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.

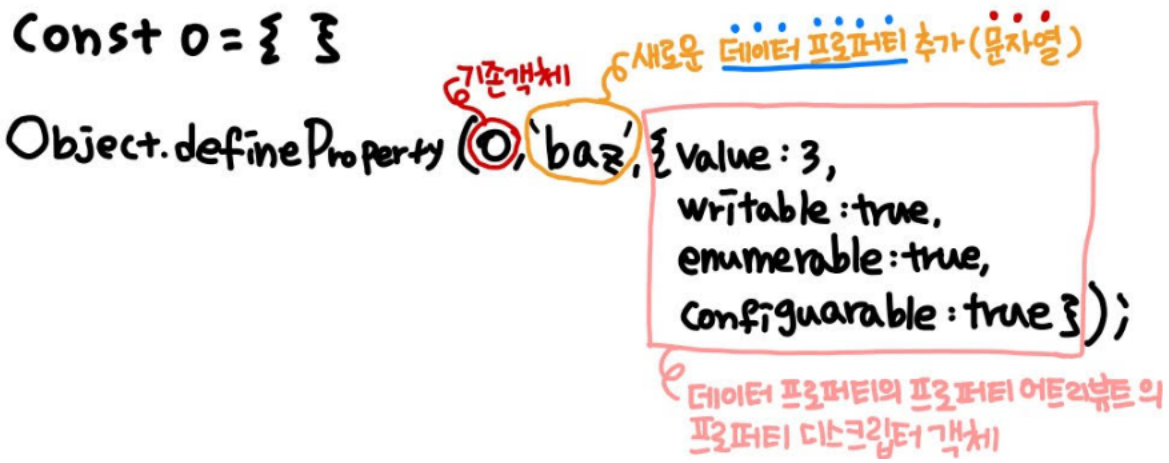
```
Object getOwnPropertyDecriptor(Object.prototype, '__proto__');
// {get: f, set: f, enumerable: true, configuarable: true}

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.

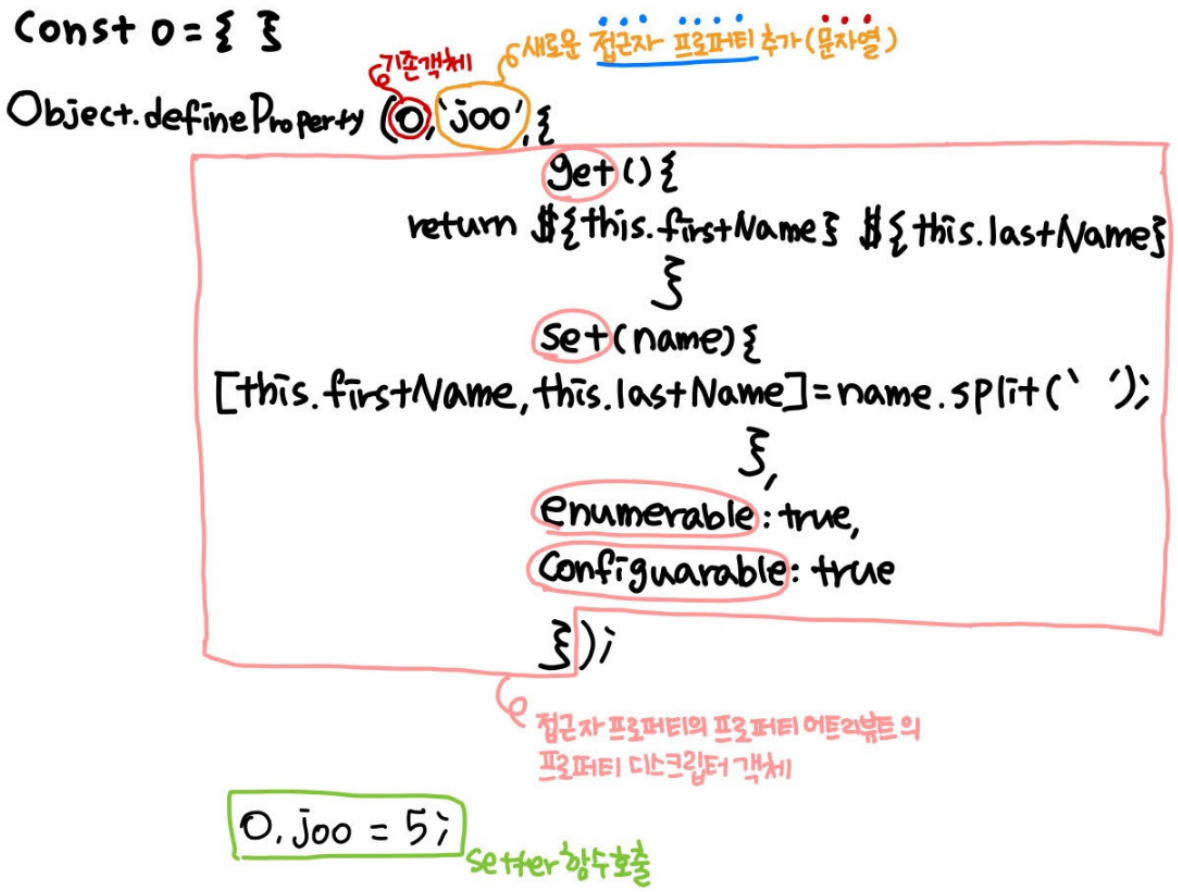
데이터 프로퍼티 정의



- 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

우리가 객체리터럴을 통해서 객체 만드는 방법은 잘 배웠다. 너무 쉽잖아.

객체 리터럴 을 통해서 객체 만드는건.

다른 언어들 얘기를 좀 하면, 객체 지향언어들 얘기를 하면 시쁠쁠이나 자바 이런 언어들은 기본적으로 객체를 만들때 리터럴을 통해서 만드는 방법이 없다. 다 어케 만드냐면, 클래스라는걸 선 정의하고 객체 만들기 전에 클래스라는걸 먼저 만들어야 한다. 함수를 호출하기 전에 함수를 정의 하듯이, 객체를 만들기 전에 그 객체에 객체를 만들수 있는 클래스라는걸 미리 정의해야 한다. 그 다음에 클래스를 뉴 연산자와 함께 호출한다. 그럼 객체가 만들어진다. 그 객체를 인스턴스라고 부른다. 그러면 왜 그 언어들은 그런식으로 할까?

