19장 프로토타입

- 자바스크립트는 프로토타입 기반의 객체지향 프로그래밍 언어
- 자바스크립트는 객체 기반의 프로그래밍 언어이며 자바스크립트를 이루고 있는 거의 모든 것이 객체다.(원시 타입 값 제외한 나머지 값들(함수, 배열, 정규 표현식 등 모두 객체)

객체지향 프로그래밍

- 객체지향 프로그래밍은 실세계의 실체를 인식하는 철학적 사고를 프로그래밍에 접목하려는 시도에서 시작
- 실체는 특징이나 성질을 나타내는 **속성**을 가지고 있고 이를 통해 실체를 인식 및 구별
- **추상화**: 프로그램에 필요한 속성만 간추려 내어 표현하는 것(사람 객체에서 '이름'과 '주소'라는 속성을 갖게 함)
- **객체**: 속성을 통해 여러 개의 값을 하나의 단위로 구성한 복합적인 자료 구조
- 객체지향 프로그래밍은 객체의 상태를 나타내는 데이터와 상태 데이터를 조작할 수 있는 동작을 하나의 논리적인 단위로 묶어 생각한다.
- 따라서 객체는 상태 데이터(property)와 동작(method)을 하나의 논리적인 단위로 묶은 복합적인 자료 구조

상속과 프로토타입

- 상속: 어떤 객체의 프로퍼티 또는 메서드를 다른 객체가 상속받아 그대로 사용할 수 있는 것
- 자바스크립트는 프로토타입을 기반으로 상속을 구현해 불필요한 중복을 제거

```
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);

// 생성자 함수는 동일한 프로퍼티 구조를 갖는 객체를 여러 개 생성할 때 유용
// radius 프로퍼티 값은 인스턴스마다 다르지만 getArea 메서드는 모든 인스턴스가 동일한 내용의 메서드를 사용
// 모든 인스턴스가 동일한 메서드를 중복 소유하는 것은 메모리 낭비, 퍼포먼스에도 악영향
// 상속을 통해 불필요한 중복을 줄일 수 있다.
```

```
function Circle(radius) {
    this.radius = radius;
}

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

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

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

- 생성자 함수가 생성할 **모든 인스턴스가 공통적으로 사용할 프로퍼티나 메서드를 프로토타입에 미리 구현**해 두면 생성자 함수가 생성할 모든 인스턴스는 별도의 구현 없이 상위(부모) 객체인 프로토타입의 자산을 공유하여 사용할 수 있다.
-

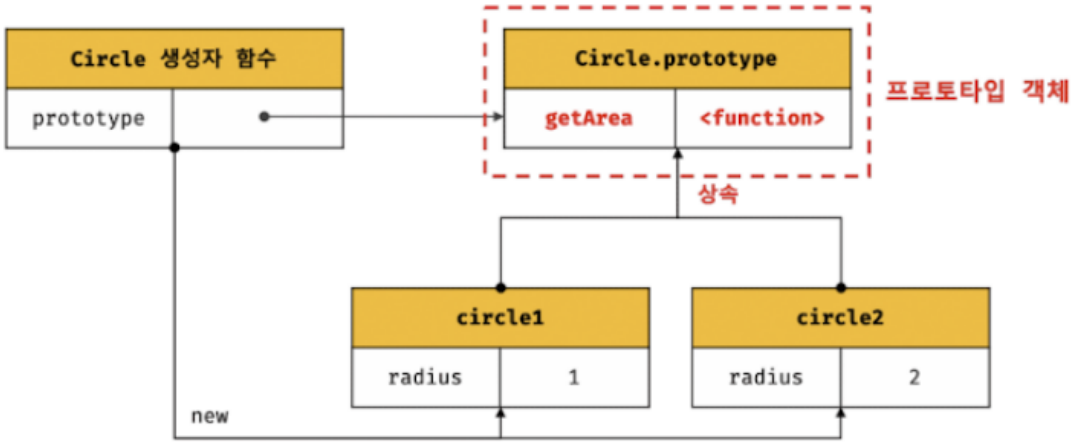


그림 19-2 상속에 의한 메서드 공유

프로토타입 객체

- 프로토타입은 객체지향 프로그래밍의 근간을 이루는 **객체 간 상속**을 구현하기 위해 사용된다.
- 프로토타입은 어떤 객체의 **상위(부모) 객체의 역할을 하는 객체**로서 다른 객체에 **공유 프로퍼티**를 제공
- 프로토타입을 상속받은 하위(자식) 객체는 상위 객체의 프로퍼티를 자신의 프로퍼티처럼 자유롭게 사용 가능
- 모든 객체는 [[Prototype]] 이라는 내부 슬롯을 가지며, 이 내부 슬롯의 값은 프로토타입의 참조.
  - [[Prototype]]에 저장되는 프로토타입은 객체 생성 방식에 의해 결정됨
  - 객체가 생성될 때 객체 생성 방식에 따라 프로토타입이 결정되고 [[Prototype]]에 저장된다.
- 모든 객체는 하나의 프로토타입을 갖는다.
- 모든 프로토타입은 생성자 함수와 연결되어 있다.
-

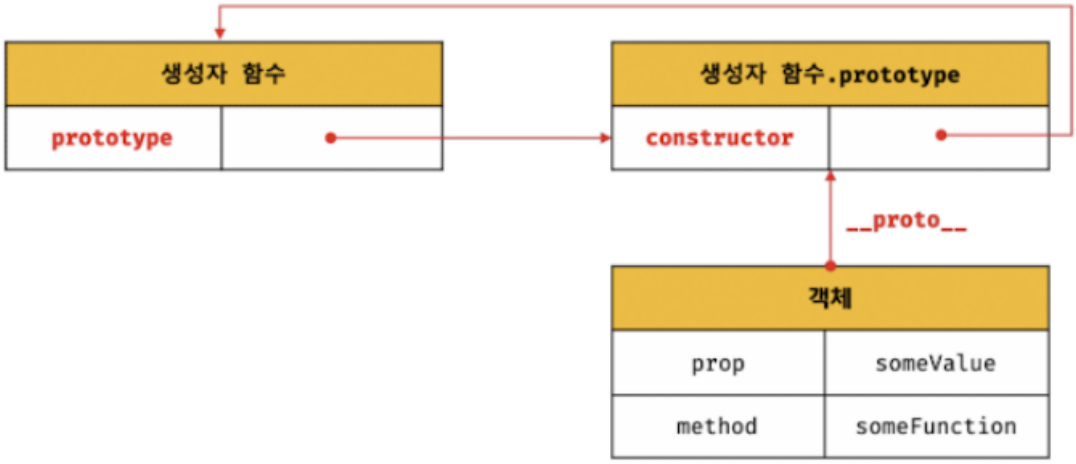


그림 19-3 객체와 프로토타입과 생성자 함수는 서로 연결되어 있다.

- [[Prototype]] 내부 슬롯에는 직접 접근 불가능
- __ proto __ 접근자 프로퍼티를 통해 자신의 프로토타입, 즉 자신의 [[Prototype]] 내부 슬롯이 가리키는 프로토타입에 간접적으로 접근 가능
- 프로토타입은 자신의 constructor 프로퍼티를 통해 생성자 함수에 접근 가능
- 생성자 함수는 자신의 prototype 프로퍼티를 통해 프로토타입에 접근 가능

- `__proto__` 접근자 프로퍼티
  - 모든 객체는 __ proto __ 접근자 프로퍼티를 통해 자신의 프로토타입, 즉 [[Prototype]] 내부 슬롯에 간접적 접근 가능
  - [[Prototype]]은 내부 슬롯이므로 원칙적으로 직접 접근은 불가능하나 __ proto __ 접근자 프로퍼티를 통해 간접적으로 접근 가능하다.
  - 접근자 프로퍼티
    - 접근자 프로퍼티는 자체적으로 값을 갖지 않고
    - 다른 데이터 프로퍼티의 값을 읽거나 저장할 때 사용하는 접근자 함수
    - 즉, [[Get]], [[Set]] 프로퍼티 어트리뷰트로 구성된 프로퍼티
  - Object.prototype의 접근자 프로퍼티인 __ proto __ 는 getter/setter 함수라고 부르는 접근자 함수([[Get]], [[Set]])를 통해 [[Prototype]] 내부 슬롯의 값, 즉 프로토타입을 취득하거나 할당한다.
  - __ proto __ 접근자 프로퍼티를 통해 프로토타입에 접근하면 __ proto __ 접근자 프로퍼티의 getter 함수인 [[Get]] 호출
  - __ proto __ 접근자 프로퍼티를 통해 새로운 프로토타입을 할당하면 __ proto __ 접근자 프로퍼티의 setter 함수인 [[Set]] 호출
  - ```
    const obj = {};
    const parent = { x : 1 };

