프로퍼티 어트리뷰트

- 내부 슬롯
  - JS의 엔진을 구현하는 알고리즘을 설명하기 위해 ECMAScript에서 사용하는 의사 프로퍼티이다.
  - 감춰진 프로퍼티 라고 말할 수도 있다.
- 내부 메소드 : JS의 엔진을 구현하는 알고리즘을 설명하기 위해 ECMAScript에서 사용하는 의사 메소드이다.

내부 슬롯과 내부 메소드는 [[ ]] 으로 감싸서 나타낸다.

내부 슬롯과 내부 메소드는 원래 외부로 공개된 객체의 프로퍼티와 메소드가 아니다. 그래서 직접적으로 접근하거나 호출 할 수 있는 방법이 없지만, 간접적으로는 접근할 수 있다.

1. [[prototype]] 이란 내부 슬롯에는 원래 직접적으로 접근하는것은 불가능하지만, _ _ prototype _ _이라는 것으로 간접적으로 접근이 가능하다.(모든 객체는 [[prototype]]) 이란 내부슬롯을 가지고있다)

   ```
   const o ={};

   o. [[prototype]] // SyntaxError <- [[prototype]]이라는 내부 슬롯에 직접적으로 접근했더니 에러가 났다.
   o.__protot__ // object.prototype <- [[prototypr]] 이라는 내부 슬롯에 __proto__ 라는 것을 사용하여
   간접적으로 점근한 상태이다. 에러가 안난걸 보니 정상적으로 동작하는것이다.
   ```

2. Object.getOwnPropertyDescript 라는 메소드를 사용하여 `프로퍼티 어트리뷰트` 라는 내부슬롯에 간접적으로 접근이 가능하며, 접근을 하면 프로퍼티 디스크립터 객체를 반환한다.
   - Object.getOwnPropertyDescript 라는 메소드를 호출하면, 첫번째 매개변수에는 객체의 참조를 전달하고, 두번째 매개변수에는 프로퍼티 키를 문자열로 전달한다.

> **데이터 프로퍼티 어트리뷰트**(=내부 슬롯)
>
> 프로퍼티의 상태를 나타내는 것이다.
>
> ① [[Value]] : 프로퍼티의 값을 나타낸다.
>
> - 프로퍼티 키를 통해 프로퍼티 값을 변경하면 [[Value]]에 값을 재할당한다.

- 프로퍼티가 업승면 프로퍼티를 동적 생성하고 생성된 프로퍼티의 [[Value]]에 값을 저장한다.

② [[Writable]] : 값의 갱신이 가능한지, 프로퍼티 값의 변경이 가능한지 여부를 나타낸다.
(Value 의 값을 쓸수 있느냐?)

- [[Writable]] 값이 false인 경우에는 [[Value]] 의 값을 변경할수 없는 readOnly한 프로퍼티가 된다.
- 단, [[Writable]]이 true면 [[Value]] 의 변경과 [[Writable]]을 false로 변경이 가능하다.

③ [[Enumerable]] : 열거가 가능한지 여부를 나타낸다.

- for in 문으로 열거
- object 라는 객체의 keys로 열거
(객체의 keys : 프로퍼티의 키들을 뽑아서 배열로 리턴해준다.)
- [[Enumerable]] 의 값이 false인 경우에는 for in문이나 Object.keys 메소드 등으로 열거할수 없다.

④ [[Configurable]] : 재정의가 가능한지 여부를 나타낸다.

- [[Configurable]]값이 false인 경우에는, 해당 프로퍼티의 삭제, 값의 변경, 프로터피 재정의가 금지된다.

> **프로퍼티 디스크립터 객체**
>
> 프로퍼티 어트리뷰트들을 모은 객체

```
const person = {
    name: 'Lee'
};
console.log(object.getOwnPropertyDescriptor(person,'name');
// {value: "Lee", writable: true, enumerable: true,
configurable: true}
-----------------------------------------------------------
----
// value: "Lee" <- 프로퍼티의 값인 'Lee'를 문자열로 반환했다.
// writable: true <- 객체 리터럴로 프로퍼티를 만들면 내부슬롯의 값은 기본적으로 true로 세팅이 된다.
//enumerable: true <- 객체 리터럴로 프로퍼티를 만들면 내부슬롯의 값은 기본적으로 true로 세팅이 된다.
//confiurable: true <- 객체 리터럴로 프로퍼티를 만들면 내부슬롯의 값은 기본적으로 true로 세팅이 된다.
```

즉, 프로퍼티에는 프로퍼티 어트리뷰트라는 내부슬롯이 있고, 그렇기 때문에 프로퍼티 자체가 객체라는 뜻이다.
또한, 모든 객체는 [[Prototype] 이란 내부슬롯을 가지고있는 것이고, [[Value]], [[Writable]], [[Enumarable]]. [[Configuarable]], [[Prototype]] 각각의 내부슬롯을 **프로퍼티 키**라고 생각하면 된다.

