[클래스란?]

오늘날 자바스크립트는 가장 빠르게 발전하고 있는 언어이다.

JS는 머랄까… 전통적인 객체지향 언어의 문법들을 채택해서 이미 객체지향을 사용할 수 있는 사람들이 JS에 쉽게 익숙해질 수 있도록 하는 요소들을 대거 채택중에 있다. 그래서 JS에도 클래스가 있는 것이다.

클래스는 파이썬, 자바 등 여러 객체지향 언어들이 “객체”를 만드는 생성자로서 사용하는 것이다.

JS는 클래스를 사용하고있지 않았다. JS는 Constructor function으로 객체를 생성하고 있었지만, 지원하게 됐다.

Class 문법은 JS의 명세 표준 버전 중 ECMA Script6(ES6)에서 추가된 기능이다. ES 5이전에선 안된다는 것이다.

BABEL 이라는 사이트(컴파일러, 트랜스파일러 라고도 불림)에서, 이런 문제를 해결 해준다. Class로 선언된 객체 생성자 텍스트를 기존의 JS constructor function 형식으로 구현하는 텍스트로 바꿔주는 기능을 제공한다.

하지만 class, constructor function의 혼용이 큰 문제가 되지 않는 것은 오늘날 많은 웹브라우저, Node.js, 프레임워크들이 다 ES6을 지원하기 때문이다. IE 빼고 말이다.

클래스는 새로운 것이 아니다. JS가 이미 갖고 있는 것이다. 이미 JS는 객체지향 언어이기 때문이다.

[클래스 만들기 1]

자 그럼 클래스를 만들어보자. 이전에 constructor와 prototype을 써서 객체 만들기 예제를 진행해 본 적이 있으므로, 간단하게 그 예제와 똑같이 동작하는 클래스를 만들어보도록 하자.

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

    this.name = name;

    this.first = first;

    this.second = second;

}

Person.prototype.sum = function(){

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

}

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

kim.sum = function(){

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

}

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

console.log("kim.sum()", kim.sum());

console.log(lee.sum);

위 예제와 똑같이 동작하는 클래스를 만들어보자.

class Person{

}

우선 이렇게 선언하자.

class Person{

}

var kim = new Person();

console.log('kim', kim);

일단 kim에 아무런 파라미터도 주지 말고 kim을 출력해보자.

텍스트이(가) 표시된 사진

자동 생성된 설명이렇게 kim은 Person 객체라는 것을 보여준다.

[클래스 만들기 2 – 생성자 구현]

Constructor 에서는 this.name = name 이런 식으로 프로퍼티들을 초기화해줬다. 과연 class에서는 어떤 방식으로 이루어질까?

class Person{

    function sum(){

    }

}

Method를 만들어봤다. 이렇게 만들면 될까?

미안하지만.. 안된다! 우리가 지금까지 js에서 function을 선언할 때 다 이렇게 만들었는데, 왜 안될까?

아마 function이라고 글을 써서 method를 선언하는 것이 쓸데없이 자판을 두들기는 것이라고 생각했을지도 모른다.

class Person{

    sum(){

    }

}

JS class에서 method를 선언할 때에는 이렇게 function을 뺀 채 method명만으로 선언한다.

자 그런데 class에는 이미 옛날에 배웠지만, initialize 해주는 함수가 있다. 생성자 말이다.

그런 생성자(constructor, 예약어이므로 바꿀 수 없다.)를 이용하면 객체들의 프로퍼티들을 다 지정해줄 수 있다.

class Person{

    constructor(){

        console.log('constructor')

    }

}

이렇게!

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

}

var kim = new Person();

console.log('kim', kim);

그럼 이렇게 하면? Constructor에서 함수 내에서 사용될 프로퍼티들을 다 지정해줬다!

메소드를 이제 만들어보자.

[클래스 만들기 3 – 메소드 구현]

Person.prototype.sum = function(){

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

}

원래 JS는 위와같이 prototype을 통해서 만들곤 했다.

그런데 class에서는?

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

}

Person.prototype.sum = function(){

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

}

방법 1. 이렇게 prototype을 붙여넣는다 ㅋㅋㅋ 간단하다.

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

}

Person.prototype.sum = function(){

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

}

var kim = new Person('kim', 10, 20);

console.log('kim', kim);

console.log("kim.sum()", kim.sum());

방법 1로 이렇게 하면 실제로 잘 된다.

