프로퍼티 어트리뷰트

내부슬롯과 내부 메소드라는 용어가 있다.

에크마스크립트 엔진을 구현할때 참고하라고 준 표준 문서인 에크마스크립트 사양서에 잇는 이름이다.

이름에서 알수 있듯이 내부 슬롯과 내부 메소드는 "내부"라는 이름에서 알수 있듯이 자스엔진이 내부에서 사용하는 데이터다.

그런데 일부는 우리 개발자들한테 노출해준게 잇다. 기본적으로는 다 감춘다.

여기서 헷갈리는게 뭐냐면 우리가 코딩한건 결국 뭐라고 하냐? 파일로 만들거 아냐.

순수한 텍스트 문서일것이다. 이렇게 자스파일을 만들거다. 통상 이걸 뭐라고 하냐? 소스 코드라고 한다.

소스는 재료가 되는 코드라고 한다. 뭐에 재료가 되는 코드?

이렇게 쓰면 바로 실행되는게 아니다. 우리가 쓴건 자스엔진 입장에서는 외국어다. 못알아먹는다. 못알아먹는 걸 어케 해줘야 하냐? 통역을 해줘야 할거 아냐.

인터프리터 라는 프로그램이 있는데, 얘가 저걸 통역해줘야 한다. 그럼 인터프리터 라는 프로그램을 우리가 깔았냐? 그런 프로그램이 어디있냐. 브라우저에 내장되있는 거다.

브라우저를 깔면 알아서 깔려지는거다. 있는거다.

그러면

인터프리터 : 말그대로, JS를 기계어로 바꿔주는 거다. 자스엔진이 이해할수 있는 언어로 바꾸는 애다.

인터프리터만 있으면 안된다. 그렇게 단순하지가 않다. 우리가 코딩한건 서버에 갖다놓을건데 그럼 서버에 있는걸 가져와야 한다. 왜냐하면 얘네는 브라우저에서 실행되기 때문에. 그러면 가져오는 애도 있어야 한다. 그 애를 로더 라고 한다.

브라우저 안에 내장되있는 프로그램이다.

문자열에서 얘를 해석해야 한다. 토큰으로 죄다 쪼개서 그것을 하나의 자료구조로 만들어야 한다는 것이다.

그게 asp 라는 자료구조를 만드는데, 그걸 인터프리터한테 준다.

그러면 asp를 만드는 걸 뭐라고 할까? 파서라고 한다.

로더, 파서, 인터프리터 얘네들은 브라우저 안에 내장 되있는 애들이다.

그럼 여태까지 우리가 에크마스크립터 사양서는 누구를 위한거라고 할수있나?

로더, 파서, 인터프리터 나왓는데.

우리가 얘기하는 자스엔진은 누구 소속일까? 파스오브 소속이다. 실행하는거에는 신경 안쓴다. 해석하는거에만 관심이 있다.

여기서 우리가 만드는 자스코드들이 결국엔 해석되어져서 자스어로 갈거다.

그러면 아까 얘기했던 브라우저 안에잇던 프로그램들은 자스일까? 아까 로더도 있고 파서도 잇고, 인터프리터도 있다고 햇는데 얘넨 자스일까?

왜 에크마스크립터라는 표준 문서를 왜 만들었을까? 그걸 안만들면 각자 알아서 만들게 되는데, 그렇게 되면 달라질거다. 호환이 안될거다.

완전히 호환되는게 아니라 적당히 호환되는 문제가 발생한다. 그러면 결국은 적당히 호환되는건 전혀 호환안되는거나 마찬가지다.

그래서 에크마스크립트 1버전이 나온거다.

에크마스크립트 사양서는 엔진 만드는 사람들한테 만들어준거라서 우리는 뭔말인지 모르고, 어떤 부분은 되게 자세하게 설명해주고, 어떤 부분은 되게 추상적으로 설명한 부분도 있어서 어렵다. 그래서 자스를 만들었다고 해서 그냥 동작하는게 아니라 자스엔진이라는 애가 자스를 해석할거란말이지.

그 해석하는 자스엔진도 결국엔 프로그램이다. 별도의 프로그램인것이다. 별도의 프로그램이 얘넬 읽어들여서 일을 하는거다. 걔네는 자스로 안만들어졌다. 느리기 때문에. 씨쁠쁠로 만들어진 자스엔진이 자스를 해석하는 것이다.

그래서 지금부터 얘기하는 애들을 자스랑 연관지으면 안된다. 씨쁠쁠로 만들어진 자스엔진 내부의 얘기인것이다.

