17장 생성자 함수에 의한 객체 생성

17.0 머릿말

- 앞서서 객체 리터럴을 사용하여 객체를 만들어봤다.
- 다양한 객체 생성방식 중 생성자 함수를 사용하여 객체 생성하는 방식과 장단점을 살펴보자.

17.1 Object 생성자 함수

- New 연산자와 함께 Object 생성자 함수를 호출하면 빈 객체를 생성 — 프로퍼티, 메서드 후에 추가 가능 — Ex) new Object( );
- 생성자 함수란? — new 연산자와 함께 호출하여 객체(인스턴스)를 생성하는 함수 — 생성자 함수에 의해 생성된 객체를 인스턴스라고 한다.
- Object 생성자 함수 이외 여러가지 빌트인 생성자 함수
  - String, Number, Boolean, Function, Array, RegExp, Promise 등
  - new로 생성한 값의 타입은 모두 object 이다. — Ex) strObj = new String("Kim"); => { "Kim" }
- Object 생성자 함수를 사용하는 것은 그다지 유용하지않다. 별로 쓸 일 없다. — 객체 리터럴 사용 권장

17.2 생성자 함수

1. 객체 리터럴에 의한 객체 생성 방식의 문제점

- 객체 리터럴 생성방식은 직관적이고 간편하다.
- 하지만 단 하나의 객체만 생성 여러개 객체를 만들어야 할 때는 비효율적인 문제
- 객체는 프로퍼티를 통해 객체 고유의 상태를 표현 — 메서드는 내용이 동일한 경우가 일반적

2. 생성자 함수에 의한 객체 생성 방식의 장점

- 객체를 생성하기 위한 템플릿(클래스)처럼 생성자 함수를 사용하여 여러개 객체 생성 가능 — 예제
- new 연산자와 함께 호출해야 생성자 함수로 동작한다. new가 없으면 일반 함수로 동작하게 된다. — 일반 함수로 호출될 경우 this.변수는 전역객체에 바인딩된다.

** this

3. 생성자 함수의 인스턴스 생성 과정

- (0) 머릿말
  - 생성자 함수의 역할
    - 인스턴스 생성 — 필수
    - 생성된 인스턴스를 초기화(프로퍼티 추가 및 초기값 할당) — 옵션
  - 생성자 함수 내에 return 명령어는 쓰지 않는다.
  - Js 엔진은 암묵적인 처리를 통해 인스턴스를 생성하고 반환한다.
- (1) 인스턴스 생성과 this 바인딩
  - 암묵적으로 빈 객체가 생성
  - 이때 생성된 인스턴스는 this에 바인딩
  - 이 처리는 함수 몸체의 코드가 한 줄씩 실행되는 런타임 이전에 실행
- ** 바인딩
- (2) 인스턴스 초기화 — 생성자 함수에 있는 코드가 한 줄씩 실행되어 this에 바인딩되어 있는 인스턴스를 초기화한다. 이 처리는 개발자가 기술
- (3) 인스턴스 반환
  - 생성자 함수 내부의 모든 처리가 끝나면 완성된 인스턴스가 바인딩된 this가 return 문 없이 암묵적으로 반환된다.
  - 만약 return 값으로 객체를 반환하면 this가 반환되지 못하고 return문에 명시한 객체가 반환 — Ex) return { }
  - 하지만 명시적으로 원시 값을 반환하면 원시 값 반환은 무시되고 암묵적으로 this가 반환 — Ex) return 100
  - 따라서 생성자 함수 내부에서 return 문은 반드시 생략해야 한다.