왜 자스는 객체 리터럴이란 방법을 제공할까? 그냥은 없다. 이유가 있다. 그럼 여태까지 우리가 클래스라는 개념 모른다치고, 객체 리터럴 쓰면서 느끼는게 있나? 객체 리터럴이 어떤 느낌이냐? 엄청 편한거다. 객체를 런타임에 만들수 있다. 클래스 함수를 미리 정의 해놓고 그 함수를 호출해서 객체를 만드는 방법이 있는데 그게 클래스인데 얘네는 런타임에 만들수 없고 미리 정의해야 한다.

그럼 클래스라는걸 생각해볼 필요가 있는데, 걔네는 왜 그런식으로 할까? 왜 클래스얘길 하냐면, 자스에서는 클래스가 생성자 함수다.

객체 리터럴은 무슨 문제점이 있지? 객체가 여러개 만들어질때 객체 리터럴이 조금 거시기 해. 왜냐면 첨부터 다 써야 하니까. 그리고 객체는 메소드 갖고 잇는데, 객체 여러개 잇을때 메소드는 거의 안바뀌지만 상태 데이터만 거의 바뀐다. 그래서 우리가 100개의 객체 리터럴 만드는 것은 불합리하다.

그래서 객체를 만들어내는 함수를 만들어야 한다. 그래서 그 함수를 호출하면 객체가 툭툭툭 만들어진다.

따라서 객체는 만드는 방법이 여러개 있다. 객체 리터럴로만 만들수 있는게 아니다.

객체 리터럴로 만드는 방법이 있고, 오브젝트 생성자로 만드는 방법이 있고 생성자 함수라는 것으로 만드는 방법이 있고 오브젝트.크리에이트 라는 메소드를 써서 만드는 방법이 있다.

오브젝트 생성자 함수

아까 오브젝트. 어쩌고 저쩌고. 겟오운프로퍼티디스크립터.

