[객체와 함수]

Class로 상속을 하지 않고 상속 속성을 \_\_proto\_\_ 값을 바꿈으로써 유연하게 바꿀 수 있다는 것은 사실 전통적인 클래스 형식의 객체지향 개념과는 아주 동떨어진 기능이다.

또, 함수를 호출할 때 new를 호출하면 객체를 생성하는 생성자가 된다.(constructor function)

함수는 이렇게 객체를 생성시키기도 하지만, 객체에 종속되기도 하며, 어떤 객체의 메소드로 들어갈 수도 있다.

[call 에 대해서 배우기]

var kim = {name: 'kim', first: 10, second: 20}

var lee = {name: 'lee', first: 10, second: 10}

위와 같이 kim, lee라는 아무 관련 없는 객체 둘을 만든다.

var kim = {name: 'kim', first: 10, second: 20}

var lee = {name: 'lee', first: 10, second: 10}

lee.\_\_proto\_\_ = kim;

이렇게 하면 lee가 kim의 자식 객체가 된다는 것도 배웠다.

그런데 이렇게 하지않고 object.create()를 쓴다는 것도 배웠다.

여기서 우리가 그러면 이 두 객체의 first와 second를 더해주는 sum 함수를 정의하려면 어떻게 하면 될까.

이번에 그것과 관련된 것을 배운다. call이다.

var kim = {name: 'kim', first: 10, second: 20}

var lee = {name: 'lee', first: 10, second: 10}

function sum(){

    return this.first+this.second;

}

객체 Kim, lee와 아무 관계가 없는 sum이라는 function을 정의했다.

원래대로라면 이 함수를 kim의 프로퍼티로 두고 lee가 그것을 상속하던가 해야겠지만 이젠 아니고 call을 써서 해결하겠다.

var kim = {name: 'kim', first: 10, second: 20}

var lee = {name: 'lee', first: 10, second: 10}

function sum(){

    return this.first+this.second;

}

console.log("sum.call(kim)", sum.call(kim));

결론은, 이렇게 하면 된다.

Call() 이라는 메소드는 모든 함수 객체에 들어있는 메소드이며, call(객체명) 해주면 그객체가 call 메소드로 들어가서 함수를 실행한다.

무슨 말이냐면 call 메소드의 매개변수로 들어간 객체가 그 함수를 실행할 때의 this가 된다는 말이다.

즉 sum.call(kim) 해주면 sum 함수를 실행하되, 그 함수의 this는 kim이 되는 것이다.

아주 기괴한 방식이고, 이건 사실 더 복잡하다고 하는데, 일단 여기까지 이해해보자.

var kim = {name: 'kim', first: 10, second: 20}

var lee = {name: 'lee', first: 10, second: 10}

function sum(prefix){

    return prefix+this.first+this.second;

}

console.log("sum.call(kim) : ", sum.call(kim));

console.log("sum.call(lee) : ", sum.call(lee));

위 코드를 보면 sum 함수는 prefix라는 파라미터를 받는다.

이 파라미터를 우리가 지정해주지 않았기 때문에 sum.call(kim)이나 sum.call(lee) 같은 것들은 모두

NaN이 뜬다.

Call 함수에서, call(this를 해줄 값, 함수의 파라미터1, 파라미터2 …) 이다.

즉 call의 두번째 파라미터부터는 해당 함수에 넣어줄 파라미터 값을 넣어줄 수 있는 것이다.

var kim = {name: 'kim', first: 10, second: 20}

var lee = {name: 'lee', first: 10, second: 10}

function sum(prefix){

    return prefix+this.first+this.second;

}

console.log("sum.call(kim) ", sum.call(kim, '=> '));

console.log("sum.call(lee) ", sum.call(lee, '= '));

// sum.call(kim)  => 1020

// sum.call(lee)  = 1010

위 코드가 잘 동작한다는 것이다.(1020, 1010으로 붙은건 파라미터로 문자열을 줬기 때문에 자동 캐스팅됐기 때문이다.)

이 call은 잘 사용하면 참 좋은데, 잘 사용하기가 어렵다.ㅠㅜ

추가로 apply라는, 똑같이 call과 비슷하고 기괴한 메소드가 존재하는데 나중에 혼자 알아보도록 하자.

var kim = {name: 'kim', first: 10, second: 20}

console.log(kim.call());

참고로 위와 같이 일반 객체는 call이 안붙고 함수에 붙더라. 다른 것도 다 실험해보긴 해야한다.

[bind 배우기]

이전 시간에는 call을 이용해서 어떤 객체의 this, 다시말해서 컨텍스트를 바꾸는 명령어를 배웠다.