데이터 프로퍼티와 접근자 프로퍼티

데이터 프로퍼티

" 키와 값으로 구성된 **일반적인 프로퍼티** "

- 프로퍼티가 생성될 때 [[Value]]의 값은 프로퍼티 값으로 초기화가 되며, [[writable]]과 [[Enumarable]], [[Configuarable]] 의 기본값인 true 로 초기화 된다.
  이는, 프로퍼티를 동적 추가하여도 마찬가지다.

접근자 프로퍼티(_ _ proto _ _)

" [[Value]]을 가지고 있지 않고, 다른 데이터 프로퍼티의 값을 읽거나 저장할 때 사용하는 **접근자 함수로 구성된 프로퍼티**"

프로퍼티 처럼 생긴 함수

① [[Get]] : 접근자 프로퍼티를 통해 데이터 프로퍼티의 값을 읽을 때 호출되는 접근자 함수

- 접근자 프로터피 키로 프로퍼티 값에 **접근하면** `getter 함수`가 호출이 되고, 그 결과가 프로퍼티 값으로 반환된다.
- 반드시 리턴을 해야 한다.
- 인수를 전달할 방법이 없기때문에 매개변수를 쓸 이유가 없다.

> **getter 함수**
>
> [[Get]] 이라고 하는 프로퍼티 어트리뷰트의 값인 함수

```
get fullname(){
    return `${this.firstName} ${this.lastName}`;
}
console.log(person.fullName);
```

② [[Set]] : 접근자 프로퍼티를 통해 데이터 프로퍼티의 값을 읽을 때 호출되는 접근자 함수

- 접근자 프로터피 키로 프로퍼티 값에 **저장하면** setter 함수가 호출이 되고, 그 결과가 프로퍼티 값으로 반환된다. (프로퍼티 값을 할당하면 `setter 함수`가 호출이 된다.)
- 반드시 리턴이 없어야 한다.
- 반드시 인수를 받아야 하는데, 반드시 1개만 받아야 한다.

> **setter 함수**
>
> [[Set]] 이라고 하는 프로퍼티 어트리뷰트의 값

```
set fullName(name){
    [this.first, this.lastName] = name.split(' ');
}
person.fullName = 'Heegun Lee';
```

③ [[Enumerable]] : 열거가 가능한지 여부를 나타낸다.

- for in 문으로 열거
- object 라는 객체의 keys로 열거
  (객체의 keys : 프로퍼티의 키들을 뽑아서 배열로 리턴해준다.)
- [[Enumerable]] 의 값이 false인 경우에는 for in문이나 Object.keys 메소드 등으로 열거할수 없다.

④ [[Configurable]] : 재정의가 가능한지 여부를 나타낸다.

- [[Configurable]]값이 false인 경우에는, 해당 프로퍼티의 삭제, 값의 변경이 금지된다.

예시 코드)

```
const person = {
  // 데이터 프로퍼티
  firstName: 'Ungmo',
  lastName: 'Lee',

  // fullName은 접근자 함수로 구성된 접근자 프로퍼티이다.
  // getter 함수
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
  // setter 함수
  set fullName(name) {
    // 배열 디스트럭처링 할당: "31.1 배열 디스트럭처링 할당" 참고
    [this.firstName, this.lastName] = name.split(' ');
  }
};

// 데이터 프로퍼티를 통한 프로퍼티 값의 참조
console.log(person.firstName + ' ' + person.lastName); // Ungmo Lee

// 접근자 프로퍼티를 통한 프로퍼티 값의 저장
// 접근자 프로퍼티 fullName에 값을 저장하면 setter 함수가 호출된다.
person.fullName = 'Heegun Lee';
console.log(person); // {firstName: "Heegun", lastName: "Lee"}

// 접근자 프로퍼티를 통한 프로퍼티 값의 참조
// 접근자 프로퍼티 fullName에 접근하면 getter 함수가 호출된다.
console.log(person.fullName); // Heegun Lee

// firstName는 데이터 프로퍼티이다.
// 데이터 프로퍼티는 value, writable, enumerable, configurable 프로퍼티 어트리뷰트를 갖는다.
let descriptor = Object.getOwnPropertyDescriptor(person, 'firstName');
console.log(descriptor);
// {value: "Heegun", writable: true, enumerable: true, configurable: true}
```

```
// fullName는 접근자 프로퍼티이다.
// 접근자 프로퍼티는 get, set, enumerable, configurable 프로퍼티 어트리뷰트를 갖는다.
descriptor = Object.getOwnPropertyDescriptor(person, 'fullName');
console.log(descriptor);
// {get: ƒ, set: ƒ, enumerable: true, configurable: true}
```