씨쁠쁠은 객체지향 언어다. 얘네도 객체가 있을거다. 걔네도 객체가 있으니까 걔네도 프로퍼티 만들고 메소드 만들어서 일을 내부에서 할거 아냐?

그걸 어케 만들어라 하라는게 에크마스크립트 사양서인것이다.

그래서 우리가 코딩을 할때

의사코드? 수도. 짝퉁이란 것이다. 코드가 아닌데 코드 비스무리하게 대충 쓰는 짝퉁 코드.

언어를 정확히 모르기때문에 의사코드를 써라. 정 모르겠으면 한글로 써라. 이런식으로 연습해야 머리가 로직화해준다.

예를 들어서 메소드를 만들어야 하면 얘네들이 메소드의 이름을 지어준다. 이름을 불러줘야 할거 아냐. 가상적으로 메소드 이름을 만들어준다. 그래서 이 메소드를 어떻게 동작해라. 다른데서 호출해라 이런식으로 한다.

그래서 거기서 메소드를 내부 메소드라고 부른다. 객체 지향 언어니까 프로퍼티가 있을거 아냐. 거기서 내부에서 쓰는 프로퍼티를 내부 슬롯이라고 하는 것이다.

그러면 내부 슬롯과 내부 프로퍼티는 자스엔진을 만들때 필요한 것들이다.

기본적으로 내부 슬롯과 내부 프로퍼티는 비공개다. 안 알려준다. 씨쁠쁠로 만들어진 프로퍼티와 메소드는 우리에게 안 알려준다.

근데 우리가 코딩을 할때 여기서 일부 필요한것들이 잇다. 그것들은 자스코드로 우리에게 접근할 방법을 간접적으로 제공한다.

그래서 에크마스크립트에서 기술 안된것들은 예를 들어서 숫자라는 것을 만들때 부동소수점 배정밀도 로 만들어라 라고만 써져잇다. 그리고 불리언이라는게 잇다. 그건 트루 풜스 두개다. 그럼 불리언 몇바이트 써야 해? 모르는거다.

에크마 스크립트 기술서 보면 대괄호 두개 쳐놓은게 있는데 이것들이 내부 슬롯 또는 내부 메소드다.

우리가 왜 이 용어들을 알아야 하느냐, 예를 들어서, 클로저라는게 뭐냐면 함수는 태어날때, 함수 객체가 만들어질때 상위 스코프를 기억한다. 기억한다는 건 정확히 말하면 함수는 객체이므로 프로퍼티가 있는데, 예를 들어서 펑션 푸() 이거 함수잖아. 푸. x = 2 이렇게 써도 된다. 함수니까 메소드 할수 있죠? 함수가 함수를 갖고 있어. 객체니까. 이게 중요한게 아니고, 함수는 객체이므로 프로퍼티가 있다. 아까 얘기했던 자기가 태어날때 에크마 스크립트 사양서를 보면 [[en]] 여기에 상위 스코프의 참조 기억을 저장하여라. 라고 쓰여있다. 즉, 저 대괄호 안에 있는건 내부 슬롯이다.

이 얘기는 곧 모든 함수 객체는 내부 슬롯 인바이러넘먼트를 갖고 잇다. 여기에 자신의 상위 스코프를 저장한다. 라고 설명해야 한다.

그러면 내부 슬롯이 뭐에요? 감춰진 프로퍼티 라고도 말할수 잇다. 그럼 감춰진 프로퍼티가 뭐에요? 라고 할수있다.

이 푸라는 함수는 내부슬롯을 갖고잇다는 거잖아. 그러면 이렇게 자스문법으로 접근할수 있을거 같아? 없을거 같아? 이게 프로퍼티 키냐고. 이건 그냥 의사코드일뿐이다. 이건 정식명칠일뿐 이런식으로 자스를 사용할수 있는 프로퍼티 키가 아니라는거다. 그래서 접근할수 없다. 원래 접근할수 있는 방법이 없는데 일부 프로퍼티나 내부 슬롯이나 내부 메소드에 대해서는 우리가 접근해야 할 경우가 잇을 수도 잇다.

걔네들은 자스로 우리에게 알려주는 프로퍼티 또는 메소드가 존재하는 경우가 일부 있다.

예를 들어서

const o 가 있다. (메모장 보기 5월4일꺼)

객체 리터럴 안에 푸라고 하는 프로퍼티를 만들고 이 안에다가 1 이라는 값을 할당 햇다.