그런데 call 못지않게 독특한 이상한 것이 있다. bind이다.

호출될 때마다 this를 바꾸는 Call과 다르게, 아예 함수 내부적으로 this를 어떤 것으로 고정시키는 것이 bind이다.

sum.bind(kim);

이렇게 하면, 이 sum이라는 함수는 똑같이 존재하는데, 내부적으로 this를 kim으로 하는 새로운 함수가 만들어지는 것이다.

그리고 그것을

var kimSum = sum.bind(kim);

이렇게 저장한다.

var kimSum = sum.bind(kim, '-> ');

sum 함수에는 파라미터(prefix, 전 시간 참조)도 있으므로 정확히는 이렇게 해주자.

var kimSum = sum.bind(kim, '-> ');

console.log('kimSum() ', kimSum());

//kimSum()  -> 30

자 이렇게 출력하면? Sum을 복제해서 kim을 컨텍스르토하는 함수를 만들었다.

Call과 bind는 다른듯 비슷해보인다.

그런 차이를 명확하게 해보자.

참 그러고보면 자바스크립트에서 함수란 특별한 지위를 가지고 있는 것이 아닌가?

그럼, Call과 bind는 어떻게 다른가? 얘기해보자.

Call은 함수를 실행할 때 객체를 파라미터로 받아 해당 함수의 컨텍스트, this 값을 그 객체로 한다.

Bind는 똑같이 객체를 파라미터로 받아 해당 함수의 컨텍스트, this 값을 그 객체로 하는 함수를 새로 만들어 리턴한다.

[prototype과 \_\_proto\_\_]

너무 어려워하지 마라. 이걸로 씨름할 시간에 앞에서 배웠던 것을 더 많이 다루고자 할 것 같다.

근데 노력해야 이해할 수 있는 요소들이다.

function Person(){}

이걸 보면 함수는 무엇이라고 생각하는가?

함수는 객체같지 않은가?

var kim = {name: 'kim', first: 10, second: 20}

이것과 비슷하게 생겼으니 말이다.

사실 그렇게 생기지 않았다고 따박따박 우겨대도 별 수 없다.

함수는 객체다.

var Person = new Function();

이렇게 선언해도 된다는 말이다. 물론 무슨 기능을 하는 함수인지 설명하기 힘들지만.

function Person(name){

    this.name = name;

}

이렇게 함수 형태로 constructor function을 선언하면, javascript 내부적으로 Person이라는 function 객체를 만든다. 여기까진 이해할 수 있을 것이다.

그런데 function 객체를 만들 적에는 그 객체에 해당하는 prototype 이라는 객체를 만든다.

console.log(Person.prototype);

그리고 그것을 위 Person.prototype과 같이 참조할 수 있게 한다.

Person.prototype 은 객체이다. javascript에서 함수(function) 객체에는 prototype 객체가 붙는다고 위에 말했다.

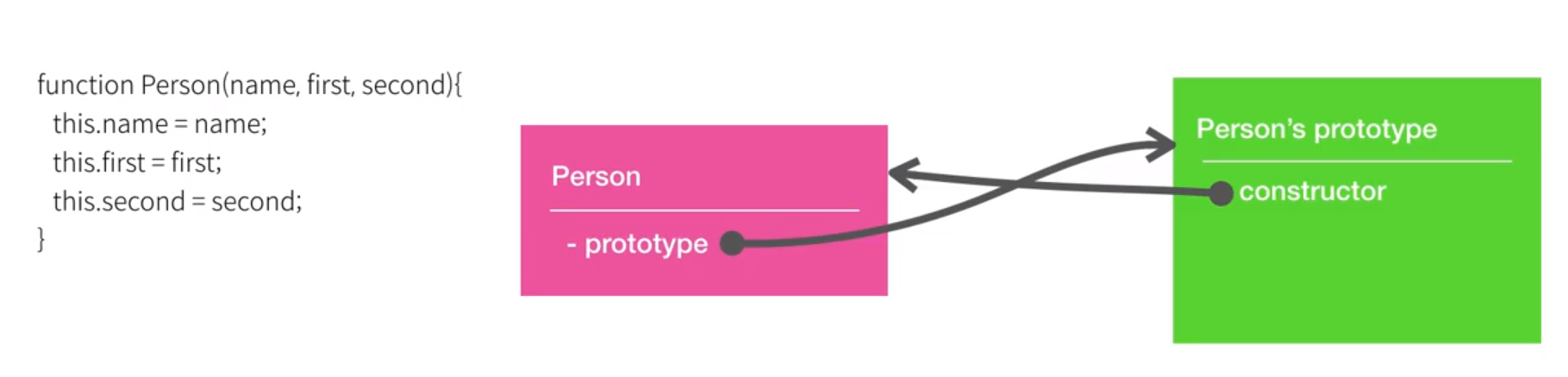
Javascript에서 모든 객체에는 \_\_proto\_\_ 프로퍼티가 붙어있고 이 \_\_proto\_\_는 해당 객체의 constructor function의 prototype을 가리킨다. \_\_proto\_\_ 프로퍼티는 상속 관계를 나타낸다고 전에 배웠으며, 기본적으로 모든 객체는 Object라는 constructor function에 의해 생성되므로,