    // getter 함수 호출
    obj.__proto__ ;

    // setter 함수 호출
    obj.__proto__ = parent;

    console.log(obj.x); // 1
    ```
  - __ proto __ 접근자 프로퍼티는 객체가 직접 소유하는 프로퍼티가 아니라 Object.prototype의 프로퍼티
    ```

- 모든 객체는 상속을 통해 Object.prototype.__ proto __ 접근자 프로퍼티를 사용 가능

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

console.log(person.hasOwnProperty('__proto__')); // false

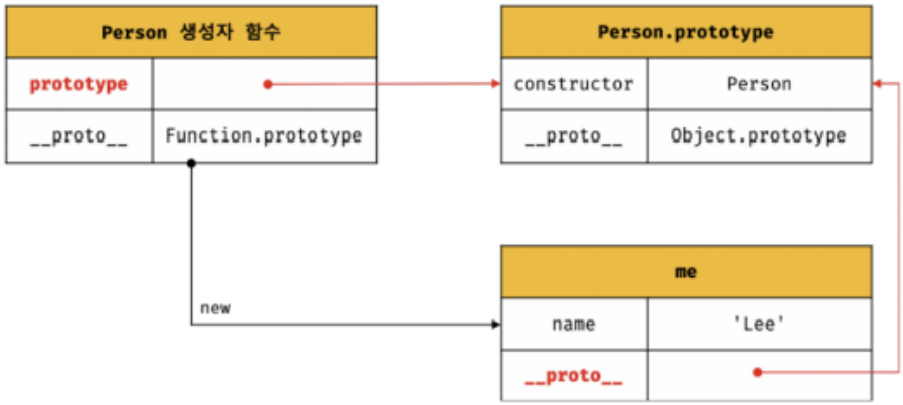
console.log(Object.getOwnPropertyDescriptor(Object.prototype,
'__proto__'));
// {get: f, set: f, enumerable: false, configurable: false}

console.log({}.__proto__ === Object.prototype); // true
// 모든 객체는 Object.prototype의 접근자 프로퍼티 __proto__를 상속받아 사용할
수 있다.
```

- __ proto __ 접근자 프로퍼티를 통해 프로토타입에 접근하는 이유
  - 상호 참조에 의해 프로토타입 체인이 생성되는 것을 방지하기 위해
  - ```
    const parent = {};
    const child = {};

    child.__proto__ = parent;
    parent.__proto__ = child;

    // 이러한 코드가 에러 없이 정상적으로 처리되면서
    // 서로가 자신의 프로토타입이 되는 비정상적인 프로토타입 체인이 만들어짐
    ```
  - 프로토타입 체인은 단방향 링크드 리스트로 구현되어야 한다.
  - 따라서, 아무런 체크 없이 무조건적으로 프로토타입을 교체할 수 없도록 __ proto __ 접근자 프로퍼티를 통해 프로토타입에 접근하고 교체하도록 구현
- __ proto __ 접근자 프로퍼티를 코드 내에서 직접 사용하는 것은 권하지 않는다.
  - 모든 객체가 __ proto __ 접근자 프로퍼티를 사용할 수 있는 것은 아니기 때문
  - 따라서 __ proto __ 접근자 프로퍼티 대신 프로토타입의 참조를 취득하고 싶은 경우에는 Object.getPrototypeOf 메서드를 사용
  - 프로토타입을 교체하고 싶은 경우에는 Object.setPrototypeOf 메서드를 사용할 것을 권장
  - ```
    const obj = {};
    const parent = { x : 1 };

    Object.getPrototypeOf(obj);
    Object.setPrototypeOf(obj, parent);

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

- 함수 객체의 prototype 프로퍼티
  - 함수 객체만이 소유하는 prototype 프로퍼티는 생성자 함수가 생성할 인스턴스의 프로토타입을 가리킴
  - 일반 객체는 prototype 프로퍼티를 소유하지 않는다.
  - prototype 프로퍼티는 생성자 함수가 생성할 객체의 프로토타입을 가리킨다.
  - 따라서 생성자 함수로서 호출할 수 없는 함수(non-constructor)인 화살표 함수와, 메서드 축약 표현은 prototype 프로퍼티를 소유하지 않으며, 프로토타입도 생성하지 않음

- 생성자 함수로 호출하기 위해 정의하지 않은 일반 함수도 prototype 프로퍼티를 소유하지만 객체를 생성하지 않는 일반 함수의 prototype 프로퍼티는 아무 의미도 없다
- 모든 객체가 가지고 있는 __proto__ 접근자 프로퍼티와 함수 객체만이 가지고 있는 prototype 프로퍼티는 결국 동일한 프로토타입을 가리킨다.

구분	소유	값	사용 주체	사용 목적
__proto__ 접근자 프로퍼티	모든 객체	프로토타입의 참조	모든 객체	객체가 자신의 프로토타입에 접근 또는 교체하기 위해 사용
prototype 프로퍼티	constructor	프로토타입의 참조	생성자 함수	생성자 함수가 자신이 생성할 객체(인스턴스)의 프로토타입을 할당하기 위해 사용

  

  그림 19-7 객체의 __proto__ 접근자 프로퍼티와 함수 객체의 prototype 프로퍼티는 결국 동일한 프로토타입을 가리킨다.

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

  const me = new Person('Lee');

  console.log(me.__proto__ === Person.prototype); // true
  ```

