



Chap19 프로토타입

들어가기 앞서...

- [19-1 객체지향 프로그래밍 \(OOP\)](#)
- [19-2 상속과 프로토타입](#)
- [19-3 프로토타입 객체](#)
- [19-4 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입](#)
- [19-5 프로토타입의 생성시점](#)
- [19-6 객체 생성 방식과 프로토타입의 결정](#)
- [19-7 프로토타입 체인](#)
- [19-8 오버라이딩과 프로퍼티 새도잉](#)
- [19-9 프로토타입의 교체](#)
- [19-10 instanceof 연산자](#)
- [19-11 직접 상속](#)
- [19-12 정적 프로퍼티 / 메서드](#)
- [19-13 프로퍼티 존재 확인](#)
- [19-14 프로퍼티 열거](#)

들어가기 앞서...

- 자바스크립트는 명령형, 함수형, **프로토타입 기반 객체지향 프로그래밍**을 지원하는 멀티 패러다임 프로그래밍 언어이다.
- 자바스크립트는 **객체 기반의 프로그래밍 언어**이며 자바스크립트를 이루고 있는 거의 “모든 것”이 **객체**이다.

19-1 객체지향 프로그래밍 (OOP)

객체지향 프로그래밍

- **객체의 집합**으로 프로그램을 표현하려는 프로그래밍 패러다임
- 4가지 특징
 - **추상화** : 핵심적이거나 공통되는 속성 및 기능을 간추리는 것 (객체, 클래스로 정리하는 것 모두 추상화이다.)

- **상속** : 기본클래스(base class)의 특징을 파생클래스(derived class)가 상속 받는 것. ex) **프로토타입, ES6이후 클래스**
- **다형성** : 하나의 메서드가 다른 동작을 하게끔 구현할 수 있다.
- **캡슐화** : 관련된 속성과 동작을 객체 내부에 묶는 것을 의미한다. 이를 통해 객체의 내부 구현을 외부로부터 숨기고, 외부에서는 객체의 인터페이스를 통해 객체와 상호작용할 수 있게 됩니다.

▼ 자바스크립트에서 캡슐화 하기

Java 같은 경우에는 public, default, protected, private 총 4가지 종류의 **접근 제어자**를 통해 제어가 가능하다.

하지만 **JavaScript에는 이런 접근 제어자가 없다.** 어떻게 해야할까?

- 클로저 사용
- Symbol을 사용한 프라이빗 필드

(이러한 방법이 있다고 한다. 이 내용은 뒤 챕터에서 자세히 배우고 알아볼 예정)

▼ 객체란?

- 상태 데이터와 동작을 하나의 논리적인 단위로 묶은 복합적인 자료구조
- 상태 데이터 = 프로퍼티 = 속성, 동작 = 메서드

19-2 상속과 프로토타입

상속

- 어떤 객체의 프로퍼티 또는 메서드를 다른 객체가 상속받아 그대로 사용할 수 있는 것
- 자바스크립트에서는 **프로토타입을 기반으로 상속을 구현**한다

▼ 프로토타입

자바스크립트 객체가 다른 객체로부터 메서드와 속성을 상속받기 위해 사용하는 메커니즘(체제).

- 예시

▼ 인스턴스

특정 클래스나 생성자 함수로부터 생성된 객체

```

// 생성자 함수
function Circle(radius) {
  this.radius = radius;
}

// Circle 생성자 함수가 생성한 모든 인스턴스가 getArea 메서드를
// 공유해서 사용할 수 있도록 프로토타입에 추가한다.
// 프로토타입은 Circle 생성자 함수의 prototype 프로퍼티에 바인딩!
Circle.prototype.getArea = function () {
  return Math.PI * this.radius ** 2;
};

// 인스턴스 생성
const circle1 = new Circle(1);
const circle2 = new Circle(2);

// Circle 생성자 함수가 생성한 모든 인스턴스는 부모 객체의 역할을
// 프로토타입 Circle.prototype으로부터 getArea 메서드를 상속받는다.
// 즉, Circle 생성자 함수가 생성하는 모든 인스턴스는 하나의 getArea 메서드를 공유한다.
console.log(circle1.getArea === circle2.getArea); // true

console.log(circle1.getArea()); // 3.141592653589793
console.log(circle2.getArea()); // 12.566370614359172

```

Circle 생성자 함수가 생성한 모든 인스턴스는 자신의 프로토타입, 즉 상위(부모)객체의 역할을 하는 Circle.prototype의 모든 프로퍼티와 메서드를 상속받는다.

