```
function foo() {}
```

함수 내부에는 디스라는 암묵적인 식별자가 있다.

```
console.log(this);
```

위 코드에서 디스하면.....

```
function foo() {}
console.log(this);
```

여기에서 디스는 뭘까? -> 정답은 모른다. 아직 결정이 안되었다. 동적으로 결정된다. 언제? 호출되었을때.

```
function foo() {
console.log(this);
    }
foo();
```

위 코드는 일반함수 호출했을때다. 전역객체다.

```
function foo() { // <- 빈객체 만든다.
console.log(this);
    }
new foo();
```

위 코드처럼 호출할수도 있나? new 는 연산자다.

위 코드처럼 호출하면.... 함수가 암묵적인 처리를 한다.

```
function foo() {
console.log(this);
    }
new foo();
```

위 코드처럼 호출하면... 콘솔로그 출력하면 foo{} 라고 나올것이다.

```
function foo() {
console.log(this);  // foo {}
this.name = 'Lee';
    console.log(this); // foo {name: 'Lee'}
    }
foo();
```

콘솔로그 디스하면 위 코드처럼 나온다.

... 반환되어질것이다.

```
function foo() {
this.name = 'Lee';
    }
const f = new foo();
```

위 코드의 생성자 함수의 문제점은? 이런식으로 생성자 함수 만들었다는 것은 의미가 없다.

프로퍼티의 값을 매개변수에게 전달해줘야 한다.

메소드 하나 추가해볼까?

```
function foo() {
    this.name = 'Lee';
    this.sayHi = function () {}
    }
const f = new foo('Lee');
```

생성자 함수에서 메소드는 일반함수다. 쎄이하이 앞에 뉴를 붙일수 잇다 없다?

```
function foo() {
    this.name = 'Lee';
    this.sayHi = function () {    // 여기서 this 만든다. 여기서 this는 빈객체(언디파인드가 나온다)
    console.log(`Hi! my name is ${this.name}.`);
    };
}
const f = new foo('Lee');
f.sayHi();
// Hi! my name is Lee.
```

위 코드는 메소드로 호출한것이다. 디스가 만약에 잇다고 가정한다면 위 코드처럼 된다.

.... 통일해서 위 코드라고 말할수 있다.

```
function Person(name) {
this.name = naeme;
}
const me = new Person();
```

계속 생성자 함수를 함수 선언문으로 만들고 있는거다.

위 코드는 콘스트 문을 펑션문 위로 올릴수 있나? 아래에 있는거와 차이점은? 똑같다. 펑션문은 런타임 이전에 생성되니까. 근데 표현식으로 하면 안된다.(아래 코드가 표현식으로 나타낸거다)

```
const me = new Person('Lee');
function Person(name) {
this.name = naeme;
}
```

```
function Person(name) {
this.name = naeme;
this.sayHi = function () {
    console.log(`Hi! my name is ${this.name}.`);
    };
}
const me = new Person('Lee');
const you = new Person('Kim')
```

위 코드에서 문제점을 찾아봐라.

... 여러개 만들어진다.

```
function Person(name) {
this.name = naeme;
this.sayHi = function () {
    console.log(`Hi! my name is ${this.name}.`);
    };
}
const me = new Person('Lee');
const you = new Person('Kim')
console.log(me.sayHi === you.sayHi); //false
```

위 코드 왜 퓔스일까? <- 이거 이해하기.

메소드는 같다. 프로퍼티만 틀리다. 그럼 프로퍼티는 무조건 틀려야 하나? 그건 아니다.

예를 들어서 이름이 있고 성별이 있다고 하자. 그건 괜찮다.

위 코드에서 중복을 방지하는 방법을 생각해내라.

객체 지향이라는 키워드 하에 생성자 함수를 배우고 있다. 월요일부터 나누자면 그 전은 자스 기초고,

월요일부터는 객체 지향 프로그램이다. 그럼 객체 지향 관점에서 생각해보자.

위 코드를 어떤식으로 처리하냐면, 상속을 하자.

상속은 내가 있고, 누가 있어야 해, 부모가 있어야 함. 자식은 됐어. 부모가 있어야 상속이 된다.