- 프로토타입의 constructor 프로퍼티와 생성자 함수
  - 모든 프로토타입은 constructor 프로퍼티를 갖는다.
  - 이 constructor 프로퍼티는 prototype 프로퍼티로 자신을 참조하고 있는 생성자 함수를 가리킨다.
  - 이 연결은 생성자 함수가 생성될 때(함수 객체가 생성될 때) 이루어진다.
  - ```
    function Person(name) {
        this.name = name;
    }

    const me = new Person('Lee');

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

    // me 객체에는 constructor가 없다.
    // 하지만 me 객체의 프로토타입인 Person.prototype에는 constructor 프로퍼티가 있다.
    ```
  -
    ```

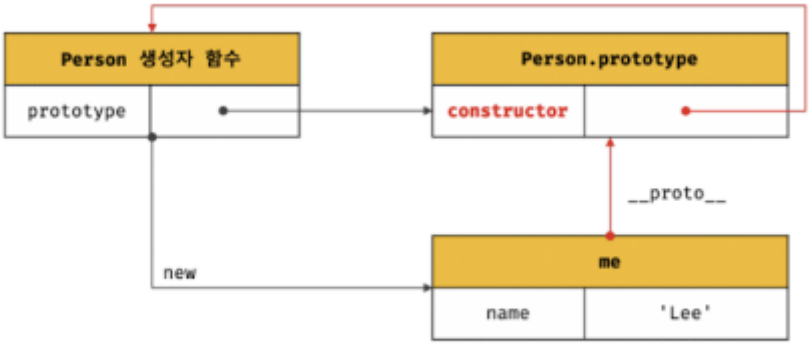


그림 19-8 프로토타입의 constructor 프로퍼티

리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

- 리터럴 표기법에 의한 객체 생성 방식과 같이 명시적으로 new 연산자와 함께 생성자 함수를 호출해 인스턴스를 생성하지 않는 객체 생성 방식도 있다.
- 하지만, 리터럴 표기법에 의해 생성된 객체의 경우 프로토타입의 constructor 프로퍼티가 가리키는 생성자 함수가 반드시 객체를 생성한 생성자 함수라고 단정할 수 없다.
  - Object 생성자 함수
    - **19.1.1.1 Object ([*value*])**

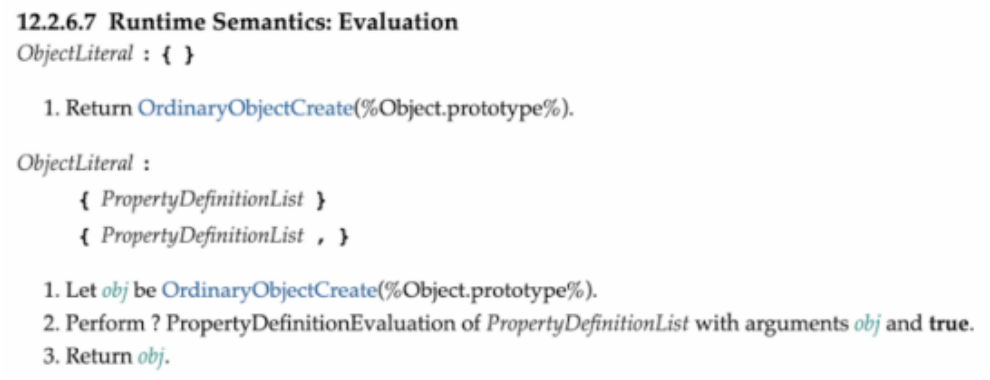
      When the **Object** function is called with optional argument *value*, the following steps are taken:

      1. If NewTarget is neither **undefined** nor the active function, then
         a. Return ? OrdinaryCreateFromConstructor(NewTarget, "%**Object.prototype**%").
      2. If *value* is **undefined** or **null**, return OrdinaryObjectCreate(%Object.prototype%).
      3. Return ! ToObject(*value*).

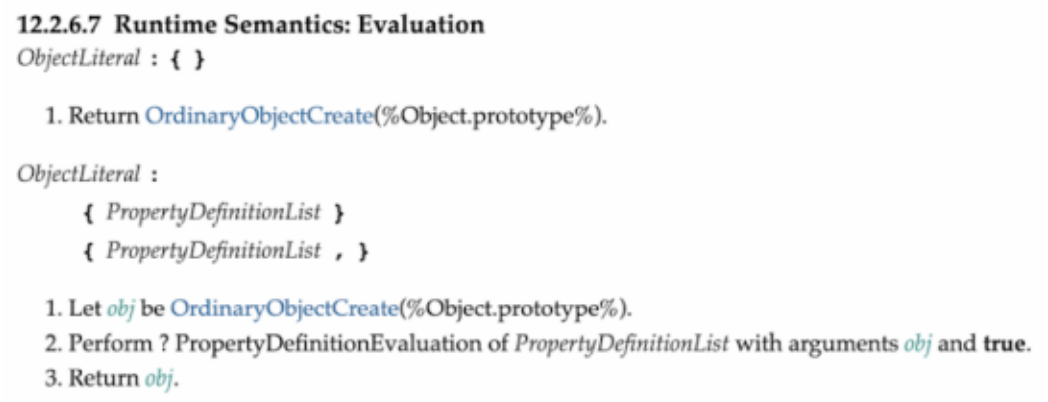
      The "**length**" property of the **Object** constructor function is 1.

      그림 19-9 Object 생성자 함수[11]

    - ```
      // 1. new.target이 undefined나 Object가 아닌 경우
      // (new.target을 사용하면 new 연산자와 함께 생성자 함수로서 호출되었는지 확인)
      // 즉, new.target이 undefined: 일반 함수로 생성됨
      // 인스턴스 → Foo.prototype → Object.prototype 순으로 프로토타입 체인이 생성
      class Foo extends Object {}
      new Foo();

      // 2. Object 생성자 함수에 의한 객체 생성
      // 인수가 전달되지 않았으므로 OrdinaryObjectCreate를 호출해 빈 객체를 생성
      obj = new Object();
      console.log(obj); // {}

      // 3. 인수가 전달된 경우에는 인수를 객체로 변환
      obj = new Object(123);
      console.log(obj); // Number {123}
      ```

  - 객체 리터럴의 평가
      ```



12.2.6.7 Runtime Semantics: Evaluation

ObjectLiteral : { }

1. Return OrdinaryObjectCreate(%Object.prototype%).

ObjectLiteral :

{ *PropertyDefinitionList* }

{ *PropertyDefinitionList* , }

1. Let *obj* be OrdinaryObjectCreate(%Object.prototype%).
2. Perform ? PropertyDefinitionEvaluation of *PropertyDefinitionList* with arguments *obj* and **true**.
3. Return *obj*.

