**JAVA SCRIPT**

1. C Family (C++, JAVA, etc…)
2. **What is JavaScript?**

Mozzilla development network (MDN)

* Often as JS
* Lightweight (easy to run)
* Interpreted (no compilation, executed directly)
* Object oriented
* First class functions (can send a function as an argument)
* Scripting (instruction are written for runtime environment, like batch file, shell script)

**Why learn JS?**

* Client side development (web browser)

Rich functionality basis for JQuery

Base for React, Anguler, Ember

* Server side framework based on JS: NodeJS (Express) Chrome-Plugins (add-ons)

**History?**

* Brenden Eich at Netscape (called live script)
* Created in 10 days ☺
* Based on Applet which gave the idea of rich client
* Called JS just for marketing reasons
* Made easy for use 🡪 goal of the language (for anyone to get into)
* To make a standard (specifications) they made it by ECMA therefor ECMAScript

Now it is ECMAScript 6 but not fully supported (therefor gonna use ECMAScript 5)

**Heads-up**

* Somtims confusing since it looks like a familiar language but acts differently

1. HTML + JS (changes the DOM tree)
2. Working environment

Use fire-fox scratchpad (while theConsole is open F12), or download Chrome extensions Scratch JS and the same (cooler)

* Print in the Console: console.log("hello world!");
* Run in Scratch few lines to show the ability: console.log("hello world!"); …

PART I

1. Primitives:

**There is no type definition, only var!!**

**Number** = double precision 64-bit (no integer, float, etc.)

**String** = Only strings (no characters. Character = string size 1)

**Boolean**

**Symbol (ECMA Script 6) – enumeration**

1. var x = 42

x = "hello";

console.log(x);

hello

console.log(typeof x);

string

1. var b;

console.log(b);

undefined

console.log(typeof b);

undefined

1. y = **null**;

console.log(typeof y);

object ( js bug! Cannot fix it now since it will break older code….)

console.log(y);

null

1. There are no modifiers (public, private, etc.) -- only scope
2. **Type coersion**

Type coercion means that when the operands of an operator are different types, one of them will be converted to an "equivalent" value of the other operand's type. For instance, if you do:

boolean == integer

var a = "5"

var b = 5

if (a **==** b)

{

console.log("equal");

}

1. **The === Also check the type**

var a = "5";

var b = 5;

if (a **===** b)

{

console.log("equal");

}

else

{

console.log("**not equal**");

}

1. **For all non-zero will be true (also string which is not "" and any-other not undefined and not null)**

var b=42;

**if (b)**

{

console.log("here");

}

PART II – Classes/Objects

1. **Not Class-Based!!**
2. **Easiest way to create in-line (literal)**

**No rules, can add anything to it**

var mystring = "hello"

var myobject = { } // empty object

console.log(myobject);

Object {}

Myobject.prop = "hello"; // adding a property

console.log(myobject);

Object { prop: "hello" }

Myobject.prop2 = 42; // adding a property

console.log(myobject);

Object { prop: "hello" , prop2: 42}

console.log(myobject.prop2);

42

1. **Can also do it in-line (also can add later a property)**

**Objects can also have methods**

var myobject = {

"number" : 123,

"name" : "itay"

}

console.log(myobject);

Object {number: 123, name: "itay"}

1. **All properties are public!! (there is a hack)**

**If you access a property which does not exist you get undefined**

1. **Dot notation + []**

var myobject = {

"number" : 42,

"name" : "itay"

}

console.log(myobject["number"]);

42

When to use it?

* + - When the property name is a saved-keyword (i.e. "1")
    - When you want to use a pre-defined string.

var myobject = {

"if" : 123,

"name" : "itay"

}

console.log(myobject);

var fieldName = "if";

console.log(myobject[fieldName]);

* + - Dot notation is faster (due to optimization)

1. **Objects are also location in the memory, you can have another pointer to the same location.**

**Object property could also be an Object! (nested)**

var myobject = {

"if" : 123,

"name" : "itay",

"objPropert" : {

"innerProp" : "inside"

}

}

console.log(myobject["objPropert"]);

Object {innerProp: "inside"}

console.log(myobject.objPropert.innerProp);

inside

1. **==, ===**

Object1 = object2

If ( object1 == object2) // equal

If ( object1 === object2) // equal

var myobject = {

"x" : 5,

"y" : 10

}

var myobject2 = {

"x" : 5,

"y" : 10

}

If (myobject == myobject2) // NOT equal

If (myobject === myobject2) // NOT equal

1. **You can set a property to undefined**

myObject.propertyName = undefined

1. **If you want to remove a property from the object use delete**

delete myobject.propertyName;

