[자바스크립트 객체지향]

객체란? 서로 연관된 변수와 함수를 그룹핑하고 이름을 붙인 것이다.

이후 공부해 나가면서 객체의 또다른 모습들을 배워가면 된다.

[객체선언]

var memberObject = {

    'manager':'joonyi'

}

매니저 = 준이 라는 뜻이다.

객체 내의 어떠한 정보(준이)를 설명해주는 정보(매니저)를 쓸 수있다. = 준이는 매니저다 라는 정보를 썼다.

var memberObject = {

    'manager':'joonyi',

    manager : 'joonyi'

}

이렇게 manager는 굳이 string이 아니어도 된다.

[객체에 접근하기]

객체를 읽는 방법, 객체에 접근하는 방법을 알아보자.

아래와 같이 객체명에 “.” 마침표를 붙이고 부를 데이터를 쓰면 된다.

var memberObject = {

    manager:'chulsu',

    developer:'joonyi',

    designer:'egoing'

}

console.log("memberObject.developer", memberObject.developer);

memberObject 객체에서 developer를 부르면 ? joonyi가 나온다.

객체선언에서, developer는 “developer”를 써도 상관이 없다고 했는데 과연?

var memberObject = {

    manager:'chulsu',

    developer:'joonyi',

    designer:'egoing',

    "developer":'zz'

}

console.log("memberObject.developer", memberObject.developer);

//memberObject.developer zz

궁금하면 해보면 된다.

Developer가 둘이다. 하나는 문자열로 선언, 하나는 그냥 선언됐다.

결과는 zz가 나왔다. Developer로 부를 데이터가 두개인데, 가장 나중의 데이터를 부르는 것이다.

var memberObject = {

    manager:'chulsu',

    developer:'joonyi',

    designer:'egoing'

}

console.log("memberObject['developer']", memberObject['developer']);

// memberObject['developer'] joonyi

위와 같이 대괄호를 사용해서 접근할 수 있다.

[객체 데이터 수정하기]

대충 눈치 채겠지만 우리는 CRUD 배우고 있는 것이다. 이번에는 수정(Update)에 대해서 배운다.

Create, Read, Update, Delete!

객체의 데이터를 수정하려면 우선 데이터에 접근해야한다. 객체의 데이터에 접근하는 방법은 위에서 배웠으니 그것을 활용하면 좋을 것 같다.

var memberObject = {

    manager:'chulsu',

    developer:'johnyi',

    designer:'egoing'

}

memberObject.developer = 'joonyi';

console.log("memberObject.developer",memberObject.developer);

//memberObject.developer joonyi

위에서 developer에 내 이름이 들어가야하는데 johnyi로 오타가 났다.

그래서 developer에 접근해서 joonyi로 값을 바꿨다.

바꾸고 출력하면? 객체의 데이터가 수정되어 잘 출력된다.

[객체 데이터 삭제하기]

var memberObject = {

    manager:'chulsu',

    developer:'johnyi',

    designer:'egoing'

}

delete memberObject.designer;

console.log('memberObject.designer', memberObject.designer);

//memberObject.designer undefined

CRUD 마지막! delete이다.

객체의 데이터를 삭제하고 싶다면 delete를 앞에 붙이고 삭제할 데이터를 읽으면 된다.

Designer가 퇴사했다는 가정하에 designer 데이터를 삭제하고 출력하면?

Undefined, 정의되지 않았다 = 읽을 수가 없다 = 데이터가 없기 때문에 = 데이터가 삭제되었다.

[깨알팁 1 : console 꿀팁]

깨알팁 1이라고 했는데 2, 3이 있을지는 미지수다. 일단 있을수도 있으니까 1이라고 넘버링한다.

var memberArray = ['egoing', 'graphittie', 'leezhce'];

console.group('array loop')

var i = 0;

while(i < memberArray.length) {

    console.log(i, memberArray[i]);

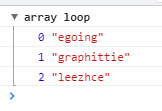
    i += 1;

}

console.groupEnd('array loop')

위를 잘 보면 코드의 내용은 배열의 출력이다.