그림 19–10 객체 리터럴의 평가[13]

        - OrdinaryObjectCreate를 호출하여 빈 객체를 생성하고 프로퍼티를 추가하도록 정의됨
    - 즉, Object 생성자 함수 호출과 객체 리터럴의 평가는 추상 연산 OrdinaryObjectCreate를 호출하여 빈 객체를 생성하는 점에서는 동일
    - 하지만 new.target 확인이나 프로퍼티를 추가하는 처리 등 세부 내용은 다름
    - **따라서 객체 리터럴에 의해 생성된 객체는 Object 생성자 함수가 생성한 객체가 아니다.**
    - 함수 객체의 경우 차이가 더 명확
        - Function 생성자 함수를 호출하여 생성한 함수는 렉시컬 스코프를 만들지 않고 전역 함수인 것처럼 스코프를 생성하며 클로저도 만들지 않는다.
        - 따라서 **함수 선언문과 함수 표현식을 평가하여 함수 객체를 생성한 것은 Function 생성자가 아니다.**
        - 하지만 constructor 프로퍼티를 통해 확인해보면 Function 생성자 함수이다.
        - ```
          function foo () {} // 함수 선언문으로 생성

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

- 리터럴 표기법에 의해 생성된 객체도 상속을 위해 프로토타입이 필요
- 따라서 리터럴 표기법에 의해 생성된 객체도 가상적인 생성자 함수를 가진다.
- **프로토타입과 생성자 함수는 단독으로 존재할 수 없고 언제나 쌍으로 존재**하기 때문
- 리터럴 표기법에 의해 생성된 객체는 생성자 함수에 의해 생성된 객체는 아니지만 본질적인 면에서 큰 차이는 없다.
    - **객체 리터럴에 의해 생성된 객체와 Object 생성자 함수에 의해 생성한 객체**는 생성 과정에서의 차이는 있지만 결국 **객체로서 동일한 특성**을 갖는다.
    - **함수 리터럴에 의해 생성한 함수와 Function 생성자 함수에 의해 생성한 함수**는 스코프, 클로저 등의 차이가 있지만 결국 **함수로서 동일한 특성**을 갖는다.
- 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입
          ```

- 

12.2.6.7 Runtime Semantics: Evaluation

ObjectLiteral : { }

1. Return OrdinaryObjectCreate(%Object.prototype%).

ObjectLiteral :

{ *PropertyDefinitionList* }

{ *PropertyDefinitionList* , }

1. Let *obj* be OrdinaryObjectCreate(%Object.prototype%).
2. Perform ? PropertyDefinitionEvaluation of *PropertyDefinitionList* with arguments *obj* and **true**.
3. Return *obj*.

그림 19–10 객체 리터럴의 평가[13]

프로토타입의 생성 시점

- 모든 객체는 생성자 함수와 연결되어 있다(리터럴 표기법에 의해 생성된 객체도 생성자 함수와 연결되어 있음)
- 프로토타입은 생성자 함수가 생성되는 시점에 더불어 생성
- 생성자 함수
  - 사용자가 직접 정의한 사용자 정의 생성자 함수
    - 일반 함수로 정의한 함수 객체(화살표 함수나 메서드 축약 표현이 아닌 함수)는 new 연산자와 함께 생성자 함수로서 호출 가능
    - constructor(생성자 함수로서 호출할 수 있는 함수)는 함수 정의가 평가되어 함수 객체를 생성하는 시점에 프로토타입도 더불어 생성
    - ```
      console.log(Person1.prototype); // {constructor: f}

      function Person1(name) {
          this.name = name;
      }
      // 함수 선언문으로 정의된 Person1 생성자 함수는
      // 런타임 이전에 함수 객체를 생성, 식별자를 생성, 할당까지 마친 상태이므로 함수 호이스팅이 일어남
      // 이 때 프로토타입도 더불어 생성


      function Person2 = name => {
          this.name = name;
      }

      console.log(Person2.prototype); //undefined
      // 화살표 함수로 정의된 Person2 함수는(non-constructor)
      // 프로토타입이 생성되지 않는다.
      ```
    - Person1 함수로 생성된 프로토타입은 오직 constructor 프로퍼티만을 갖는 객체이다.
      - 프로토타입도 객체이고 모든 객체는 프로토타입을 가지므로 프로토타입도 자신의 프로토타입을 갖기 때문에
      - 생성된 프로토타입의 프로토타입은 Object.prototype 이다.
      - 즉, **사용자 정의 생성자 함수는 자신이 평가되어 함수 객체로 생성되는 시점에 프로토타입도 더불어 생성되며, 생성된 프로토타입의 프로토타입은 언제나**
      ```

Object.prototype이다.

- 자바스크립트가 기본 제공하는 빌트인 생성자 함수
  - 빌트인 생성자 함수도 일반 함수와 마찬가지로 빌트인 생성자 함수가 생성되는 시점에 프로토타입이 생성된다.
  - 모든 빌트인 생성자 함수는 전역 객체가 생성되는 시점에 생성된다.
  - 생성된 프로토타입은 빌트인 생성자 함수의 prototype 프로퍼티에 바인딩된다.
  - 이후 생성자 함수 또는 리터럴 표기법으로 객체를 생성하면 프로토타입은 생성된 객체의 [[Prototype]] 내부 슬롯에 할당된다.

> 빌트인 생성자 함수는 전역 객체가 생성되는 시점에 생성되고 이와 동시에 프로토타입도 생성된다. 이 프로토타입은 생성자 함수의 prototype 프로퍼티에 바인딩된다. 그 후 객체가 생성될 때 프로토타입은 생성된 객체의 [[Prototype]] 내부 슬롯에 할당된다.