**Undefined vrs NULL**

**Arrays:**

**(1)**

var arrString = ["first", "second", "third"];

var arr=[1,2,3];

var arr2 = arr;

console.log(arr[0]);

console.log(arr);

console.log(arr.length);

arr[3] = 12; // adding on-the-fly

console.log(arr);

console.log(arr.length); //4

console.log(arr2.length); //4

arr[500] = 1;

console.log(arr);

console.log(arr.length); // the array is in length of 501

console.log(arr[300]); //undefined

**(2)push, pop, shift, unshift**

var arr = [1,2,3];

arr.push(10); // [1,2,3,4]

arr.pop();// [1,2,3]

arr.pop();// [1,2]

console.log(arr); // [1,2]

unshift(x): The unshift() method is like the push() method, only it works at the beginning of the array

shift(): The shift() method is like the pop() method, only it works at the beginning of the array

var complexArray = {1, 2, 3, "hello" , {} };

**(3) foreach**

var printElement = function(item)

{

console.log(item);

}

var arr = [1,2,3];

arr.forEach(printElement); // 1,2,3

**Primitives/Objects**

In JavaScript there are 5 primitive types: undefined, null, boolean, string and number

typeof true; //"boolean"

typeof Boolean(true); //"boolean"

typeof new Boolean(true); //"object"

typeof (new Boolean(true)).valueOf(); //"boolean"

typeof "abc"; //"string"

typeof String("abc"); //"string"

typeof new String("abc"); //"object"

typeof (new String("abc")).valueOf(); //"string"

typeof 123; //"number"

typeof Number(123); //"number"

typeof new Number(123); //"object"

typeof (new Number(123)).valueOf(); //"number"

**If primitives have no properties, why does "abc".length return a value?**

Because JavaScript will readily coerce between primitives and objects. In this case the string value is **coerced** to a string object in order to access the property length

PART III – Functions

function sayHello()

{

console.log("hello");

}

sayHello();

**no method ovewrloading**

function sayHello( a, b)

{

console.log("hello1");

}

function sayHello( a, b, c)

{

console.log("hello2");

}

sayHello(1);

sayHello(1, 2, 3);

**// will always print hello2**

**window.sayHello**

**function sayHello(a, b, c) { … }**

**arguments** JavaScript functions have a built-in object called the arguments object (not an array!!)

function sayHello( a, b)

{

console.log(a);

console.log(b);

console.log(arguments[2]);

}

sayHello(1, 2, "suprise!");

**the third argument was passed (despite not appearing in the signature!!)**

**Return value**

(1)

function calc()

{

return 5;

}

var x = calc();

console.log(x);

**// 5**

(2)

function nothing()

{

return ;

}

var y = nothing();

console.log(y);

**// undefined**

**Function Expressions**

var f = function func()

{

return 5;

}

console.log(f); **// prints method**

console.log( f() ); **// 5**

**Function as an Argument**

**(1)**

var executor = function (funcName)

{

console.log ( funcName() );

}

var f = function func()

{

return 5;

}

executor(f); **// 5**

**(2)**

var executor = function (funcName, number)

{

console.log ( funcName(number) );

}

var f = function func(number)

{

return number;

}

executor(f, 3);  **// 3**

**Function inside an object**

(1)

var Calculator = {

"x" : 0,

"y" : 0,

"add" : function ()

{

console.log(Calculator.x);

console.log(Calculator.y);

console.log(Calculator.x + Calculator.y);

}

}

var calc1 = Calculator;

calc1.x = 5;

calc1.y = 9;

calc1.add();**// 5, 9, 14**

**But if I create another object, i.e.**

**var calc2 = Calculator I will always be stuck the same object!!**

**So better use this**

(2)

var Calculator = {

"x" : 0,

"y" : 0,

"add" : function ()

{

console.log(**this**.x);

console.log(**this**.y);

console.log(**this**.x + **this**.y);

}

}

var calc1 = Calculator;

calc1.x = 5;

calc1.y = 9;

calc1.add(); **// 5, 9, 14**

**Nested Objects**

(1)

var Person = {

"first\_name" : "",

"last\_name" : "",

"get\_name" : function ()