근데 얘는 정체가 뭐다? 함수다. 함수인데, 뭐하는 함수다? 생성자 라는 말이 붙으면 얘의 존재 목적이 뭐냐면 객체를 만들어내는 목적을 가지고 잇는 함수다. 생성자 함수는 어케 호출하냐면, 우리가 익히 알고잇는 함수 호출방법으로 호출하는게 아니라, 뉴 라는 연산자를 앞에 꼭 붙여줘야 한다. 문법이다. 붙여주고 안붙여주고 큰 차이가 있다. 자스는 뉴 안붙여줘도 에러가 안난다. 그러면서 내부 동작이 이뤄진다.

```
// 빈 객체의 생성
const person = new Object();

// 프로퍼티 추가
person.name = 'Lee';
person.sayHello = function () {
  console.log('Hi! My name is ' + this.name);
};

console.log(person); // {name: "Lee", sayHello: f}
person.sayHello(); // Hi! My name is Lee
```

오브젝트는 전역객체에 있다. 오브젝트 앞에 윈도우. 이 숨겨져있다. 전역객체는 브라우저가 켜지자마자 그 안에 자스가 제공하는 빌트인 객체가 주루룩 있다.

우리가 코딩하는 시점에는 반드시 오브젝트 가 있다. 생성자 함수의 이름이 오브젝트인거 보니까 얘는 객체를 만드는 애다 . 근데 인수가 없으면 빈객체를 만든다. 빈객체 만들고 프로퍼티 추가해주면 객체 만들수 있다. 그러면 이렇게 객체를 만들 필요가 있을까? 빈객체 만드는 방법이 얘밖에 없나?

객체 리터럴로 빈객체 만들어줘도 되고, 만약에 빈객체 만들 필요가 왜 있지? 그냥 이것들을 프로퍼티 안에다가, 객체 리터럴 안에다 쓰면 되잖아. 그럼 빈객체가 왜 있어야 할까? 유용하지 않다. 만들일 없다.

오브젝트도 생성자 함수지만, 이거 말고도 자스에는 생성자 함수가 여러개 있다.

대표적인게 스트링, 넘버, 불리언, 펑션, 어레이, 데이트, regexp 라는 정규표현식 등등이 있다.

문자열이 아닌것을 문자열로 바꾸는 방법중에 우리가 선택한건 뭔가? + 빈 문자열

숫자가 아닌것을 숫자로 바꾸는건? + 단항연산자를 바꾸고 싶은 값 앞에 둔다.

그럼 불리언이 아닌것을 불리언으로 바꾸는건? 느낌표를 두개 붙인다.

스트링이라는 생성자함수 앞에 뉴를 붙이고 문자열을 주면, 이 문자열을 내포하고 있는 문자열 객체라는걸 만든다. 문자열도 객체가 잇다는 얘기네? 객체가 있다.

숫자 객체라는게 있다는 말이고, 불리언 객체라는게 있다는 말이다.

함수도 펑션 생성자 함수로 만들수도 잇다고 햇지만 무슨 문제점이 있다고? 클로저를 안만들고 내부 동작이 틀려서 이건 쓰지 말라고했다.

그럼 지금 생성자 함수를 자스가 제공하는게 있고, 우리가 따로 안만들고 들어가면 그냥 있는 그런애들을 빌트인 이라고 한다.

그럼 여기서 말하는 생성자 함수는 빌트인이 아니라 우리가 만드는거다.

객체 리터럴로 객체를 생성했을때 문제점이 있다.

생성자 함수

```
// 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
```

생성자 함수

객체 리터럴에 의한 객체 생성 방식의 문제점

```
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
```

위 코드는 원을 나타내는것이다. 원에 여러가지 프로퍼티(속성)들이 잇겟지만 우리가 관리하고싶은 속성은 반지름이다. 반지름을 알면 지름도 알수 있고, 넓이도 알수 있고, 원주의 둘레도 알수 있고, 알수 잇는게 많다. 그래서 반지름을 관리하겠다.

그리고 메소드가 하나 있는데, 겟다이아미터라고 하고, 2 곱하기 반지름 한거잖아. 그럼 이건 뭐지? 지름을 나타낸거다. 이렇게 객체를 만든거다.