var lee = {name:'lee'}

위와 같이 손수 선언한 lee 객체의 \_\_proto\_\_ 는 object라는 function의 prototype 객체를 가리킨다.

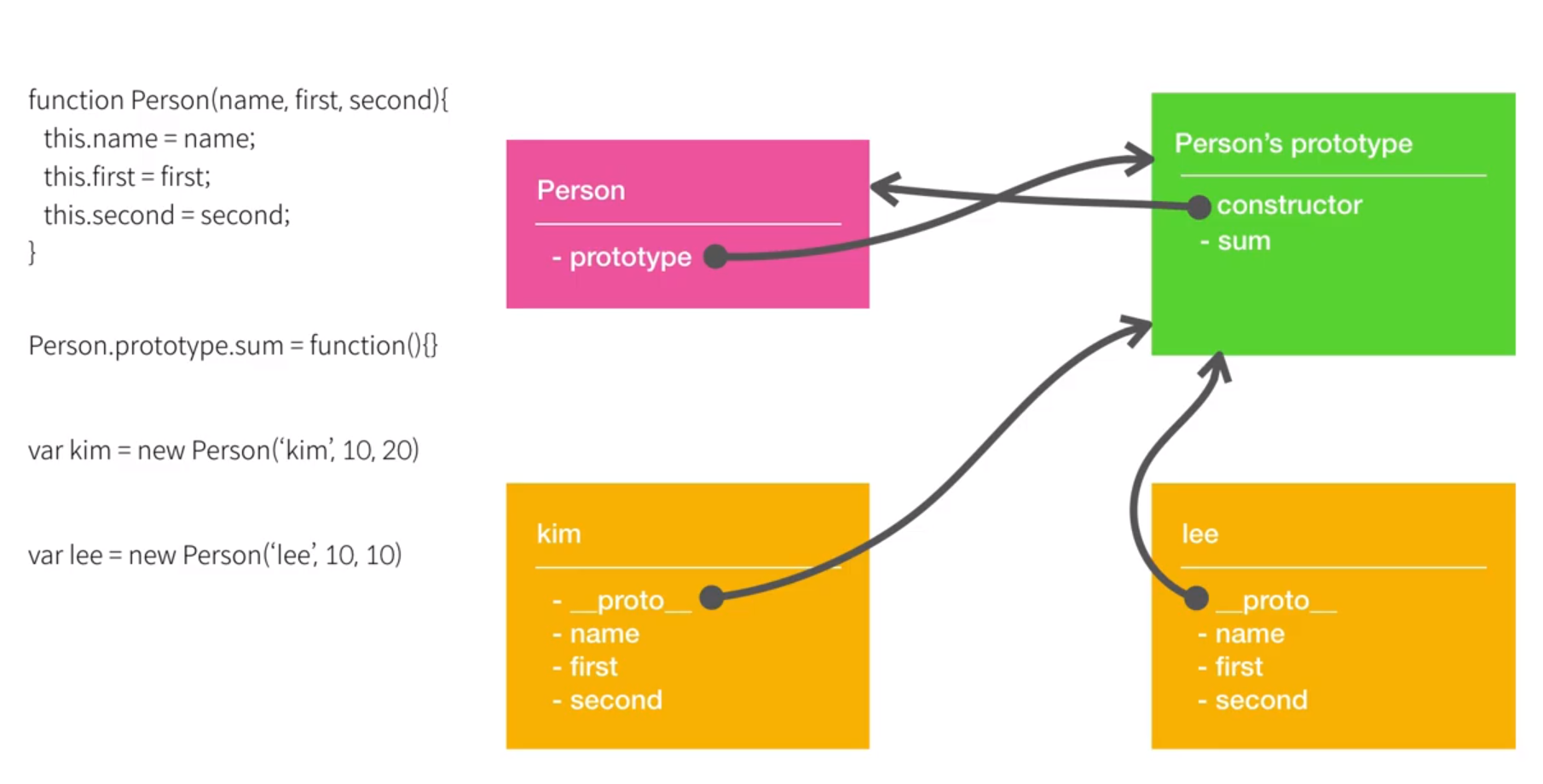
1. 모든 함수(함수도 객체고, 함수 객체다)는 선언될 때 해당 함수의 prototype 객체를 갖는다.
2. 모든 객체는(함수가 아니더라도, 함수여도) 해당 객체의 생성자 함수의 prototype 객체를 가리키는 \_\_proto\_\_ 속성(프로퍼티)을 갖는다.
3. 어떤 객체의 프로퍼티를 찾을 때, 객체가 그 프로퍼티를 갖고 있지 않다면 객체의 \_\_proto\_\_(해당 객체의 생성자의 prototype 객체)를 참조한다.