{

return this.first\_name + " " + this.last\_name;

} ,

}

var itay = Person;

itay.first\_name = "itay";

itay.last\_name = "hau";

console.log( itay.get\_name() );

(2)

var Person = {

"first\_name" : "",

"last\_name" : "",

"get\_name" : function ()

{

return this.first\_name + " " + this.last\_name;

} ,

"print" : function ()

{

console.log ( "name = " + this.get\_name() );

console.log ( "address = " + this.Address.street + " " + this.Address.city );

} ,

"Address" :

{

"street": "",

"city" : ""

}

}

var itay = Person;

itay.first\_name = "itay";

itay.last\_name = "hau";

itay.Address.street = "101 Hertzel";

itay.Address.city = "Tel Aviv";

itay.print();

**//result:**

**name = itay hau address = 101 Hertzel Tel Aviv**

PART IV – Scopes and Closures

**All objects are global:**

{

var y = 12;

}

console.log(y); **// 12**

Whenever you create variable it is added to **window.<name>**

**Unless inside functions**

function hello()

{

var x2 = 19;

}

console.log(x2); **// undefined**

**Inner value wins:**

var inner = 50;

function print()

{

var inner = 20;

console.log(inner); **// 20**

}

print();

console.log(inner); **// 50**

**Parameters are value types: (unless they are objects!)**

var name = "itay";

function printName(name)

{

name = "new";

console.log(name);

}

printName(name);

console.log(name);

**Parameters-Objects are ref types:**

function modify(product)

{

product.name = "computer";

}

var product = Product;

product.name = "TV";

modify(product);

console.log(product); **// computer**

**Immediately-Invoked Function Expression (IIFE)**

(function saveSpace()

{

var arr = [1,2,3,4,5,6,7];

console.log(typeof(arr));

console.log(arr.length);

})();

Console.log(arr.length) **// ERROR – does not exist!**

**Read Vrs Write**

Read will create an error when reference is not **VALUE/NULL/UNDEFINED**

Read will create an error when reference is not **VALUE/NULL/UNDEFINED**

1. var a = 10;

console.log(d); **// ERROR – does not exist!**

1. var d = undefined;

console.log(d); **// undefined**

1. new1 = undefined;

console.log(new1); **// ERROR – undefined needs a var**

1. var new2 = null;

console.log(new2); **// undefined**

1. new3 = null;

console.log(new3); **// ERROR – null needs a var**

1. function test(param1)

{

console.log(param1);

}

test();**// undefined – no error since it is like var param1**

1. var test\_new;

console.log(test\_new);**// undefined – no error due to var**

Window object

var a = 5;

console.log(window.a); **// 5**

var hello = "hello";

console.log(window.hello); **// hello**

**METHODS – are also window objects-**

function foo()

{

console.log("foo");

}

console.log(window.foo); **// will print the foo method**

**Scope**

The compiler reads the code and prepares the vars (Global, func)

(1)

var a = 10;

function func() {

var b = 20;

console.log(a + b);

}

func();

**Global: a, func()**

**func: b**

**(2)**

var a = 10;

var b = 20;

function func() {

var b = 20;

console.log(a + b);

**// or: console.log(a + window.b);**

}

func();

**Global: a, b, func()**

**func: b**

**(3)**

var a = 10;

function func() {

c = 20;

console.log(c);

}

func();

console.log(c);

**Global: a, func(), c**

**If we do not write var – it will write to global!!!! (see solution below)**

**solution : How to avoid that? Answer: strict mode** JavaScript in strict mode does not allow variables to be used if they are not declared

"use strict"

var long\_name = 10;

function func() {

lone\_Name = 20;

console.log(long\_Name);

}

func();

**This will cause an error and by that prevent the global variable**

Use strict could be at function level:

var x455 = 10;

function importantFunction()

{

"use strict"

x456 = 12;

console.log(x456);

}

importantFunction();

**(4)**

**Sequence:**

var a = 10;

function outer()

{

var b = a;

console.log(b);

function inner()

{

var c = b;

console.log(c);

}

inner();

}

outer();

what will be printed? Why?