객체 생성 방식과 프로토타입의 결정

- 객체는 다양한 생성 방법이 있고 각 방식마다 세부적인 객체 생성 방식의 차이는 있으나
- 추상 연산 `OrdinaryObjectCreate`에 의해 생성된다는 공통점이 있다.
  - OrdinaryObjectCreate
    - 필수적으로 자신이 생성할 객체의 프로토타입을 인수로 전달받음
    - 자신이 생성할 객체에 추가할 프로퍼티 목록을 옵션으로 전달할 수 있다.
    - 빈 객체를 생성 → 객체에 추가할 프로퍼티 목록이 인수로 전달된 경우 프로퍼티를 객체에 추가 → 인수로 전달받은 프로토타입을 자신이 생성한 객체의 [[Prototype]] 내부 슬롯에 할당 → 생성한 객체를 반환
- 객체 리터럴에 의해 생성된 객체의 프로토타입
  - 자바스크립트 엔진은 객체 리터럴을 평가해 객체를 생성할 때 OrdinaryObjectCreate를 호출 → 이 때 OrdinaryObjectCreate에 전달되는 프로토타입은 Object.prototype이다.
  - 객체 리터럴이 평가되면 OrdinaryObjectCreate에 의해 Object 생성자 함수와 Object.prototype과 생성된 객체 사이에 연결이 만들어짐
  - ```
    const obj = {x : 1};

    console.log(obj.constructor === Object); // true
    console.log(obj.hasOwnProperty('x')); // true

    // obj 객체는 constructor 프로퍼티와 hasOwnProperty 메서드 등을 소유하지 않지만
    // Object.prototype 객체를 상속받았기 때문에
    // 자신의 프로토타입인 Object.prototype의 constructor 프로퍼티와 hasOwnProperty 메서드를 자유롭게 사용 가능
    ```

- Object 생성자 함수에 의해 생성된 객체의 프로토타입
  - Object 생성자 함수를 인수 없이 호출하면 빈 객체가 생성
  - Object 생성자 함수를 호출하면 객체 리터럴과 마찬가지로 OrdinaryObjectCreate가 호출된다.
  - 이 때, OrdinaryObjectCreate에 전달되는 프로토타입은 Object.prototype
  - 즉, Object 생성자 함수에 의해 생성되는 객체의 프로토타입은 Object.prototype
    ```

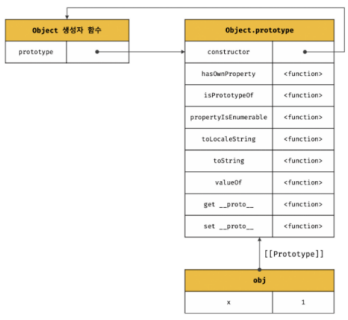
    - OrdinaryObjectCreate에 의해 Object 생성자 함수와 Object.prototype과 생성된 객체 사이에 연결이 만들어짐
    - 객체 리터럴에 의해 생성된 객체와 동일한 구조를 갖는다.
    - ```
      const obj = new Object();
      obj.x = 1;

      console.log(obj.constructor === Object); // true
      console.log(obj.hasOwnProperty('x')); // true
      ```
    - 객체 리터럴과 Object 생성자 함수에 의한 객체 생성 방식의 차이
        - 프로퍼티 추가 방식이 다름
        - 객체 리터럴 → 객체 리터럴 내부에 프로퍼티 추가
        - Object 생성자 함수 → 일단 빈 객체를 생성한 후 프로퍼티 추가
- 생성자 함수에 의해 생성된 객체의 프로토타입
    - new 연산자와 함께 생성자 함수를 호출해 인스턴스를 생성하면 마찬가지로 OrdinaryObjectCreate가 호출된다.
    - 이 때, OrdinaryObjectCreate에 전달되는 프로토타입은 생성자 함수의 prototype 프로퍼티에 바인딩되어 있는 객체
    - 즉, 생성자 함수에 의해 생성되는 객체의 프로토타입은 생성자 함수의 prototype 프로퍼티에 바인딩되어 있는 객체
    - ```
      function Person(name) {
          this.name = name;
      }

      const me = new Person('Lee');

      // Person.prototype의 프로퍼티는 constructor 뿐이다.
      ```
객체 생성 방식	엔진의 객체 생성	인스턴스의 prototype 객체
객체 리터럴	Object() 생성자 함수	Object.prototype
Object() 생성자 함수	Object() 생성자 함수	Object.prototype
생성자 함수	생성자 함수	생성자 함수 이름.prototype
- 

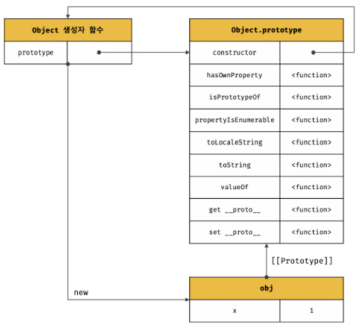
  그림 19-14 객체 리터럴에 의해 생성된 객체의 프로토타입
  

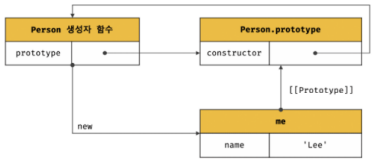
  그림 19-15 Object 생성자 함수에 의해 생성된 객체의 프로토타입
  

  그림 19-16 생성자 함수에 의해 생성된 객체의 프로토타입

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')); // true

  // me 객체는 Object.prototype의 메서드인 hasOwnProperty를 호출 가능
  // me 객체가 Person.prototype 뿐만 아니라 Object.prototype도 상속받았다는 것을 의미
  ```
- ```
  Object.getPrototypeOf(me) === Person.prototype; // true
  Object.getPrototypeOf(Person.prototype) === Object.prototype; // ture
  ```
- 프로토타입의 프로토타입은 언제나 Object.prototype
-

