생성자 함수에 의한 객체 생성

모던 자바스크립트 17장

17.1 Object 생성자 함수

생성자 함수란?

new 연산자와 함께 호출하여 객체(인스턴스)를 생성하는 함수

```
// String 생성자 함수에 의한 String 객체 생성
const strObj = new String('Lee');
console.log(typeof strObj); // object
console.log(strObj);          // String {"Lee"}
```

17.2 생성자 함수

17.2.1 객체 리터럴에 의한 객체 생성 방식 문제점/ 17.2.2 생성자 함수에 의한 객체 생성 방식 장점

객체 리터럴

중괄호{ }를 이용하여 표현

직관적이고 간편

but

단 하나의 객체만 생성

【 예제 17-03 】

```
const circle1 = {
  radius: 5,
  getDiameter() {
    return 2 * this.radius;
  }
};

console.log(circle1.getDiameter()); // 10

const circle2 = {
  radius: 10,
  getDiameter() {
    return 2 * this.radius;
  }
};

console.log(circle2.getDiameter()); // 20
```

VS

생성자 함수

형식 지정 x

new 연산자와 함께 호출하면

생성자 함수로 동작

프로퍼티 구조 동일한 객체

간편하게 여러 개 생성 가능

【 예제 17-04 】

```
// 생성자 함수
function Circle(radius) {
  // 생성자 함수 내부의 this는 생성자 함수가 생성할 인스턴스를 가리킨다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}

// 인스턴스의 생성
const circle1 = new Circle(5);  // 반지름이 5인 Circle 객체를 생성
const circle2 = new Circle(10); // 반지름이 10인 Circle 객체를 생성

console.log(circle1.getDiameter()); // 10
console.log(circle2.getDiameter()); // 20
```

17.2 생성자 함수

17.2.3 생성자 함수의 인스턴스 생성 과정

자바스크립트 엔진은 3가지 과정을 거쳐 암묵적으로 인스턴스를 생성하고 초기화한 후 반환

1) 인스턴스 생성과 this 바인딩

2) 인스턴스 초기화

3) 인스턴스 반환

17.2 생성자 함수

17.2.3 생성자 함수의 인스턴스 생성 과정

1) 인스턴스 생성과 this 바인딩

변수가 객체를 가리키는 것

【 예제 17-08 】

```
function Circle(radius) {
  // 1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.
  console.log(this); // Circle {}

  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}
```

17.2 생성자 함수

17.2.3 생성자 함수의 인스턴스 생성 과정

2) 인스턴스 초기화

매개변수를 통해 사용자가 전달한 값을 이용하여 객체 초기 상태 설정

【 예제 17-09 】

```
function Circle(radius) {
  // 1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.

  // 2. this에 바인딩되어 있는 인스턴스를 초기화한다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}
```

17.2 생성자 함수

17.2.3 생성자 함수의 인스턴스 생성 과정

1) 인스턴스 반환

완성된 인스턴스가 바인딩된 this가 암묵적으로 반환됨

(return문이 없을 때 해당)

【 예제 17-10 】

```
function Circle(radius) {
  // 1. 암묵적으로 빈 객체가 생성되고 this에 바인딩된다.

  // 2. this에 바인딩되어 있는 인스턴스를 초기화한다.
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };

  // 3. 완성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.
}

// 인스턴스 생성. Circle 생성자 함수는 암묵적으로 this를 반환한다.
const circle = new Circle(1);
console.log(circle); // Circle {radius: 1, getDiameter: f}
```

this가 아닌 다른 객체를 명시적으로 반환한다면?

```
  // 3. 암묵적으로 this를 반환한다.
  // 명시적으로 객체를 반환하면 암묵적인 this 반환이 무시된다.
  return {};
}

// 인스턴스 생성. Circle 생성자 함수는 명시적으로 반환한 객체를 반환한다.
const circle = new Circle(1);
console.log(circle); // {}
```

명시적으로 원시 값을 반환한다면?

```
  // 3. 암묵적으로 this를 반환한다.
  // 명시적으로 원시 값을 반환하면 원시 값 반환은 무시되고 암묵적으로 this가 반환된다.
  return 100;
}

// 인스턴스 생성. Circle 생성자 함수는 명시적으로 반환한 객체를 반환한다.
const circle = new Circle(1);
console.log(circle); // Circle {radius: 1, getDiameter: f}
```