따라서 이부분이 true로 나오는 것이다.

```

console.log(circle1.getArea === circle2.getArea); // true

```

19-3 프로토타입 객체

1. 프로토타입 객체(=프로토타입)

- 다른 객체에 의해 상속될 수 있는 객체
- 상속을 구현하기 위한 메커니즘

- 모든 객체는 `[[Prototype]]`이라는 내부 슬롯을 갖는다. 그리고 이 `[[Prototype]]`에 저장되는 프로토타입은 객체 생성 방식에 따라 달라진다.
- 내부슬롯에 직접적으로 접근하는 것은 불가능하지만 `[[Prototype]]`만 `__proto__`라는 접근자 프로퍼티를 통해 **간접적으로 접근** 가능하다


▼ 내부슬롯, 내부 메서드란?

- 자바스크립트 엔진의 내부 동작을 설명하기 위해 ECMAScript에서 정의한 의사 프로퍼티(pseudo property)와 의사 메서드(pseudo method)이다.
- 즉 ECMAScript 사양에 등장하는 `[[...]]`로 감싼 이름들이 **내부 슬롯과 내부 메서드**이다.
- 조금 쉽게 이야기 하면 ECMAScript 문서에서 자바스크립트 내부 동작의 설명을 위해 정의해 놓은 가상 메서드라고 이해하면 될 듯 하다.
- 내부슬롯 중 하나가 프로토타입인것이다.

2. `__proto__` 접근자 프로퍼티

모든 객체는 `__proto__` 접근자 프로퍼티를 통해 자신의 프로토타입, 즉 `[[Prototype]]` 내부 슬롯에 **간접적으로 접근**할 수 있다.

```
> const person={name:'Lee'};
< undefined
> person
< {name: 'Lee'}
  name: "Lee"
  [[Prototype]]: Object
    ▶ 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__: (...)
```



`__proto__`에 대해서 더 자세히 알아보자.

- `__proto__`는 접근자 프로퍼티이다

▼ 접근자 프로퍼티

자체적인 값 `[[Value]]`을 갖지 않고, 접근자 함수 `[[Get]]`, `[[Set]]` 프로퍼티 어트리뷰트로 구성된 프로퍼티다.

`Object.prototype`의 접근자 프로퍼티인 `__proto__`는 getter/setter 함수라고 부르는 접근자 함수(`[[Get]]`, `[[Set]]` 프로퍼티 어트리뷰트에 할당된 함수)를 통해 `[[Prototype]]` 내부 슬롯의 값, 즉 프로토타입을 취득하거나 할당한다.

▼ 취득 과정

`__proto__`을 통해 프로토타입에 접근하면 내부적으로 `__proto__` 접근자 프로퍼티의 getter 함수인 `[[Get]]`이 호출된다. `__proto__` 접근자 프로퍼티를 통해 새로운 프로토타입을 할당하면 `__proto__` 접근자 프로퍼티의 setter 함수인 `[[Set]]`이 호출된다.

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

//getter 함수인 get __proto__가 호출되어 obj 객체의 프로토타입
obj.__proto__;

