(번외) 프로토타입과 프로토타입 상속

- __proto__ 게터나 Object.getPrototypeOf 메서드를 통해서 **숨겨진 [[Prototype]] 객체**에 접근 가능

```
let o = {};
// __proto__ 게터를 통해서 [[Prototype]] 객체에 접근 가능
console.log(o.__proto__);
// Object.getPrototypeOf 메서드를 통해서 [[Prototype]] 객체에 접근 가능
console.log(Object.getPrototypeOf(o));
// 게터를 통해서 접근하나 메서드를 통해서 접근하나 완전히 같은 객체에 접근 가능
console.log(o.__proto__ === Object.getPrototypeOf(o));
```

- 객체 속성이나 메서드에 접근 시 만약 해당 객체에서 속성, 메서드를 발견하지 못하면 **객체의 [[Prototype]] 객체에 접근 후 해당 속성, 메서드가 있는지 검색**하여 활용함
  - 이러한 검색을 **프로토타입 체인을 따라서 검색**한다고 표현하기도 함

```
let animal = {
    isAnimal: true,
    eat: function(food) {
        console.log(`${food}를 먹습니다.`);
    }
};

let dog = {
    bark: function() {
        console.log("멍멍");
    }
};

let cat = {
    meow: function() {
        console.log("냥냥");
    }
};

// __proto__ 세터를 통해서 [[Prototype]] 객체 지정
dog.__proto__ = animal;
// Object.setPrototypeOf 메소드를 통해 [[Prototype]] 객체 지정
// Object.setPrototypeOf(프로토타입을 지정할 객체, 프로토타입으로 사용될 객체);
Object.setPrototypeOf(cat, animal);

// 고유의 메서드에는 당연히 접근 가능
dog.bark();
cat.meow();

// [[Prototype]] 객체를 통해 접근하여 isAnimal 속성 접근 가능
console.log(dog.isAnimal);
console.log(cat.isAnimal);

// [[Prototype]] 객체를 통해 접근하여 eat 메서드 접근 가능
dog.eat('사료');
cat.eat('사료');
```

```
// 해당 객체에도 없고, [[Prototype]] 객체에도 존재하지 않는 속성에 접근하면 undefined 값 반환
console.log(dog.asdf);
// 해당 객체에도 없고, [[Prototype]] 객체에도 존재하지 않는 메서드를 호출하려하면 TypeError 발생
// dog.hello();
```

- 프로토타입 체인은 원하는 만큼 길어질 수 있음 (=> 상속 관계가 더 깊어질 수 있음)

```
let animal = {
    eat: function(food) { console.log(`${food}를 먹습니다.`); }
};
let human = {
    talk: function() { console.log("대화를 합니다."); }
};
let superhuman = {
    fly: function() { console.log("날아다닙니다."); }
};

// animal => human => superhuman 의 프로토타입 체인 형성
human.__proto__ = animal;
superhuman.__proto__ = human;

// superhuman의 fly 메서드 접근
superhuman.fly();
// superhuman의 프로토타입 객체인 human의 talk 메서드 접근
superhuman.talk();
// superhuman의 프로토타입 객체인 human의 프로토타입 객체인 animal의 eat 메서드 접근
superhuman.eat("밥");
```

- 프로토타입 체인을 검색하는 과정에서 **가장 먼저 만나게 된 속성, 메서드에 접근**함
    - 자바의 **메서드 오버라이드**와 비슷한 방식이라고 생각해도 무방함

```
let animal = {
    eat: function(food) { console.log(`${food}를 먹습니다.`); }
};
let human = {
    eat: function(food) { console.log(`${food}를 즐겁게 먹습니다.`); },
    talk: function() { console.log("대화를 합니다."); }
};
let superhuman = {
    eat: function(food) { console.log(`${food}를 흡수하여 먹습니다.`); },
    talk: function() { console.log("텔레파시를 통해서 대화합니다."); },
    fly: function() { console.log("날아다닙니다."); }
};

human.__proto__ = animal;
superhuman.__proto__ = human;

// 객체에 해당 메서드(eat)가 이미 정의되어 있으므로, 프로토타입 체인 및 객체(animal) 접근이 불필요
human.eat("밥"); // 밥를 즐겁게 먹습니다.

// 객체에 해당 메서드(eat, talk)가 이미 정의되어 있으므로, 프로토타입 체인 및 객체(human) 접근이 불필요
```

```
superhuman.eat("밥"); // 밥를 흡수하여 먹습니다.
superhuman.talk(); // 텔레파시를 통해서 대화합니다.
```