내가 자식이라고 생각하자. 부모가 잇어야 내가 상속받을수 있다.

```
function Person(name) {
this.name = naeme;
this.sayHi = function () {
    console.log(`Hi! my name is ${this.name}.`);
    };
}
const me = new Person('Lee');
const you = new Person('Kim')
console.log(me.sayHi === you.sayHi); //false

-------
//콘스트 문 포함 세개. 얘네가 상속받을수 있을까? 미의 부모가 가지고 있게 함. 부모가 가지고 있으면 어떤 식별자를 찾을때 프로토타입 내에서...
// 상속도 결국 재사용이다.
// 상속: 변수, 함수, 프로토타입 <- 다 관점이 재사용이다.
// 그럼 재사용을 왜 할까? 우리가 객체 3개 프로그램 150줄이라고 가정하면, 중요한거는 총 라인수가 20만줄 되는 프로젝트 할때, 패러다임이 중구난방 되면 지옥이 됨.
////
```

함수와 일급객체

함수는 객체다.

왜 자스엔진은 함수를 객체로 취급할까?

콜백함수 기억나? 얘의 특징은 함수를 인수에 줄수 잇다. 인수에 뭘 줄수 잇나?

매개변수 x 가 있다고 가정했을때 값이 올수 있는 자리다. 매개변수 x 에 할당되어야 하기 때문에.

그래서 매개변수 {} 뒤에 값이 올수 있다.

함수형 프로그래밍: 함수가 값이어야 한다. 함수형 프로그램 패러다임을 실현하기 위해서는 함수가 값이어야 한다.함수형 프로그래밍이 없는 건 함수가 값일 필요가 없다.

일급객체를 공부한다는 얘기는 고차함수에서 고생한다는 말이다.

일급객체란 말은 뭐냐면, 일급이 있으면 이급도 잇나? 일급 객체는 다른 책에서는 퓔스트 클래스 시티즌이라고 한다. 시티즌 이 시민이잖아. 우린 시민이란 말이 익숙치 않잖아. 서양애들은 시민이란 말 쓴다.

옛날에 시민이 여러가지 있었잖아. 시민이라고 하면 투표권 있는 사람들만 시민이었다. 그 시민들을 일급시민이라고 했다. 일급은 뭐냐면 완전히 값과 똑같은거다.

값과 완전히 똑같은 걸 일급객체라고 한다.

무명리터럴 : 변수에 할당하라는 말이다. 무명 리터럴을 변수에 할당하는 것을 함수 표현식이라고 했잖아. 함수 표현식의 할당은 런타임에 생성이 가능하다. 함수를 어케 만든다? 함수 정의를 미리 한다음 한다. 이런 코스가 아니라 호출하기 직전에 만들수 있다.

무명 리터럴로 만들수 있다는 것 : 함수를 호출하면서 (foo(...)만들수 있다.

배열이나 객체에 함수를 집어넣을수 있다. 배열은 모르겠는데 객체는 넣을수 있음. 그게 메소드라고 하는거다.

이런거 될까? [function(){},...]

함수의 리턴값이 될수 있나? 될수잇다.

이 4개를 만족하는 걸 일급객체라고 한다. 이게 되면 함수형 프로그래밍이 된다는 거다.

그런데, 함수는 객체다 라고 햇는데, 그러면 함수하고는 객체하고는 똑같은거냐? 다르다. 함수 고유의 특성이 많다.

함수 고유의 속성(프로퍼티) : [[call]] <- 일반객체는 이거 없다. 함수는 무조건 [[call]] 이 있다.

[[construct]] <- 객체가 이거 갖고 있을수 있나? 갖고있을 수 있다. 이게 있는 애는 컨스트럭트 라고 한다. 있는 애가 있고 없는 애가 있다.

클래스는 일반 함수 호출 못하니까 [[call]] 이 없다.

함수와 객체의 또 다른 차이점이 뭐가 있을까? 객체는 호출할수 없으나 함수는 호출할수 있다.

내부 메소드는 감춰진 프로퍼티다.

객체들이 갖고 잇는 프로퍼티는 함수는 다 가지고 있는데 함수만 가지고 있는 프로퍼티가 뭐가 있을까?