//setter 함수인 set __proto__가 호출되어 obj 객체의 프로토타입
obj.__proto__=parent;
console.log(obj.x); //1
```

- `__proto__` 접근자 프로퍼티는 상속을 통해 사용된다
 - `__proto__` 접근자 프로퍼티는 객체가 직접 소유하는 프로퍼티가 아니라 `Object.prototype`의 프로퍼티이다.
 - 따라서 상속을 통해 `Object.prototype.__proto__` 접근자 프로퍼티를 사용할 수 있다.

▼ 위 내용이 무슨 뜻인지 모르겠어서 찾아봄

JavaScript의 모든 객체는 `Object.prototype`을 상속받습니다. 따라서 모든 객체는 `Object.prototype`에 정의된 메서드와 프로퍼티를 사용할 수 있습니다.

`__proto__` 접근자 프로퍼티도 이러한 상속 구조를 통해 모든 객체에서 사용할 수 있습니다.

- `__proto__` 접근자 프로퍼티를 통해 프로토타입에 접근하는 이유

상호 참조에 의해 프로토 타입 체인이 생성되는 것을 방지하기 위해서다.(상호 참조(순환 참조)를 시도하면 TypeError를 발생시킨다.)

왜 에러가 나는가?

프로토타입 체인은 **단방향 링크드 리스트**로 구현되어야 한다.

따라서

아무런 제지없이 프로토타입을 교체할수 없도록 간접적으로 접근자프로퍼티를 이용하여 접근하는것이다.

- `__proto__` 접근자 사용 권장X

모든 객체가 `__proto__` 접근자 프로퍼티를 사용하는 것은 아니기 때문이다. 직접 상속을 통해 Object.prototype을 상속받지 않는 객체를 생성할 수 있어 `__proto__` 접근자 프로퍼티를 사용할 수 없는 경우가 있다.

3. 함수 객체의 프로토타입

함수 객체만이 소유하는 prototype프로퍼티는 생성자 함수가 생성할 인스턴스의 프로토타입을 가리킨다.

따라서 생성자 함수로 호출할 수 없는 함수, 즉 non-constructor인 화살표함수, ES6 메서드 축약 표현으로 정의한 메서드는 prototype프로퍼티를 소유하지 않으며 프로토타입도 생성하지 않는다.



`__proto__` 접근자 프로퍼티 === 함수 객체의 `prototype` 프로퍼티

4. 프로토타입의 constructor 프로퍼티와 생성자 함수

- constructor 프로퍼티 : prototype 프로퍼티로 자신을 참조하고 있는 생성자 함수

19-4 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법에 의해 생성된 객체의 종류

```
//객체 리터럴
const obj={};

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

//배열 리터럴
const arr=[1,2,3];