17.2 생성자 함수

17.2.4 내부 메서드 [[Call]]와 [[Construct]]

함수는 일반 객체와 동일하게 동작할 수 있음
하지만 일반 객체와는 다름

일반 객체: 호출 X
함수: 호출 O

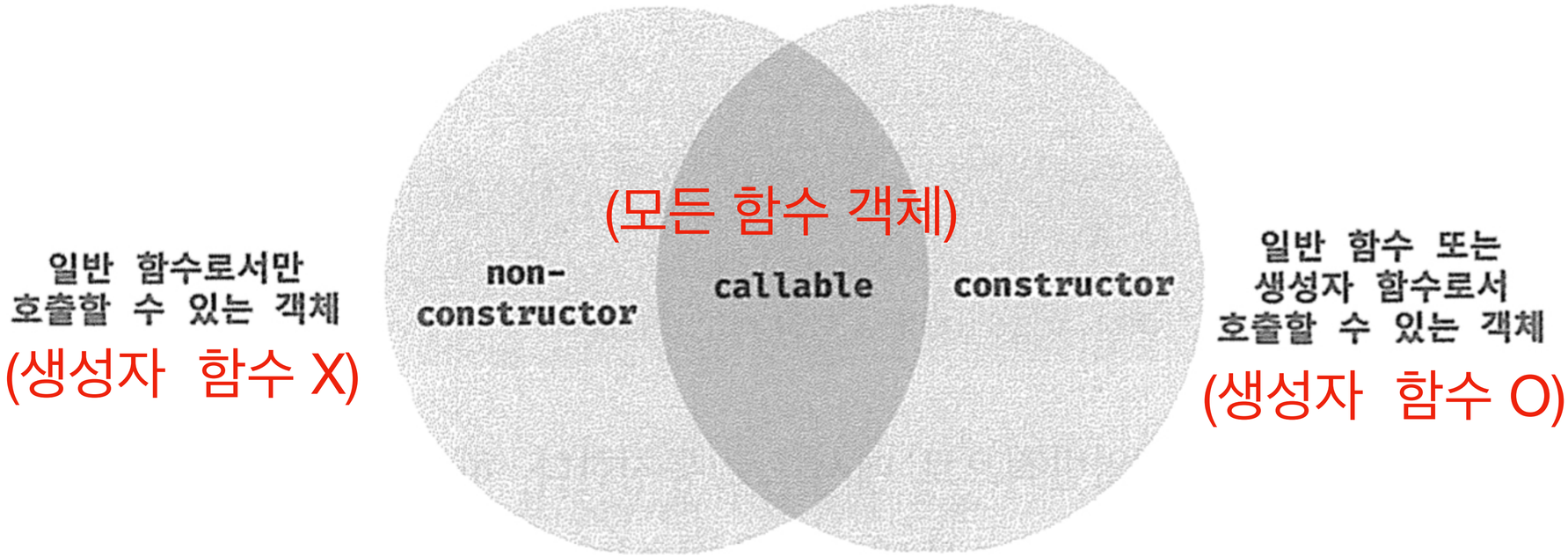

그림 17-1 모든 함수 객체는 callable이지만 모든 함수 객체가 constructor인 것은 아니다.

17.2 생성자 함수

17.2.5 constructor와 non-constructor의 구분

함수 정의 방식에 따라서

constructor와 non-constructor로 구분

〖 예제 17-15 〗

function 키워드로 시작하며 이름이 지정됨

변수에 함수를 할당

```
// 일반 함수 정의: 함수 선언문, 함수 표현식
function foo() {}
const bar = function () {};
// 프로퍼티 x의 값으로 할당된 것은 일반 함수로 정의된 함수다. 이는 메서드로 인정하지 않는다.
const baz = {
  x: function () {}
};

// 일반 함수로 정의된 함수만이 constructor다.
new foo(); // → foo {}
new bar(); // → bar {}
```