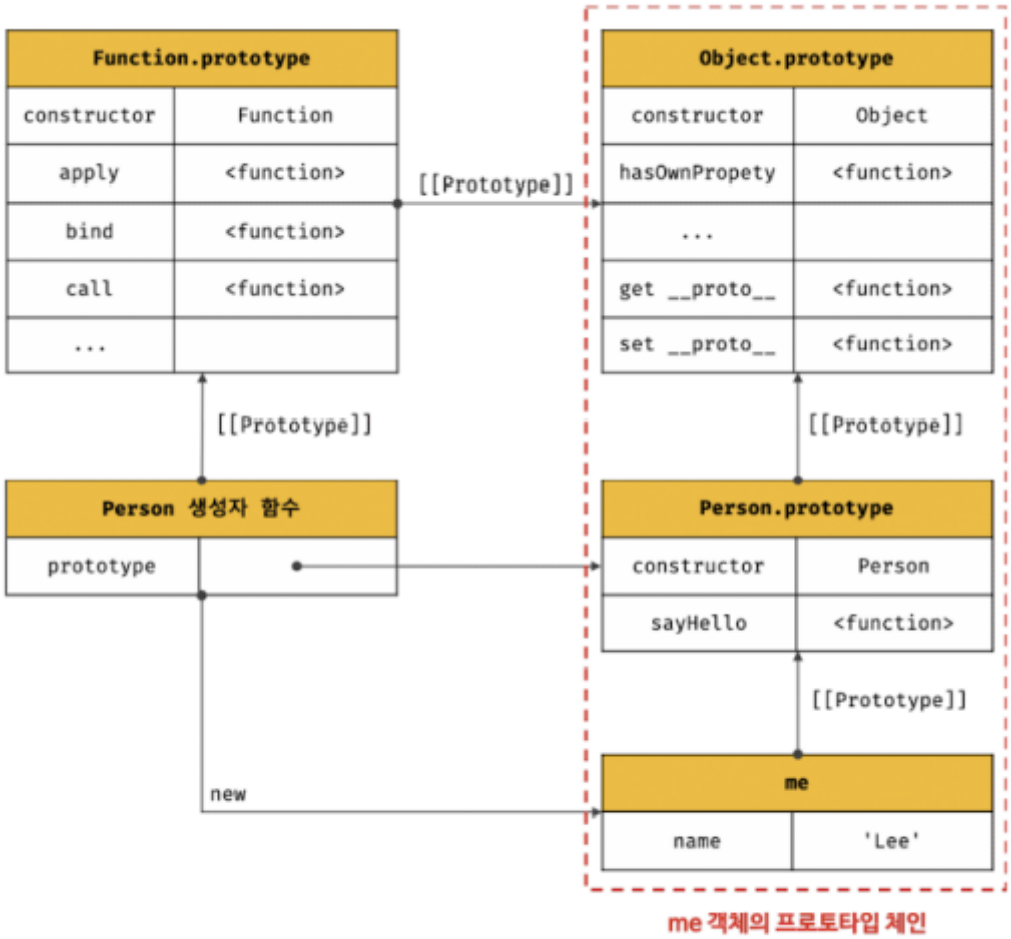


그림 19-18 프로토타입 체인

- 자바스크립트는 객체의 프로퍼티에 접근하려고 할 때 해당 객체에 접근하려는 프로퍼티가 없다면 **[[Prototype]] 내부 슬롯의 참조를 따라 자신의 부모 역할을 하는 프로토타입의 프로퍼티를 순차적으로 검색 : 프로토타입 체인**

- 프로토타입 체인은 자바스크립트가 객체지향 프로그래밍의 상속을 구현하는 메커니즘
- ```
  me.hasOwnProperty('name');
  ```

  1. 먼저 hasOwnProperty 메서드를 호출한 me 객체에서 hasOwnProperty 메서드를 검색
  2. 없을 경우, 프로토타입 체인을 따라, [[Prototype]] 내부 슬롯에 바인딩되어 있는 프로토타입으로 이동해 hasOwnProperty 메서드를 검색
  3. Person.prototype에도 hasOwnProperty가 없으므로 프로토타입 체인을 따라 Object.prototype으로 이동해 hasOwnProperty 메서드를 검색
  4. Object.prototype에는 hasOwnProperty 메서드 존재
  5. 이 때, Object.prototype.hasOwnProperty 메서드의 this에는 me 객체 바인딩(this로 사용할 me 객체를 전달하며 Object.prototype.hasOwnProperty 메서드를 호출한다고 이해한다)
- 프로토타입 체인의 최상단에 위치하는 객체는 언제나 Object.prototype
- 따라서, 모든 객체는 Object.prototype을 상속받는다.
- **Object.prototype을 프로토타입 체인의 종점**이라 한다.
- Object.prototype의 프로토타입, 즉 [[Prototype]]의 내부 슬롯 값은 null
- 프로토타입 체인 종점인 Object.prototype에서도 프로퍼티를 검색할 수 없는 경우 undefined 반환
- **프로토타입 체인은 상속과 프로퍼티 검색을 위한 메커니즘**
- 스코프 체인과 프로토타입 체인은 서로 협력하여 식별자와 프로퍼티를 검색하는데 사용된다.

오버라이딩과 프로퍼티 섀도잉

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

    Person.prototype.sayHello = function () {
      console.log(`${this.name} 프로토타입`);
    };

    return Person;
  })();

  const me = new Person("Lee");

  me.sayHello = function () {
    console.log(`${this.name} 인스턴스`);
  };