그런데 비스무리한 객체를 하나 더 만들 필요가 있다. 반지름만 다르다. 프로퍼티 값만 다르다. 메소드는 거의 다 동일한 경우가 많다. 왜냐면 메소드는 뭐에 대한 행동이죠? 대부분 프로퍼티가 다른 행동이다. 얘를 조작하는 행위를 하는거다. 그럼 조작하는 놈들이 객체만 다르고 똑같기 때문에 메소드는 거의 다 비슷하다. 값만 바뀌는거다. 근데 위 코드는 고정값이 바뀐거잖아. 고정값이 바뀌었으니까 리터럴을 쓸때 고정값을 여기다 주어야 얘를 지정할수가 있겠지. 그럼 이게 늘어날수록 똑같은 것들을 계속 써줘야 한다는 얘기다. 불합리하다는 거다.

그래서 프로퍼티 구조가 똑같은 객체라면 그것을 함수로 만들수 있는 방법을 제공하는데 그게 바로 생성자 함수라는거다.

생성자 함수가 왜 필요한지 생각해보자.

멍멍개를 본적있나? 우리가 생각하는 개는 다 틀리다. 그럼 개가 뭔데? 다 틀리다. 그러면 개의 이상이 있는거다. 개의 원형이 잇는거다. 그게 뭘까? 예를 들어서 개는 다 다른가? 특징들이 있을거 아냐. 예를 들어서 짓는 소리가 멍멍해야해. 야옹하면 안 되. 그리고 색깔은 어떤 색이다. 그런것들을 미리 정해놓을수 있을까 없을까?

예를 들어서, 나열해보자고.

털색깔 , 짓는 행동, 다리의 개수, 꼬랑지가 있다없다 트루 퓔스. 이런식으로. 그래서 걔를 클래스로 만드는거다. 즉, 틀을 만드는것이다.

 그럼 개의 실체는 우리가 언젠가 본 개인거다. 집에 있는 개일수도 있고. 걔네들은 그 틀로 인해서 만들어진 실체다. 클래스는 이데아 같은거다. 개하면 개의 이데아가 잇는거다.

그러면 이데아의 실체도 있는거다. 걔네들을 프로그램세상에서 이데아를 클래스라고 하고,

클래스를 통해서 만들어진 애들을 인스턴스라고 한다.

실체를 인스턴스라고 부른다. 여기서 클래스라고 하는것은 클래스 기반 언어를 말하는거고, 자스는 클래스를 생성자 함수라고 한다. 클래스도 함수다.

자 그러면 우리 사람의 이데아를 만들어보자. 아직 실체는 없고 사람은 이런거야. 라고 정의를 해보자.

예를들어서, 함수라고 했으니까 펑션 키워드 쓰고, 함수 선언문을 만들어보자. 여기다 사람. 이라고 쓰자.

(메모장 보기)

생성자 함수라는 다른 문법이 있는게 아니고, 일반 함수랑 똑같이 생겼다. 그래서

생성자 함수로 호출할때 다르게 동작하고, 일반 함수로 호출할때 다르게 동작한다. 따라서 얘는 생성자 함수로 만든거라고 개발자들한테 알려줘야 한다. 그래서 컨벤션을 파스칼케이스를 쓰면 생성자 함수라는 거다. 그러고 함수 바디 만든다.

생성자 함수로 만든것이다 라는걸 알리기 위해 함수 이름의 첫머리를 대문자로 한다.

이때 여기서 디스라는 걸 써줘야 한다.

여기서 this 는 문맥에 따라서 가리키는 값이 달라진다. 그 문맥이 6가지 정도 된다.

오늘은 그 6가지 중 3가지를 배울거다.

생성자 함수 내부안에서의 디스는 이 생성자 함수가 만들어낼 인스턴스를 가리킨다.
<- 이건 그냥 외우자.

생성자 함수 내부안에서의 디스는 얘가 언제 객체를 만드는 인스턴스를 만들거 아냐, 그 인스턴스를 가리킨다. 미래에 만들겟지. 이것을 우리가 디스라고 쓰고있을때는 인스턴스를 아직 안만든거다.