4. 내부 메서드 [[Call]] 과 [[Construct]]

- 함수 선언문 또는 함수 표현식으로 정의한 함수는 일반 함수, 생성자 함수로서 호출할 수 있다. — 생성자 함수로서 호출할 수 있다는 것은 new연산자와 함께 호출하여 객체를 생성하는 것을 의미
- 함수는 객체이므로 일반 객체와 동일하게 동작할 수 있다.
  - 함수 객체는 일반 객체가 가지고 있는 내부 슬롯과 내부 메서드를 모두 가지고 있기 때문
  - 함수는 프로퍼티, 메서드를 소유할 수 있다. — 예제
- 함수(객체) vs 일반 객체 차이점 — 일반 객체는 호출할 수 없지만 함수는 호출할 수 있다.
- 함수 객체만 가지고 있는 프로퍼티
  - 내부 슬롯
    - [[Environment]]
    - [[FormalParameters]]
  - 내부 메서드
    - [[Call]] — 일반 함수 호출시 작동
    - [[Construct]] — 생성자 함수 호출시 작동
- 내부 메서드 call을 갖는 함수 객체를 callable 이라고 하며, 내부 메서드 [[Construct]] 를 갖는 함수 객체를 constructor , 갖지 않는 것은 non-constructor 이다.
  - Callable : 호출할 수 있는 객체, 즉 함수
  - Constructor : 생성자 함수로 호출할 수 있는 함수
  - 모든 함수가 call 메서드를 갖고 있지만, constructor 을 모든 함수객체가 갖는건 아니다.

5. constructor 와 non-constructor 구분

- 함수 객체를 생성할 때 함수 정의 방식에 따라 나뉜다.
  - 코드 내 Function 키워드 유무 차이
    - non-constructor
      - ES6 화살표 함수 — const bar = ( ) => { }
      - 메서드 축약 표현 — { bar( ) { } }
    - constructor
      - 함수 선언식 — function bar( ) { ... }
      - 함수 표현식 — const bar = function ( ) { ... }
  - 객체 내 메서드는 축약 표현만 메서드로 인정한다.
    - Ex) x: function ( ) { } || x: ( ) => { } -> 인정 X
    - Ex) x() { } -> 인정 O
- constructor
  - 함수 선언문, 함수 표현식, 클래스(클래스도 함수)
  - 일반 함수처럼 사용 가능 (주의)
- Non-constructor
  - 메서드(ES6 메서드 축약 표현 객체내)
  - 화살표 함수
  - 생성자 함수 new 사용시 에러

6. new 연산자

- 일반함수와 생성자 함수에 특별한 형식적 차이가 없다.
- This 는 생성자 함수 호출시 인스턴스를 가리키고, 일반 함수 호출시 window 전역객체를 가리킨다.
- 그렇기에 생성자 함수는 첫 문자를 대문자로 기술하는 파스칼 케이스로 명명하여 일반 함수와 구별한다.

7. new.target

- new 연산자 없이 생성자 호출 방지책
- ES6 부터 지원
- 메타 프로퍼티로 this 와 유사하게 constructor인 모든 함수 내부에서 암묵적인 지역 변수와 같이 사용 — IE는 지원 x — IE를 위해서 스코프 세이프 생성자 패턴이 있다. — new.target 대신에 !(this instanceof 생성자함수) 를 사용
- new 연산자와 함께 생성자 함수로서 호출되면 함수 내부의 new.target은 함수 자신을 가리킨다. — new 연산자 없이 일반 함수로서 호출하면 함수 내부의 new.target은 undefined 이다.
- 재귀 호출을 통해 에러를 발생 없이 return 문에 " new 생성자함수() " 를 써준다. — 예제
- 대부분의 빌트인 생성자 함수는 new 연산자와 함께 호출되었는지 확인 후 적절한 값을 반환한다.
  - Object와 Function 생성자 함수는 new 연산자 없이 호출해도 new를 쓴 것과 동일하게 동작한다. — 그렇기때문에 new Array( ) 대신 Array( ) 만 써도 잘 작동하는 것이다.
  - 하지만, String, Number, Boolean 생성자 함수
    - new 호출시 — 객체를 생성하여 객체반환
    - new 호출 x — 타입변환