우리가 여태까지 불렀던 프로퍼티는 프로퍼티 키, 프로퍼티 값의 쌍인데, 프로퍼티라는 것도 하나의 객체잖아. 실체가 잇을거잖아. 사실은 객체 내부에 이런 객체가 또 있는 거다.

얘네들은 언제 만들어지나? 프로퍼티를 하나의 얘가 가지고 잇는 조그만 객체라고 생각해봐라. 객체라는 큰 덩어리가 있는데, 그 안에 객체가 잇을수도 잇잖아. 그렇게 갖고있다고 가정해봐라.

그럼 언젠가 얘네도 만들어지겟지.

일부 내부 슬롯과 내부 메소드에 관해서는 우리가 접근하는 일이 있을수도 잇다. 걔네는 자스로 우리에게 알려주는 프로퍼티 또는 메소드로 존재하는 경우가 있다.

예를 들어서 객체 내부에 이런 객체가 또 있는거다.

= 이거 하기 직전이다.

무명리터럴: 이름이 없다라고 한다.

무명 리터럴의 의미가 뭘까? 이름이 있고 없고. 이름이 없다라는 것은, 이름이 없는 리터럴 문법이 있다는 것은 얘가 어디에 할당된다는 뜻이다. 이름이 붙지않는 리터럴이 문법이 존재한다는 것은, 그 리터럴로 만들어진 무언가가 반드시 변수에 할당된다는 전제를 갖고 있는것이다. 아니면 무명 리터럴이란 문법이 있을수가 없다.

무명리터럴이 가능하다는 얘기는 뭐와 동일한 얘기냐면 런타임에 생성된다는 뜻이다. 무명리터럴이란 얘기는 런타임에 생성된다는 뜻이다.

프로퍼티는 객체가 생성될때 만들어짐. 그 객체는 언제 생성되어지냐면 이 변수에 할당되기 직전에 만들어진다. 얘가 만들어져야 객체가 완성되는 것이다.

자 우리는 얘를 지금까지 프로퍼티라고 불럿든데 얘는 특징을 가지고 잇다.

이 프로퍼티는 어트리뷰트가 있다는 말이다. 엄격히 얘기하면 자산이라고 말해야 한다.

프로퍼티: 프로퍼티는 어트리뷰트를 가지고 있다. 프로퍼티는 가지고 있는 자산이라고 해석해야 한다.

프로퍼티들은 속성이 있다. 즉 어트리뷰트가 있다는 얘기다. 흡사 객체와 같은 애다.

내부적으로 4가지를 갖고 있는데 첫번째로는 벨류라는걸 갖고 잇다.

프로퍼티가 2개가 잇는데 데이터 프로퍼티 / 접근자 프로퍼티 라는게 있다.

우리가 지금까지 경험한 프로퍼티는 데이터 프로퍼티 라고 한다.

데이터 프로퍼티의 특징은 왼쪽에 값이 온다.

데이터 프로퍼티는 4가지의 어트리뷰트를 가지고 잇다.

1. [[벨류]] : (프로퍼티 값)을 가리킨다.
2. [[라이터블]]: 쓸수 있는가 라는 얘기다. 이 벨류 프로퍼티의 값을 쓸수 있느냐 없느냐 라는 말이다.(쓸수 있다는 말은 갱신할수 있다는 말이다), 쓸수 없다는 얘기는 리드온리하다고 하는거다. 그럼 얘는 어떤 값을 가지고 잇을까? 이거 다 내부슬롯이다. 얘는 값이 뭘까? 내부슬롯은 프로퍼티라고 생각하면 된다. 값이 잇다는 거다. 얘의 값은 뭘까? 콜론 뒤에 써준 게 값이 내부에 들어온다는 거다.기본적으로 트루 풸스를 갖고 있다.

   우리가 객체 리터럴로 만들었을때 모든 프로퍼티는 다 트루다. 객체 리터럴로 만들면 내부적으로 이 값을 다 트루로 갖고 잇다. 그래서 우리가 갱신이 가능한것이다.
3. [[이뉴머러블]] : 열거할수 있는가 라는 얘기다. 열거할수 있느냐라는 뭔 얘기냐면 객체에 프로퍼티들이 존재하지? 이 프로퍼티들을 포문에 통해서 나열이 할수 잇을까?? 배열의 경우에 순서대로 하나씩 루프를 돌면서 순회할수 있었지? 객체의 경우는 어떨까? 못 돈다. 왜냐면 렝스가 없어서 포문에서 못돈다. 그럼 이게 되려면 어케 하지? 두가지가 있다. 포 인문 이라는게 있고 오브젝트라는 객체에 키스(keys) 라는게 있다.

