Javascript - 상속

- ChildClassName.prototype = new ParentClass(); 형식으로 상속받을 수 있다.
- ChildClassName.prototype.constructor=ChildClassName 를 사용하여 생성자 속성을 초기화하는 것을 잊지 말아라.
- 오버라이딩 된 함수를 갖는 자식 클래스에서 Function.call()을 이용하여 부모 클래스의 함수를 호출할 수 있다.
- JavaScript는 protected 멤버를 지원하지 않는다.

Example

```
function Mammal(name){
    this.name=name;
    this.offspring=[];
}
Mammal.prototype.haveABaby=function(){
    var newBaby=new Mammal("Baby "+this.name);
    this.offspring.push(newBaby);
    return newBaby;
}
Mammal.prototype.toString=function(){
    return '[Mammal "'+this.name+'"]';
}


Cat.prototype = new Mammal();        // Here's where the inheritance occurs
Cat.prototype.constructor=Cat;       // Otherwise instances of Cat would
have a constructor of Mammal
function Cat(name){
    this.name=name;
}
Cat.prototype.toString=function(){
    return '[Cat "'+this.name+'"]';
}


var someAnimal = new Mammal('Mr. Biggles');
var myPet = new Cat('Felix');
alert('someAnimal is '+someAnimal);   // results in 'someAnimal is [Mammal
"Mr. Biggles"]'
alert('myPet is '+myPet);             // results in 'myPet is [Cat "Felix"]'

myPet.haveABaby();                    // calls a method inherited from
Mammal
alert(myPet.offspring.length);        // shows that the cat has one baby now
alert(myPet.offspring[0]);            // results in '[Mammal "Baby Felix"]'
```

Using the .constructor property

Look at the last line in the above example. The baby of a Cat should be a Cat, right? While the haveABaby() method worked, that method specifically asks to create a new Mammal. While we could make a new haveABaby() method for the Cat subclass like this.offspring.push(new Cat("Baby "+this.name)), it would be better to have the ancestor class make an object of the correct type.

Every object instance in JS has a property named constructor that points to its parent class. For example, `someAnimal.constructor==Mammmal` is true. Armed with this knowledge, we can remake the haveABaby() method like this:

```
Mammal.prototype.haveABaby=function(){
    var newBaby=new this.constructor("Baby "+this.name);
    this.offspring.push(newBaby);
    return newBaby;
}
...
myPet.haveABaby();                    // Same as before: calls the method
inherited from Mammal
alert(myPet.offspring[0]);            // Now results in '[Cat "Baby Felix"]'
```

Calling 'super' methods

Let's extend the example now so that when baby kittens are created, they 'mew' right after being born. To do this, we want to write our own custom Cat.prototype.haveABaby() method, which is able to call the original Mammal.prototype.haveABaby() method:

```
Cat.prototype.haveABaby=function(){
    Mammal.prototype.haveABaby.call(this);
    alert("mew!");
}
```

The above may look a little bit bizarre. Javascript does not have any sort of 'super' property, which would point to its parent class. Instead, you use the call() method of a Function object, which allows you to run a function using a different object as context for it. If you needed to pass parameters to this function, they would go after the 'this'. For more information on the Function.call() method, see the MSDN docs for call().

Making your own 'super' property

Rather than having to know that Cat inherits from Mammal, and having to type in Mammal.prototype each time you wanted to call an ancestor method, wouldn't it be nice to have your own property of the cat pointing to its ancestor class? Those familiar with other OOP languages may be tempted to call this property 'super', but JS reserves this word for future use. The word 'parent', while used in some DOM items, is free for the JS language itself, so let's call it parent in this example:

```
Cat.prototype = new Mammal();
Cat.prototype.constructor=Cat;
Cat.prototype.parent = Mammal.prototype;
...
Cat.prototype.haveABaby=function(){
    var theKitten = this.parent.haveABaby.call(this);
    alert("mew!");
    return theKitten;
}
```

Spoofing pure virtual classes

Some OOP languages have the concept of a pure virtual class...one which cannot be instantiated itself, but only inherited from. For example, you might have a LivingThing class which Mammal inherited from, but you didn't want someone to be able to make a LivingThing without specifying what type of thing it was. You can do this in JS by making the virtual class an object instead of a function.

The following example shows how this could be used to simulate a pure virtual ancestor:

```
LivingThing = {
    beBorn : function(){
        this.alive=true;
    }
}
...
Mammal.prototype = LivingThing;
Mammal.prototype.parent = LivingThing;   //Note: not 'LivingThing.prototype'
Mammal.prototype.haveABaby=function(){
    this.parent.beBorn.call(this);
    var newBaby=new this.constructor("Baby "+this.name);
    this.offspring.push(newBaby);
    return newBaby;
}
```

With the above, doing something like `var spirit = new LivingThing()` would result in an error, since LivingThing is not a function, and hence can't be used as a constructor.

Convenient Inheritance

Rather than writing 3 lines every time you want to inherit one class from another, it's convenient to extend the Function object to do it for you:

```
Function.prototype.inheritsFrom = function( parentClassOrObject ){
    if ( parentClassOrObject.constructor == Function )
    {
        //Normal Inheritance
        this.prototype = new parentClassOrObject;
```

```
        this.prototype.constructor = this;
        this.prototype.parent = parentClassOrObject.prototype;
    }
    else
    {
        //Pure Virtual Inheritance
        this.prototype = parentClassOrObject;
        this.prototype.constructor = this;
        this.prototype.parent = parentClassOrObject;
    }
    return this;
}
//
//
LivingThing = {
    beBorn : function(){
        this.alive = true;
    }
}
//
//
function Mammal(name){
    this.name=name;
    this.offspring=[];
}
Mammal.inheritsFrom( LivingThing );
Mammal.prototype.haveABaby=function(){
    this.parent.beBorn.call(this);
    var newBaby = new this.constructor( "Baby " + this.name );
    this.offspring.push(newBaby);
    return newBaby;
}
//
//
function Cat( name ){
    this.name=name;
}
Cat.inheritsFrom( Mammal );
Cat.prototype.haveABaby=function(){
    var theKitten = this.parent.haveABaby.call(this);
    alert("mew!");
    return theKitten;
}
Cat.prototype.toString=function(){
    return '[Cat "'+this.name+'"]';
}
//
//
var felix = new Cat( "Felix" );
var kitten = felix.haveABaby( ); // mew!
alert( kitten );                 // [Cat "Baby Felix"]
```

Just make sure you call this method immediately after your constructor, before you extend the prototype for the object.

Protected methods?

Some OOP languages have the concept of 'protected' methods—methods that exist in a parent or ancestor class that can only be called by descendants of the object (on each other), but not by external objects. These are not supported in JS. If you need such, you will have to write your own framework, ensuring that each class has a 'parent' or some such property, and walking up the tree to find ancestors and checking whether or not the calling object is the same type. Doable, but not enjoyable.

참고

- Simple JavaScript Inheritance