- 모든 함수는 정의가 되자마자 **자동으로 prototype 속성을 갖게됨**
  - 단, 화살표 함수는 예외적으로 prototype 속성을 가지지 않음
  - 보통 프로토타입 객체와 연관된 작업을 수행할 함수는 **함수선언식을 통해서 정의**함
  - (주의) 함수의 __proto__ 와 prototype은 다른 객체임을 유의
- prototype 속성에 저장된 값은 **객체**이며 해당 객체는 **constructor라는 속성**을 가지고 있음
- constructor 속성에는 자기 자신을 가리키는 함수가 저장됨

```
// 함수 선언식으로 함수를 정의하나
function hello() {};
// 함수 표현식 정의하나 모두 prototype 속성을 갖게 됨
let world = function() {};

console.log(hello.prototype); // { constructor: hello }
console.log(hello.prototype.constructor === hello); // true
console.log(world.prototype); // { constructor: world }
console.log(world.prototype.constructor === world); // true
```

- 일반적으로 **프로토타입 객체와 연관된 작업을 수행하는 함수는 생성자 함수**임
- **new 키워드**와 함께 함수를 호출하면 this와 관련된 작업(this에 새 객체 할당 및 자동 this 반환)을 수행함 (이미 배운 사실)
- 여기에 추가로 **반환될 객체의 [[Prototype]] 객체를 해당 객체를 생성할 때 사용할 생성자 함수의 prototype으로 연결해 주는 작업도 수행**함

```
function Animal(name, age) {
    // new 키워드와 함께 생성자 함수가 호출될 경우, 마치 아래 코드와 같은 작업을 자동으로 해주게 됨
    // Object.setPrototypeOf(this, Animal.prototype);

    this.name = name;
    this.age = age;
};

let sam = new Animal("Sam", 2);
console.log(sam.__proto__); // { constructor: Animal }
console.log(sam.__proto__ === Animal.prototype); // true
console.log(sam.__proto__.constructor === Animal); // true

// sam.__proto__.constructor는 Animal 생성자 함수를 가리키므로 다음과 같이 활용 가능 (권장하는 것은 아님!)
let boksun = new sam.__proto__.constructor("Boksun", 1);
console.log(boksun.__proto__ === Animal.prototype); // true
```

- 생성자 함수를 통해 생성된 객체가 생성자 함수의 prototype 속성에 저장된 객체를 [[Prototype]] 객체로 사용하므로 해당 객체에 **필요한 속성과 메서드를 정의 가능**
  - 거의 대부분의 경우 **prototype 객체에는 공용으로 사용할 메서드를 추가**함

```
// Animal 생성자 함수를 통해서 생성될 객체의 __proto__는 Animal.prototype
function Animal(name, age) {
    this.name = name;
    this.age = age;
```

```
};

// 공통적으로 사용할 속성 혹은 메서드 정의 (보통은 속성보다는 메서드만 정의함)
Animal.prototype.isAnimal = true;
Animal.prototype.eat = function(food) {
    console.log(`${food}를 먹습니다.`);
};

let sam = new Animal("Sam", 2);
let boksun = new Animal("Boksun", 1);

// 프로토타입 체인을 따라가 Animal.prototype에 정의된 속성과 메서드에 접근하게 됨
console.log(sam.isAnimal); // true
console.log(boksun.isAnimal); // true
sam.eat("사료"); // 사료를 먹습니다.
boksun.eat("고급 사료"); // 고급 사료를 먹습니다.

// 객체에 직접 메서드 정의
boksun.eat = function(food) {
    console.log(`${food}를 복스럽게 먹습니다.`);
};
// 이후에는 해당 객체(boksun)에서 직접적으로 메서드(eat)에 접근할 수 있으므로, 프로토타입 체인을 검색하지 않고 프로토타입 객체(Animal.prototype)에도 접근하지 않음
boksun.eat("고급 사료"); // 고급 사료를 복스럽게 먹습니다.
```

- prototype 객체에 공통 메서드를 정의하더라도 메서드 내부의 **this의 값은 해당 메서드를 호출하는 객체(점(.) 앞의 객체)를 가리키므로** 문제 없이 사용 가능

```
function Animal(name, age) {
    this.name = name;
    this.age = age;
};

Animal.prototype.eat = function(food) {
    console.log(`${this.name}이 ${food}를 먹습니다.`);
};

let sam = new Animal("Sam", 2);
let boksun = new Animal("Boksun", 1);

// eat 메서드 내부에서 this는 sam
sam.eat("사료"); // Sam이 사료를 먹습니다.
// eat 메서드 내부에서 this는 boksun
boksun.eat("고급 사료"); // Boksun이 고급 사료를 먹습니다.
```