** getOwnPropertyDecriptor 함수로 데이터 프로퍼티만 가져올수 있는게 아니라 접근자 프로퍼티도 가져올수 있다.

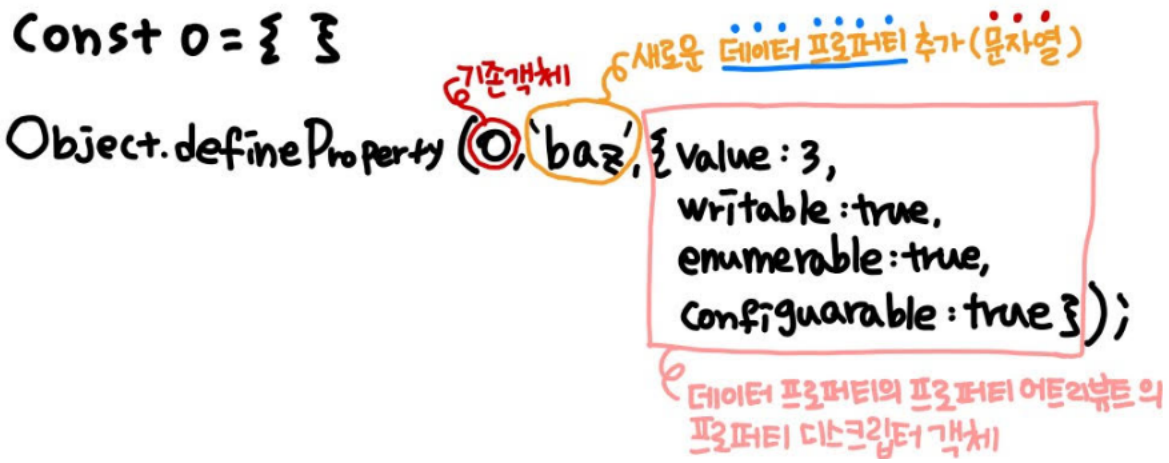
```
Object getOwnPropertyDecriptor(Object.prototype, '__proto__');
// {get: f, set: f, enumerable: true, configuarable: true}

Object.getOwnPropertyDecriptor(function(){}, 'prototype');
//  {value: {...}, writable: true, enumerable: true,
configuarable: true}
```

프로퍼티 정의

프로퍼티 정의

- **새로운 프로퍼티를 추가** 하면서 **프로퍼티 어트리뷰트를 명시적으로 정의** 하거나,
  기존 프로퍼티의 프로퍼티 어트리뷰트를 **재정의** 하는 것을 말한다.
- **Object.defineProperty 메소드**를 사용하면 **프로퍼티의 어트리뷰트를 정의** 할 수 있다.
- Object.defineProperty 메소드는 한번에 하나의 프로퍼티만 정의할수 있지만, Object.defineProperties 메소느는 여러개의 프로퍼티를 한번에 정의 할수 있다.