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

}

Person.prototype.sum = function(){

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

}

var kim = new Person('kim', 10, 20);

console.log('kim', kim);

kim.sum = function(){

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

}

console.log("kim.sum()", kim.sum());

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

console.log(lee.sum());

이렇게 해도 다 잘된다. (kim.sum을 재정의해도 된다는 말이다)

그 다음, 이 방법 말고 다른 방법!

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

    sum(){

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

    }

}

이렇게 안에다가 sum을 넣어주는 것이다. 결과는 똑같다.

[클래스의 메소드 형식]

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

    sum(){

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

    }

    sum = function(){

        return this.first + this.second;

    }

}

추가적으로, sum() {}가 메소드를 정의해주는 형식이라고 했다.

Sum(){} 이런식으로 메소드를 짜면 해당 메소드가 Person의 prototype 객체의 프로퍼티로 들어간다. 무슨 말인지몰라도 prototype 객체는 나중에 설명한다.

근데 sum = function(){} 이런 식으로 짜면 이건 메소드가 아니다.

이건 그냥 Person 객체가 만들어주는 함수다. 그러니까 var a = new Person() 했을 때 a의 프로퍼티로 들어가는 값이다. 상속시에도 똑같다. Person의 Constructor 안에 있어서 Person 객체가 생성될 때 같이 생긴다고 생각하면 된다.

만약 Person을 상속하는 Personplus 클래스가 있다고 하자. Var a = new Personplus() 하게되면 그 a라는 객체 안에 Person의 sum 객체가 프로퍼티로 들어간다.

[상속이 왜 필요한가?]

상속이 왜 필요한가 알아보자.

상속에 익숙하지 않은데 억지로 “쓰려고” 하면 잘 안될 것이다.

차차 익숙해져 나가면서 상속을 자연스럽게 활용할 수 있게 되어보자.

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

    sum(){

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

    }

}

이게 우리가 만들었던 클래스다.

그런데, 기능이 좀 부족하다. 평균을 구하고 싶을 땐 어떻게 할까?

    avg(){

        return (this.furst+this.second)/2;

    }

이런 메소드를 추가해도 되겠지만, 일단 저걸 언제나 할 수 있는 것은 아니다. 주석처리하고 생각해보자.

만약에 저 Person이라는 클래스가 지금처럼 우리가 직접 만든 것이 아니라 다른 사람이 만들었던 라이브러리라고 한다면 어떨까. 그 코드를 수정해버리면? 그 라이브러리가 업데이트 되었을 때, 우리가 수정한 내용이 사라질 수도 있다. 아니면 라이브러리를 업데이트 할 수 없게 될 수도 있다.

내가 짠 코드라고 하더라도, 내가 만든 Person 클래스의 객체들 전부가 avg를 모두 쓸 수 있게 하는 것이 부담되거나, 나의 의도에 맞지 않을 수 있다.

이런 경우에 상속을 사용할 수 있다. 물론 쓰지 않고도 목적을 달성할 수 있다.

똑 같은 Class를 하나 더 만들어서 그 클래스에는 avg를 넣고, 앞으로 avg가 필요한 객체는 그 클래스로 만들면 되겠지. 그런데 이건 당연히 비효율적이다.

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

    sum(){

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

    }

    // avg(){

    //     return (this.furst+this.second)/2;

    // }

}

class Personplus{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

    sum(){

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

    }

    avg(){

        return (this.first+this.second)/2;

    }

}

이런식으로 구현해도 되긴 한다는 것이다. 여기서 avg가 필요한 객체는 Personplus, 아니면 Person으로 가면 된다. 그런데, 중복이 너무 크잖아. 이런걸 해결할 수 있다는 말이다. 이제 상속 해보자.

[상속 구현]

class Personplus extends Person{

}

우선 이렇게 해보자. Personplus extends Person. Personplus는 Person 클래스를 상속한다는 뜻으로 이해하자.

class Personplus extends Person{

    avg(){

        return (this.first+this.second)/2;

    }

}

그러고는 Person class에서 추가할 내용(avg)을 넣는다.

이렇게 중복되는 코드를 제거할 수 있다.

Personplus 객체들은? Person의 기능까지 모두 사용할 수 있다.

[super 배우기]

우리가 뭔가 배웠다고 그걸 바로 도입하면, 복잡해지면서 어려워진다. 그러니까 좀더 탄탄히 알아보자.

