is not a function
```



# 정적 프로퍼티/메서드

인스턴스에서는 정적 프로퍼티/메서드를 사용할 수 없는 이유

```
me.staticMethod(); // TypeError: me.staticMethod is not a function
```



# 프로퍼티 존재 확인

in 연산자

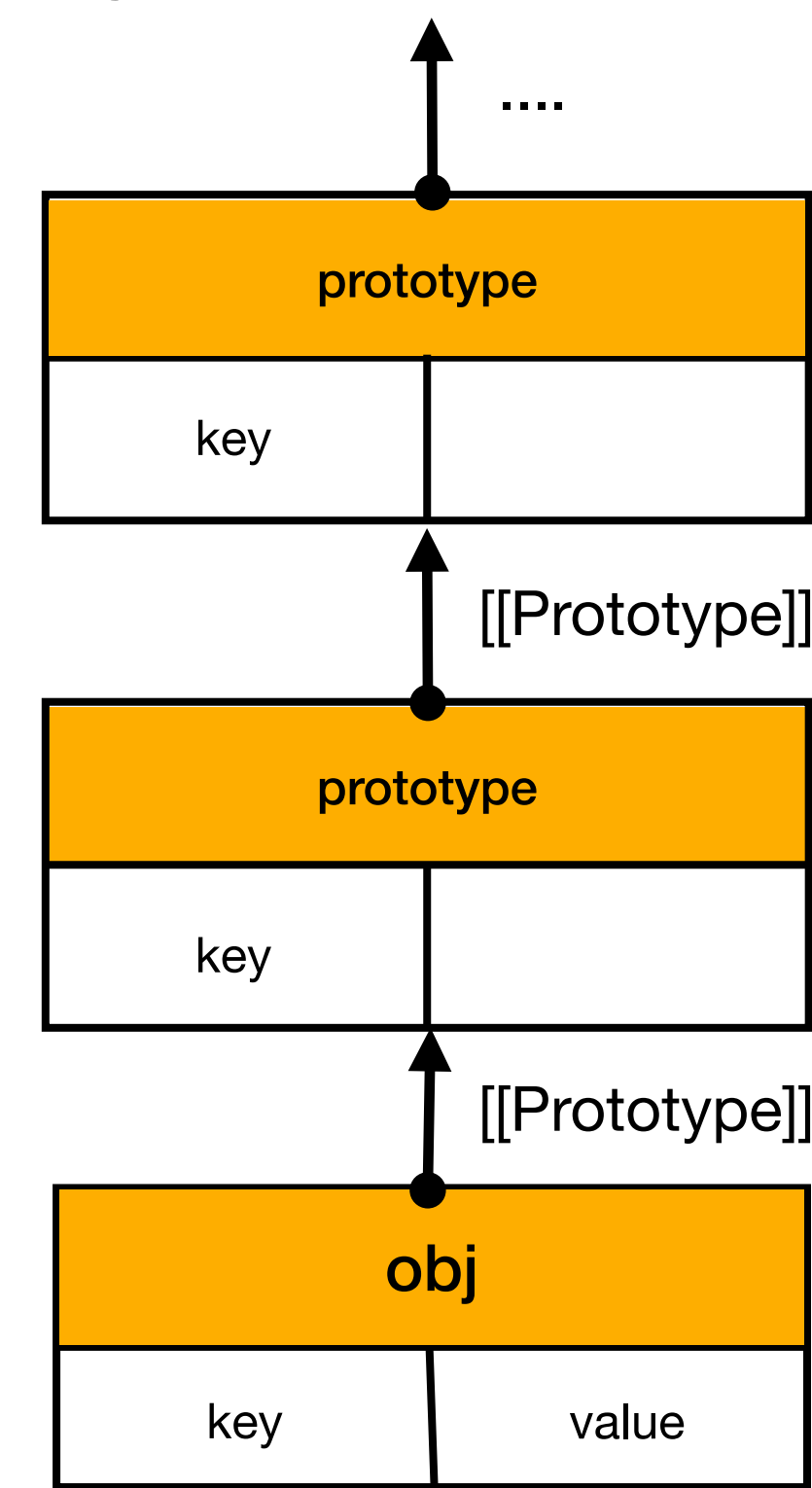
key **in** object



# 프로퍼티 존재 확인

in 연산자