   포인문 : 얘는 포문과 유사한데, 뭐 전용이냐면, 객체에 프로퍼티를 열거하는 전용 문법 (근데 이거 쓰지마라.)

   객체의 키 : 객체가 가지고 있는 키. 여기서 키는 뭘 얘기하냐면 프로퍼티 키들을 뽑아서 배열로 리턴해준다. 그럼 그 배열로 가지고 순회하면서 키를 갖고 있어서 값을 땡겨올수가 있다.

   이 두가지의 방법으로 열거 할수 잇다
4. [[컴피규어러블]] : 설정할수 있다 라는 얘기다. 얘가 만약에 풸스면 어떤 일이 벌어지냐면, 이 객체 내에서 프로퍼티에 값을 갱신할수도 없고 쓸수도 없다. (읽기 전용이 되고 추가도 안되고 지울수도 없다)

기본적으로 이런것들을 갖고 잇다. 이걸 확인해보려면 어케 해야 할까? 일~4보니까 다 내부슬롯이다.

내부 슬롯에 접근할수 없지만 일부 필요한건 열어놨다.

열어놧다는게 다이렉트가 아니라 간접적인 방법으로 제공한다.

그게 대부분 메소드 아니면 접근자 프로퍼티다.

(메모장 보기) cosnt o = {

foo: 1

};

Object . 하면 뒤에 여러가지가 나오는데 얘네가 메소드 또는 프로퍼티일것이다.

아~! 오브젝트라는게 메소드 또는 프로프티를 갖고잇다는 얘기네. 그럼 얘는 뭐다? 객체다 라는 말이다. 근데 앞에가 대문자잖아. 대문자면 생성자 함수다. 뉴라고 호출하면 객체가 만들어진다.

아까도 얘기햇자나 푸라는 함수를 만들었을때 걔는 메소드를 갖고 있을수 있다고.

얘도 함수다. 생성자 함수다. 함수니까 객체다. Object 이름이 객체잖아. 객체에 대한 유용한 메소드들을 제공한다는 얘기다

여기서 우리가 사용하고 싶은게 뭐냐면 getOwnPropertyDescriptor 이걸 딱 보면 생각이 나야 한다.

(메모장보기)

키면 스트링 하고 심볼은 오케이. 넘버도 줄수 잇었죠? 왜? 아까 얘기했던거 처럼 유사배열은 넘버를 오케이 해주기 때문에 넘버도 받아준다.

우리는 키에 관심이 잇는거다.

프로퍼티 디스크립터라는 객체를 리턴할거다. 저 프로퍼티 디스크립터라는 객체는 무슨얘기냐면, 객체는 프로퍼티가 다 다르다. 그래서 어떠한 프로퍼티 구조를 가지고 잇느냐를 이 객체라고 부르자.

프로퍼티 디스크립터라는 객체를 리턴할거다. 그럼 이걸 콘솔로그로 확인해보자. 그럼 뭐가 나올거 같아?

느낌상. 이 4가지 프로퍼티 어트리뷰트 정보를 제공한다.

얘는 에크마스크립터에 잇는 내부슬롯이고, 얘는 그 내부슬롯에 대응하는 프로퍼티 키다.

아~ 내부슬롯 벨류에는 1이 들어가있고 내부슬롯 라이터블에는 트루 가 들어가잇고, 내부슬론 이뉴 에는 트루가 있고 컴뷰러블 에는 트루가 들어잇다라고 알수 잇다. 왜 다 자동으로 트루가 들어가잇지?

객체 리터럴로 프로퍼티를 만들면 기본 트루로 세팅이 된다.

이 얘기는 객체 리터럴로 만든 모든 데이터 프토퍼티는 읽기쓰기가 가능하고 열거가능하며 재정의 가능한다.

프로퍼티라는게 단순하지는 않단 얘기다. 자스엔진 내부에서는 프로퍼티가 어떻게 관리되고 잇다는거지?

프로퍼티 자체가 객체다. 내부 슬롯이 있잖아. 내부 슬롯을 프로퍼티 키라고 생각하면 된다. .... 하나의 객체로 다뤄진다. 그 ... 집합체를 객체라고 한다.

우리가 자스엔진을 만든다고 생각해봐라. 어떤 자료구조를 만들어야 하잖아. 지금까지 살펴본건 데이터 프로퍼티다.

프로퍼티 처럼 생긴 함수.

