Chap17 생성자 함수에 의한 객체 생성

이번 장에서는

- 생성자 함수를 사용하여 객체를 생성하는 방식
- 객체 리터럴을 사용하여 객체를 생성하는 방식과 생성자 함수를 사용하여 객체를 생성하는 방식의 장점과 단점

살펴볼 것이다.

17-1 Object 생성자 함수

▼ 생성자 함수

new 연산자와 함께 호출하여 객체(인스턴스)를 생성하는 함수

> 인스턴스 : 생성자 함수에 의해 생성된 객체

참고사항 자바스크립트에는 Object 생성자 함수 이외에 String, Number, Boolean, Function, Array, Date, RegExp, Promise 등의 빌트인 생성자 함수를 제공

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

```
// Number 생성자 함수에 의한 Number 객체 생성
const numObj = new Number(123);
console.log(typeof numObj); // object
console.log(numObj);        // Number {123}

// Boolean 생성자 함수에 의한 Boolean 객체 생성
const boolObj= new Boolean(true);
console.log(typeof boolObj); // object
console.log(boolObj);        // Boolean {true}

// Function 생성자 함수에 의한 Function 객체(함수) 생성
const func = new Function('x', 'return x * x');
console.log(typeof func); // function
console.dir(func);        // ƒ anonymous(x)

// Array 생성자 함수에 의한 Array 객체(배열) 생성
const arr = new Array(1, 2, 3);
console.log(typeof arr); // object
console.log(arr);        // [1, 2, 3]

// RegExp 생성자 함수에 의한 RegExp 객체(정규 표현식) 생성
const regExp = new RegExp(/ab+c/i);
console.log(typeof regExp); // object
console.log(regExp);        // /ab+c/i

// Date 생성자 함수에 의한 Date 객체 생성
const date = new Date();
console.log(typeof date); // object
console.log(date);        // Mon May 04 2020 08:36:33 GMT+
```

17-2 생성자 함수

▼ 객체 리터럴에 의한 객체 생성 방식의 문제점

프로퍼티가 구조가 동일할때에는 객체 리터럴을 이용하는 방식은 매번 같은 프로퍼티를 기술해야하므로 비효율적이다.

```
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
```

객체 고유의 상태 데이터인 redius 프로퍼티의 값은 객체마다 다를 수 있지만 getDiameter 메서드는 완전히 동일하여 같은 코드를 반복적으로 적어야한다.

▼ 생성자 함수에 의한 객체 생성 방식의 장점

생성자 함수를 사용하면 프로퍼티 구조가 동일한 객체를 여러개 간편하게 생성할 수 있다.

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

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

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

`생성자 함수`는 일반 함수와 동일한 방법으로 정의하고 `new 연산자`와 함께 호출하는 형식이다. 그런데 new 연산자를 함께 호출하지 않으면 이는 `생성자 함수`가 아니라 `일반 함수`로 동작한다.

▼ this

객체 자신의 프로퍼티나 메서드를 참조하기 위한 자기 참조 변수(self-referencing variable)다. this가 가리키는 값, 즉 this 바인딩은 함수 호출 방식에 따라 동적으로 결정된다.

함수 호출 방식	this가 가리키는 값(this 바인딩)
일반 함수로서 호출	전역 객체
메서드로서 호출	메서드를 호출한 객체(마침표 앞의 객체)
생성자 함수로서 호출	생성자 함수가 (미래에)생성할 인스턴스

```
// 함수는 다양한 방식으로 호출될 수 있다.
function foo() {
  console.log(this);
}

// 일반적인 함수로서 호출
// 전역 객체는 브라우저 환경에서는 window, Node.js 환경에서는 gl
foo(); // window

// 메서드로서 호출
const obj = { foo }; // ES6 프로퍼티 축약 표현
obj.foo(); // obj => {foo: f}

// 생성자 함수로서 호출
const inst = new foo(); // inst => foo {}
```

▼ 생성자 함수의 인스턴스 생성 과정

1. 인스턴스 생성과 this 바인딩

this에 반인딩 되어 있는 인스턴스에 프로퍼티나 메서드를 추가하고 생성자 함수가 인수로 전달받은 초기값을 인스턴스 프로퍼티에 할당하여 초기화하거나 고정값을 할당한다.

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

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

▼ 바인딩

식별자와 값을 연결하는 과정. 예를 들어 변수 선언은 변수이름(식별자)과 확보된 메모리 공간의 주소를 바인딩하는 것이다. this 바인딩은 this(키워드로 분류되지만 식별자 역할을 한다.)와 this가 가리킬 객체를 바인딩하는 것이다.

2. 인스턴스 초기화

this에 바인딩되어 있는 인스턴스를 초기화한다.

```
function Circle(radius){
 //1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.
 console.log(this); //Circle {}
 //2. this에 바인딩되어 있는 인스턴스를 초기화한다.
 this.radius=radius;
 this.getDiameter = function (){
  return 2*this.radius;
 };
}
```

3. 인스턴스 반환

생성자 함수 내부에서 모든 처리가 끝나면완성된 인스턴스가 바인딩된 this를 암묵적으로 반환한다.

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

```
 console.log(this); //Circle {}
 //2. this에 바인딩되어 있는 인스턴스를 초기화한다.
 this.radius=radius;
 this.getDiameter = function (){
  return 2*this.radius;
 };
 //3. Cicle 생성자 함수는 암묵적으로 this를 반환한다.
}

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

만약 this가 아닌 다른 객체를 명시적으로 반환하면 this가 반환되지 못하고 return 문에 명시한 객체가 반환된다.