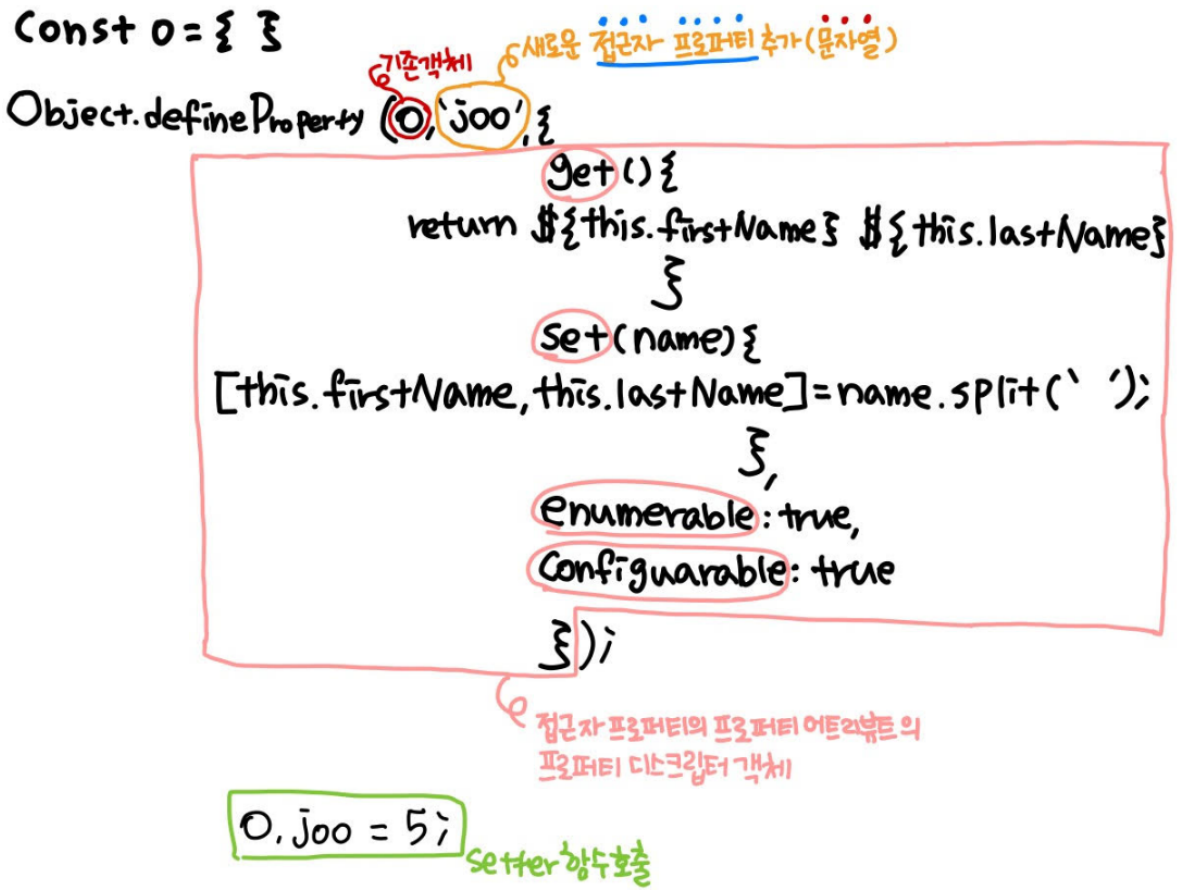
데이터 프로퍼티 정의



- Object.defineProperty 메소드로 프로퍼티를 정의할때 프로퍼티 디스크립터 객체에서 일부의
  프로퍼티를 생략하면 프로퍼티의 기본값이 적용된다.

  [[Value]] 의 기본값 : undefined
  [[Writable]] 의 기본값 : false
  [[Enumerable]] 의 기본값 : false
  [[Configuarable]] 의 기본값 : false

접근자 프로퍼티 정의



- Object.defineProperty 메소드로 프로퍼티를 정의할때 프로퍼티 디스크립터 객체에서 일부의
  프로퍼티를 생략하면 프로퍼티의 기본값이 적용된다.

  [[Get]] 의 기본값 : undefined
  [[Set]] 의 기본값 : undefined
  [[Enumerable]] 의 기본값 : false
  [[Configuarable]] 의 기본값 : false

객체 변경 방지

객체를 변경을 방지할수 있는 3가지 메소드가 있다.

1. 객체 확장 금지
2. 객체 밀봉
3. 객체 동결
   - 재할당이 안 이루어지는 상황에서 객체를 원시값처럼 쓰고자 할때에 객체 동결(Object.freeze)를 통해서 객체를 얼려서 readOnly하게 만든다.

==> ∴ 이런게 있다고만 알고 넘어갑시다. 왜냐면, 실무에서는 다 라이브러리를 쓰기 때문에 이것들이 있다고만 알고 넘어가자. <- *라이브러리 사용하는게 훨씬 안전하다*

생성자 함수에 의한 객체 생성

객체를 생성하는 방법은 다양하다.

1. 객체 리터럴을 사용하여 생성
2. Object 생성자 함수로 생성
3. 생성자 함수로 생성
4. Object.creat 라는 메소드로 생성

위 방법들 중 1번 객체 리터럴을 사용하여 생성하는 건 앞서 다룬 내용이므로, 이번엔 2번부터 시작하는 생성자 함수로 객체를 생성하는 방법을 알아보겠다.

Object 생성자 함수

new연산자 와 함께 **Object 생성자 함수를 호출** 하면 **빈 객체를 생성**하여 반환한다. 이 이후에 **프로퍼티 또는 메소드를 추가**하여 빈 객체를 채워서, 내용을 채운 객체를 완성할 수 있다.

반드시 Object 생성자 함수를 사용해서 빈 객체를 생성해야 하는 것은 아니므로, 특별한 이유가 없다면
사용하지마라. 유용하지 않은 함수다.

생성자 함수

- 생성자 함수 라고 함은, 객체를 만들어내는 목적을 가지고 있는 함수다
- new 연산자와 함께 호출하여 객체(인스턴스)를 생성하는 함수
  즉, 생성자 함수에 의해 생성된 객체를 인스턴스라고 한다.
- 생성자 함수와 일반 함수가 생긴 것은 똑같이 생겼지만, 구분하기 위해서 생성자 함수의 함수 이름 첫글자를 대문자로 사용한다.(<-*소문자로 해도 Error는 나지 않지만, 구분을 해주자*)