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

    sum(){

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

    }

}

class Personplus extends Person{

    avg(){

        return (this.first+this.second)/2;

    }

}

자 이렇게 보니까 좀 아쉽다. Person에는 first, second 프로퍼티만 있지만, Personplus에는 third 프로퍼티까지 있으면 얼마나 좋을까?

class Personplus extends Person{

    constructor(third){

        this.third = third;

    }

    avg(){

        return (this.first+this.second)/2;

    }

}

위와 같이 Personplus의 constructor에 정의하면 될까?

안된다. ㅠㅠ

class Personplus extends Person{

    constructor(name, first, second, third){

        this.name = name;

        this.first = first;

        this.second = second;

        this.third = third;

    }

    avg(){

        return (this.first+this.second+this.third)/3;

    }

}

이렇게 해도 안된다. ㅠㅠ

안되는 이유는 설명할 수 있지만 아직 중요하지 않으니까 일단 넘어가자.

근데 이렇게 하면 코드가 동작한다고 생각을 해보자.

Personplus의 constructor를 아예 새로 재정의하면 코드가 동작한다고 생각을 해보잔 말이다.

이러면 Person을 extends 할 이유가 없게된다.

이 때 사용하는 키워드가 super다.

부모 클래스가 하지 못하는 일을 할 수 있게 한다!

[super 써보기]

class Person{

    constructor(name, first, second){

        this.name = name;

        this.first = first;

        this.second = second;

    }

    sum(){

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

    }

}

class Personplus extends Person{

    constructor(name, first, second, third){

        super(name, first, second);

        this.third = third;

    }

    avg(){

        return (this.first+this.second+this.third)/3;

    }

}

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

이렇게 쓰면 된다. Personplus 클래스를 잘 보자.

Super() 를 하게 되면 부모 클래스의 생성자(constructor)를 호출하게 된다. 호출할 때 매개변수가 필요하므로 name, first, second를 super에 집어넣어줬다.

class Personplus extends Person{

    constructor(name, first, second, third){

        super(name, first, second);

        this.third = third;

    }

    sum(){

        return super.sum()+this.third

    }

    avg(){

        return (this.first+this.second+this.third)/3;

    }

}

Sum이 바뀐다면 어떨까? 부모 클래스는 first + second만 해줬는데 자식인 Personplus는 third까지 해줘야한다면?

위와 같이 불러주면 된다.

자 그럼 super라는 것은 무엇인가?

Super()는 부모의 컨스트럭터, 생성자를 호출하는 것이다.

Super는 부모 클래스 자체를 말한다.

[객체끼리의 상속 object inheritance]

텍스트이(가) 표시된 사진

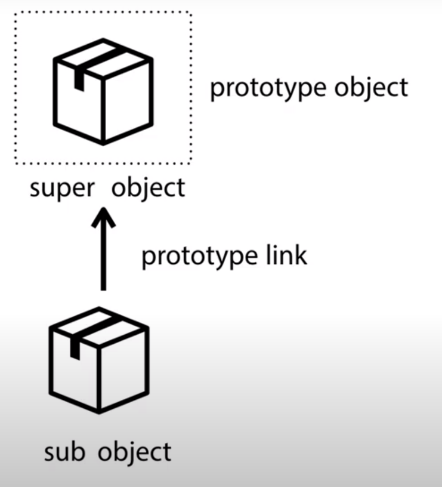
자동 생성된 설명객체지향은 대체로 객체를 만드는 class, 그 class로부터 만들어지는 객체로 나눠진다.

자바스크립트는 같은 객체지향언어인 자바와 이름은 비슷하지만 시스템은 아주 다르다.

자바와 같은 객체지향언어는 왼쪽 그림과 같이 super class, sub class간 상속 관계를 통해서 객체사이의 상속을 구현한다. 객체 사이의 상속은? 클래스의 상속 관계가 만들어낸다. 객체 사이의 상속 관계는 클래스에서 객체를 생성할 때 형성되는 것이다.

물론 자바스크립트에서도 class, extends라는 키워드는 있지만 그냥 장식에 불과하다.

그냥 class, extends라는 것이라 쓸 뿐이지 내부적으로 자바스크립트가 그것을 읽어내는 것은 왼쪽의 방식과는 다르다.

[자바스크립트의 객체상속]

그럼 자바스크립트는 객체끼리의 상속관계를 어떻게 쓸까.

자바스크립트는 객체가 직접 다른 객체의 상속을 받을 수 있다.