//정규 표현식 리터럴
const regexp=/is/ig;
```

객체 리터럴 constructor 프로퍼티출력

```
const obj={};
//하지만 obj 객체의 생성자 함수는 Object 생성자 함수다.(?????왜 true일
console.log(obj.constructor===Object); //true
```

→ 객체 리터럴에 의해 생성된 위 객체들은 생성자 함수로 생성된걸까?? 라는 의문이 든다.

▼ 답

리터럴 표기법에 의해 생성된 객체도 상속을 위해 프로토타입이 필요하다. (why? 프로토타입이 생성자 함수와 더불어 생성되고 prototype, constructor 프로퍼티에 의해 연결되어 있기 때문)따라서 리터럴 표기법에 의해 생성된 객체도 가상의 생성자 함수를 갖는다.

정리

- 리터럴 표기법(객체 리터럴, 배열 리터럴, 정규 표현식 리터럴 등)에 의해 생성된 객체는 생성자 함수에 의해 생성된 객체는 아니지만 생성자 함수로 생성한 객체와 본질적인 면에서 큰 차이가 없다. (가상 생성자 함수로 생성된다고 생각하자.)
 - why? 둘다 객체이기 때문에
- 따라서 constructor 프로퍼티를 통해 연결되어 있는 생성자 함수를 리터럴 표기법으로 생성한 객체를 생성한 생성자 함수로 생각해도 크게 무리가 없다.

19-5 프로토타입의 생성시점

프로토타입은 **생성자 함수가 생성되는 시점에 더불어 생성된다**. 프로토 타입과 생성자 함수는 단독으로 존재할 수 없고 **쌍**으로 존재한다.

1. 사용자 정의 생성자 함수와 프로토타입 생성 시점

- 생성 시점 : 함수 객체를 생성하는 시점에 프로토타입도 생성된다.
- 런타임 이전에 자바스크립트 엔진에 의해 먼저 실행되는 **함수선언문으로 정의된 생성자 함수**는 당연히 어떤 코드보다 먼저 평가되어 함수 객체가 된다. 즉 **호이스팅 적용!**

2. 빌트인 생성자 함수와 프로토타입 생성 시점

▼ 빌트인 생성자 함수?

- 자바스크립트 언어 사양에 내장되어 있는 기본 제공 생성자 함수.
- 종류 : `Object` , `String` , `Number` , `Function` , `Array` , `RegExp` , `Date` , `Promise`
- 생성 시점 : 전역 객체가 생성되는 시점에 생성된다.

모든 빌트인 생성자 함수는 전역 객체(window)가 생성되는 시점에 생성된다.

정리

빌트인 생성자 함수들은 자바스크립트 엔진이 실행될 때 필요한 시점(=전역 객체 생성 시점)에 생성되고, 객체가 생성되기 이전에는 단순히 함수와 프로토타입 객체만이 존재하게 됩니다. 이들은 객체가 생성될 때 해당 객체의 프로토타입 체인에 연결되어 사용된다.

19-6 객체 생성 방식과 프로토타입의 결정

1. 객체 생성 방식

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

위와 같이 여러가지 객체 생성 방식이 있으나 **모두 추상연산 `OrdinaryObjectCreate`에 의해 생성된다.**

▼ 추상연산 `OrdinaryObjectCreate`에 의한 객체 생성 순서

1. 생성할 객체의 프로토타입을 인수로 전달 받는다.

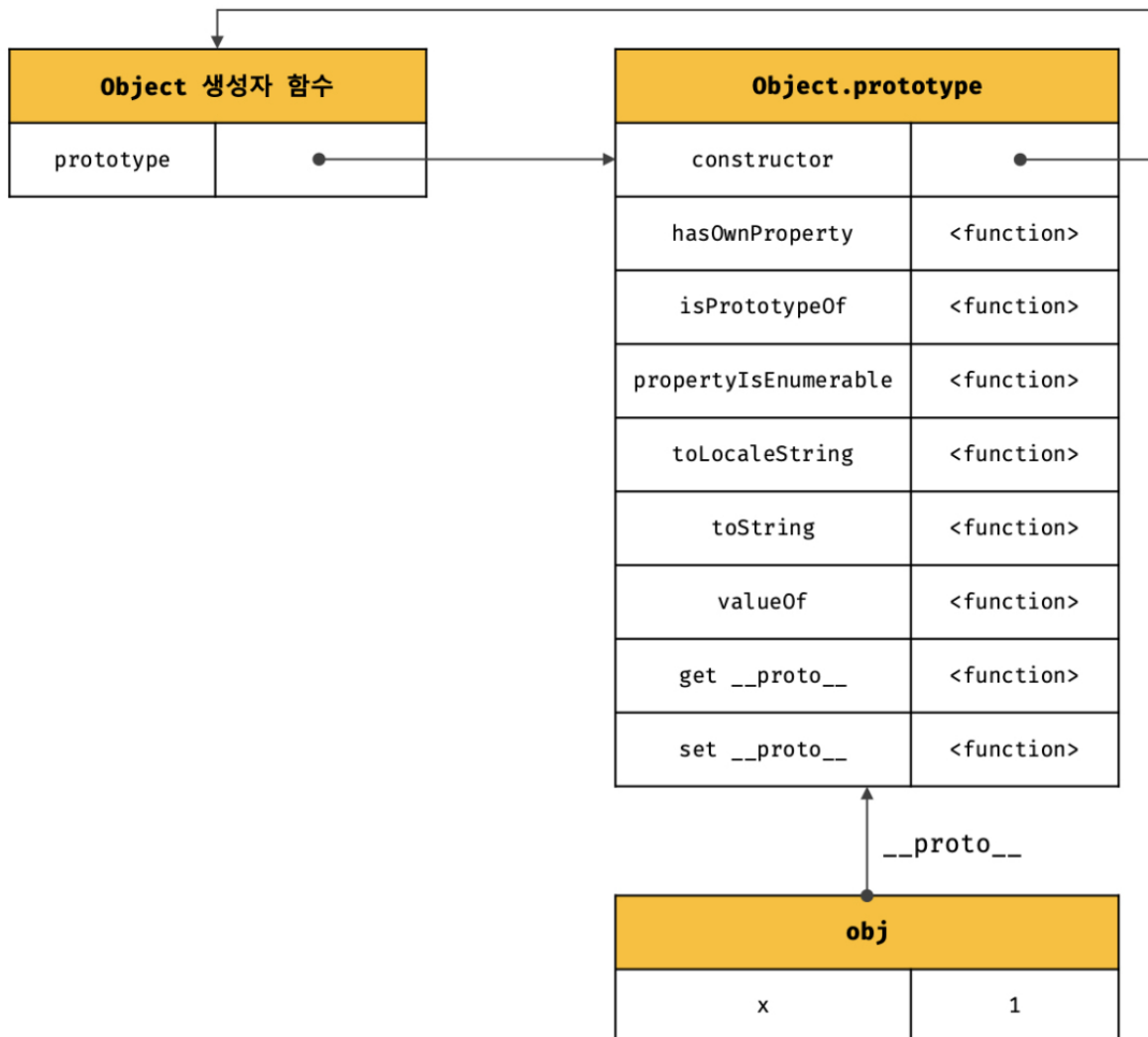
2. 자신이 생성할 객체에 추가할 프로퍼티 목록을 옵션으로 전달한다.
3. 빈 객체를 생성한다.
4. 객체에 추가할 프로퍼티 목록이 인수로 전달된 경우 객체에 프로퍼티를 추가한다.
5. 인수로 전달받은 프로토타입을 생성한 객체의 `[[Prototype]]` 내부 슬롯에 할당한다.
6. 생성한 객체를 반환한다.

2. 객체 리터럴에 의해 생성된 객체의 프로토타입

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

//객체 리터럴에 의해 생성된 obj 객체는 Object.prototype을 상속받는다.
console.log(obj.constructor ===Object);//true
console.log(obj.hasOwnProperty('x')); //true
```

