**JAVASCRIPT OOP/Advanced JavaScript**

**Functions**

Each functions gets 2 special objects– **arguments** and  **this.**

**Functions are one if the most powerful featres in JS and one of the most important**

**In JavaScript has first-class functions:**

(Може да подаваме като параметър, да я връщаме като стойност от друга функция и т.н.)

* they can be assigned to variables or properties, passed as arguments and returned by other functions
* they have **properties** and **methods** of their own:

properties: **length, name**

methods: **caller, apply, call, toString, callee**

**length**

**function** *someFunc(a, b, c)*

*someFunc*.**length** – броя на параметрите

**name**

**function** *someFunc*(a){**………**};  
**var** *someFunc* = **function** *gosho*(a) {**………**};

**toString**

**function** *someFunc*(x, y) {  
 **return** x + y;  
}

*someFunc*.toString() //function someFunc(x, y) {

return x + y;

}

**call(obj, args)**

**apply(obj, [args])**

Invokes the function **with this set to obj** and the given **args**

**function** *printMessage*(a) {  
 **console**.log(**this**.name + **' '** + a);  
}  
  
*printMessage*.call({**name**: **'Pesho'**}, 5); *//Pesho 5  
printMessage*.apply({**name**: **'Pesho'**}, [5]); *//Pesho 5*

**callee**

**function** *factorial*(n){  
 **if**(n === 0) {  
 **return** 1;  
 }  
 *//return n \* factorial(n-1); //NaN* **return** n \* arguments.**callee**(n - 1); *//120*}  
**var *factorialCopy*** = *factorial*;  
*factorial* = **function**() {  
 **return 'Other'**;  
}  
**console**.log(***factorialCopy***(5));

**caller**

**Връща коя функция извиква функцията. Лоша практика ☺**

**IIFE Immediately Invoked Function Expression**

**IIFEs are primary used to create function scope and prevent naming collisions. Also prevent global scope pollution.**

**Module**

**A module is the result that is returned from an IIFE . Всеки модул е IIFE, ама не всяко IIFE е модул.**

**- allows hidden data**

**- evades the polluting of the global scope**

**//1st way**

**var *getNextId*** = **function**() {  
 **var** lastId = 0;  
 **return function** () {  
 **return** lastId += 1;  
 }  
}();  
**console**.log(***getNextId***());*//1***console**.log(***getNextId***());*//2*

**console**.log(**typeof *getNextId***);*//function*

//2nd way

**var *generator*** = **function**() {  
 **var** lastId = 0;  
 **return** {  
 getNextId: **function**(){  
 **return** lastId += 1;  
 }  
 }  
}();  
**console**.log(***generator***.getNextId());*//1***console**.log(***generator***.getNextId());*//2*

**console**.log(**typeof *generator***);*//object*

//Like 2nd way, but not wrapped in IIFE

**var *lastId*** = 0;  
**var *generator*** = {  
 getNextId: **function**(){  
 **return *lastId*** += 1;  
 }  
}

**console**.log(***generator***.getNextId());*//1***console**.log(***generator***.getNextId());*//2*

**Scope**

**Global and function scope**

**Global scope is the scope of the web page(Window) or node.js app(Global). Objects belong to the global scope if they are defined outside of a function scope or without var(fixable with ‘use strict’);**

**let x = 5; // let създава block scope //EcmaScript 6**

**Closure**

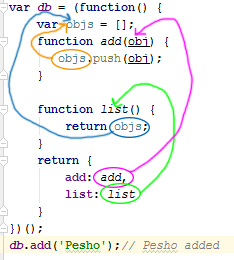
**Closure-а дава възможност от една функция да достъпваме стойности, които са над нейния scope. Closures are a special kind of structure. They combine a function and the context of this function.**

**Closure е начин, по който вътрешната функция запазва контекста, в който е и всички променливи и неща, които са около нея...(Владо ☺)**

**function** *outer*(x) {  
 **function** *inner*(y) { //inner() forms a closure.It holds a reference to x  
 **return** x + **' '** + y;   
 }  
 **return** *inner*;  
}

**var *f*** = *outer*(**'Gosho'**);  
**console**.log(***f***(**'Petrov'**)); *//Gosho Petrov***console**.log(***f***(**'Dimitrov'**)); *//Gosho Dimitrov*

**Когато връщаме функцията inner като резултат от изпълнениеето на outer, всъщност inner прави closure със x. Closure е възможността една функция да достъпва и да запазва стойности на променливи обекти, които са в нейния си scope.**

**Използва се често при правене на модули.**

**4 closure –a**

**Important!!!**

**// with var**

**var *arr*** = [];  
**for** (**var *i*** = 0; ***i*** < 10; ***i*** += 1) {  
 ***arr***.push(**function**() {**console**.log(***i***);})  
}  
  