또한 얼마든지 그 상속 관계를 바꿀 수 있다.

그림으로 그리자면 오른쪽 그림의 관계와 같다.

자 그러면 prototype object와 prototype link에 대해서 더 자세하게 다루고 배워보도록 하자.

var superObj = {

    superVal: 'super'

}

var subObj = {

    subVal: 'sub'

}

위와 같이 객체를 만들었다. 여기서 superObj와 subObj는 하드코딩으로 선언된 객체인데, 서로 남남 관계에 있다.

자바스크립트는 이렇게 남남인 객체끼리도 상속을 해줄 수 있다. 그러기 위해서는 prototype link를 설정해주면 되는데, 그것은 \_\_proto\_\_(던더 프로토) 프로퍼티를 바꿔줘야한다.

subObj.\_\_proto\_\_ = superObj;

console.log('subObj.subVal => ', subObj.subVal);

// sub

console.log('subObj.superVal => ', subObj.superVal);

// super

이렇게 해줬더니? ㅋㅋ 놀랍게도 \_\_proto\_\_ 값으로 설정한 superObj의 값을 참조하게 됐다.

subObj.superVal = 'subsub';

그럼 이렇게 해주면 어떻게 될까?

subObj.superVal = 'subsub';

console.log('superObj.superVal => ', superObj.superVal);

// super

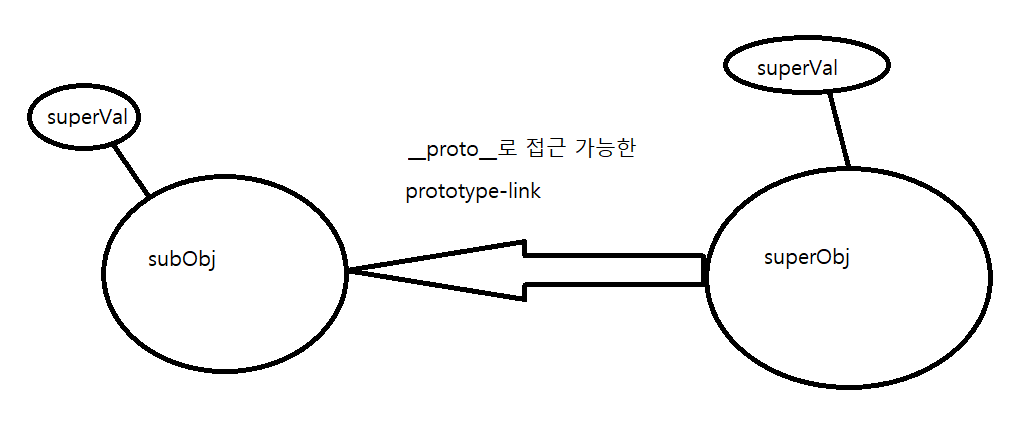
이런 경우에는 superVal의 값이 subsub로 안바뀌었다! 왤까?

subObj.superVal = 'subsub';

위와 같이 코드를 짠다면, subObj의 superVal 프로퍼티의 값을 subsub으로 바꾸는 것이다.

subObj의 부모 객체라고 할 수 있는 superObj로 접근한 것이 아니다.

다음 페이지에서 더 알아보자.



위 그림과 같이 된다는 말이다.

subObj.\_\_proto\_\_.superVal = 'asdfasdf';

console.log('superObj.superVal => ', superObj.superVal);

// asdfasdf

그래서 subObj.\_\_proto\_\_.superVal 로 접근하면 부모 오브젝트인 superObj의 superVal 프로퍼티에 접근해서 값을 만져줄 수 있는 것이다.

꼭 좋은것만은 아니다!!!! 하지만, 아무튼 subObj.\_\_proto\_\_ = 부모 오브젝트 이런 식으로 우리는 상속 관계를 쉽고 유연하게 만져줄 수가 있다! 꼭 좋은것만은 아니다!!! 복잡해지고, 착오가 일어날 수 있기 때문.

\_\_proto\_\_는 이름도, 생긴 것도 이상하지만 자바스크립트 표준은 아니다. 하지만 수많은 자바스크립트 환경에서 \_\_proto\_\_를 지원하기 때문에 사실상 표준이나 다름없다.

그런데 상속을 저런식으로 하면 말했다시피 좋은 방법이 아니다. 정석이 아니라는 말이다.