객체 리터럴에 의해 생성된 객체는 `Object.prototype`을 프로토타입으로 갖게 된다. 즉 `Object.prototype`을 상속는 것이다.



따라서 Object.prototype에서 상속받았기때문에 자체적으로 가지고 있지는 않지만 constructor프로퍼티와 hasOwnProperty 메서드 등을 자유롭게 사용할 수 있다.

3. Object 생성자 함수에 의해 생성된 객체의 프로토타입

```
const obj=new Object();
obj.x=1;
```

추상연산에 의해 전달되는 프로토타입은 Object.prototype 이다.

4. 생성자 함수에 의해 생성된 객체의 프로토타입

추상연산에 의해 전달되는 프로토타입은 생성자 함수의 prototype 프로퍼티에 바인딩 되어 있는 객체이다.

19-7 프로토타입 체인

자바스크립트가 객체의 프로퍼티(메서드 포함)에 접근하려고 할 때 해당 객체에 접근하려는 프로퍼티가 없다면 [[Prototype]] 내부슬롯의 참조를 따라 자신의 부모 역할을 하는 프로토타입의 프로퍼티를 순차적으로 검색하는 것.

- 프로토타입 체인의 **최상위 객체**는 언제나 `Object.prototype` 이다. `Object.prototype` 을 **체인의 종점(end of prototype chain)**이라한다.
- 프로토타입 체인의 종점에서도 프로퍼티를 검색할 수 없는 경우 `undefined` 를 반환한다.
- 프로토타입은 **상속과 프로퍼티 검색을 위한 메커니즘**

▼ 스코프 체인

프로퍼티는 프로토타입 체인에서 검색을 하고 식별자는 스코프 체인에서 검색을 한다고 한다.

따라서 **스코프 체인**은 식별자 검색을 위한 메커니즘이다.

스코프 체인과 **프로토타입 체인**은 서로 협력하여 식별자와 프로퍼티를 검색하는데 사용된다.

19-8 오버라이딩과 프로퍼티 새도잉

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

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

  //생성자 함수 반환
}
```

```

    return Person;
  }());

  const me = new Person('Lee');

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

  //인스턴스 메서드가 호출되다. 프로토타입메서드인 인스턴스 메서드에 의해 가려
  me.sayHello();

```

▼ 코드 해석

IIFE(Immediately Invoked Function Expression)로 생성자 함수 정의

```

const Person = (function () {
  // 생성자 함수
  function Person(name) {
    this.name = name;
  }

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

  // 생성자 함수 반환
  return Person;
})();

```

- IIFE를 사용하여 `Person` 생성자 함수를 정의하고 반환합니다.
- IIFE는 즉시 실행되며, `Person` 생성자 함수는 반환된 값을 받아 전역 변수 `Person`에 할당됩니다.

- 생성자 함수 `Person` 은 인스턴스를 생성할 때 `name` 을 받아서 `this.name` 에 할당합니다.
- `Person.prototype` 에 `sayHello` 메서드를 정의합니다. 이 메서드는 `Person` 의 인스턴스에서 사용할 수 있습니다.

인스턴스 생성

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

- `new` 키워드를 사용하여 `Person` 의 인스턴스를 생성합니다.
- 생성된 인스턴스 `me` 의 `name` 속성은 `'Lee'` 로 설정됩니다.

인스턴스 메서드 추가 및 호출

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

// 인스턴스 메서드가 호출되다. 프로토타입 메서드는 인스턴스 메서드에 의해 가려진다.
me.sayHello();
```