쉽게 이고잉이 그린 그림으로도 설명한다.



이렇게 Person을 정의하면 Person이라는 객체, Person의 prototype 객체가 생성된다.

이 둘은 서로 알아야 되기 때문에, Person 객체에는 prototype이라는 프로퍼티가 생기고, Person’s prototype에는 contructor 라는 프로퍼티가 생긴다.



그러면 Person 이라는 constructor function으로 생성된 kim, lee 객체들은 Person function 에 따라 name, first, second properties를 갖고, \_\_proto\_\_를 갖는다.

\_\_proto\_\_는 Person function의 prototype 객체를 가리킨다.

[복잡한 거 풀면서 이해해보기]

function a(){

}

function b(){

}

b.prototype = new a();

var obc = new b();

위 코드를 보자.

객체들의 연결 관계를 이해할 수 있을까? 설명을 해보겠다. 대소문자는 구분하지 말자. 힘드니까.

우리가 생성자를 통해 객체를 생성할 때, 우리는 아래와 같이 생성한다.

var someone = new something();

이 상황에서 someone이라는 이름의 객체는 something 생성자를 통해 something 방식으로 생성되고(something 함수 내에 있는 프로퍼티들을 갖고 생성된다) someone.\_\_proto\_\_는 something.prototype을 가리키게 된다. 이를 알고 이해해보자.

Function a는 prototype 객체를 prototype 프로퍼티를 사용해 가지고 있다.

그러니까 a.prototype 이런 식으로 prototype 객체에 접근할 수 있다.

Function b도 a랑 똑같이 이해할 수 있다.

그런데 function b의 prototype을 a의 객체라고 선언해줬다.

그러면 b.prototype은 a 형식의 객체이고, b.prototype.\_\_proto\_\_ 는 a.prototype을 가리키게 된다.

그런데 이 상황에서 obc 를 b 형식의 객체로 생성했다.

그러면 obc의 \_\_proto\_\_는?

Obc의 \_\_proto\_\_는 b.prototype을 가리킨다. B 타입의 객체니까 말이다.

그런데 b.prototype 객체는 a 형식의 객체이다. 이건 중요하진 않다.

그런데 a 형식의 객체임에 따라서, b.prototype.\_\_proto\_\_는 a.prototype을 가리킨다.

즉 obc.\_\_proto\_\_.\_\_proto\_\_ 는 a.prototype을 가리킨다.

만약 obc 객체의 어떤 프로퍼티를 호출할 때 obc 객체에 그 프로퍼티가 없다면, 다음과 같이 호출할 것이다.

1. b.prototype 을 탐색한다(주의해야할 점 = b.prototype은 여기서 a형식의 객체이다.)
2. a.prototype 을 탐색한다
3. object.prototype을 탐색한다

function a(){

    this.sum = function(){

        return 10;

    }

}

function b(){

    this.sum = function(){

        return 1;

    }

}

a.prototype.sum = function(){

    return 100;

}

b.prototype = new a();

var obc = new b();

console.log(obc.sum());

위 코드에서, obc.sum() 은 무엇을 반환하는가?

답은 1이다. Obc는 곧죽어도 b 형식의 객체이고, b는 생성된 객체의 프로퍼티에 1을 return하는 sum 프로퍼티를 주기 때문이다. 다음 문제는 밑에 있다.

function a(){

    this.sum = function(){

        return 10;

    }

}

function b(){

}

a.prototype.sum = function(){

    return 100;

}

b.prototype = new a();

var obc = new b();

console.log(obc.sum());

그럼 이러면 obc.sum()은 무엇이 리턴되는가?

답은 10이다.