특정한 동작을 수행하지 않기 때문.. 단순 속성값으로 사용됨

17.2 생성자 함수

17.2.5 constructor와 non-constructor의 구분

함수 정의 방식에 따라서
constructor와 non-constructor로 구분

```
// 화살표 함수 정의
const arrow = () => {};

new arrow(); // TypeError: arrow is not a constructor

// 메서드 정의: ES6의 메서드 축약 표현만 메서드로 인정한다.
const obj = {
  x() {}
};

new obj.x(); // TypeError: obj.x is not a constructor
```

함수 선언문, 표현식으로 정의된 함수만 construtor
화살표 함수, 메서드 축약 표현으로 정의된 함수는 non-constructor

non-constructor인 함수 객체를 생성자 함수로 호출 시 에러 발생

17.2 생성자 함수

17.2.6 new 연산자

new 연산자와 함께 함수 호출 => 생성자 함수로 작동

함수 객체 내부 메서드 [[constructor]] 호출

반대로

new 연산자 없이 생성자 함수 호출 => 일반 함수로 호출됨

일반 함수와 생성자 함수에 특별한 형식적 차이 X

따라서 생성자 함수는 파스칼 케이스로 명명하여 일반 함수와 구별할 수 있도록 해야 함

첫 문자를 대문자로 기술하는 것

17.2 생성자 함수

17.2.7 new.target

new.target(ES6에서 지원)

- 생성자 함수가 new 연산자 없이 호출되는 것을 방지함

  일반 함수로서 호출된 함수 내부의 new.target => undefined
- 메타 프로퍼티: 함수 내부에서 암묵적인 지역 변수와 같이 사용되는 것
- 함수 내부의 new.target은 함수 자신을 가리킴

【 예제 17-19 】

```
// 생성자 함수
function Circle(radius) {
  // 이 함수가 new 연산자와 함께 호출되지 않았다면 new.target은 undefined다.
  if (!new.target) {
    // new 연산자와 함께 생성자 함수를 재귀 호출하여 생성된 인스턴스를 반환한다.
    return new Circle(radius);
  }

  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}

// new 연산자 없이 생성자 함수를 호출하여도 new.target을 통해 생성자 함수로서 호출된다.
const circle = Circle(5);
console.log(circle.getDiameter());
```

스코프 세이프 생성자 패턴scope-safe constructor

new.target은 ES6에서 도입된 최신 문법으로 IE에서는 지원하지 않는다. new.target을 사용할 수 없는 상황이라면 스코프 세이프 생성자 패턴을 사용할 수 있다.

【 예제 17-20 】

```
// Scope-Safe Constructor Pattern
function Circle(radius) {
  // 생성자 함수가 new 연산자와 함께 호출되면 함수의 선두에서 빈 객체를 생성하고
  // this에 바인딩한다. 이때 this와 Circle은 프로토타입에 의해 연결된다.

  // 이 함수가 new 연산자와 함께 호출되지 않았다면 이 시점의 this는 전역 객체 window를 가리킨다.
  // 즉, this와 Circle은 프로토타입에 의해 연결되지 않는다.
  if (!(this instanceof Circle)) {
    // new 연산자와 함께 호출하여 생성된 인스턴스를 반환한다.
    return new Circle(radius);
  }

  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}

// new 연산자 없이 생성자 함수를 호출하여도 생성자 함수로서 호출된다.
const circle = Circle(5);
console.log(circle.getDiameter()); // 10
```

17.2 생성자 함수

17.2.7 new.target

Object와 Function 생성자 함수는
new 연산자 없이 호출해도 new 연산자와 함께 호출했을 때와 동일하게 동작

String, Number, Boolean 생성자 함수는 다르게 동작함

new 연산자 있을 때 => 객체 생성하여 반환
new 연산자 없을 때 => 문자열, 숫자, 불리언 값을 반환(데이터 타입이 변환하기도 함)

【 예제 17-22 】

```
const str = String(123);
console.log(str, typeof str); // 123 string
```

```
// String 생성자 함수에 의한 String 객체 생성
const strObj = new String('Lee');
console.log(typeof strObj); // object
console.log(strObj);        // String {"Lee"}
```