**for** (**var *i*** = 11; ***i*** <= 15; ***i*** += 1) {  
 ***arr***[***i***-10]();*//11, 12, 13, 14, 15*}

**//with let**

**var *arr*** = [];  
**for** (**let *i*** = 0; ***i*** < 10; ***i*** += 1) {  
 ***arr***.push(**function**() {**console**.log(***i***);})  
}  
  
**for** (**let *i*** = 11; ***i*** <= 15; ***i*** += 1) {  
 ***arr***[***i***-10]();*//1, 2, 3, 4, 5*}

**//with var**

**function** *solve*() {  
 **var** arr = [];  
 **for** (**var** i = 0; i < 10; i += 1) {  
 arr.push(**function**() {**console**.log(i);})  
 }  
 **return** arr;  
}  
**var *arr*** = *solve*();  
  
**for** (**var *i*** = 11; ***i*** <= 15; ***i*** += 1) {  
 ***arr***[***i***-10]();*//10, 10, 10, 10, 10*}

**//with let**

**function** *solve*() {  
 **var** arr = [];  
 **for** (**let** i = 0; i < 10; i += 1) {  
 arr.push(**function**() {**console**.log(i);})  
 }  
 **return** arr;  
}  
**var *arr*** = *solve*();  
  
**for** (**var *i*** = 11; ***i*** <= 15; ***i*** += 1) {  
 ***arr***[***i***-10]();*//1, 2, 3, 4, 5*}

Примери:

**//**

**var *module*** = (**function**(argument) {  
 **var** name = **'Vlado'**;  
 **function** *printMsg*() { // closure  
 **console**.log(name);  
 **console**.log(argument);  
 }  
  
 **return** *printMsg*;  
})(**'Gosho'**);  
  
***module***(**'Pesho'**); *//Vlado \n Gosho*

**//**

**var *module*** = (**function**() {  
 **console**.log(arguments);  
 **function** *printMsg*() {  
 **console**.log(arguments);  
 }  
  
 **return** *printMsg*;  
})(**'Gosho'**, 5, 6);  
  
***module***(**'Pesho'**); *// '0': 'Gosho', '1': 5, '2': 6 } \n { '0': 'Pesho' }*

*Syntax sugar – не е нова функционалност, а улеснява писането. Старите неща могат да се правят по по-лесен начин.(Дончо)*

*Lasy loading – a*[*design pattern*](https://en.wikipedia.org/wiki/Design_pattern_(computer_science))*commonly used in computer programming to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used. The opposite of lazy loading is****eager loading****. Обектите се инициализират, чак когато потрябват(когато интерпретаторът стигне до реда на инициализащията в JS)*

Traceur и Babel(6-to-5) – инструменти, които превеждат Ecma-Script 6 to 5.

**Classical (Functional) Inheritance in JavaScript**

**Duck typing -** The rule of "[Duck Typing](http://en.wikipedia.org/wiki/Duck_typing)" is

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

In a class-based object-oriented programming language (C++, for example) to make both objects look like a duck you must inherit their classes from a common "interface" class, so the compiler would let you call duck methods on them. That is called a strong typing.

На практика замества **полиморфизма**, който има в другите езици.

В JS не се интересуваме дали някакъв обект е от даден тип(клас) , ами по-скоро дали обектът има дадено пропърти.

http://stackoverflow.com/questions/3379529/duck-typing-in-javascript

**var *arrObj*** = {  
 **length**: 3,  
 **0**: 5,  
 **'1'**: 6,  
 **2**: 8  
}  
  
**console**.log([].slice.call(***arrObj***, 1)); *//[6, 8] за да работи slice-а трябва да има*

**console**.log([].slice.call(***arrObj***)); *//[5, 6, 8] length и индекси на array от 0 до*

*length - 1*

**//**

**function** *Person*(name, age) {  
 **this**.**name** = name; *//когато тук използваме this, това означава обекта, който*  
 **this**.**age** = age; *//се създава, т.е. конкретната инстанция*  
}  
  
**var *pesho*** = **new** *Person*(**'Pesho'**, 19);  
**console**.log(***pesho* instanceof** *Person*); *//true Дончо: Това е единственото смислено*

*нещо, което ни дава класическото ООП, което*

*не е чак толкова смислено, но е единствено ☺*

**Prototype**

JS е прототипно-ориентиран език. Всеки обект има прототип.