Console.log 만 써봤지만, console.group(그룹이름)을 쓰고 console.groupEnd(그룹이름)로 묶으면 이용하면 콘솔에 출력될 때 텍스트를 잘 정리해서 출력하게 한다.

요렇게 말이다.

[객체와 반복문]

배열을 아는가? 배열의 핵심은 반복문이다. 반복문과 함께할 때 그 배열내의 여러 개의 데이터를 처리할 수 있게 되는 것이다.

객체도 여러 개의 데이터를 갖고 있다. 그렇기 때문에 객체 역시 반복문과 활용될 수 있다.

보통 배열에선 for, while문을 썼을 것이다. 객체에서도 그걸 쓰면 된다.

그런데 객체에서 for 문을 쓸 적엔 for(var i=0; i<something; i++) 이런 식으로 쓰는게 아니라

For(i in object) 이런 형식으로 쓰는 것이다.

var memberObject = {

    manager:'egoing',

    developer:'johnyi',

    designer:'chulsu'

}

console.group('object loop');

for(var name in memberObject) {

    console.log(name, memberObject.name);

}

console.groupEnd('object loop');

위를 잘 보자. name에는 memberObject의 값들이 들어간다.

결과는 manager, developer, designer가 될 것이다.



근데 그럼 egoing, johnyi chulsu를 뽑으려면 memberObject.name 이렇게 쓰면 될까?

이렇게 쓰면 안된다. Undefined가 나온다.

객체의 “.”(마침표) 뒤에는 변수(여기서는 name 변수를 썼다)가 오면 안되기 때문이다.

객체의 “.”(마침표) 뒤에는 객체의 각각의 원소(속성)의 이름이 와야 한다.

자 그럼 어쩌란 말인가?

console.group('object loop');

for(var name in memberObject) {

    console.log(name, memberObject[name]);

}

console.groupEnd('object loop');

위와 같이 쓰면 좋을 것 같다는 말이다.



좀 더 잘 보이게 확대하면 memberObject[불러올 변수]를 쓰면 되는 것이다.

다 배워 놓고 좀 미안하지만, 배열에서 반복문을 쓰는 것은 필수적인데에 비해, 객체에서 반복문을 쓰는 것은 그렇게 중요한 것이 아니었다.

[객체를 왜 쓰는가?]

console.log("Math.PI ", Math.PI);

console.log("Math.random() ", Math.random());

console.log("Math.floor(3.9) ", Math.floor(3.9));

위를 보자. Math는 이미 만들어져있는 자바스크립트 내장객체이다.

Math라는 객체에는 random, floor 함수, PI라는 변수 등을 통해서 랜덤으로 수를 뽑던지, 반내림을 하던지를 할 수 있다.

Math 객체는 그런 상수와 함수들을 구현하고 저장해고 묶어서 보관해놓은 것이다.

이렇게 정리를 편하게 할 수 있다.

이렇게 우리는 객체를 모른다고 하면서도 사실 객체의 도움을 받고 살고 있다.

[객체와 메소드]

우리가 함수라고 부르는 것이 객체의 원소로 들어간다면 그것은 메소드(Method)라 불린다.

[객체에 함수 넣어서 메소드 만들기!]

var MyMath = {

    PI: Math.PI,

    random: function(){

        return Math.random();

    },

    floor: function(val){

        return Math.floor(val);

    }

}

console.log("MyMath.PI", MyMath.PI);

// MyMath.PI 3.141592653589793

console.log("MyMath.random() ", MyMath.random());

// MyMath.random()  0.07168215013577628

console.log("MyMath.floor(3.9) ", MyMath.floor(3.9));

// MyMath.floor(3.9)  3

내용은 별거 없다.

Math 객체의 PI상수와, random, floor 함수들을 MyMath 객체에 추가시킨 것이다.