- `me` 인스턴스에 새로운 `sayHello` 메서드를 정의합니다. 이 메서드는 인스턴스 레벨에서 정의되었기 때문에 프로토타입 메서드를 덮어씁니다.
- 인스턴스 메서드를 호출하면, `Hey, My name is Lee` 가 출력됩니다. 이 때, **인스턴스 메서드가 프로토타입 메서드를 가리기 때문에 인스턴스 메서드가 우선적으로 호출** 됩니다. (프로퍼티 shadowing 발생)

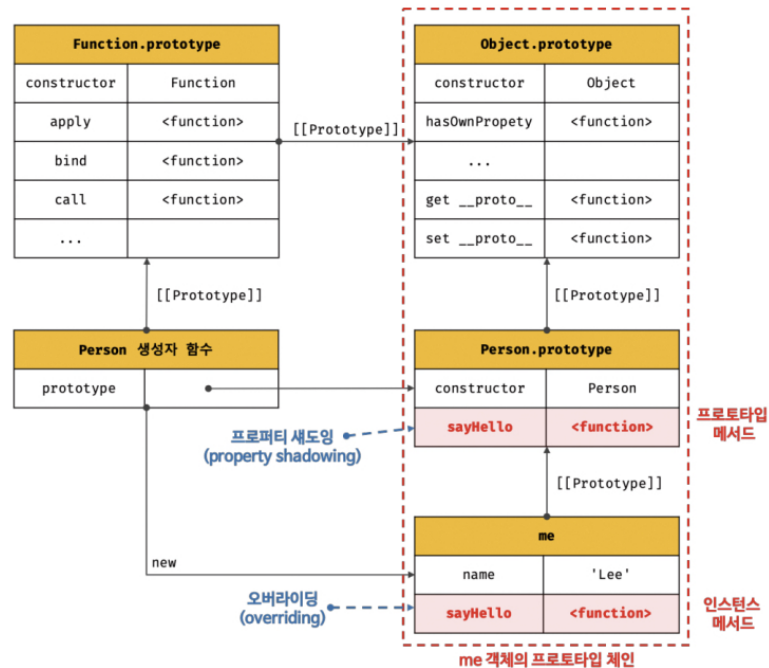


그림 19-19 오버라이딩과 프로퍼티 새도잉

프로토타입 프로퍼티와 같은 이름의 프로퍼티를 인스턴스에 추가하면 프로토타입 체인을 따라 프로토타입 프로퍼티를 검색하여 프로토타입 프로퍼티를 덮어쓰는 것이 아니라 인스턴스 프로퍼티로 추가한다. 이때 인스턴스 메서드 `sayHello`는 프로토타입 메서드 `sayHello`를 오버라이딩 했고 프로토타입 메서드 `sayHello`는 가려진다. 이러한 현상을 **프로퍼티 새도잉** 이라한다.

삭제 시에도 같다.

```
//인스턴스 메서드를 삭제
delete me.sayHello;
//인스턴스에는 sayHello 메서드가 없으므로 프로토타입 메서드가 호출된다.
me.sayHello(); //Hi! My name is Lee

//인스턴스 메서드가 삭제되었으니 프로토타입 체인을 타고 프로토타입 메서드가
//->x
delete me.sayHello;
//프로토타입 메서드가 삭제되지 않고 호출됨
me.sayHello(); //Hi! My name is Lee
```

▼ 프로퍼티 새도잉

상속관계에 의해 프로퍼티가 가려지는 현상

▼ 오버라이딩

상위 클래스가 가지고 있는 메서드를 하위 클래스가 재정의하여 사용하는 방식

▼ 오버로딩

함수의 이름은 동일하지만 매개변수의 타입 또는 개수가 다른 메서드를 구현하고 매개변수에 의해 메서드를 구별하여 호출하는 방식. js는 오버로딩을 지원하지 않지만 arguments 객체를 사용하여 구현할 수는 있다.

정리

- 하위 하위 객체(인스턴스)를 통해 프로토타입에 get 액세스는 허용되나 set 액세스는 허용되지 않는다.
- 프로토타입 프로퍼티를 변경 또는 삭제하려면 하위 객체(인스턴스)를 통해 프로토타입 체인으로 접근하는 것이 아니라 프로토타입에 직접 접근해야 한다.