```
function Circle(radius){
 //1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.
 console.log(this); //Circle {}
 //2. this에 바인딩되어 있는 인스턴스를 초기화한다.
 this.radius=radius;
 this.getDiameter = function (){
  return 2*this.radius;
 };
 //명시적으로 객체를 반환하면 암묵적인 this반환이 무시된다.
 return {};
}

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

그러나 명시적으로 `원시값` 을 반환하면 `원시 값`은 무시되고 암묵적으로 `this`가 반환된다.

```
function Circle(radius){
 //1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.
 console.log(this); //Circle {}
 //2. this에 바인딩되어 있는 인스턴스를 초기화한다.
 this.radius=radius;
 this.getDiameter = function (){
  return 2*this.radius;
 };
 //명시적으로 원시값을 반환하면 암묵적으로 this반환된다.
 return 100;
}

//인스턴스 생성.
const circle=new Circle(1);
//Cicle 생성자 함수는 암묵적으로 this가 반환된다.
console.log(circle); ////Circle {fadius:1, getDiameter:
```

▼ 내부 메서드 [[Call]과 [[Construct]]

함수는 객체이므로 일반객체와 동일하게 동작할 수 있다. 그러나 일반 객체는 함수처럼 호출을 할 수 없다.

따라서 함수 객체는 일반 객체가 가지고 있는 내부슬롯과 내부 메서드는 물론, 함수로서 동작하기 위해 함수 객체만을 위한 `[[Environment]]`, `[[FormalParameters]]` 등의 내부슬롯과 `[[Call]]`, `[[Construct]]` 같은 내부 메서드를 추가로 가지고 있다.

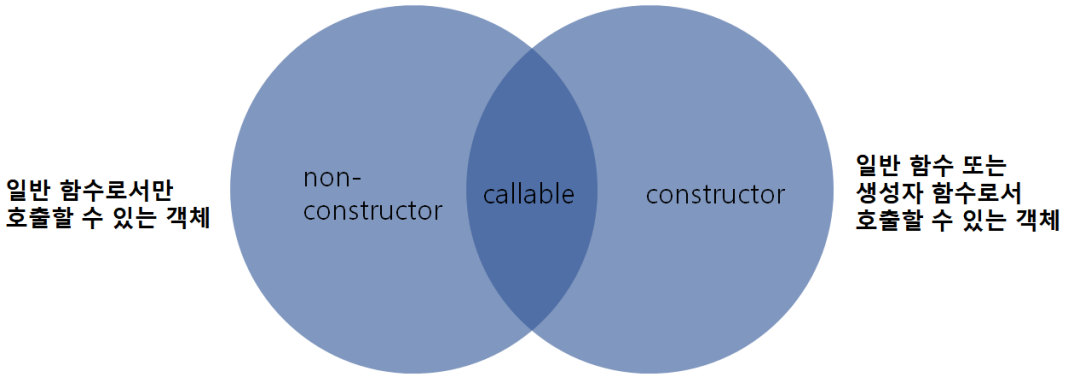
함수가 일반 함수로로서 호출되면 함수 객체 내부메서드 `[[Call]]` 이 호출되고 new 연산자와 함께 생성자 함수로서 호출되면 내부 메서드 `[[Construct]]`가 호출된다.

```
function foo(){}

//일반적인 함수로서 호출 : [[Call]]
foo();

//생성자 함수로서 호출 : [[Construct]]가 호출된다.
new foo();
```

함수 객체는 반드시 `callable`이어야 한다. 따라서 모든 함수 객체는 내부 메서드 `[[Call]]`을 갖고 있으므로 호출할 수 있다.
하지만 모든 함수 객체가
`[[Construct]]`를 갖는 것은 아니다. 함수 객체는 `constructor`일 수도 있고 non-constructor일 수도 있다.



▼ constructor와 none-constructor 구분

ECMAScript사양에서는 함수 정의 방식에 따라 `constructor`와 `non-constructor`를 구분 한다. 따라서

- `constructor` → 일반함수, 함수선언문, 함수 표현식으로 정의된 함수
- `none-constructo`r → 화살표 함수와 메서드 축약 표현으로 정의된 함수

▼ new 연산자

- 사용자 정의 객체 타입 또는 내장 객체 타입의 인스턴스를 생성한다.
- **new 연산자**와 함께 함수를 호출하면 해당함수는 생성자 함수로 동작한다. 따라서 `[[Construct]]`가 호출된다.

전에도 언급했듯이 **new 연산자**에 유무로 생성자 함수인지 일반 함수인지 나뉘고 this의 역할도 바뀐다.

- 생성자 함수에서 this → 생성자 함수가 생성할 인스턴스
- 일반함수에서 this → 전역 객체 window

▼ window?

Window 객체란?

window 객체는 두 가지 역할을 하는데,

1. 브라우저 안의 모든 요소들이 소속된 객체로, 최상위에 있기 때문에 어디서든 접근이 가능하다고 해서 '전역 객체'라고도 부른다.

2. 일반적으로 우리가 열고 있는 브라우저의 창(browser window)을 의미하고, 이 창을 제어하는 다양한 메서드를 제공한다.

▼ new.target

ES6에서 지원하는 것으로 생성자함수로서 호출됐는지 안됐는지 확인할 수 있다.

new 연산자와 함께 생성자 함수로서 호출되면 new.target은 함수 자신을 가리키고 new 연산자 없이 일반 함수로서 호출되면 new.target은 undefined이다.

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

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

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