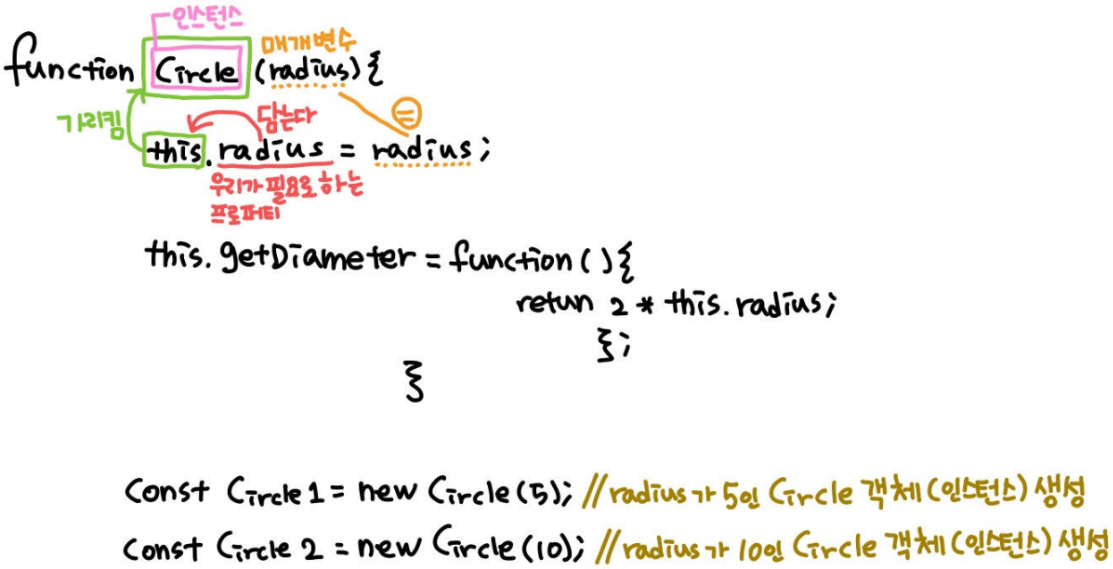
- 생성자 함수에는 여러 빌트인 함수들이 있는데, 그 중에 우리가 만들수 있는 생성자 함수는
  Object 생성자 함수 / String 생성자 함수 / Number 생성자 함수 / Boolean 생성자 함수
  / Function 생성자 함수 / Array 생성자 함수/ Date 생성자 함수 / RegExp 생성자 함수 이다.

```
1. String 생성자 함수를 사용하여 생성한 String 객체
const strObj = new String('Lee');
console.log(typeof strObj); // object (문자열 객체)
console.log(strObj); // String {"Lee"}

2. Number 생성자 함수를 사용하여 생성한 Number 객체
const numObj = new Number(123);
console.log(typeof munObj); // object(number 객체)
console.log(numObj); // Number {123}

3. Boolean 생성자 함수를 사용하여 생성한 Boolean 객체
const booObj = new Boolean(true);
console.log(typeof booObj); // object(boolean 객체)
console.log(booObj); // Boolean {true}

4. Function 생성자 함수를 사용하여 생성한 Function 객체
const func = new Function('x', 'return x * x');
console.log(typeof func); // object(function 함수 객체)
console.log(func); // ƒ anonymous(x)

5. Array 생성자 함수를 사용하여 생성한 Array 객체
const arr = new Array(1, 2, 3);
consoel.log(typeof arr); // object(array 객체)
console.log(arr); // [1, 2, 3]

6. Date 생성자 함수를 사용하여 생성한 Date 객체
const date = new Date();
conosole.log(typeof date); // object(date 객체)
console.log(date); // Sun May 17 2020 21:17:00 GMT+0900

7. RegExp 생성자 함수를 사용하여 생성한 RegExp 객체
```

- 객체 리터럴에 의한 객체 생성은 단 하나의 객체만 생성하기 때문에 프로퍼티 구조가 똑같으면 매번 똑같은 객체를 만들어야 하므로 비효율 적이다 -> 프로퍼티 구조가 똑같은 객체라면, 생성자 함수로
  객체를 짧고 간결하게 만들 수 있다.

(대부분 객체마다 프로퍼티 값이 다를수 있지만 메소드는 내용이 동일한 경우가 일반적이기 때문이다.)

- 객체 리털



위 코드에서 볼수 있듯이,

1. 생성자 함수 내부에서의 this
   - 생성자 함수가 미래에 생성할 인스턴스를 가리킨다.
   - this에는 우리가 필요로 하는 프로퍼티를 담는다.(그러므로, this 뒤에는 꼭 프로퍼티가 와야 한다)
   - 생성자 함수 내부에서의 this는 문맥에 따라서 가리키는 값이 달라진다.
2. news 연산자를 통해서 생성자 함수를 호출했다.
   - new를 쓰지 않고 호출하면 undefined 가 나오고, new를 붙여야 값이 호출된다.

     (new 연산자와 함께 호출하면 해당 함수는 행성자 함수로 동작하지만, new 연산자와 함께 호출하지 않으면 생성자 함수가 아니라 일반 함수로 동작한다.)