19-9 프로토타입의 교체

프로토타입은 임의의 다른 객체로 변경할 수 있다 === 부모 객체인 프로토타입을 동적으로 변경할 수 있다.

그렇다면 어떻게 변경할까?

방법1. 생성자 함수에 의한 프로토타입의 교체

```
const Person=(function (){
  function Person(name){
    this.name=name;
  }
  //생성자 함수의 prototype 프로퍼티를 통해 프로토타입을 교체
  Person.prototype={
    sayHello(){
      console.log(`Hi My name is ${this.name}`);
    }
  };
};
```

```

return Person;
})();

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

```

프로토타입으로 교체한 객체 리터럴에는 constructor 프로퍼티가 없다.

→ constructor 프로퍼티와 생성자 함수 간의 연결 파괴

그렇다면 이 연결을 되살리기 위해서는 어떻게 해야할까?

→ 명시적으로 constructor 프로퍼티를 추가하기

```

const Person=(function (){
  function Person(name){
    this.name=name;
  }
  //생성자 함수의 prototype 프로퍼티를 통해 프로토타입을 교체
  Person.prototype={
    //constructor 프로퍼티와 생성자 함수 간의 연결을 설정
    constructor:Person,
    sayHello(){
      console.log(`Hi My name is ${this.name}`);
    }
  };
  return Person;
})();

const me = new Person('Lee');

//consturctor 프로퍼티가 생성자 함수를 가리킨다.
console.log(me.constructor === Person); //true
console.log(me.constructor === Object); //false

```

방법2 인스턴스에 의한 프로토타입의 교체

프로토타입은 인스턴스의 `__proto__` 접근자 프로퍼티를 통해 접근할 수 있으므로 교체도 할 수 있다.

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

const me = new Person('Lee');

//프로토타입으로 교체할 객체
const parent ={
  sayHello(){
    console.log(`Hi! My name is ${this.name}`);
  }
};

//me 객체의 프로토타입을 parent 객체로 교체한다.
Object.setPrototypeOf(me, parent);
// 위 코드는 아래의 코드와 동일하게 동작한다.
//me.__proto__ =parent;

me.sayHello(); //Hi! My name is Lee
//프로토타입을 교체하면 constructor 프로퍼티와 생성자 함수 간의 연결이 파
console.log(me.constructor ===Person); //false
//프로토타입 체인을 따라 Object.prototype의 constructor 프로퍼티가 건
console.log(me.constructor ===Object); //true
```

교체된 객체에는 constructor 프로퍼티 없으므로 constructor 프로퍼티와 생성자 함수 간의 연결이 파괴된다.

다시 연결하려면?

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

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

```

// 프로토타입으로 교체할 객체
const parent = {
  // ① constructor 프로퍼티와 생성자 함수 간의 연결을 설정
  constructor: Person,
  sayHello() {
    console.log(`Hi! My name is ${this.name}`);
  }
};

// ② 생성자 함수의 prototype 프로퍼티와 프로토타입 간의 연결을 설정
Person.prototype = parent;

// ③ me 객체의 프로토타입을 parent 객체로 교체한다.
Object.setPrototypeOf(me, parent);
// 위 코드는 아래의 코드와 동일하게 동작한다.
// me.__proto__ = parent;

me.sayHello(); // Hi! My name is Lee

// constructor 프로퍼티가 생성자 함수를 가리킨다.
console.log(me.constructor === Person); // true
console.log(me.constructor === Object); // false