그 디스는 미래에 만들어낼 인스턴스를 가리킨다. 디스 뒤에 프로퍼티를 쓴다. 예를 들어서 name이란 프로퍼티가 필요하다. 그러면 이건 무슨 뜻이야? 얘가 나중에 인스턴스를 만들거잖아. 그걸 얘가 가리킨다고 햇자나. 거기에 프로퍼티를 담는거다. 동적추가. 그러면 할당이 있어야 하잖아. 이렇게 하면 참조고, 할당을 해줘야 하는데, 이렇게 정해질수도 잇다. 세상의 인간은 다 이씨다. 디스.네임 뒤에 'Lee' 라고 써야 하는 것이다.

객체 리터럴 내부는 프로퍼티를 , 로 구분하지만 함수 바디 안에는 문들이 오니까 세미콜론으로 끝내줘야 함.

이 객체를 이 함수로 호출하면 이름이 이 이고 나이가 20살인 객체들만 호출된다.

10개고 100개고.

만들땐 어케 한다고? 반드시 뉴를 붙히고, 함수를 호출하는거다.

그럼 뭐가 리턴되? 얘의 리턴값이 뭐라고? 보니까 리턴이 업네? 그럼 언디파인드겠네? 근데 아니다.

왜 아닐까? 뉴 때문에 아닌거다. 그러면 뉴를 띄면 언디파인드. 뉴를 붙히면 언디파인드가 아니다. 뉴가 중요한거다. 뉴가 암묵적인 역할을 해주는거다.

뉴를 붙히면 이 함수를 어케 호출했다라고 하냐면, 생성자 함수로서 호출했다는 뜻이다.

그러면 펄슨을 생성자 함수가 아닌 일반함수로서 호출할수도 있다는 얘기다.

뉴를 안붙히면 일반함수로서 호출햇다는 거다.

즉, 이 함수는 2가지의 가능성을 갖고 잇는데, 생성자 함수로 호출될수도 잇고, 일반함수로 호출될수도 있다는 거다. 그래서 헷갈리니까 이렇게 쓴다는 말이다.

이게 중의적인 의미를 가지고 있는거니까, 반드시 실수를 한다.

그러면 이렇게 호출하면 안되는거다. 실수를 하니까 방어코드가 있어야 한다.

어떤식으로 방어코드가 있어야 할까? 얘가 뉴와 함께 호출되엇는지, 안되었는지를 알아차려야 한다. 그게 바로 뉴 타겟이라는 거다.

메소드도 하나 추가해볼까? sayHi.. (메모장 보기)

이름을 어케 참조할까? 왜 참조할까? 메소드는 누누히 얘기하지만 프로퍼리를 조작하거나 참조하는 애들이다. 그러니까 얘가 이 함수내부에 접근할수있는 방법이 있어야지, 조작하거나 참조할수 잇는거잖아. 그럼 어케 접근할래? 이거 얘기 하기 전에 다른 얘기를 해보자.

(메모장보기)

아~ 객체 리터럴 내부에서 디스는 자신이 이 메소드가 포함되어져있는 객체를 가리키는게 아니고 이 메소드를 호출한 객체를 가리키는거다.

객체 리터럴 내부에서 디스는, 그 메소드를 호출할때 쩜 찍었을거잖아. 그럼 쩜 앞에 잇는거는?

 호출할때 쩜을 썻잖아. 쩜 앞에 잇는거다.

함수몸체는 호출될때 실행된다. 호출될때마다 실행된다. 이 함수는 호출될때마다 만들어진다. 그니까 모든 인스턴스들은 내용이 같은 쎄이하이 메소드들을 중복소유한다. 그럼 어카지? 방법은 이 함수를 펄슨의 부모에게 준다. 상속으로 한다. 부모가 저거 하나 가지고 잇으면 자식들이 다 갖는다.

생성자 함수에 의한 객체 생성 방식의 장점

여러개 만들어야 한다. 몇개? 하여간 여러개. 그렇단 얘기죠.