생성자 함수에 의한 객체 생성 방식의 장점

생성자 함수 내부에서 this는 생성자 함수가 생성할 인스턴스다.
객체 리터럴 내부의 메소드 내부에서 사용한 this는 그 메소드를 호출한 객체다.

그러면 일반함수에서 this를 쓸수 있을까?
쓸 수는 있지만, 그때의 this 는 전역객체를 가리킨다.

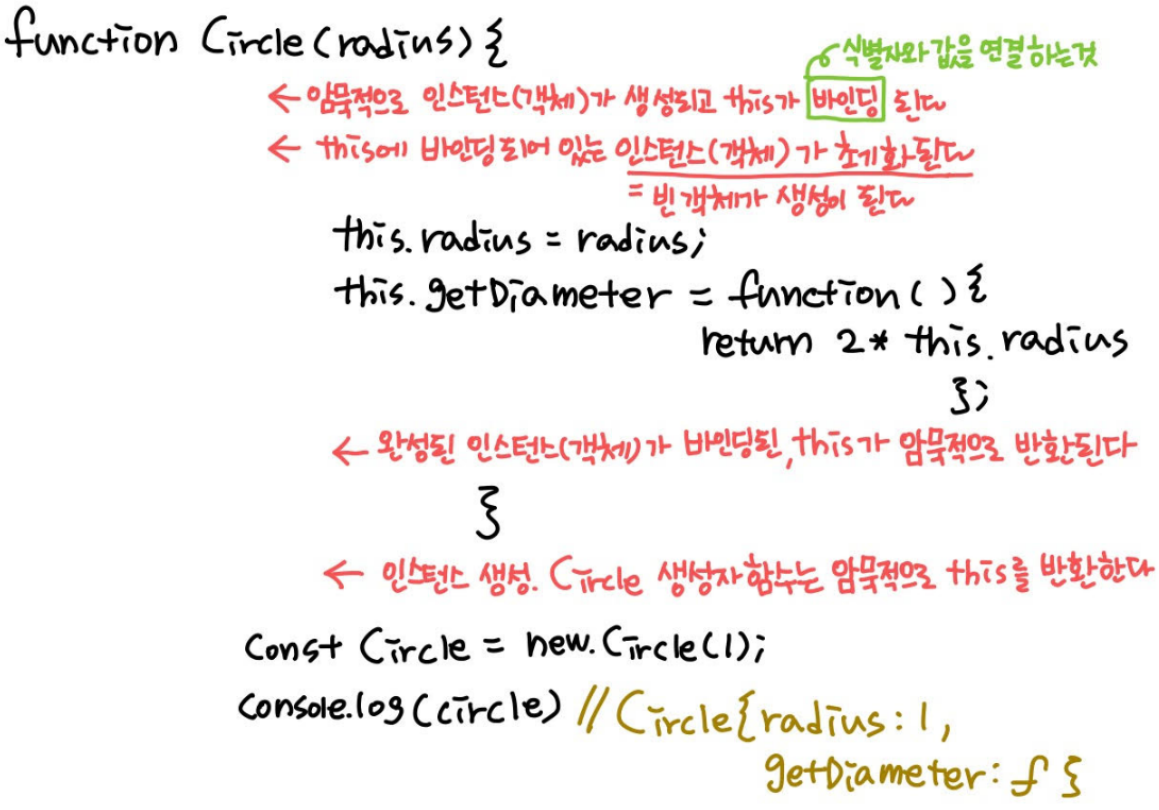
함수 호출 방식	this가 가리키는 값(this 바인딩)
일반 함수로서 호출	전역 객체
메소드로서 호출	메소드를 호출한 객체(마침표 앞의 객체)
생성자 함수로서 호출	생성자 함수가 (미래에) 생성할 인스턴스

생성자 함수의 인스턴스 생성 과정

1. 인스턴스(객체) 생성과 this 바인딩
   - 암묵적으로 빈 객체가 생성되고, 생성된 인스턴스, 즉 객체는 this에 `바인딩` 된다.
   - 이 처리는 런타임 이전에 실행된다.

     > **바인딩**
     >
     > 식별자와 값을 연결하는 과정

2. 인스턴스 초기화
   - this에 바인딩되어 있는 인스턴스를 초기화한다
3. 인스턴스 반환
   - 생성자 함수 내부의 모든 처리가 끝나면 완성된 인스턴스가 바인딩된, this가 암묵적으로 반환된다.



- 3번의 과정에서 명시적으로 다른 객체를 return 하면 반환해야 하는 this 가 반환되지 못하고,
  개발자 본인이 명시작 객체가 반환된다.
- 3번 과정에서 명시적으로 원시값을 return 하면 원시값 반환은 무시되고 원래 반환해야 하는 this가 반환된다.

  ==> 따라서 생성자 함수 내부에서 return 문은 절대 명시적으로 쓰면 안된다.