물론 기능적으로 이렇게 할 이유는 없지만… 아무튼 내가 나만의 메소드를 만들어본 것이다!

[객체를 쓰지 않는다면?]

// 객체를 쓰지 않는다면..

MyMath\_PI = Math.PI;

function MyMath\_random() {

    return Math.random();

}

function MyMath\_floor(val) {

    return Math.floor(val);

}

객체를 만들어 봤는데, 객체를 쓰지 않고도 저 기능을 구현할 수 있다. 위와 같이 말이다.

그런데 다 “MyMath\_” 라는 접두사가 붙어있다. 이는 당연히 다른 코드에 붙어있을지 모르는 불확실한 어떤 데이터가 PI라는 이름을, random이라는 이름을, floor라는 이름을 가지고 있을 때 그것이 나중에 우리가 그러한 이름을 호출했을 때 잘못된 동작을 하는 것을 막기 위해서이다.

알잖아 사실. 이름이 같으면 나중에 선언된 것이 호출된다. 먼저 선언된 것을 호출할 방법이 없어지게 되는 것이다.

[this 배우기]

var kim = {

    name: 'kim',

    age: 36,

    first: 10,

    second: 20,

    sum: function(){

        return kim.first + kim.second;

    }

}

//console.log("kim.sum(kim.first, kim.second) ",kim.sum(kim.first, kim.second));

console.log("kim.sum()", kim.sum());

kim 이라는 객체의 내부 데이터에 접근하는 법은 무엇일까?

위 처럼 kim.first 와 같이 접근할 수 있을 것이다. 그런데 이 방법은 유연성이 많이 떨어진다.

객체 명이 K로 바뀐다면? 또 코드를 수정해줘야 할 것이다.

var k = {

    name: 'kim',

    age: 36,

    first: 10,

    second: 20,

    sum: function(){

        return this.first + this.second;

    }

}

console.log("kim.sum()", k.sum());

위와 같이, 이런 상황에서 자기자신 객체를 콜할 매개변수로써 쓰는 예약어가 this 라는 키워드이다.

[constructor 주세요~]

var k = {

    name: 'kim',

    age: 36,

    first: 10,

    second: 20,

    sum: function(){

        return this.first + this.second;

    }

}

지금까지 배운 것으로 객체를 만든다면 이런 모양새가 될 것이다. 근데, 이런 건 가내 수공업으로 객체를 만드는 것과 같다. 이름이 lee인 사람 객체를 추가할 때 똑같이 이 방식으로 만들어야 한다. 첫번째 성적과 두번째 성적을 더하는 sum 메소드가 앞으로 세번째 성적까지 더해줘야 한다면 kim 객체와 lee 객체에서 모두 코드를 수정해줘야 한다.

이 말은, 어딘가 자동으로 객체를 만드는, 공장과 같이 라인공정을 만들 수 있는 무언가가 존재한다는 말이다.

var d1 = new Date('2021-7-17');

잘 보자. D1이라는 변수에 new 예약어로 Date형식의 객체가 ‘2021-7-17’ 이라는 값을 받아 저장됐다!

이렇게, 새로운 객체를 선언할 때 자동으로 값을 받아 생성할 수 있다.

너무 해보고싶다! 해보기 싫으면 손해임 ㅅㄱ ^^

[constructor 만들자~]

var d1 = new Date('2021-7-17');

이걸 보자. 마치 함수를 호출하는 것 같지 않나?

그 이유는 바로 Date가 진짜 함수라서 그렇다.

console.log('Date ', Date);

//Date  ƒ Date() { [native code] }

자 위 코드를 보면 Date의 출력 결과는 Date는 함수라고 나오고 그것도 native code 라고 나온다.

Native code, 즉 내장된 함수라는 말이다.