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

여기서 디스는 누굴 가리킬까요, 외우세요. 생성자 함수 내부에서 디스는 누구다? 생성자 함수가 생성할 인스턴스다. 객체 리터럴 내부에 메소드 내부에서 사용한 디스는 그 메소드를 호출할 객체다. 그 메소드가 소속한 객체가 아니고

일반함수에서 디스를 쓸수 있을까? 일반함수라는 것은 기존에 우리가 호출한거. 쓸수 있을까? 왜 있지?

원래 쓰면 안된다. 디스는 왜 쓰지? 디스는 원래 객체지향에서 의미가 있는거다.

일반함수에서 디스 왜써? 의미 없는거다. 근데 자스는 디스가 있어야 함.

일반함수에서 디스는 무조건 전역객체다.

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

디스가 가리키는 값이 중요하다

우리가 함수라고 불리는 것이 일반 함수로서 호출될때가 있고 메소드로서 호출될때가 있고 생성자 함수로서 호출될때 장면이 잇다.

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

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

위에 보면 일반함수같다. 여기서 디스는 전역객체다.

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

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

// 메소드로서 호출
const obj = { foo }; // ES6 프로퍼티 축약 표현
obj.foo(); // obj

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

생성자 함수로서 호출했다는 것은 뉴와 함께 호출했다는 것이다.

```
// 생성자 함수
function Circle(radius) {
  // 인스턴스 초기화
  this.radius = radius;
  this.getDiameter = function () {
    return 2 * this.radius;
  };
}

// 인스턴스 생성
const circle1 = new Circle(5);  // 반지름이 5인 Circle 객체를 생성
```

그럼 우리가 명시적으로 리턴하면 어케 될까? 그래서 어떤 객체를 리턴했다고 하면 우리가 명시적으로 리턴한 객체가 된다. 그러니까 주의해야 한다.

원시값을 리턴하면 무시당한다.

내부 메소드

new : 연산자다.

얘 뒤에는, 우항에는 함수가 온다. 생성자 함수라는 문법이 있나요? 생성자 함수라는 문법을 지키면 생긴다는 식의 문법은 없다. ... 로도 될수 있다.

따라서 일반함수는 일반함수라는건 함수만드는거 몇가지?

펑션 푸()

콘스트 푸 = 펑션() {}

위 두개는 생성자함수도, .. 도 호출할수 있다.

콘스트 푸 = ( )=> {})

위 하나는 생성자 함수로 호출 못한다.

우리가 일반적으로 메소드라는거. 객체 지향언어에서 다 메소드라고 한다.

우리가 메소드라고 통상 부르는거랑 에크마 에서 부르는 메소드는 다른거다.

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

에크마에서 메소드라고 지칭하는건 생성자 함수로 호출할수 없다. 메소드를 왜 생성자 함수로 호출하냐고~ 당연한거다.

함수들의 종류가 있다.

생성자 함수로 호출할수 있고 없는 함수가 있다.

생성자 함수로서 호출 가능한거 함수 선언문, 함수 표현식

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.

그런데 함수만의 특징을 갖고잇다. 그 특징은 쉽게 말해서 호출할수 있다.

객체는 호출할수 없다

함수를 호출하면 함수가 내부적으로 [[call]] [[construct]] 얘네 내부 메소드다

함수 객체를 가리키는 식별자 ( )

위 두개를 호출하는거다. 내부동작이.

어쩔때 콜, 어쩔대 콘스트럭 호출하느냐.

함수가 함수 선언문이랑 함수 표현식이 잇는데 [[콜]] [[콘스트럭트]] 둘다 가능하다. 둘다 가지고 있다.

근데 콘스트럭트 화살표함수는 콜을 갖고잇다. 콜을 안갖고잇으면 함수가 아니다.

근데 콘스트럭트는 있는애 가 잇고 없는 애가 있다. 없는 애를 논 컨스트럭트 , 있는 애를 콘스트럭트라고 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 메소드는 콘스트럭트다. 뉴와 함께 호출할수 있다.

그러면 콜 메소드도 갖고 잇고 내부에서 콘스트럭트고 갖고 있어서 함구 객체가 크다는 얘기다.

이렇게 저렇게 호출할수도 있다. 쓸데없는 기능을 갖고잇다는 얘기다.

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

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