위에꺼 무조건 외워야 한다. 디스가 가리키는 값이 중요하다.

new : 이건 연산자다.

new 뒤에는, 우항에는 함수가 온다. 생성자 함수라는 문법이 있나요? 어떤 문법을 지켜서 함수를 만들면 생성자 함수다 라는게 있냐고. 그런건 없다. 그냥 함수를 만들면 그 함수는 생성자 함수로도 호출될수 있는거다. 따라서 일반 함수는 (일반함수는, 함수 만드는 방법 몇가지? 일반함수는 일반함수라는건 함수만드는거 몇가지?)

1. function foo()
2. const foo = finction () {}

위 두개는 생성자함수로도, 호출할수 잇고 일반함수로도 호출할수 있다. 즉, 뉴와 함께 호출할수도 있고 뉴와 함께 호출안할수도 잇다.

const foo = ( ) => {}

위 하나는 생성자 함수로 호출 못한다. 그리고 메소드라는게 있다. 여기서 주의해야 하는데 우리가 일반적으로 메소드라고 얘기하는데, 메소드는 자스의 전용용어가 아니다. 객체지향 언어에서 다 메소드라고 부른다. 우리가 메소드라고 통상 부르는거랑 에크마 스크립트에서 부르는 메소드는 다른거다.

에크마에서 메소드라고 부르는 애는 단하나. 메소드 축약형. (푸 () {}) 얘네만 문법상 메소드라고 한다.

근데 우리가 통상 어떤걸 다 메소드라고 하나? 통상적으로 말할때. 프로퍼티의 값이 함수인 애를 다 메소드라고 한다. 이걸 구별할줄 알아야 한다.

에크마스크립트 사양에서 메소드라고 명칭하는건 생성자 함수로 호출할수가 없다. 당연한거다. 메소드를 왜 생성자 함수로 호출하냐고~ 당연한거다.

(메모장 보기)

엄격하게 따지면 함수들의 종류가 있다 라는 얘기다.

생성자 함수로서 호출할수 있는 함수가 있고, 생성자 함수로서 호출 할수 없는 함수가 있다.

생성자 함수로서 호출할수 잇는 함수는 함수 선언문, 함수 표현식이다. 옛날거.

근데 생성자 함수로서 호출 안되는건 다 이엑스 식스 문법이다. 메소드랑 화살표함수.

함수라는게 기본적으로 어떤 특성을 갖고 있어야 하나? 함수도 객체라매. 객체랑 함수의 차이는?

객체가 함수보다 큰 개념이다.

그러면 함수는 객체의 특성을 모두 갖고 잇을까? 싹 다 갖고 있다.

함수는 객체로서의 특징은 싹다 갖고 잇다는 거다. 근데 함수만의 특징을 갖고 있을까? 갖고 잇다.

함수는 함수만의 특징도 갖고 잇고, 객체의 특징도 싹다 갖고 잇다.

그 함수만의 특징은 쉽게 말해서 호출할수 있다는 것이다.

객체는 호출할수 없다.

근데 함수를 호출하면, 내부적으로 [[call]] [[construct]] 얘네 내부 메소드다

함수 객체를 가리키는 식별자+ ( ) <- 이 식별자를 찾으러 가서 이 식별자가 가지고 있는 위 두개중의 하나를 내부 메소드를 호출하는거다. 내부동작이.

그러면 어떨때 콜, 어떨때 콘스트럭 호출하느냐.

얘는 일반 함수로 호출되엇을때 [[콜]] , 생성자 함수로 호출되었을때 [[컨스트럭트]] 다.

함수가, 함수 선언문이랑 함수 표현식이 잇는데 얘네들은 [[콜]] [[콘스트럭트]] 둘다 호출이 가능하다.

내부슬롯 콜과 내부슬롯 컨스트럭트 를 둘다 가지고 잇는것이다.

근데 콘스트럭터, 즉 생성자 함수로서 호출 못하는 메소드하고 화살표 함수는 콜은 갖고잇따.

콜을 안갖고잇으면 함수가 아니다.

함수는 호출할수 있는 객체라고 했는데, 그럼 호출을 못한다는 거다.

es6 메소드랑 화살표는 컨스트럭트가 없다.

근데 콘스트럭트는 있는애 가 잇고 없는 애가 있다. 없는 애를 논 컨스트럭터 , 있는 애를 콘스트럭터라고 한다.

```
// 일반 함수 정의: 함수 선언문, 함수 표현식
function foo() {}
const bar = function () {};
// 프로퍼티 x의 값으로 할당된 것은 일반 함수 정의에 의해 생성된 함수 객체이다.
// 이는 메소드로 인정하지 않는다.
const baz = {
  x: function () {}
};

// 일반 함수로 정의된 함수만이 constructor이다.
new foo(); // OK
new bar(); // OK
new baz.x(); // OK

// 화살표 함수 정의
const arrow = () => {};

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

// 메소드 정의: ES6의 메소드 축약 표현만을 메소드로 인정한다.
const obj = {
  x() {}
};
```

위의 코드에서 푸는 콘스트럭트일까, 논컨스트럭트일까?