- 생성자 함수와 프로토타입을 이용한 상속(=프로토타입 기반 상속, prototypal inheritance) 구현

Object.create 메서드 공부

```
let animal = {
    eat: function(food) { console.log(`${food}를 먹습니다.`); }
};
// Object.create 메서드를 호출하면 호출 시 전달한 객체를 [[Prototype]] 객체로 사용하는 빈 객체를 생성하여 반환함
let human = Object.create(animal);
// 비어있는 객체
```

```
console.log(human); // { }
// 그러나 __proto__는 Object.create 메서드로 전달한 객체로 설정되어있는 것을 확인 가능
console.log(human.__proto__); // { eat: f }
human.talk = function() { console.log("대화를 합니다."); }
// 이후 human 객체의 모습은 다음과 같음
// { talk: f, __proto__: { eat: f } }

// 객체에 eat 메서드가 없으므로 프로토타입 체인 검색 후 프로토타입 객체(animal)에 접근하여 eat 메서드 호출
human.eat("밥"); // 밥을 먹습니다.
// 객체에 정의한 고유 메서드이므로 프로토타입 체인 검색 없이 바로 호출
human.talk(); // 대화를 합니다.
```

프로토타입과 생성자 함수를 이용한 상속 구현 (가장 중요한 예제)

```
// Shape - 부모 클래스(superclass)
function Shape(x, y) {
    this.x = x;
    this.y = y;
}

// 부모 클래스의 메서드 정의
Shape.prototype.move = function(x, y) {
    console.log(`Shape moved from ${this.x},${this.y} to ${x},${y}`);
    this.x += x;
    this.y += y;
}

// Rectangle - 자식 클래스(subclass)
function Rectangle(x, y, w, h) {
    // 부모 생성자 함수 호출
    // 자바의 super와 비슷한 기능이라고 생각해도 무방 (new 키워드를 이용해서 호출하지 않았음을 유의!)
    Shape.call(this, x, y);
    this.w = w;
    this.h = h;
}

// Object.create 메서드를 호출하여 Shape.prototype을 프로토타입 객체로 사용하는 비어있는 객체를 생성 후 Rectangle 생성자 함수의 prototype 속성 지정
// Rectangle.prototype에 대입될 객체의 모습 => { __proto__: Shape.prototype }
Rectangle.prototype = Object.create(Shape.prototype);
// Rectangle.prototype 객체의 constructor가 대입으로 인하여 사라졌으므로 다시 설정
// (엄밀히 말하면 사라졌다기 보다는, constructor 속성에 접근할 경우 constructor 속성을 Rectangle.prototype 객체 내부에서 찾을 수 없으므로 프로토타입 체인을 검색하여 만나게 된 Shape.prototype.constructor(=Shape)를 constructor로 사용하게 됨)
Rectangle.prototype.constructor = Rectangle;
// Rectangle 생성자 함수를 통해서 생성된 객체가 사용할 공통 메서드 정의
Rectangle.prototype.size = function() {
    return this.w * this.h;
}

let rect = new Rectangle(0, 0, 20, 10);
// Shape.prototype 접근을 통해서 move 메서드를 호출
console.log(rect.move(5, 5));
// rect 객체에 size 메서드가 없으므로 프로토타입 체인을 검색하여 프로토타입 객체인 Rectangle.prototype에 정의된 size 메서드를 호출
```

```
console.log(rect.size());
```

Q) 위와 비슷한 방식으로 Shape.prototype 객체를 프로토타입 체인에서 검색할 수 있는 Circle 생성자 함수 정의하고 고유의 area 메서드 정의하기
공식은 π*r^2

- 기본적으로 "빈 객체"를 생성하면 **Object 생성자 함수의 prototype 객체를 프로토타입 객체로 사용**함

```
// let o = {};
// 위의 코드와 동일한 작업 수행
let o = new Object();

// Object 생성자 함수를 이용해서 생성한 객체이므로 당연히 o의 __proto__는 Object.prototype을 가리킴
console.log(o.__proto__ === Object.prototype); // true
// toString과 같은 정의되지 않은 메서드를 쓸 수 있는 것도 프로토타입 체인을 거슬러 올라가 Object.prototype에 정의된 메서드를 호출하기 때문임
console.log(o.toString());
// Object.prototype이 프로토타입 체인이 검색할 마지막 프로토타입 객체이므로 상위의 프로토타입 객체이 접근시 null이 반환됨
console.log(o.__proto__.__proto__); // null
// 바꾸어 말하자면 Object.prototype에 저장된 객체에서 접근할 프로토타입 객체는 존재하지 않음
console.log(Object.prototype.__proto__); // null
```

- 앞서 살펴본 Shape, Rectangle 예제 다시 살펴보기

```
console.log(Shape.prototype); // { move: f, constructor: Shape }
// 최상위 생성자 함수(Shape)의 prototype 객체의 프로토타입 객체는 Object.prototype임을 유의
console.log(Shape.prototype.__proto__);
// 따라서 가장 최상위 생성자 함수의 prototype 객체에 지정된 프로토타입 객체는 Object 생성자 함수의 prototype 객체가 됨
console.log(Shape.prototype.__proto__ === Object.prototype);
// (중요) 바꾸어 말하자면 어떤 생성자 함수를 통해서 만들어진 객체건 프로토타입 체인을 거슬러 올라가면 가장 마지막으로 Object.prototype 객체에 접근할 수 있다는 말이 됨
```