우리도 그럼 함수를 만들자

function Person(){

    this.name = 'kim',

    this.first = 10,

    this.second = 20,

    this.third = 30,

    this.sum = function(){

        return this.first + this.second + this.third;

    }

}

console.log('Person() ', Person());

//Person()  undefined

console.log('new Person() ', new Person());

/\*

new Person()

Person {name: "kim", first: 10, second: 20, third: 30, sum: ƒ}

first: 10

name: "kim"

second: 20

sum: ƒ ()

third: 30

\_\_proto\_\_: Object

\*/

이렇게 만들고, 결과를 출력했더니 주석처럼 나왔다!

Person에 new를 안 붙이고 그 객체를 출력했더니 리턴값이 없어서 undefined로 나왔지만, new 예약어를 붙이고 Person()을 작성하니까 객체를 뱉었다.

정리하자.

함수를 그냥 호출하면, 그냥 호출이다.

근데 예약어 new를 붙이고 함수를 호출하면? 객체를 생성하는, 생성자가된다.

생성자는, constructor라고 한다.

여기서 this.name 이라고 왜 붙일까?

그 이유는… 모르겠다. 그냥 Person이라는 생성자로 객체를 생성할 때 그 객체 자체를 지정할 때 this를 쓰는듯 하다.

조금 생각해봤다. 추측은 이렇다. 우리가 객체를 만들 때 var a = { aaa = “123” } 이런 식으로 선언된다.

근데 사실 aaa는 그냥 aaa가 아니라, a.aaa라는 변수를 지정하는 것 아닌가? 즉, aaa를 만드는 것이 아니다.

a.aaa를 만드는 것이다. 따라서 Person이라는 생성자에서도 객체를 생성할 때, 객체명 + 속성명을 써준다. This.name처럼 말이다.

[객체 파라미터 받기]

function Person(name, first, second,  third){

    this.name = 'kim',

    this.first = first,

    this.second = second,

    this.third = third,

    this.sum = function(){

        return this.first + this.second + this.third;

    }

}

var kim = new Person('kim', 10, 20, 30);

var lee = new Person('lee', 10, 10, 10);

console.log("kim.sum()", kim.sum());

console.log("lee.sum()", lee.sum());

//kim.sum() 60

//lee.sum() 30

쪼금만 바꾸면 파라미터도 받게 할 수 있다.

Kim과 lee객체에 각각의 정보를 줬더니 출력이 서로 다르게 됐다!

이렇게 하니까 유지보수가 쉬워진다.

Third 변수가 사라지면? ㅋㅋ 원래는 각각의 객체를 모두 바꿔줬어야 했는데, constructor function만 바꿔주면 다 바뀐다!

[프로토타입을 배우는 이유]

프로토타입은 원형이라는 말이다. 프로토타입은 사물의 원형, 본래의 모습을 말하긴 한다.

js에서 프로토타입은 중고급으로 넘어가는 길목에 있는 아주아주 중요한 개념이다.

Javascript를 prototype based language라고도 부를만큼 prototype이라는 것은 js의 기본이자 가장 중요한 개념이고, 깊게 들어가면 진짜 끝도 없으니까 실용적인 수준까지 알아보도록 하자.

function Person(name, first, second,  third){

    this.name = name,

    this.first = first,

    this.second = second,

    this.sum = function(){

        return this.first + this.second;

    }

}

var kim = new Person('kim', 10, 20, 30);

var lee = new Person('lee', 10, 10, 10);

자 우리가 만든 객체의 생성자(constructor)와 객체들(kim, lee)이다. 그런데 this.sum = function() { … }구문이 보이는가? 이 구문은 객체가 생성될 때마다 function 크기 만큼의 함수가 계속 정의될 것이다. 객체마다 함수를 계속 정의하는 것이다. 객체를 1억개 만든다면? 1억개의 데이터를 저장하면 되는 것이 아니라 1억개 + 1억개의 객체내의 데이터를 다룰 함수 각각(sum이라는 함수)이 저장될 것이라는 거다.

이해 어려울 수도 있으니 한번 더 말하자면 아래와 같다.

우리가 객체를 선언할 때 this.name = name, … 이렇게 변수들을 저장하곤 한다.

