Course link:
https://classroom.udacity.com/courses/ud015

Scopes:

```
//Outer scope

function try(){

//Inner scope

};



if(){

//does not create scope limitations, only functions do

};
```

Prototype Chain:

Almost every function relates to a global object, which has functions and properties linked to the created object.
So if the toString object has a new function the, the yellow function will be able to use this function as well.

Classes:

The differentiation between a class and a decorator is the following:
Classes:
 - Classes has no recieved object arguments
 - Classes creates similar instances of itself
 - The functions of the classes are called constructors
 - Starts with a capital letter
Decorators:
- Recieve the objects as arguments

```
//Decorator, object recieved as an argument and at the end the modified object is returned

var carLike= function(obj,loc){

obj.loc = loc;

obj.move = function(){

    obj.loc++;

};

return obj;

};

var amy = carLike({},1);

amy.move();

var ben = carLike({},9);

ben.move();


//Class, no object has been recieved, but at the end an ojbect will be created.

var Car = function(loc){

var obj = {loc :loc};

obj.move = function(){

    obj.loc++;

};

return obj;

};


var amy = Car(1);

amy.move();

var ben = Car(9);

ben.move();
```

Class Duplicity reduction:

In the previous code, we can see that the Car class creates an object with the same function move() for every object instead of having one and referencing to it.

```
//So here we have the same class, but for each created an object, we just point to the already
 existing move function so we don't have to create it every time
var Car = function(loc){
var obj = {loc :loc};
obj.move = move();
return obj;
};
//This. will refer to the function which called it.
var move = function(){
    this.loc++;
};
var amy = Car(1);
amy.move();
var ben = Car(9);
ben.move();
```

Property accessing for shared functional patterns: (Class pattern)

- Created with object literal ({})
- Custom property defined where all of the shared functions are stored
- Usage of extend(obj, Custom property defined);

Basically in the previous example if we create a new shared function we always have to go to the Car class to add a property name of the new function example
obj.functionName, + we have to declare them seperately in the function code.
Instead of doing this, we can **create a property for the Car function** called as Car.methods , which will store an object, with properties as functions.
So the only thing what we have to to is to combine the Car class with its property called Car.methods, to create an object.

(Picture)
There is a Car variable, which has a actually a function, with property methods, this property has the move function as a property.
the created variable ben,amy has the same function as the Car varialbe, the everyting is poiting to the same function code for moving the car.

```
//Extend requires jquery plugin.

//Car is a class which creates similar objects + extends itself with the Car.methods property o
bject

var Car = function(loc){

var obj = {loc :loc};     //Created with object literal, means  with directly defining the obje
ct with "{}"

extend(obj, Car.methods);

//We just merge the Car class object +  the created property of the Car class such as Car.metho
ds, so the Car object will have all of the properties of the Car.methods() funciton.property.

return obj;

};

//The car function is an object as well, which is capabale of being called and storing properti
es as well.

Car.methods = {

        move:function(){this.loc++},

};

var amy = Car(1);

amy.move();

var ben = Car(9);

ben.move();
```

Prototypal Class:

- Using Object.create(x.prototype);
- Storing the shared functions in the built in prototype property.
- Creating instances of the prototype object

(Picture)
- **Car** class has an **function with property of prototype**. This prototype property has a other functional properties such as move, which have a function object.
- Ben,Amy is the **created instances of the Car** class, which has the own **.loc property** + the **delegated**/shared **prototype functions(_.proto_.move)**
- As a result, the move function object is created once and referenced by every instance created by the Car class.