이건 지금부터 배우자.

```
function square(number) {
  return number * number;
}

square.f = function () {}  <- 이거 되는거다.
square.a = 1; <- 이것도 되는거다.

string. 하고 나오는 목록들은 전부 프로퍼티다.
```

아~ 이 앞에 O 자가 붙는걸 생성자 함수라고 했는데, 얘가 메소드들을 제공한다. 얘는 함수이지만 객체이다.

```
펄슨 생성자 함수도 프로퍼티를 갖고 잇겠네.
function square(number) {
  return number * number;
}    // 일반함수인가? 모르는거다. 어케 호출될지 모르니까.

console.dir(square);
```

콜론 붙은것들은 다 프로퍼티들이다.

```
> function square(number) {
    return number * number;
  }

  console.dir(square);
```

```
▼ ƒ square(number)
    arguments: null
    caller: null
    length: 1
    name: "square"
  ▶ prototype: {constructor: ƒ}
  ▶ __proto__: ƒ ()
    [[FunctionLocation]]: VM341:1
  ▶ [[Scopes]]: Scopes[1]
```

함수 객체의 프로퍼티

직접이 아니라 상속받아 쓰는 애다.

```
function square(number) {
  return number * number;
}

console.log(Object.getOwnPropertyDescriptors(square));
/*
{
  length: {value: 1, writable: false, enumerable: false, configurable: true},
  name: {value: "square", writable: false, enumerable: false, configurable: true},
  arguments: {value: null, writable: false, enumerable: false, configurable: false},
  caller: {value: null, writable: false, enumerable: false, configurable: false},
  prototype: {value: { ... }, writable: true, enumerable: false, configurable: false}
}
*/

// __proto__는 square 함수의 프로퍼티가 아니다.
console.log(Object.getOwnPropertyDescriptor(square, '__proto__'));
// undefined

// __proto__는 Object.prototype 객체의 접근자 프로퍼티이다.
// square 함수는 Object.prototype 객체로부터 __proto__ 접근자 프로퍼티를 상속받는다.
console.log(Object.getOwnPropertyDescriptor(Object.prototype, '__proto__'));
// {get: ƒ, set: ƒ, enumerable: false, configurable: true}
```

.... 저 꼬랑지 부분은..... 없으니까 언디파인드가 나온다.

아규먼츠 : 얘는 프로퍼티인가? 프로퍼티는 맞아요? 네. 그럼 왜 프로퍼티가 맞지? 위에 콜론 붙어서 나오니까. 근데 표준으로는 프로퍼티가 아니다. 폐지되었다. 근데 왜 크롬은 프로퍼티로 출력하고 있지?

모든 자스엔진이 다 표준을 깔고 있지 않다. 이렇게 아규먼츠 프로퍼티가 있다는것은 어케 참조할수 잇는거지? 스퀘어.아규먼츠로 참조하라는 말이잖아.

그럼 우리는 아규먼츠를 어케 들여다봐야해?

아규먼츠라는 프로퍼티가 이전에 존재했엇잖아. 걔 값이 아규먼츠 객체다. 아규먼츠 객체를 보려면 함수이름.아규먼츠 해서 참조했었다. 함수내부에서 참조했었다.

프로퍼티값은 인수다.

렝스가 있는 것을 유사배열이라고 한다.

```
const sum = function () {
let res = 0;
for (let i = 0; i < arguments.length; i++){
res += arguments[i];
}
return res;
};
console.log(sum(1, 2, 3)); //6
```

엄청 많이 돌때 포문을 많이 쓴다.

포문 돌면서 배열 만들수 있다.

문제의 발단 : 매개변수를 정의할수 없었다는게 문제였는데 매개변수 앞에 '...' 붙여주면.....

콜러 프로퍼티

콜러는 비표준이다.

에크마스크립트 사양서에 안올라와있다. 그래서 몰라서 된다.

그 함수를 호출한 함수를 말한다.

몰라도 된다. 넘어가도 된다.

렝스 프로퍼티

함수의 렝스다.

매개변수의 개수를 가리킨다.

거의 쓸일 없다.

네임 프로퍼티

함수 이름이 들어간다.