그러면, 김철수라는 사람을 객체화시켜 저장할 때 우리가 저장할 데이터는 김철수라는 사람의 이름과, 뭐, 여기선 first second라는, 첫번째 시험, 두번째 시험의 점수를 저장할 것이다. 그러면 김철수의 이름이라는 데이터를 이름1이라고 하자. 앞으로 김영희라는 사람이 또 객체화되어 저장된다면? 이름2가 되겠지.

그런데 객체를 만들 때 객체별로 sum이라는 함수까지 저장된다. 객체를 만들수록 함수1, 함수2를 만들게 되는 것이다.

이게 왜 문제가 되는가?

데이터는 고유한 값(여기서는 이름이 되겠다. 실제로 이름은 동명이인도 있으니 고유값이 되진 않지만, 그런걸로 치자)과 그것에 포함되는 값(first, second) 등 어차피 그 데이터를 저장할 저장공간을 차지해야 한다.

하지만 함수는?

저 함수는 그냥 첫번째 데이터와 둘째 데이터를 더해주는 함수이다.

그런데 객체별로 저장된다면 이것은 마치 김철수의 first, second(시험점수)를 더해주는 함수, 김영희의 시험점수를 더해주는 함수… 또 더 객체가 있다면 정준이의 시험점수를 더해주는 함수… 계속 만들어지는 것이다.

함수는 고유할 필요가 없다. 어차피 결국 시험점수를 더해주기만 하는 기능이 필요한 것이니까. 김철수라는 객체만 쓸 수 있는 함수일 필요가 없는 것이다.

function Person(name, first, second,  third){

    this.name = name,

    this.first = first,

    this.second = second,

    this.sum = function(){

        return this.first + this.second;

    }

}

var kim = new Person('kim', 10, 20, 30);

kim.sum = function(){

    return 'modified : '+(this.first+this.second);

}

var lee = new Person('lee', 10, 10, 10);

console.log("kim.sum()", kim.sum());

console.log("lee.sum()", lee.sum());

위 코드를 또 보자. 이 코드에선 kim이라는 객체에 대해서 sum 함수를 새로 재정의 했다.

이렇게 우리는 재정의 할 수도 있다. 이러면 Person에서 만들어졌더라도 기능을 바꿀수도 있다는 걸 알아두자.

그런데 여기서 lee의 함수도 저렇게 바꾸고 싶다면?

Lee도 바꿔줘야 한다.

var kim = new Person('kim', 10, 20, 30);

kim.sum = function(){

    return 'modified : '+(this.first+this.second);

}

var lee = new Person('lee', 10, 10, 10);

lee.sum = function(){

    return 'modified : '+(this.first+this.second);

}

이렇게 말이다. 아. Person.sum = function() { … } 하면 안되냐고? 안된다.

바로 이것이 우리가 Person 같은 생성자 안에서 sum과 같은 메소드를 만드는 것의 단점이다.

그럼 어떻게 해야하지?

Person이라고 하는 생성자를 이용해서 만들어진 모든 객체가 “공통적으로” 사용하는 함수를 만들면 얼마나 좋을까. 속성을 만들 수 있다면 얼마나 좋을까.

[프로토타입을 이용해서 객체들이 모두 사용하는 속성을 만들기]

function Person(name, first, second,  third){

    this.name = name;

    this.first = first;

    this.second = second;

}

Person.prototype.sum = function(){

    return this.first + this.second;

}

어려우니까 그러려니 하자.

Person.prototype.함수명 을 하면 Person 생성자에 정의될 “함수명”이라는 이름의 함수의 원형을 설정하는 것이 된다.

이전에 만들었던 Person이라는 생성자로 만들어진 객체들에 이걸 적용하더라도 결과는 달라지지 않는다.

그럼 뭐가 다른가?