key in object

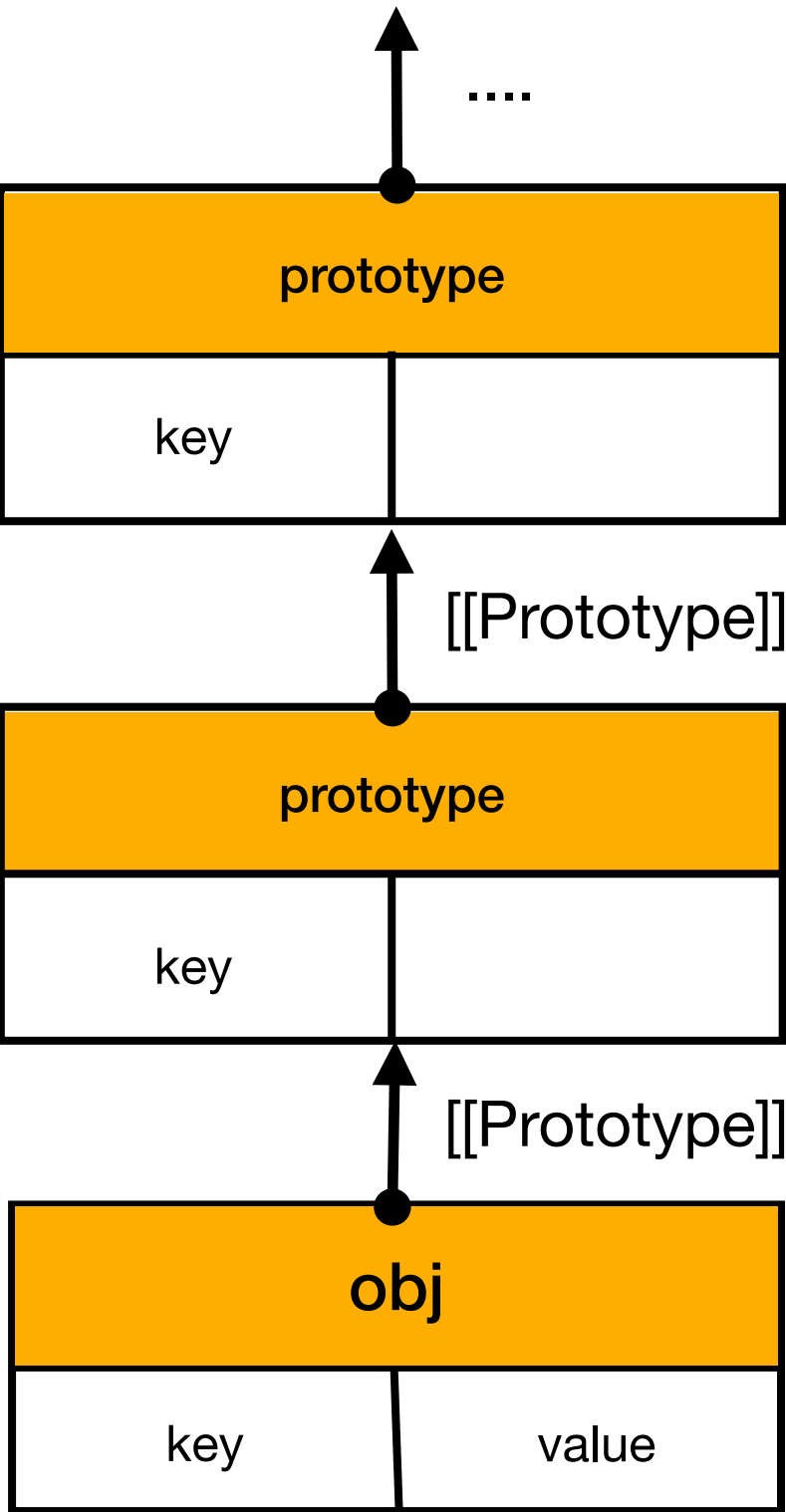


# 프로퍼티 존재 확인

in 연산자

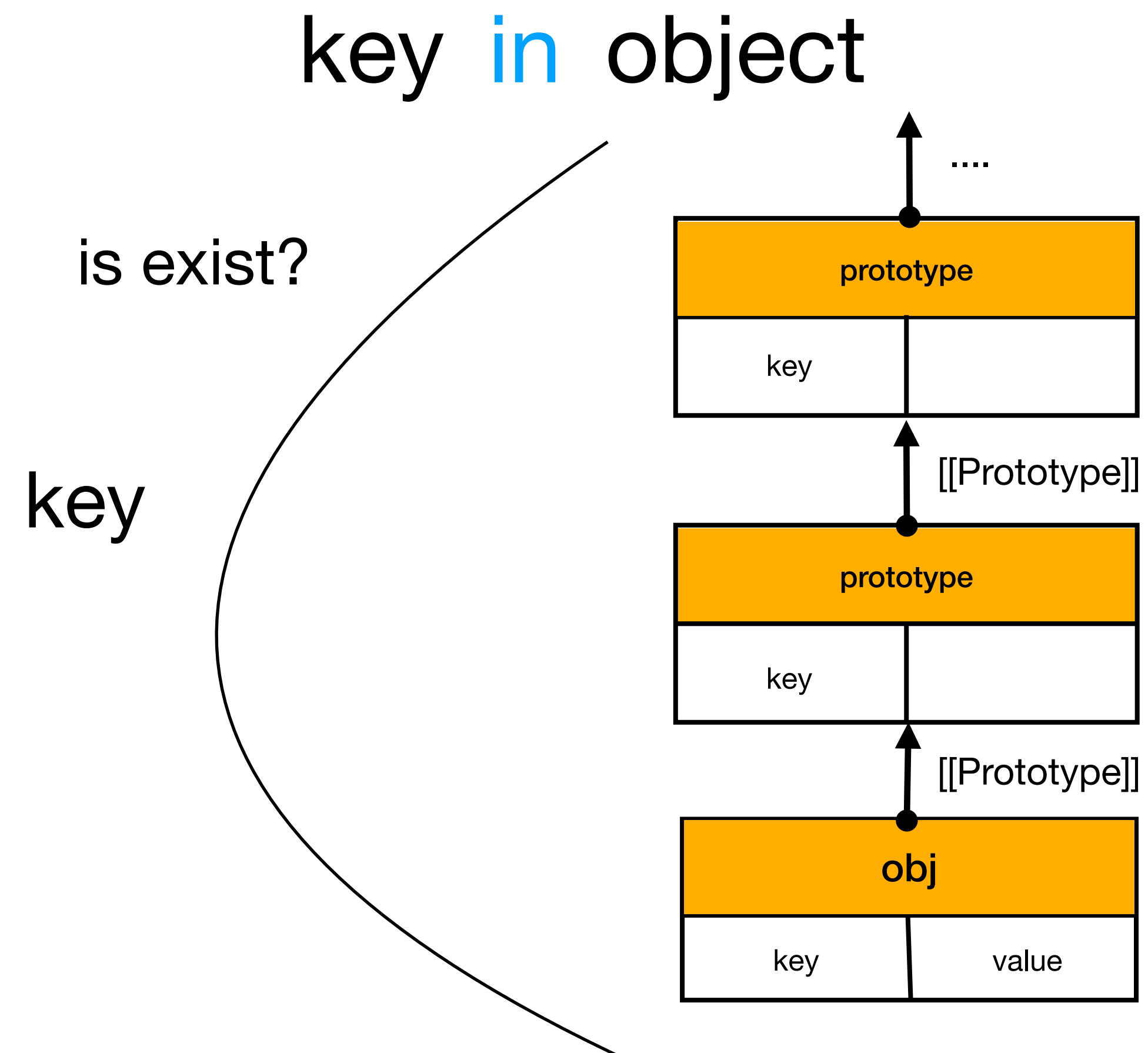