왜냐면 obc 객체의 프로퍼티에 sum이 없어서 b.prototype을 탐색하는데, b.prototype은 a 형식의 객체로 선언되어있다. A는 a형식으로 생성된 객체에게 10을 return 하는 sum 프로퍼티를 준다. 따라서 obc.sum은 b.prototype.sum 을 탐색한다.

function a(){

}

function b(){

}

a.prototype.sum = function(){

    return 100;

}

b.prototype = new a();

var obc = new b();

console.log(obc.sum());

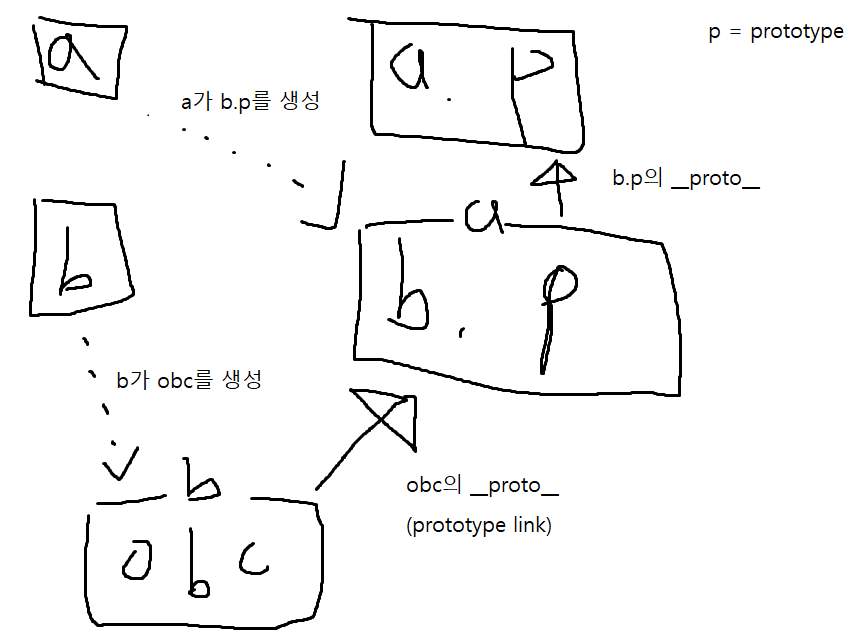
자 이러면 obc.sum()은 무엇을 리턴하는가?

답은 100이다.

Obc에게 sum 프로퍼티가 없으므로 b.prototype을 참조하는데 b.prototype 객체에도 sum 프로퍼티가 없으므로 탐색을 못한다. 그런데 b.prototype 객체는 a 객체이므로 prototype link(\_\_proto\_\_)로서 a.prototype을 가리킨다. 따라서 b.prototype 객체에서 sum을 못찾으므로 b.prototype.\_\_proto\_\_에서 sum을 찾게 된다. 이는 a.prototype 에서 sum 프로퍼티를 찾는 것이다.

따라서 a.prototype.sum()을 호출한다.

그림으로 나타내보겠다.



그림판으로 그려서 좀 못그렸지만..

네모난 사각형은 객체를 뜻한다. A, b도 다 객체다 function 객체 말이다.

A.prototype는 무슨 객체일까? Object 객체다.

function a(){

}

function b(){

}

a.prototype.sum = function(){

    return 100;

}

b.prototype = new a();

var obc = new b();

console.log(obc);

console.log(obc.\_\_proto\_\_);

console.log(b.prototype);

console.log(b.prototype.\_\_proto\_\_);

console.log(a.prototype.\_\_proto\_\_);

그럼 위 코드의 결과를 알 수 있을까?

텍스트이(가) 표시된 사진

자동 생성된 설명

결과는 위와 같다. 그런데 a{}\_\_proto\_\_ 가 Object인데 그 밑에 있는 놈도 \_\_proto\_\_가 Object로 같은게 신경쓰이지 않는가?

텍스트이(가) 표시된 사진

자동 생성된 설명

궁금해서 열어보니

이렇게 console의 \_\_proto\_\_라는 것은 \_\_proto\_\_로 가리키는 객체가 어떤 형식의 객체인지를 나타내는 것이다 라는 걸 알 수 있었다.

B의 prototype은 A형식의 객체이고, 그러다보니 b.prototype.\_\_proto\_\_는 a.prototype을 가리킨다.

일반적으로 어떤 생성자 함수(constructor function)의 prototype은 객체(Object)이다. Object 라는 형식인 것이다.

따라서 b.prototype의 \_\_proto\_\_는 object 형식인 것이다.

a.prototype의 \_\_proto\_\_는 당연히 object.prototype이니까 object 형식이고 말이다.