이것도 거의 쓸일 없다.

프로토 접근자 프로퍼티

얘는 함수가 가지고 있는게 아니고 모든 객체가 가지고 있다.

얘는 뭐냐면 얘로 접근하면 프로토타입 객체가 나옴.

나중에 다음시간에 살펴보자

얘는 모든 객체가 갖고 있고 "프로토타입 프로퍼티" 는 함수만 가지고 있다.

```
function Person (name) {
    this.name = name;
    this.sayHi = function () {
        console.log(`Hi! my name is ${this.name}.`);   <- sayHi
가 여러번 만들어져서 문제점.
    };
}
const me = new Person ('Lee');
me.sayHi();
----
//세이하이는 일반함수로 호출된거야, 생성자함수로 호출된거야? 메소드로 호출된거
다.
```

모든 객체는 [[prototype]] 이 있다.

식별자하고 프로퍼티를 혼동하지 않도록 주의하자.

결국은 프로퍼티도 값을 찾아내는 하나의 키잖아. 근데 문법적으로 식별자하고 프로퍼티하고 다르다.

프로토타입

자스는 멀티패러다임 언어다.

자스는 클래스가 없다. 아~ 자스는 객체지향 언어가 아니라는 오해를 받았지만

자스는 객체지향 언어이긴 하다. 근데 클래스 기반이 아니라 프로토타입 기반이다.

프로토타입 기반이 클래스기반보다 유연하다.

데이터를 주고받을수 있어야 한다. 그 중에 중요한게 상속이다. 프로퍼티를 찾을때... 부모를 찾으려고 하는게 상속구조다.

자스는 훌륭한 객체지향 언어다.

객체지향 프로그래밍

객체지향이라는 것은 여러개의 독립된 객체들을 만든것이다. 그 객체들을 연관을 지어서 프로그래밍을 하려는 패러다임을 말한다. 그럼 객체는 뭘까? 개발자 또는 프로그램이 주체고 ... 대상들이 다 객체다.

근데 그 객체를 어케 표현하냐면 속성으로 표현한다. 모든 속성ㄷ을 다 막라 해야하나요?

... 그것을 추상화라고 한다.

```
const person = {
    name: 'Lee'
};
console.log(person.constructor); // object
```

위 코드는 뭘 보려고 하는거야?

```
const person = {
    name: 'Lee'
};
console.log(person.__proto__ === object.prototype); //true

const arr = {};  <- 얘도 객체다.
console.log(arr.__proto__ === Array.prototyoe); // true
```

```
// 생성자 함수
function Circle(radius) {
  this.radius = radius;
  this.getArea = function () {
    // Math.PI는 원주율을 나타내는 상수이다.
    return Math.PI * this.radius ** 2;
  };
}

// 인스턴스 생성
// 반지름이 1인 인스턴스 생성
const circle1 = new Circle(1);
// 반지름이 2인 인스턴스 생성
const circle2 = new Circle(2);

// Circle 생성자 함수는 인스턴스를 생성할 때마다 동일한 동작을 하는
// getArea 메소드를 중복 생성하고 모든 인스턴스가 중복 소유한다.
// 따라서 getArea 메소드는 하나만 생성하여 모든 인스턴스가 공유하는 것이 바람직하다.
console.log(circle1.getArea === circle2.getArea); // false

console.log(circle1.getArea()); // 3.141592653589793
console.log(circle2.getArea()); // 12.566370614359172
```

this 레디우스는 ....

프로토타입 객체

모든 함수가 만들어질때에라고 하면 안된다. 컨스트럭터와 함께 호출할수 있는 함수들을 생성하면 함수 객체들이 만들어지는데, 프로토타입이 쌍으로 만들어진다. 동시에 만들어지는 것인가? 동시라는게 아니라 처리를 따로따로 되긴 하지만 만들고 바로 만드니까 같이 만들어진다고 이해하고 있자.

우리가 예를 들어서 생성자 함수를 만들고 ... 프로토타입이라고 하는 프로토타입 객체가 만들어질텐데.. 비어있는 상태이다. 왜 이렇게 비어놧을까? 생성자 함수를 만들었어, 이게 평가되어져서... 프로토타입 객체가 만들어질텐데 .... 뭘 넣을지 모른다. 자스엔진은. 일단 비워놓는다. 그럼 알아서 채워넣어야 한다.