key in object

key



# 프로퍼티 존재 확인

in 연산자



# 프로퍼티 존재 확인

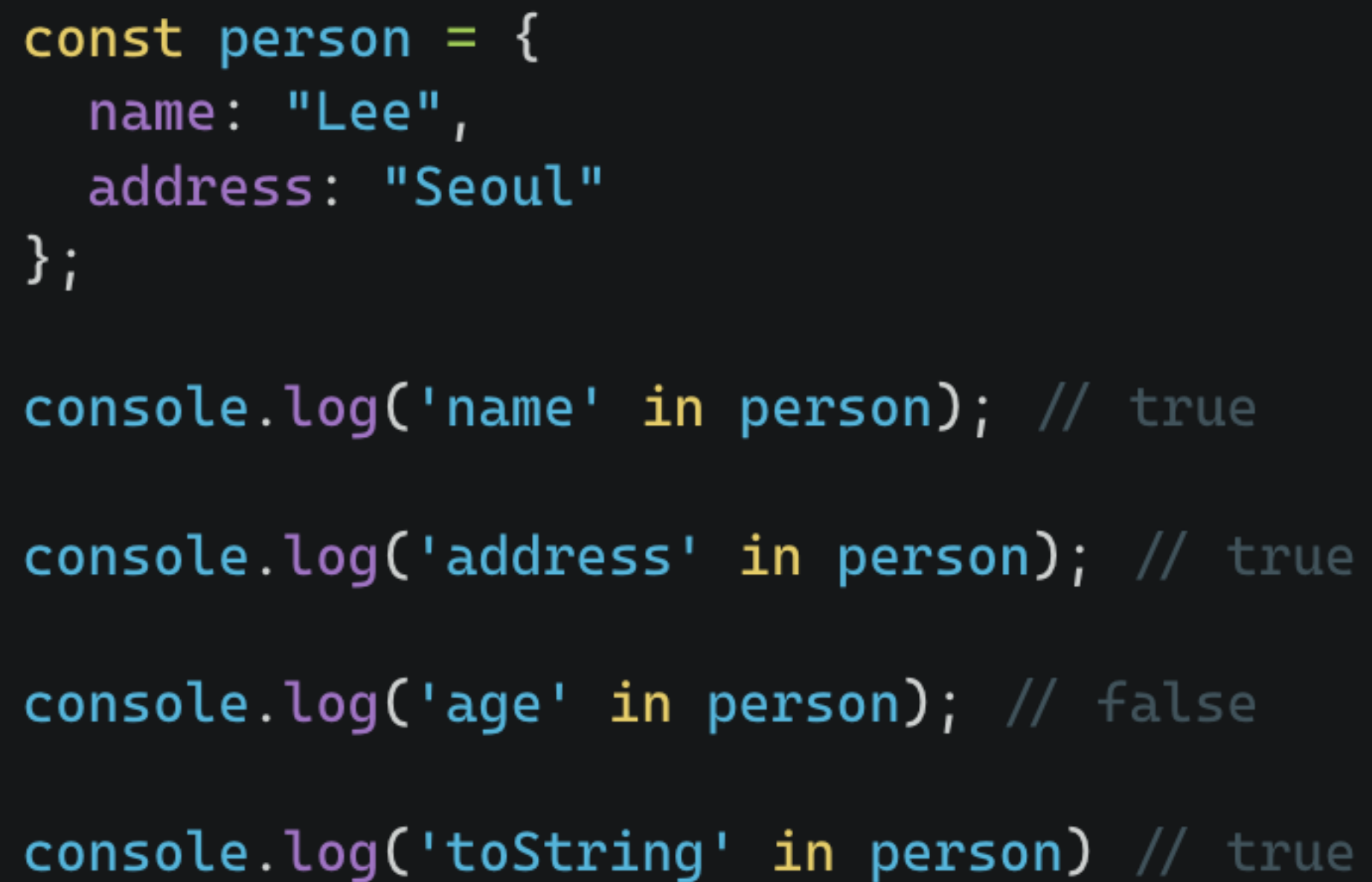
## in 연산자 - 예시