**Solution**: it always starts by looking in current scope if not found? Going one level up till Global and then error.. SCOPE CHAIN

**(4) Hoisting:**

console.log(aaa); //undefined, but not error

var aaa = 20;

…methods…

**with methods**

f1(); // will run f1

function f1()

{

console.log("hi from f1");

}

This must be like that because i.e.

function f1()

{

f2(); **// must be hoisted** }

function f2()

{

F1();}

**Conclusion: declare all your variables at the top!!!**

**(5) Closure:**

A **closure** is an inner function that has access to the outer (enclosing) function's variables—scope chain. The **closure** has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables var a = 10;

function outer()

{

var b = 20;

var inner = function() {

a++;

b++;

console.log(a);

console.log(b);

}

return inner;

}

var innerFn = outer();

innerFn();

console.log("===========");

var innerFn2 = outer();

innerFn2();

**what will be printed?**

**11**

**21**

**===========**

**12**

**21**

**(6) Closure in time-out. Will print after 1000**

var x = 10;

function print()

{

console.log(x);

}

setTimeout( print, 1000);

x = x + 100;

console.log("Done!"); // **will print 110;**

**how can we solve this to print 10?**

**Solution1** :

var x = 10;

function print(x)

{

console.log(x);

}

setTimeout( print, 1000, x);

x = x + 100;

console.log("Done!");

**Solution2** :

var i;

for (i = 0; i < 10; i++)

{

(function (currentI) {

setTimeout(function() { console.log(currentI);}, 1000);

})(i);

}

createPerson()

{

var firstName = "first";

var lastName = "last";

var returnObject = {

"getFirstName" : function()

{

return firstName;

},

"getLastName" : function()

{

return lastName;

}

}

return returnObject;

}

var person = createPerson();

console.log(person.getLastName()); **//last-name**

console.log(person.lastName); **//undefined!!**

**Now add setters**

function createPerson()

{

var firstName = "first";

var lastName = "last";

var returnObject = {

"getFirstName" : function()

{

return firstName;

},

"getLastName" : function()

{

return lastName;

},

"setLastName" : function(name)

{

lastName = name;

},

"setFirstName" : function(name)

{

firstName = name;

}

}

return returnObject;

}

var person = createPerson();

console.log(person.getLastName()); **// last**

person.setLastName("new");

console.log(person.getLastName()); **// new**

PART IV – Objects & Prototypes

**(1) Create object**

function createEmployee(fname, lname, gender, salary)

{

var newObject = {};

newObject.fname = fname;

newObject.lname = lname;

newObject.gender = gender;

newObject.salary = salary;

return newObject;

}

var employee1 = createEmployee("bob", "builder", "M", 10000);

console.log(employee1);

**(2) Improve**

function createEmployee(fname, lname, gender, salary)

{

//var newObject = {};

this.fname = fname;

this.lname = lname;

this.gender = gender;

this.salary = salary;

//return newObject;

}

var employee1 = new createEmployee("bob 2", "builder", "M", 10000);

console.log(employee1);

**(3) this**

**(1)**

function foo()

{

console.log(this); **// undefined**

}

foo();

console.log(this); **// undefined**

**(2)**

function foo()

{

console.log(this); **// foo {}**

}

new foo();

**(3)**

var obj = { "name": "newname"};

obj.foo = function() { console.log(this);} **// Object {name: "newname"}**

obj.foo();

**(4)**

function createBicycle(maxSpeed, height, tirePressure)

{

this.maxSpeed = maxSpeed;

this.height = height;

this.tirePressure = tirePressure;

this.inflateTires = function() {

this.tirePressure += 10;

}

}

var bicycle1 = new createBicycle(30, 20, 0);

function createMechanic(name) {

this.name = name;

}

var Benny = new createMechanic("Benny");

Benny.inflate = bicycle1.inflateTires;

Benny.inflate();

console.log(Benny); **// OOPS: crateMechanic {name: "Benny", tirePressure: NaN}**

console.log(bicycle1); **// createBicycle {maxSpeed: 30, height: 20, tirePressure: 0 – Still 0}**

**Solution**:

function createBicycle(maxSpeed, height, tirePressure)

{

this.maxSpeed = maxSpeed;

this.height = height;

this.tirePressure = tirePressure;

this.inflateTires = function() {

this.tirePressure += 10;

}

}

var bicycle1 = new createBicycle(30, 20, 0);

function createMechanic(name) {

this.name = name;

}

var Benny = new createMechanic("Benny");

Benny.inflate = bicycle1.inflateTires;

Benny.inflate.**call**(bicycle1);

console.log(Benny); **// createMechanic {name: "Benny"}**

console.log(bicycle1); **// createBicycle {maxSpeed: 30, height: 20, tirePressure: 10}**