우선, 객체의 sum이라는 함수를 정의하는 코드가 객체가 만들어질 때 마다 실행되어 함수를 객체마다 제각각 만들지 않을 것이다. 성능도, 메모리도! 확보가능하다. 함수를 객체들간에 share할 수 있다.

kim.sum = function(){

    return 'this : '+(this.first + this.second);

}

만약 kim이라는 객체의 sum이라는 메소드만 바꾸고 싶다면 이렇게 하면 된다.

Kim.sum = function(){ … }

이걸 통해서 프로토타입의 특징을 알 수 있다.

Js는 kim이라는 객체의 sum 메소드를 호출할 때, 우선 객체 자신(kim)이 sum이라는 속성(프로퍼티)을 가지고 있는지를 찾는다. 찾으면? 그것을 반환한다.

그런데 lee라는 다른 객체는? Sum이 새로 정의되지 않았다. 이 경우에는 이 lee 객체의 생성자인 Person의 prototype에 sum이라는 메소드가 정의되어 있는지를 찾는다. 있으면? 반환한다.

이 프로토타입은 객체를 생성할 때 메소드를 정의하는 패턴이다.

1. 프로토타입이란 어떤 의미?

나의 답) 생성자로 생성된 객체들의 디폴트 프로퍼티, 메소드 값을 정의하는 도구.

2. 프로토타입을 사용하지 않고 생성자 함수 안에서 메소드나 속성을 직접 정의하면 어떤 비효율이 발생하는가?

나의 답) 각 객체 고유의 메소드, 속성(프로퍼티)들이 생성되고, 그것을 생성하는 시간과, 그것이 저장되는 공간이라는 리소스가 소모된다.

3. 그 비효율을 프로토타입으로 어떻게 극복했는가?

나의 답) 프로퍼티의 디폴트 값을 정해줌으로써 객체가 그 프로퍼티를 실제로 가지고 있지 않더라도, 그 객체가 생성된 생성자의 프로토타입을 참조하도록 함으로써 그 객체가 그 프로퍼티를 가지고 있는 것처럼 사용할 수 있게 한다.

문서 뒤에서 요약한다.

[프로토타입, 객체의 속성?? 의문점.]

function Member(){

    Member.x="hello";

    return "origin function";

};

Member.c = function(){

    return "function in function";

}

위 코드는 정상이다. 말이 되는가?

함수에 속성이 있단다. 함수에 함수가 있는 것이다. ㅋㅋㅋ 웃기다.

근데 이게 가능한게 함수도 결국 객체라서 객체의 속성이 있게 된다.

객체는 즉 속성을 가질 수 있다. 속성은 변수거나, 메소드(함수)다.

그럼그럼?

var s = "MyString";

s.c = "asdf";

console.log(s.c);

이것도 되나?

안된다.

왜안되냐고~~~~~~~~~~~!! 하며 스택오버플로우에도 질문했다. 애꿎은 외국인들이나 괴롭힌 것이다.

혹시? 해서 이 글을 쓰며 테스트해봤는데, 망치로 뭔가 얻어맞은 기분이다.

var s = new String("MyString");

s.c = "asdf"

console.log(s.c);

//asdf

console.log(typeof s);

//object

이게 된다.

ㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋ

그러니까, var s = “쏼라쏼라”; 하면 string 객체가 아니고, string 타입이고, ㅋㅋㅋㅋㅋㅋ

Var s = new String(“내용내용임~”); 하면 string 객체가 선언되니까 s에 각종 프로퍼티들 다 갖다붙일 수 있다는 것이다.

와.. 얼척이없네

진짜 짜증난다. 두시간을 씨름했는데. 결과적으로 아래와 같다.

var s = "MyString";

s.c = "asdf";

console.log(s.c);

//undefined

console.log(typeof s);

//string

var s = new String("MyString");

s.c = "asdf"

console.log(s.c);

//asdf

console.log(typeof s);

//object

에효..

근데 여기서 의문점!!!!!!!!!!!!!!!