```
const person = {  
  name: "Lee",  
  address: "Seoul"  
};  
  
console.log('name' in person); // true  
  
console.log('address' in person); // true  
  
console.log('age' in person); // false  
  
console.log('toString' in person) // true
```

# 프로퍼티 존재 확인

## in 연산자 - 예시



```
const person = {  
  name: "Lee",  
  address: "Seoul"  
};  
  
console.log('name' in person); // true  
  
console.log('address' in person); // true  
  
console.log('age' in person); // false  
  
console.log('toString' in person) // true
```

Reflect.has 메서드도 동일하게 동작한다.  
Reflect.has(person, "name") === true

# 프로퍼티 존재 확인

## Object.prototype.hasOwnProperty 메서드



```
const person = {  
  name: "Lee",  
  address: "Seoul"  
};
```

```
console.log(person.hasOwnProperty("name")); // true
```

```
console.log(person.hasOwnProperty("address")); // true
```

```
console.log(person.hasOwnProperty("toString")); // false
```

# 프로퍼티 존재 확인

## Object.prototype.hasOwnProperty 메서드



```
const person = {  
  name: "Lee",  
  address: "Seoul"  
};  
  
console.log(person.hasOwnProperty("name")); // true  
  
console.log(person.hasOwnProperty("address")); // true  
  
console.log(person.hasOwnProperty("toString")); // false
```

객체 고유의 프로퍼티 키인 경우에만 true를 반환한다.

# 프로퍼티 열거

for ... in 문



```
const person = {  
  name: "Lee",  
  address: "Seoul"  
}  
  
for (const key in person) {  
  console.log(key + ": " + person[key]);  
}  
  
// name: Lee  
// address: Seoul
```



# 프로퍼티 열거

## for ... in 문 - 주의사항

for ... in 문은 in 연산자처럼  
상속받은 프로토타입의 프로퍼티까지 열거한다.

# 프로퍼티 열거

## for ... in 문 - 주의사항

for ... in 문은 in 연산자처럼  
상속받은 프로토타입의 프로퍼티까지 열거한다.

```
const person = {  
  name: "Lee",  
  address: "Seoul",  
  __proto__: { age: 20 }  
}  
  
for (const key in person) {  
  console.log(key + ": " + person[key]);  
}  
  
// name: Lee  
// address: Seoul  
// age: 20
```

# 프로퍼티 열거 의문점

상속받은 프로토타입의 프로퍼티를 열거하는데  
왜 toString, hasOwnProperty같은 프로퍼티는 안나올까?

# 프로퍼티 열거 의문점

상속받은 프로토타입의 프로퍼티를 열거하는데  
왜 toString, hasOwnProperty같은 프로퍼티는 안나올까?



프로토타입 체인 상에 존재하는 모든 프로토타입의 프로퍼티 중에서  
프로퍼티 어트리뷰트 `[[Enumerable]]`의 값이 `true`인 프로퍼티만 순회하며 열거한다.



```
console.log(Object.getOwnPropertyDescriptor(Object.prototype, "toString"));  
// Object { value: toString(), writable: true, enumerable: false, configurable: true }  
  
console.log(Object.getOwnPropertyDescriptor(Object.prototype, "hasOwnProperty"));  
// Object { value: hasOwnProperty(), writable: true, enumerable: false, configurable: true }
```

# 프로퍼티 열거

Symbol인 키도 열거하지 않는다.



```
const sym = Symbol();

const obj = {
  a: 1,
  [sym]: 10
};

for (const key in obj) {
  console.log(key + ": " + obj[key]);
}

// a: 1
```

# 프로퍼티 열거

객체 자신의 프로퍼티만 열거하기



```
const person = {  
  name: "Lee",  
  address: "Seoul",  
  __proto__: { age: 20 }  
};  
  
for (const key in person) {  
  if (!person.hasOwnProperty(key)) continue;  
  console.log(key + ": " + person[key]);  
}  
  
// name: Lee  
// address: Seoul
```

# 프로퍼티 열거

## 객체 자신의 프로퍼티만 열거하기

```
const person = {  
  name: "Lee",  
  address: "Seoul",  
  __proto__: { age: 20 }  
};  
  
for (const key in person) {  
  if (!person.hasOwnProperty(key)) continue;  
  console.log(key + ": " + person[key]);  
}  
  
// name: Lee  
// address: Seoul
```

상속받은 프로토타입의 프로퍼티는 제외된다

# 프로퍼티 열거

## Object.keys/values/entries 메서드

for ... in문은 상속받은 프로퍼티도 열거한다.



# 프로퍼티 열거

## Object.keys/values/entries 메서드

for ... in문은 상속받은 프로퍼티도 열거한다.



hasOwnProperty 메서드를 사용하여  
객체 자신의 프로퍼티인지 확인하는 추가 작업이 필요하다

# 프로퍼티 열거

## Object.keys/values/entries 메서드

for ... in문은 상속받은 프로퍼티도 열거한다.



hasOwnProperty 메서드를 사용하여  
객체 자신의 프로퍼티인지 확인하는 추가 작업이 필요하다



객체 자신의 고유 프로퍼티만 열거하기 위해서  
Object.keys/values/entries 메서드를 권장한다.

# 프로퍼티 열거

## Object.keys/values/entries 메서드



```
const person = {  
  name: "Lee",  
  address: "Seoul",  
  __proto__: { age: 20 }  
};
```

```
console.log(Object.keys(person)); // ['name', 'address']
```

```
console.log(Object.values(person)); // ['Lee', 'Seoul']
```

```
console.log(Object.entries(person)); // [['name', 'Lee'], ['address', 'Seoul']]
```