예를 들어서

```
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 프로퍼티 : 벨류가 있는 놈이다. 근데 접근자 프로퍼티는 벨류가 없고, 벨류 대신 겟 이란 애랑 셋 이라는 애가 있다. 겟 셋 다 프로퍼티 어트리뷰트인데 둘다 함수를 가지고 있다.

겟셋 없다고 생각하면 fullName 이 똑같다. 겟셋 두개가 게터세터이다.

프로퍼티는 두가지 용도로 쓴다. 참조할때 쓴다. 프로퍼티로 값을 참조할수 잇나? 어케 참조하나? 펄슨.펄스트 네임 . 이건 참조다. 가지고 오는거다. 즉. 겟하는 것이다.

펄슨.펄트스 네임 = 어쩌고 이게 셋이다.

접근자 프로퍼티는 .... 만들어준다.

접근자 프로퍼티는 4개의 어트리뷰트를 갖고 잇다.

1. 겟 : 게터는 결국 리턴이 반드시 있어야 한다
2. 셋 : 리턴이 반드시 없어햐 한다.
3. 이뉴머러블
4. 컨피규러벌

오운(Own): 상속은 제외하고..... 것만.

.... 프로퍼티 디스크립터 객체도 있다.

디파인 프로퍼티도, 데이터 프로퍼티를 만드는것이 가능하다는 것을 알앗고..... 만들수 잇는걸 알앗다.

객체 변경 방지

3가지 메소드가 있다.

오브젝트. 프리즈

이런게 있다. 만 알고 넘어갑시다.

지금은 몰라도 됨

객체를 동결한다. 왜 객체를 얼릴까? 메소드를 통해서 객체를 얼리면 모든 프로퍼티가 리드온니가 된다.

왜 얼릴까? 원시값처럼 주소가 바뀌지 않는 상황에서... 객체를 원시값처럼 쓰고자 할때에는 오브젝트.프리즈를 통해서...

객체를 바꾸려면 어케 해야 하나? 딥카피를 해야 한다. 객체를 새롭게 처음부터 만들어야 한다.

생성자 함수에 의한 객체 생성

다른 언어들 얘기를 좀 하면, 객체 지향언어들 얘기를 하면 그런 언어들은 기본적으로 객체를 만들때 리터럴로 만드는 방법이 없다 클래스라는걸 선정의하고 객체 만들기 전에 그 객체에 클래스라는 걸 미리 정의한다. 그 다음에 클래스를 뉴 연산자와 함께 호출한다. 그럼 객체가 만들어진다. 그 객체를 인스턴스라고 한다. 그러면 왜 그 언어들은 그런식으로 할까?

왜 자스는 객체 리터럴이란 방법을 제공할까? 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 만들어 놓고 ... 이게 클래스인데 얘네는 런타임에 만들수 없다.

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 바뀐다. 그러면 우리가 100개의 객체 리터럴 만드는 것은 불합리하다.

객체를 만들어내는 함수를 만드는 것이다. 그래서 그 함수를 호출하면 객체가 툭툭툭 만들어진다.

그러면 심플하게 만들어지는것이다.

따라서 객체는 만드는 방법이 여러개 있다.

객체 리터럴로만 만드는게 아니다.

객체 리터럴로 만드는게 있고 오브젝트 생성자 함수 로 만드는것도 있다.

오브젝트 생성자 함수

아까 오브젝트. 어쩌고 저쩌고.

근데 얘는 정체가 뭐다? 함수다. 함수인데, 생성자 라는 말이 붙으면 얘의 존재 목적이 뭐냐면 객체를 만들어내는 목적을 가지고 잇는 함수다. 생성자 함수는 어케 호출하냐면 뉴 라는 연산자를 앞에 꼭 붙여줘야 한다. 문법이다. 붙여주고 안붙여주고 큰 차이가 있다. 자스는 뉴 안붙여줘도 에러가 안난다.

```
// 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 전역객체는 브라우저가 켜지자마자 빌트인 객체가 주루룩 있다.

우리가 코딩하는 시점에는 반드시 오브젝트 가 있다. 생성자 함수가 오브젝트인거보니까 얘는 객체를 만드는 애다 . 근데 인수가 없으면 빈객체를 만든다. 빈 객체 만들고 프로퍼티 추가해주면 객체 만들수 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
```

플러스 빈문자열 하는게 좋다.

단항 연산자를...

느낌표를 두개 붙인다.

스트링 앞에 뉴를 하면 문자열 객체를 만든다. 문자열도 객체가 있다는 말이다.

숫자 객체, 불리언 객체라는게 있다는 말이다.

함수도 펑션 생성자 함수를 만들수도 있다. 이건 클로저를 안만들고 내부동작이 틀려서 쓰지 마라.

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
```

원을 나타내는것이다.