왜 var s = “asdf” 는 string 타입이고 var s = new String(“asdf”) 만 object 인가? Var s = “asdf”가 string인 이유가 뭐지? Var s = new String(“asdf”)는 String 객체를 선언하는거니까 object인건 이해한다. 근데 var s = “ㅁㄴㅇㄹ”가 객체가 아닌건 뭔지모르겠다. 숫자도 이런식이다.

var a = 1234;

a.c = "asdf"

console.log(a.c);

//undefined

console.log(typeof a);

//number

var a = new Number(1234);

a.c = 78;

console.log(a.c);

//78

console.log(typeof a);

//object

이렇게 말이다.

흠.. 그냥 알아둬야겠다. 뭔가 예약어같다. A = “asdf” 하면 string 이라는 예약된 타입으로 데이터를 저장한다.

A = 1234 하면 number라는 예약된 타입으로 데이터를 저장한다.

또한 그리고 이들은 객체가 아닌 것이다.

A = new blabla(“something”); 이런식으로 선언해줘야 A가 blabla 타입의 객체가 되는 것이다.

그니까 A = 1234 해서 속성을 추가해봐야 A는 객체가 아니니까 속성추가가 안되는 것이다.

이렇게 알아두자. 왜???????? 하는 의문이 끓어오르지만 지금 9시 30분이고 오늘 새벽 4시에 일어나서 피곤하니까 차차 커리어를 쌓고, 공부하면서 조금씩 알아두는걸로 하자.

어차피 오늘 프로토타입 이해한것만으로도 장족의 발전이다.

[요약]

프로토타입은 자바스크립트에서 상속을 구현하기 위한 객체이자, 속성이다.

어떤 객체를 생성하는 생성자(아마 함수이겠지.)가 선언될 때 자동으로 같이 선언된다.

즉, 생성자가 아니면 선언되지 않는다.

Var a = { … } 이런식이면 a는 객체지만 생성자가 아니므로 프로토타입이 없다.

이 프로토타입은 프로토타입 프로퍼티를 가질 수 있으며, 이 프로토타입 프로퍼티라는 것은 해당 프로토타입 객체를 갖고 있는 생성자가 생성한 객체들이 그 생성자의 프로토타입 객체에 접근하여 프로퍼티를 참고할 때의 그 프로퍼티를 의미한다.

즉 생성자의 프로토타입이라는 것은 생성자가 고유하게 가질 수 있는 프로퍼티들을 보관하는 객체라고 봐주면 되겠다.

A 생성자가 B 객체를 만들었다고 가정하자. A 생성자의 X 메소드는 A 생성자의 프로토타입에 정의되어있다고 하자.

B 객체에 X라는 이름의 메소드를 또 정의한다면, B.X 형식으로 그 메소드를 불러올 수 있다.

만약 B 객체에 X라는 이름의 메소드가 정의되지 않았다면, B.X 형식으로 A 생성자 프로토타입에 있는 메소드를 불러올 수 있다.

프로토타입 프로퍼티를 참조할 때 순서는 객체 내부의 프로퍼티 > 프로토타입 프로퍼티 > 그 상위 프로토타입 프로퍼티 이다.

또한 프로토타입 객체는 프로토타입 링크를 가지고 있다. 프로토타입 링크는 상속 관계를 나타낸다.

\_\_proto\_\_ 변수에 그 관계를 저장하고 있으나, 이제 deprecated 될거랜다 ㅋㅋ

아무튼 \_\_proto\_\_ 변수는 그 객체를 생성한 생성자를 지칭한다.

A생성자의 프로토타입의 \_\_proto\_\_는 무엇을 가리키고있을까? Object 생성자다.

왜냐면 A생성자의 프로토타입은 객체이다.(프로토타입 객체이니까.) 그리고 그 객체(object)라는 것을 생성하는 것은? Object 생성자다. 그래서 A.prototype.\_\_proto\_\_는 Object 인 것이다. 정확히는, Object라는 이름 그 자체가 아니라 생성자 Object의 내용이다. { 쏼라쏼라 } 이렇게 되어있겠지.