Car function is a prototypal class since it is using Object.Create(), which will automatically have a property which will hold methods called as "prototype", when you pass new functions to the prototype, it will automatically be available for all of the instances created with the prototypal class.

```
var Car = function(loc){
var obj = Object.create(Car.prototype);                //Basically this is where the object is
 created, constructed, + have a prototype property.
obj.loc = loc;
return obj;
};
//The car function is an object as well, which is capabale of being called and storing properti
es as well.
Car.prototype.move= function(){this.loc++};


console.log(Car.prototype.constructor);             //Will tell the  which class, object (funct
ios are object as well) has created this instance
console.log(amy.constructor);                       //The constructor is the amy instance is th
e Car object function
log (am instanceof Car);                            //True, since the object is created from th
e Car object function as well
```

The objects created from a decorator will be not the instances of the decorator.

```
//Decorator, object recieved as an argument and at the end the modified object is returned
var carLike= function(obj,loc){
obj.loc = loc;
obj.move = function(){
    obj.loc++;
};
return obj;
};
var amy = carLike({},1);
```

```
log (amy instanceof Car);           //False, the amy object will not be the instance of the Car
 decorator.
```

Pseudoclassical Patterns:

Basically by using the keyword `New` upon calling a function, javascript do some thing for us, in order to convert the function to be a class which generates instances.

Notes:
- In one part of the code, you should specifying how all of the instances should be similar (stored as proerties in the prototype object)
- Inside the body of the constructor function, we should specify how the instances will be differentiated.

When keyword `new` has been used it will do the following things to us:
- It will create an instance of the called function
- Will create a prototype property of the object. **// 1**
- Since it is creating a new object it can be referenced in the function context as **.this // 2**

```
var Car = function(loc){

// this = Object.create(Car.prototype);   // 1

this.loc = loc ;

// return this;                           // 2

};


Car.prototype.move = function(){

this.loc++;

};


var amy = new Car(1);

amy.move();
```

Pseudoclassical Subclasses:

(Creating a super class of `Car` with a subclass of `Van`)
But creating two instances of Car objects uncessarily.
Having a super class, which creates a **Car** object and we would like to create a new kind of Car object which will have different kind of properties (**Van**)

```
// Super class

var Car = function(loc){

// this = Object.create(Car.prototype); due to using new

    this.loc = loc;            //Since it has been called with the keyword new .this will refer

 to the newly created instance of Car

};

//Adding some prototype function

Car.prototype.move = function(){

    this.loc++;

};

//Trying to create a new Van ojbect, using the Car prototypal class.

var Van = function(loc){

// this = Object.create(Van.prototype); due to using new

    return new Car(this.loc);                //We are just calling the Car object, to rename it

 as a Van object, with the same properties,

}

var zed = new Car(3);

zed.move();

var amy = new Van(9);

amy.move();
```

Effective Pseudoclassical Subclasses:

(Picture)
1. We have a main `Car` variable, which has a prototype property which includes:
- Automatic constructor
- Methods poiting to function objects
2. We have an instance of the Car object as zed which has a unique property of .loc, and all of the shared prototype properties as `_.proto_`.
When zed.move() is called, it tries to find it in the instance of the Car object called as zed, but since it doesn't have a function like this it will fall back to it's Car.prototype property function, which delegates the task to the main Car object prototype.move function, which will run the 1 stored funcion object
3. We have a `Van` variable which is a seperate object, with a seperate prototype property which delegates to the Car.prototype property if it doesn't have the request function (like move).

```
// Super class
var Car = function(loc){
// this = Object.create(Car.prototype); due to using new
    this.loc = loc;
    }
Car.prototype.move = function(){
    this.loc++;
};

var Van = function(loc){
    Car.call(this,loc);            //this reference to the Van object since it has been created w
ith the new keyword
                                   // With call we are using the Car function with passing the
 van object
}
//We are creating a delegation from the Van.prototype to search for functions at the Car.protot
ype, so whenever van.move() is called, it will fail and fall back to the Car.prototype object p
roperty which will provide the function.
Van.prototype = Object.create(Car.prototype);

//Since we borrowed the Car.prototype function we have to delegate manually the constructor fun
ction of the prototype property
Van.prototype.constructor = Van;
//Now after we have delegated the Car.prototype function with the Object.create() function we c
an assing methods to it.
Van.prototype.grab = function (){console.log('Grabbed!')};

var zed = new Car(3);
zed.move();
var amy = new Van(9);
amy.move();                        //Delegated method to the Car.prototype function
amy.grab();                        //Delegate method to the Van.prototype function
```

```
console.log(amy.constructor);      //Since we manually set the constructor to be the Van object, we get 'Van'
```