  me.sayHello(); // Lee 인스턴스
  ```
- 프로토타입이 소유한 프로퍼티를 프로토타입 프로퍼티 / 인스턴스가 소유한 프로퍼티를 인스턴스 프로퍼티
- 프로토타입 프로퍼티와 동일한 프로퍼티를 인스턴스에 추가하면 프로토타입 프로퍼티를 덮어쓰는 게 아니라 인스턴스 프로퍼티로 추가
- 인스턴스 메서드 sayHello는 프로토타입 메서드 sayHello를 오버라이딩
- 프로토타입 메서드 sayHello는 가려짐 = 프로퍼티 섀도잉
- 프로퍼티를 삭제하는 경우도 마찬가지

    - 인스턴스 메서드 sayHello를 삭제할 경우 프로토타입 메서드가 아닌 인스턴스 메서드가 삭제
    - 하위 객체를 통해 프로토타입의 프로퍼티를 변경 또는 삭제하는 것은 불가능
- 하위 객체를 통해 프로토타입에 get 액세스는 허용되나 set 액세스는 허용되지 않는다.

프로토타입의 교체

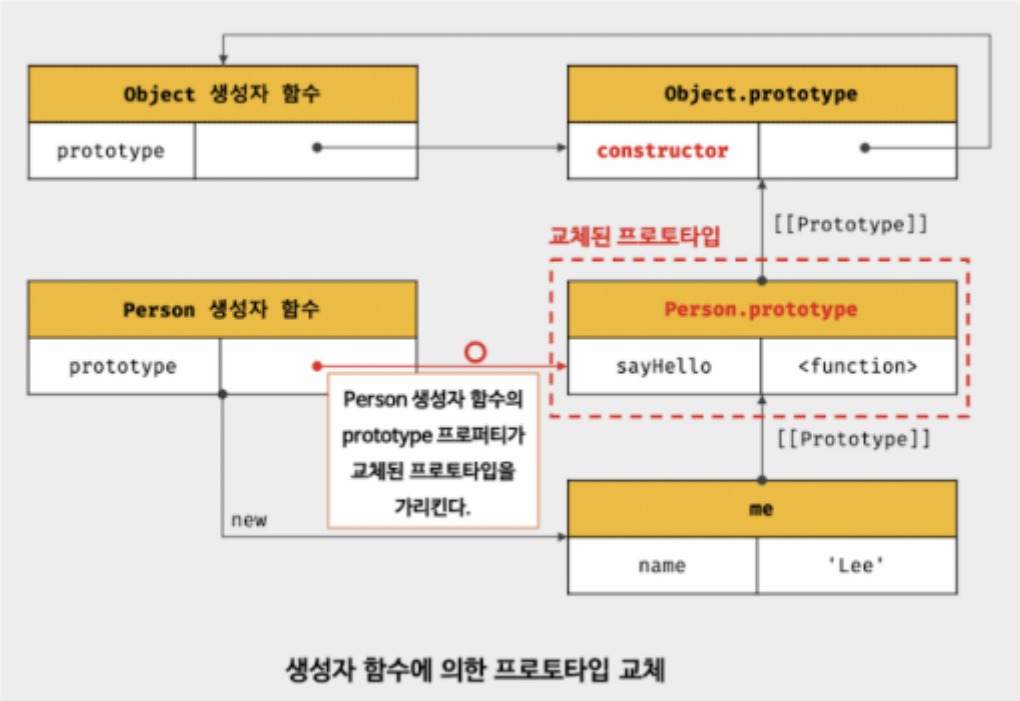
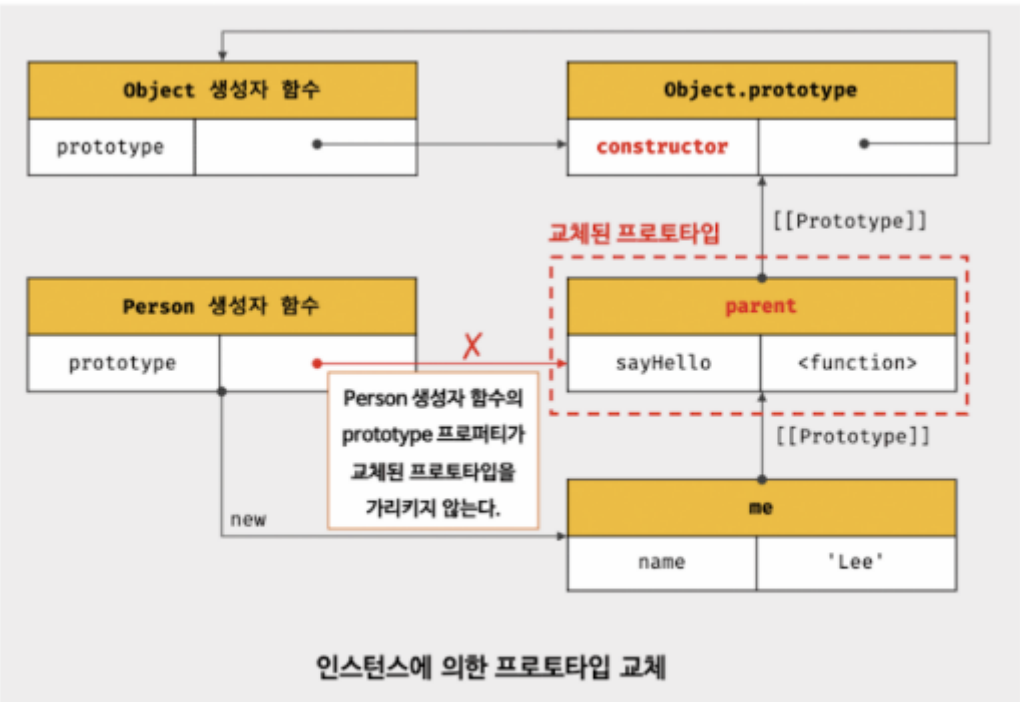
- 프로토타입은 임의의 다른 객체로 변경 가능 = 부모 객체인 프로토타입을 동적으로 변경 가능하다.
- 객체 간의 상속 관계 동적 변경 가능
- 생성자 함수에 의한 프로토타입의 교체

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

    // Person.prototype에 객체 리터럴 할당
    // 객체 리터럴에는 constructor 프로퍼티가 없음
    // 따라서 me 객체의 생성자 함수를 검색하면 Person이 아닌 Object가 나온다.
    Person.prototype = {
        sayHello() {
            console.log('안녕')
        }
    };

    return Person;
}());

const me = new Person('LEE');

console.log(me.constructor === Person); // false
console.log(me.constructor === Object); // true
// 프로토타입의 교체로 constructor 프로퍼티와 생성자 함수 간의 연결이 파괴


//// ------------------ constructor 프로퍼티 되살리기 ------------------
////
const Person(function () {
    function Person(name) {
        this.name = name;
    }

    // Person.prototype에 객체 리터럴 할당
    // 객체 리터럴에는 constructor 프로퍼티가 없음
    // 따라서 me 객체의 생성자 함수를 검색하면 Person이 아닌 Object가 나온다.
    Person.prototype = {
        constructor: Person,
        sayHello() {
            console.log('안녕')
        }
    };

    return Person;
}());

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

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

    - 생성자 함수의 prototype 프로퍼티에 다른 임의의 객체를 바인딩하는 것은 미래에 생성할 인스턴스의 프로토타입을 교체하는 것

- 인스턴스에 의한 프로토타입의 교체
    - 프로토타입은 생성자 함수의 prototype 프로퍼티 뿐만 아니라 인스턴스의 __ proto __ 접근자 프로퍼티를 통해 접근 가능(Object.getPrototypeOf도 가능)
    - 따라서, __ proto __ 또는 Object.setPrototypeOf를 통해 프로토타입 교체 가능
    - __ proto __ 접근자 프로퍼티를 통해 프로토타입을 교체하는 것은 이미 생성된 객체의 프로토타입을 교체하는 것
    - ```
      function Person(name) {
          this.name = name;
      }

      const me = new Person('Lee');

      const parent = {
          sayHello() {
              console.log('안녕');
          }
      };

      Object.setPrototypeOf(me, parent);
      // me.__proto__ = parent; (위 아래 코드 동일하게 동작함)

      me.sayHello(); // 안녕

      // 생성자 함수에 의한 프로토타입의 교체와 마찬가지로
      // 프로토타입으로 교체한 객체에 constructor 프로퍼티가 없으므로 constructor 프로퍼티와 생성자 함수 간의 연결이 파괴됨
      console.log(me.constructor === Person); // false
      console.log(me.constructor === Object); // true
      ```

- 생성자 함수에 의한 프로토타입 교체 vs 인스턴스에 의한 프로토타입 교체
      ```





그림 19–22 프로토타입 교체 방식에 의해 발생하는 차이

- 프로토타입은 직접 교체하지 않는 것이 좋다.

instanceof

- 이항 연산자
- 좌변에 객체를 가리키는 식별자 / 우변에 생성자 함수를 가리키는 식별자
- 우변의 생성자 함수의 prototype에 바인딩된 객체가 좌변의 객체의 프로토타입 체인 상에 존재하면 true로 평가 / 그렇지 않을 경우 false로 평가
- ```
  function Person(name) {
      this.name = name;
  }
  ```
  ```

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

const parent = {};

Object.setPrototypeOf(me, parent);

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

// me 객체는 프로토타입이 교체되어 프로토타입과 생성자 함수 간의 연결이 파괴되었다.
// Person.prototype이 me 객체의 프로토타입 체인 상에 존재하지 않는다.
```

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

  const me = new Person('Lee');

  const parent = {};

  Object.setPrototypeOf(me, parent);

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

  Person.prototype = parent;

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

- instanceof 연산자는 프로토타입의 constructor 프로퍼티가 가리키는 생성자 함수를 찾는 것이 아니라
- 생성자 함수의 prototype에 바인딩된 객체가 프로토타입 체인 상에 존재하는지 확인
- ```
  const Person = (function(){
      function Person(name) {
          this.name = name;
      }

      Person.prototype = {
          sayHello() {
              console.log('안녕');
          }
      };
  }());

  const me = new Person('Lee');

  const parent = {};

  Object.setPrototypeOf(me, parent);