PART V – Prototype & Objetcs

**Problem**: Whenever you create a new class you get a new function object

**First look at prototype**

var foo = function() { console.log("hello"); }

foo.prototype

**Object {}**

**What is this object? What is this prototype?**

**First look at prototype**

function Employee(name)

{

this.name = name;

}

var emp1 = new Employee("itay");

console.log (Employee.prototype);

var emp2 = new Employee.prototype.constructor("nana");

console.log(emp2);

**Emloyee {name: "nana"}**

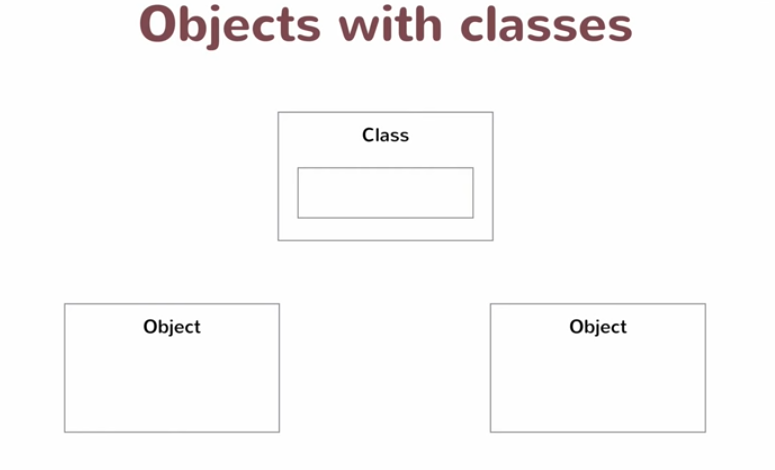
**What is going on?**

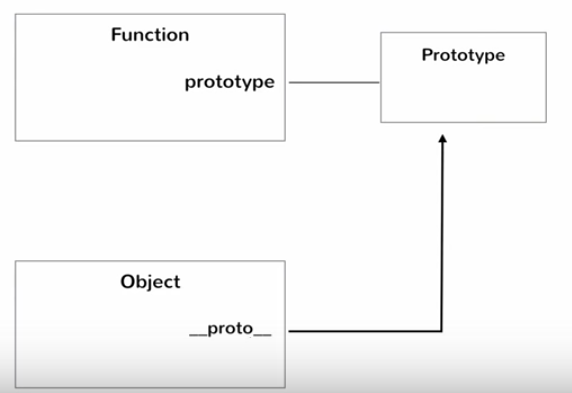
**First look at \_\_prototo\_\_**

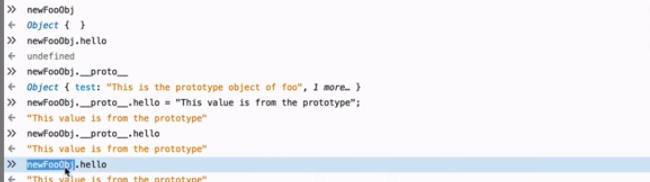
var emp = new Employee("itay");

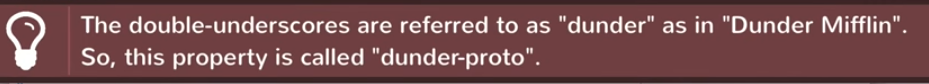
console.log(emp.\_\_proto\_\_);

**Object {}**

****

****

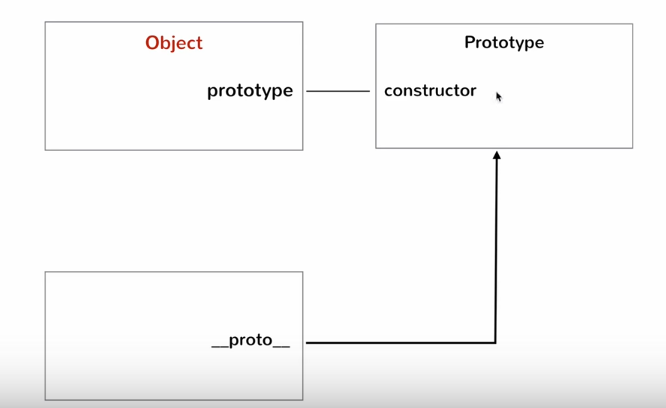
****

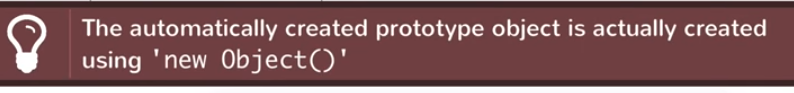
****

* **It would be better to store method in the proto type**
* **but do not store data in the highest object proto type since it will be for all objects!!!**
* **Better not use \_\_proto\_\_ but only prototype**