// 생성자 함수의 prototype 프로퍼티가 교체된 프로토타입을 가리킨다.
console.log(Person.prototype === Object.getPrototypeOf(me));

```

이렇게 하면 된다. 이처럼 프로토타입 교체를 통해 객체 간의 상속 관계를 동적으로 변경하는 것은 꽤나 번거롭다. 따라서 직접 교체하지 않는 것이 좋다.

19-10 instanceof 연산자

instanceof 연산자

- 생성자의 `prototype` 속성이 객체의 프로토타입 체인 어딘가 존재하는지 판별
- 우변의 생성자 함수의 `prototype`에 바인딩된 객체가 좌변의 객체의 프로토타입 체인 상에 존재하며 `true`로 평가되고, 그렇지 않은 경우에는 `false`로 평가 된다.

19-11 직접 상속

직접적으로 상속할 수 있는 방식

1. `Object.create`에 의한 상속

- `new` 연산자 없이도 객체를 생성할 수 있다.
- 프로토타입을 지정하면서 객체를 생성할 수 있다.
- 객체 리터럴에 의해 생성된 객체도 상속받을 수 있다.

2. 객체 리터럴 내부에서 `__proto__`에 의한 직접 상속

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

// 객체 리터럴에 의해 객체를 생성하면서 프로토타입을 지정하여 직접 상속받을
const obj = {
  y: 20,
  // 객체를 직접 상속받는다.
  // obj → myProto → Object.prototype → null
  __proto__: myProto
};

/* 위 코드는 아래와 동일하다.
const obj = Object.create(myProto, {
  y: { value: 20, writable: true, enumerable: true, configura
});
*/

console.log(obj.x, obj.y); // 10 20
console.log(Object.getPrototypeOf(obj) === myProto); // true
```

19-12 정적 프로퍼티 / 메서드

정적 프로퍼티 / 메서드

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

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

// 프로토타입 메서드
```

```

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

// ① 정적 프로퍼티
Person.staticProp = 'static prop';

// ② 정적 메서드
Person.staticMethod = function () {
  console.log('staticMethod');
};

const me = new Person('Lee');

// 생성자 함수에 추가한 정적 프로퍼티/메서드는 생성자 함수로 참조/호출한다.
Person.staticMethod(); // staticMethod

// ③ 정적 프로퍼티/메서드는 생성자 함수가 생성한 인스턴스로 참조/호출할 수 없다.
// 인스턴스로 참조/호출할 수 있는 프로퍼티/메서드는 프로토타입 체인 상에 존재한다.
me.staticMethod(); // TypeError: me.staticMethod is not a function

```

③에서 보면 정적 프로퍼티/메서드는 생성자 함수가 생성한 **인스턴스로** 참조/호출할 수 없다.

`Object.create` 같은 메서드는 **정적 메서드**이고,

`Object.prototype.hasOwnProperty` 같은 메서드는 **프로토타입 메서드**이다.

프로토타입 메서드를 호출하려면 인스턴스를 생성해야 하지만,

정적 메서드는 인스턴스를 생성하지 않아도 호출할 수 있다.

```

function Foo() {}

// 프로토타입 메서드
// this를 참조하지 않는 프로토타입 메서드는 정적 메서드로 변경해도 동일한
Foo.prototype.x = function () {
  console.log('x');
};

const foo = new Foo();

```

```
// 프로토타입 메서드를 호출하려면 인스턴스를 생성해야 한다.
foo.x(); // x

// 정적 메서드
Foo.x = function () {
  console.log('x');
};

// 정적 메서드는 인스턴스를 생성하지 않아도 호출할 수 있다.
Foo.x(); // x
```

19-13 프로퍼티 존재 확인

1. in 연산자

- 확인 대상 객체 내에 특정 프로퍼티가 존재하는지 여부를 확인한다.
- 존재 확인 여부를 확인하는 범위는 대상 객체가 상속받은 모든 프로토타입의 프로퍼티이다.
- 사용법

```
/**
 * key : 프로퍼티 키를 나타내는 문자열
 * object : 객체로 평가되는 표현식
 */
ket in object
```

2. Object.prototype.hasOwnProperty 메서드

```
const person={name:'Lee'};
console.log(person.hasOwnProperty('name')); //true
console.log(person.hasOwnProperty('age')); //false
```

3. Reflect 메서드 (ES6 부터)

```
const person={name:'Lee'};  
console.log(Reflect.has(person, 'name'));//true  
console.log(Reflect.has(person, 'age')); //false
```

19-14 프로퍼티 열거

1. for...in문

- 객체의 프로토타입 체인 상에 존재하는 모든 프로토타입의 프로퍼티 중에서 프로퍼티 어트리뷰트 [[Enumerable]]의 값이 true인 프로퍼티를 순회하며 열거한다.
- 심벌인 프로퍼티는 열거하지 않는다.
- 프로퍼티를 열거할 때 순서를 보장하지 않는다.

2. Object.keys/values/entries 메서드

- 상속받은 프로퍼티까지 열거하는 for...in문 말고 객체 자신의 고유 프로퍼티만 열거하고 싶을 때 사용
- 열거 가능한 프로퍼티 키를 배열로 반환한다.