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

직접 상속

- Object.create에 의한 직접 상속
  - 명시적으로 프로토타입을 지정하여 새로운 객체를 생성
  - OrdinaryObjectCreate를 호출
  - 첫번째 매개변수에는 생성할 객체의 프로토타입으로 지정할 객체를 전달
  - 두번째 매개변수에는 생성할 객체의 프로퍼티 키와 프로퍼티 디스크립터 객체로 이뤄진 객체를 전달(옵션)
  - new 연산자 없이도 객체 생성 가능
  - 프로토타입을 지정하면서 객체 생성 가능
  - 객체 리터럴에 의해 생성된 객체도 상속 가능
- 객체 리터럴 내부에서 __proto__ 에 의한 직접 상속
  - 일단 객체를 생성한 이후 프로퍼티를 추가하는 방법

정적 프로퍼티 / 메서드

- 생성자 함수로 인스턴스를 생성하지 않아도 참조/호출할 수 있는 프로퍼티/메서드
- 생성자 함수는 객체이므로 자신의 프로퍼티/메서드를 소유할 수 있고, 생성자 함수 객체가 소유한 프로퍼티/메서드를 정적 프로퍼티/메서드 라고 한다.
- 정적 프로퍼티/메서드는 생성자 함수가 생성한 인스턴스로 참조/호출 불가능

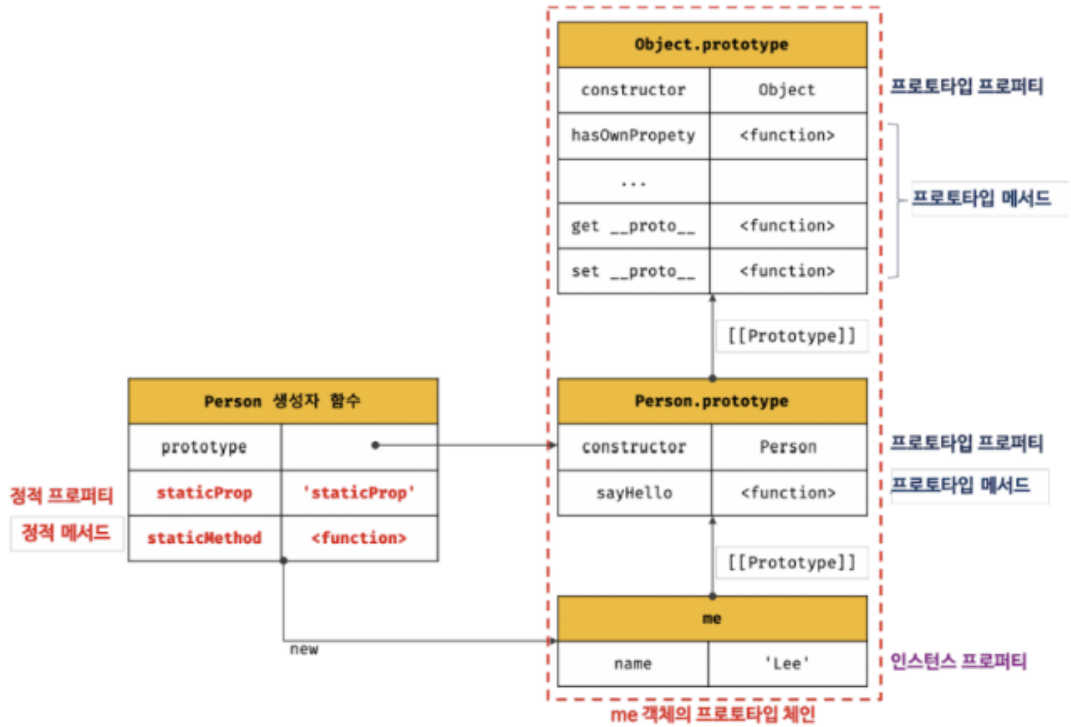


그림 19-24 정적 프로퍼티/메서드

- Object.create 메서드는 Object 생성자 함수의 정적 메서드 → Object가 생성한 객체(인스턴스)가 호출할 수 없다.
- Object.prototype.hasOwnProperty 메서드는 Object.prototype의 메서드 → 모든 객체가 호출 가능(모든 객체 프로토타입 체인의 종점)

프로퍼티 존재 확인

- in 연산자
  - 객체 내에 특정 프로퍼티가 존재하는지 여부를 확인
  - `key in object`의 형태
  - in 연산자는 확인 대상 객체의 프로퍼티뿐만 아니라 상속받은 모든 프로토타입의 프로퍼티를 확인하므로 주의가 필요
  - `Reflect.has`와 동일
  - ```
    const person = {
        name: 'Lee',
    }

    console.log('name' in person); // true
    console.log('toString' in person); // true
    ```
- Object.prototype.hasOwnProperty 메서드
  - 상속받은 프로토타입의 프로퍼티 키인 경우 false 반환

프로퍼티 열거

- for ... in 문
  - 객체의 모든 프로퍼티를 순회하며 열거
  - `for (변수 선언문 in 객체) {...}`
  - in 연산자와 마찬가지로 상속받은 프로토타입의 프로퍼티까지 열거
  - 하지만 toString과 같은 메서드가 열거되지 않는 이유는 toString 메서드가 열거할 수 없도록 정의되어 있는 프로퍼티이기 때문([[Enumerable]] 값이 false이기 때문)
  - 다시 말해, for ...in 문은 객체의 프로토타입 체인 상에 존재하는 모든 프로토타입의 프로퍼티 중에서 프로퍼티 어트리뷰트 [[Enumerable]]의 값이 true인 프로퍼티를 순회하며 열거한다.
  - 프로퍼티를 열거할 때 순서를 보장하지 않음(대부분은 보장하긴 한다.)
  - ```
    const person = {
        name: 'Lee',
        address: 'Seoul',
    }

    for (const key in person) {
        console.log(key, person[key]);
    }

    // name Lee
    // address Seoul
    ```
- Object.keys/values/entries 메서드
  - for ... in 문은 객체 자신의 고유 프로퍼티 뿐 아니라 상속받은 프로퍼티도 열거
  - Object.keys/values/entries 메서드는 객체 자신의 고유 프로퍼티만 열거
  - Object.keys 메서드
    - 객체 자신의 열거 가능한 프로퍼티 키를 배열로 반환

    - Object.values 메서드
        - 객체 자신의 열거 가능한 프로퍼티 값을 배열로 반환
    - Object.entries 메서드
        - 객체 자신의 열거 가능한 프로퍼티 키와 값의 쌍의 배열을 배열에 담아 반환

내부 슬롯과 내부 메서드

- 내부 슬롯과 내부 메서드는 자바스크립트 엔진의 구현 알고리즘을 설명하기 위해 ECMA Script 사양에서 사용하는 의사 프로퍼티와 의사 메서드
- 이중 대괄호(`[[ ]]`)로 감싼 이름들
- 자바스크립트의 내부 로직이므로 원칙적으로 직접 접근하거나 호출할 수 있는 방법을 제공하지는 않는다.
- 단, 일부 내부 슬롯과 내부 메서드에 한하여 간접적으로 접근할 수 있는 수단을 제공(ex. [[Prototype]])