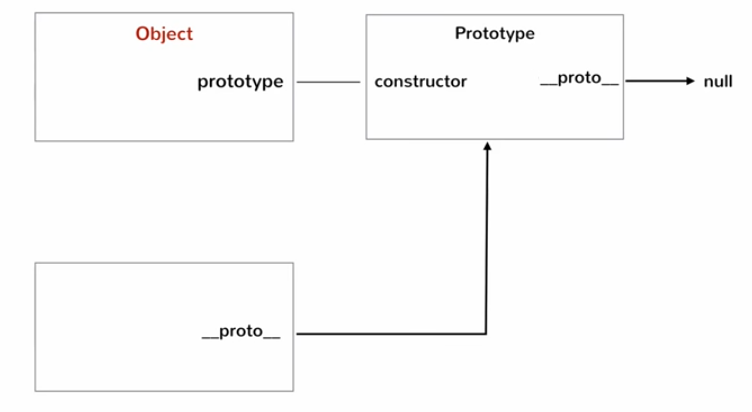
****

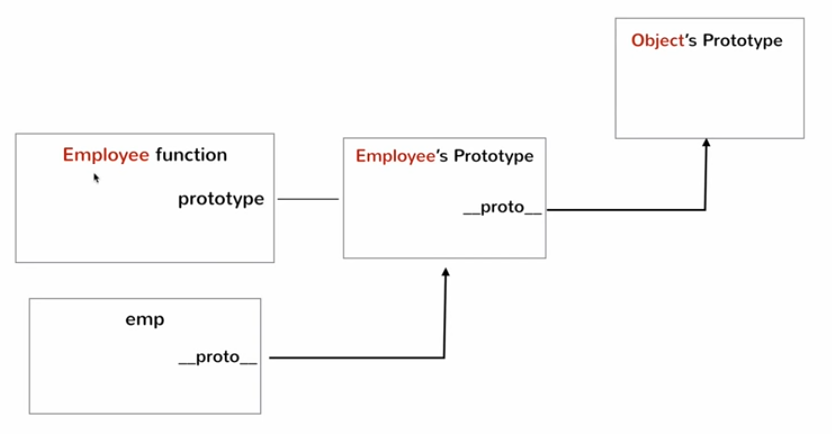
****

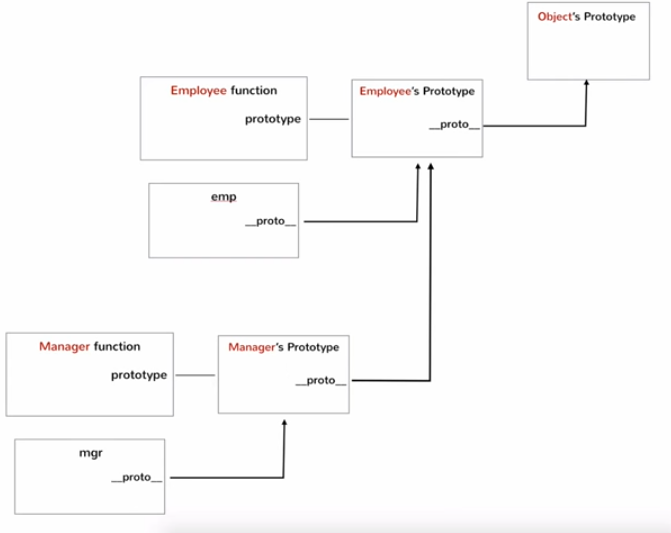
****

****

****

****

****

****

**PROTOTYPE**

Every JavaScript object has a prototype. The prototype is also an object. All JavaScript objects inherit their properties and methods from their prototype. All JavaScript objects inherit the properties and methods from their prototype.

Objects created using an object literal, or with new Object(), inherit from a prototype called **Object.prototype**. Objects created with new Date() inherit the **Date.prototype**. The Object.prototype is on the top of the prototype chain. All JavaScript objects (Date, Array, RegExp, Function, ....) inherit from the Object.prototype.