우리가 관리하고 싶은건 반지름이다. 반지름을 관리하겟다. 메소드가 하나있다. 비스무리한 객체를 또 만드려고 한다. 반지름만 다르다

이 메소드는 뭐에 대한 행동이죠? 대부분 프로퍼티에 대한 행동이다. 얘를 조작하는 애다. 위의 객체는 고정값을 갖고 있다. 그래서 리터럴을 쓸때 고정값을 ..... 똑같은걸 써줘야 한다는 얘기다.

프로퍼티 구좌가 똑같다면... 이게 생성자 함수라고 한다.

멍멍개를 본적있나? 우리가 생각하는 개는 다 틀리다. 그럼 개가 뭔데? 다 틀리다. 그러면 개의이상이 있는거다. 개의 원형이 잇는거다. 예를 들어서 개는 다 다른가? 특징들이 있을거 아냐. 짓는 소리가 멍멍해야해. 그리고 색깔은 어떤 색이다. 그런것들을 미리 정할수 있을까 없을까

예를 들어서

털색깔 , 짓는 행동, 다리의 개수, 꼬랑지가 있다없다 트루 펄스. 그래서 걔를 클래스로 만드는거다.

틀을 만드는것이다./ 그럼 개의 실체는 집에 있는 개일수도 있고.. 그 틀로 만들어진 실체다.

클래스를 통해서 만들어진 애들을 인스턴스라고 한다.

실체를 인스턴스라고 한다. 여기서 클래스라고 하는것은 클래스 기반이라고 하고 자스는 생성자 함수라고 한다.

자 그러면 우리 사람의 이데아를 만들어보자.

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

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

오늘은 3가지를 배울거다.

생성자 함수 내부안에서의 디스는 이 생성자 함수가 만들어낼 인스턴스를 가리킨다.

생성자 함수 내부안에서의 디스는 얘가 인스턴스를 만들거 아냐, 그 인스턴스를 가리킨다.

미래에 만들어낼 인스턴스를 가리킨다. 디스 뒤에 프로퍼티를 쓴다. 인간세상은 다 이씨면 디스.네임 뒤에 'Lee' 라고 써야 하는 것이다.

뉴를 붙히면 언디파인드, 띄면.....

뉴를 붙히면 생성자 함수로서 호출했다는 뜻이다.

그러면 펄슨을 생성자 함수가 아닌 일반함수로서도 호출할수 있다는 얘기다.

중의적인 의미를 가지고 잇으면 반드시 실수를 한다.

실수를 하니까 방어코드가 있어야 한다. 얘가 뉴와 함께 호출되엇는지 알아차려야 한다.

메소드는 프로퍼리를 조작하거나 동작하는 애다. 어케 접근할래?

객체 리터럴를 내부에서 쓸때 호출할때 쩜을 썻잖아. 쩜 앞에 잇는거다.

함수몸체는 호출될때 실행된다. 호출될때마다 실행된다. 호출될때마다 만들어진다. 그니까 모든 인스턴스들은 내용이 같은 쎄이하이들을 중복소유한다. 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

퀴즈퀴즈

이진검색은 반드시 정렬이 되었다고 가정하고 시작.

꼭 포문으로 안해도 된다. while문 써도 된다.

반띵반띵 해야 한다. 스타트와 엔드가 같아진다.

1. 타겟이 6.
2. 타겟이 배열에 요소로 존재하는지 확인
3. 존재하면 해당 인덱스를 반환
4. 존재 안하면 -1을 반환

```
function binarySearch(array, target) {

}

console.log(binarySearch([1, 2, 3, 4, 5, 6], 1)); // 0
console.log(binarySearch([1, 2, 3, 4, 5, 6], 3)); // 2
console.log(binarySearch([1, 2, 3, 4, 5, 6], 5)); // 4
console.log(binarySearch([1, 2, 3, 4, 5, 6], 6)); // 5
console.log(binarySearch([1, 2, 3, 4, 5, 6], -1)); // -1
console.log(binarySearch([1, 2, 3, 4, 5, 6], 0)); // -1
console.log(binarySearch([1, 2, 3, 4, 5, 6], 7)); // -1
```

```
function binarySerach(array, target) {
while (array[centerIndex] !== condition)
}
```