꼭 채워넣어야 할 필요는 없다.

proto 접근자 프로퍼티

언더바 프로토 라고 하는걸 접근자 프로퍼티라고 한다.

모든 객체들은 그 객체가 함수가 되든 배열이 되든 모든 객체는 프로토타입이라고 하는 내부 슬롯을 가지고 있다. 이 내부 슬롯의 값은 언제 결정되나? 그 값은 어케 결정되고 언제 결정되나? 자신이 태어날때 결정해야 하는데 그 결정을 어케 하냐? 자신과 연결되어진 생성자 함수가 반드시 있다 그 생성자 함수... 객체가 프로토타입의 값이 되는것이다.

어떤 객체가 태어날라고 하고 있어. 태어나면. 뭘 해야 하냐면 내 부모가 누군지 결정해야 한다. 객체가 결정하는게 아니라 자스엔진이 얘를 태어나게 해주면서 결정해주는데 그 결정에 규칙이 있는데 그 규칙이 뭐냐면 모든 객체는 생성자 함수와 연결이 되어있다. 어케 연결되어있나? 그 생성자 함수를 어케 찾을수 있지? 예를 들어서

펄슨 생성자 함수를 뉴로 해서 객체를 만들었다. 그럼 걔는 자신의 프로토타입을 어케 만들지?

... 이거의 값이다. 모든 객체들은 생성자 함수와 연결 되어잇다.

객체들은 태어날때 자기의 부모를 안다. 그 부모의 참조를 프로토타입에 연결한다.

내부슬롯의 값을 우리가 참조할수 없는데 언더바 프로토로 참조 또는 갱신을 할수가 있다.

모든 프로퍼티는 다 참조가 된다. 이렇게 언더바 프로토라고 이름을 짓지는 않을거잖아. 그리고 참조하기 어렵게 하는 이유가 있다. 심볼이 나오기 이전의 얘기다. 내부적으로 쓰려고 언더바 프로토 라고 이름을 지었다. 이런 이름을 비표준이지만 이런 이름을 가지고 어떠한 이유에서인지 브라우저 밴더들이 이런 프로퍼티 를 갖기 시작했다. 계속 비표준이다가 es6 되서 표준이 되었다.

기본적으로 참조도 가능하고 할당도 가능하다. 접근을 하면 함수가 돈다. 게터가 돈다. 게터가 돌면서 뭘하지? 그 객체에 [[프로토타입]]에 참조 값을 반환한다.