메소드는 콘스트럭트다. 뉴와 함께 호출할수 있다.

위에 셋은 콜 내부 메소드도 갖고 있고 컨스트럭트 내부 메소드도 갖고 있다. 그래서 덩치가 크다는 것이다. 함수 객체가 크다는 얘기다.

이렇게 저렇게 호출할수도 있다. 쓸데없는 기능을 갖고잇다는 얘기다.

함수라는게 생성자 함수로도 호출될수 있고 일반 함수로도 호출될수있다.

그럼 이게 생성자 함수인지 일반 함수인지 파스칼케이스를 통해서 명확히 알려줘라~

근데 명확히 알려줬다치더라도 실수가 발생할까요 안할까요? 그래도 실수를 하기 마련이다. 곤란하다.

너무 기본적인거잖아. 그럼 어케 해야 하지? 함수를 만들때, 내가 파스칼 케이스로 썻는데 생성자 함수로 호출 안했어? 할거야? 이게 책임 문제인데, 티비를 삿는데 리모콘을 주잖아. 리모콘에 빨간 버튼이 있어. 아무것도 안써있어. 그래서 눌렀는데 티비가 터졋어. 그럼 누가 잘못이야? 그럼 그걸 왜 여기에 노출해놨냐고. 우리가 함수 만들었는데 그걸 생성자 함수로 호출할수도 있고 일반 함수로도 호출할수도 있잖아.

그럼 그 버튼을 만들지 말았어야지.

그럼 우리가 앞으로 이걸 생성자 함수로 불러야 하는데 안불렀을때 어케 해야 하지?

뉴 연산자를 안붙히는 경우도 있잖아.

붙혀야 하는데 안 붙히는 경우, 안붙혀야 하는데 붙인 경우.

먼저 안붙히는 경우를 보자. 안붙히면 어케 동작하는지 생각해보자.

그래서 비효율적이다. 함수라는게 생성자, 일반 함수로도 호출할수 잇다. 이게 생성자 함수인지 일반 함수인지 파스칼케이스로 구분해서 알려줘라.

함수를 만들때 파스칼케이스로 만들어줬는데 왜 생성자 함수로 호출 안했어? 라고 하는게 아니라 그 버튼을 만들지 말았어야지.

```
// 생성자 함수로서 정의하지 않은 일반 함수
function add(x, y) {
  return x + y;
}

// 생성자 함수로서 정의하지 않은 일반 함수를 new 연산자와 함께 호출
let inst = new add();
// 함수가 객체를 반환하지 않았으므로 반환문이 무시된다. 따라서 빈 객체가 생성되어 반환된다.
console.log(inst); // {}

// 객체를 반환하는 일반 함수
function createUser(name, role) {
  return { name, role };
}

// 생성자 함수로서 정의하지 않은 일반 함수를 new 연산자와 함께 호출
inst = new createUser('Lee', 'admin');
// 함수가 생성한 객체를 반환한다.
console.log(inst); // {name: "Lee", role: "admin"}
```

뉴를 안붙이는 경우 : 일반함수로 호출 한거. 그냥 일반함수 하듯이 선언문 먼저 찾는다.

뉴 타겟

es6 문법이다.

es6 문법인데. 쩜 있으니까 이게 객체고 이게 프로퍼티 같지만 전혀 그렇지 않다. 여기서 여기까지 그냥 이름이다.

예를 들어서

```
// 생성자 함수
function Circle(radius) {
  // 이 함수가 new 연산자와 함께 호출되지 않았다면 new.target은 undefined이다.
  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());
```

뉴.타겟 : 함수 객체를 가리키고 있다. 그래서 빈객체가 아니라는거다. 뉴와 함께 호출하지 않으면 언디파인드가 된다.

```
// 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
```

방어코드가 있을까 없을까. 있다. 오브젝트도 뉴 오브젝트에서 빈객체 만들수 있었죠.

펑션 생성자 함수도 ...

문제는 얘네들이야. 빌트인 생성자 함수(Object, String, Number, Boolean, Function, Array, Date, RegExp, Promise 등) 얘네들.

행동들이 일관되자 않다.

```
let obj = new Object();
console.log(obj); // {}

obj = Object();
console.log(obj); // {}

let f = new Function('x', 'return x ** x');
console.log(f); // ƒ anonymous(x) { return x ** x }

f = Function('x', 'return x ** x');
console.log(f); // ƒ anonymous(x) { return x ** x }
```

위처럼 띄지 말고 붙여줘야 한다.

디스를 안보는 메소드를 정적 메소드라고 한다.

예습 : 18, 19