[정석대로 상속하기 Object.create]

// var subObj = {

//     subVal: 'sub'

// }

// subObj.\_\_proto\_\_ = superObj;

그럼 subObj랑 subObj의 \_\_proto\_\_ 관계를 정석적인 방법으로 만들어보자.

일단 주석처리를 했다.

var subObj = Object.create(superObj);

이렇게 하면 된다. Object 객체의 create 메소드를 사용해서 인자로 superObj를 주면 반환값은 superObj를 \_\_proto\_\_로 하는 객체를 반환하게 된다.

var superObj = {

    superVal: 'super'

}

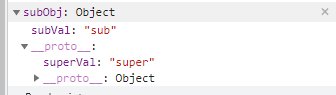
var subObj = Object.create(superObj);

subObj.subVal = 'sub';

그럼 이렇게 한다면? 주석처리한 부분과 똑같이 동작하는 코드가 된다.

Object.create(객체 혹은 constructor function)은 그 객체를 \_\_proto\_\_ 로 하는 객체를 리턴한다.

자, 결론적으로 \_\_proto\_\_보단 Object.create()를 사용해서 객체간의 상속관계, proto link를 정의해주는 것이 더 좋은 방법이다.

자 이렇게 subObj의 \_\_proto\_\_가 super를 가리키고 있는 것을 개발자 도구에서 확인할 수 있다.

[debugger 키워드]

var superObj = {

    superVal: 'super'

}

var subObj = Object.create(superObj);

subObj.subVal = 'sub';

debugger;

참고로 이런 식으로 js 코드가 읽히는 중 debugger 키워드를 만나게 되면 저 줄에서 디버깅이 멈춘다. 고 해야되나 솔직히 잘 모르겠는데 시간이 멈추듯 멈춘다.

무슨 쓸모가 있느냐? 데이터 관련 정보를 확인할 때 쓴다. 어느 시점에 어떤 데이터가 어떤 상태인지 확인하기 좋지 않은가?

[객체 상속 사용하기]

지금 우리는 아직 컨스트럭터 function으로 상속하는 것은 아니고, 수작업 객체를 상속하는 방법을 살펴보고 있다. 의문을 갖지 마라.

kim = {

    name: 'kim',

    first:10, second: 20,

    sum: function(){

        return this.first+this.second;

    }

}

console.log('kim.sum() : ',kim.sum());

자 이런 객체가 있다고 하자.

다른 객체를 만드는데 sum을 정의하기 싫다면?

kim = {

    name: 'kim',

    first:10, second: 20,

    sum: function(){

        return this.first+this.second;

    }

}

lee = {

    name: 'lee',

    first:10, second: 10

}

lee.\_\_proto\_\_ = kim;

console.log('lee.sum() : ',lee.sum());

// lee.sum() :  20

이렇게 해주면 된다. 지금까지 배운 대로라면 이해할 것이다. 맞지?

Lee.\_\_proto\_\_ 를 만져주는 것이다.

만약 lee에 함수를 추가하고 싶다면?

lee.avg = function(){

    return (lee.sum())/2;

}

이렇게 추가하거나,

lee = {

    name: 'lee',

    first:10, second: 10,

    avg:function(){

        return this.sum()/2

    }

}

이런 식으로 객체 내부에서 정의해도 된다.

참고!!!!!!!!!! 아직 그냥 일반 객체다. 하드코딩해서 만든 객체 말이다.

var kim = {

    name: 'kim',

    first:10, second: 20,

    sum: function(){

        return this.first+this.second;

    }

}

var lee = Object.create(kim);

lee.name = 'lee';

lee.first = 10;

lee.second = 10;

lee.avg = function(){

    return this.sum()/2

}

/\*

var lee = {

    name: 'lee',

    first:10, second: 10,

    avg:function(){

        return this.sum()/2

    }

}

lee.\_\_proto\_\_ = kim;

\*/

console.log('lee.sum() : ',lee.sum());

console.log('lee.avg() : ',lee.avg());

이렇게 종합적으로 보면 object.create 써서 상속시키고, 그 안의 객체 데이터들을 일일히 지정해주는 방법이 더 좋다고 한다.

그런데 원래 object.create가 있었던 것은 아니고, \_\_proto\_\_를 쓰기 싫어서 object.create를 만든거니까 아주 오래된 옛날 환경에서는 동작하지 않는다.

그런데 그 옛날이라는 것이 IE9 전후라고 한다.(이고잉 피셜, 정확하게는 기억 못하는 듯, 찾아보니 IE9 부터 사용가능)