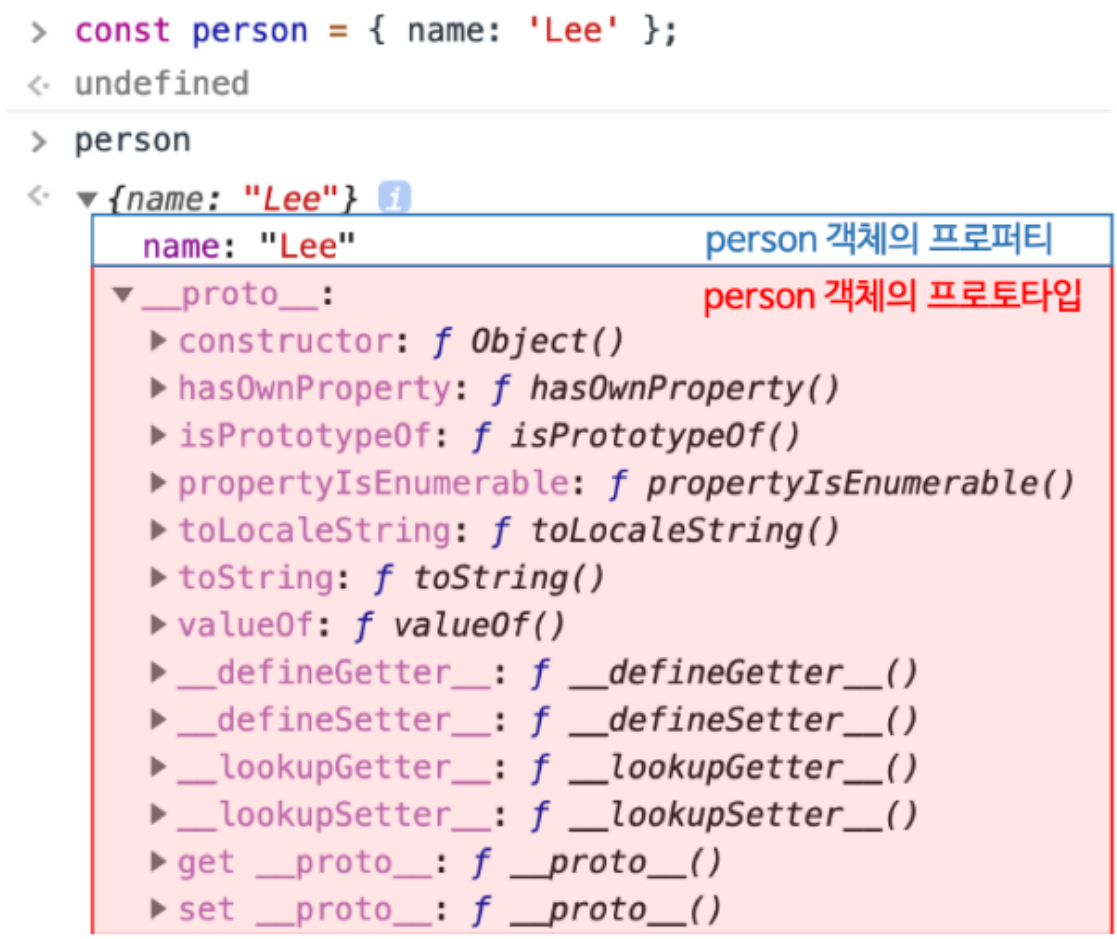
그 객체에 프로토타입에 참조를 반환한다.그리고 저 언더바 프로토에 어떤 값을 할당하면 무슨 일이 일어날까? 세터가 돈다. 세터가 돌면서 [[프로토타입]]의 참조값을 바꾼다.

.... 자신의... 를 교체할수 있다. 언더바 프로토라는 접근자 프로터피는 누가 사용할수 있을까?

이걸 생각하려면 일케 생각하자. 쟤가 하는 일이 뭐지? 프로토타입에 접근하거나 교체하는 거잖아.

그러면 프로토타입을 가지고 있는 애는 누구냐? 모든 객체다. 따라서 언더바 프로토는 누가 가지고 있어야 하는 기능이지? 모든 객체가 가지고 있어야 하는 기능이다. 그러면 모든 객체가 일일히 가지고 있어야 하나? 그럴 필요없다.

언더바 프로토는 존재의 위치가 어디냐? 오브젝트 프로토타입이다.



크롬 브라우저의 콘솔에서 출력한 객체의 프로퍼티

네임: 리는 펄슨이라는 객체...

언더바 프로토 는 프로터티 키네. 언더바 프로토 밑에 잇는 애들은 참조한 애들이다. 네임의 내용이다.

그럼 저 많은것들을 다 사용할수 있다.

```
const p = {
console.log(p.hasOwnProperty('name')); //true
console.log(object.getOwnPropertyDescripts(p));
}
```

가능성이 있는 곳이 3개다.

함수 객체의 프로토타입 프로퍼티

언더바 프로토, [[프로토타입]], 프로토타입 <- 이 3개가 헷갈리게 한다. 앞에 두개가 세트이고,

뒤에 프로토타입 앞에는 함수가 와야 하고, 언더바 프로토 앞에는 모든 객체가 와야 한다.

모든 객체긴 한데 괄호열고(오브젝트. 프로토타입) 이걸 상속받는 객체다.

맨 뒤의 프로토타입 앞에 잇는 함수가 가리키고 있는 값이 뭐지? 이 함수가 생성할 인스턴스.. 프로토타입을 가진다.

예습: 21번까지 20번은 가볍게만 읽어보고 오기. 소감 물어볼거야. 뭘 얘기하는지 주제만 찾아오면 되.