**Object.prototype**

Object {}

1. \_\_defineGetter\_\_: *\_\_defineGetter\_\_()*
2. \_\_defineSetter\_\_: *\_\_defineSetter\_\_()*
3. \_\_lookupGetter\_\_: *\_\_lookupGetter\_\_()*
4. \_\_lookupSetter\_\_: *\_\_lookupSetter\_\_()*
5. constructor: *Object()*
6. hasOwnProperty: *hasOwnProperty()*
7. isPrototypeOf: *isPrototypeOf()*
8. propertyIsEnumerable: *propertyIsEnumerable()*
9. sing: function*()*
10. toLocaleString: *toLocaleString()*
11. toString: *toString()*
12. valueOf: *valueOf()*
13. get \_\_proto\_\_: *get \_\_proto\_\_()*
14. set \_\_proto\_\_: *set \_\_proto\_\_()*

Object.prototype.sing = function(){}

**//Всяка функция има prototype, който има пропърти constructor, който сочи обратно към функцията**

**function** *f*() {}  
**console**.log(*f*.**prototype**.**constructor** === *f*); *//true*

//

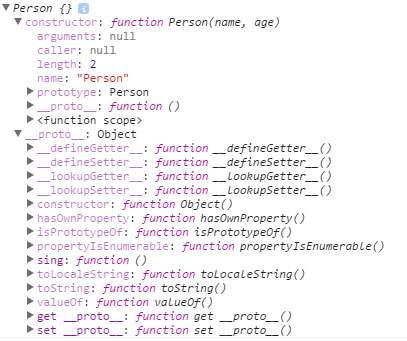
**function** *Person*(name, age) {  
 **this**.**name** = name; *//когато кажем this.name, това означава вземи прототипа на*   
 **this**.**age** = age; *//тази функция(Person.prototype) и закачи там пропъртито name*  
}  
**var *pesho*** = **new** *Person*(**'Pesho'**, 19);

**Person.\_\_proto\_\_** : function () {}

**Person.prototype** : Person {}

**pesho.\_\_proto\_\_** : Person {}

**pesho.prototype** : undefined

****

Когато закачаме нещо към прототипа, означава, че самите инстанции могат да го използват, ако го закачаме направо за фунцията, се използва от нея(като статичен метод):

**//**

**function** *Person*(name, age) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
}  
**var *pesho*** = **new** *Person*(**'Pesho'**, 19);  
  
**//**

*Person*.some = **function**() {  
 **console**.log(**this**.toString());  
};  
  
*Person*.some(); *//function Person(name, age) {this.name = name; this.age = age;}****pesho***.some(); *//Error: pesho.some is not a function*

**//**

*Person*.**prototype**.some = **function**() {  
 **console**.log(**this**);   
};  
  
*Person*.some(); *//Error****pesho***.some(); *//Person { name: 'Pesho', age: 19 }*

*Разлика между prototype и \_\_proto\_\_: \_\_proto\_\_ се използва за обекти, а Prototype върху функции.(Дончо, ама това май не е съвсем вярно, Влади го обяснява по-добре )*

**//**

**function** *Person*(name, age) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
}  
**var *pesho*** = {};  
  
***pesho***.**\_\_proto\_\_** = *Person*.**prototype**;  
**console**.log(***pesho* instanceof** *Person*);*//true*

**Properties in JS**

**//**

**function** *Person*(name) {  
 **this**.\_name = name;  
}  
Object.defineProperty(*Person*.**prototype**, **'name'**, {  
 **get**: **function**() {  
 **return this**.**\_name**;  
 },  
 **set**: **function**(value) {  
 **if**(**typeof** value !== **'string'**) {  
 **throw new** Error(**'Invalid name'**);  
 }  
 **this**.**\_name** = value;  
 **return this**;  
 }  
});  
  
**var *pesho*** = **new** *Person*(**'Pesho'**, 15);  
**var *pesho1*** = **new** *Person*(7, 15); *// throw Error Invalid name****pesho***.name = 18; *// throw Error Invalid name*

**//**

**function** *Person*(name) {  
 **this**.name = name;  
}  
Object.defineProperty(*Person*.**prototype**, **'name'**, {  
 **get**: **function**() {  
 **return this**.**\_name**;  
 },  
 **set**: **function**(value) {  
 **if**(**typeof** value !== **'string'**) {  
 **throw new** Error(**'Invalid name'**);  
 }  
 **this**.**\_name** = value;  
 **return this**;  
 },  
 **enumerable**: **true**});  
Object.defineProperty(*Person*.**prototype**, **'fullName'**, {  
 **get**: **function**() {  
 **return this**.name + **this**.name;  
 },  
 **enumerable**: **true**})  
  
**var *pesho*** = **new** *Person*(**'Pesho'**);  
**console**.log(***pesho***.fullName); *//PeshoPesho***for**(**var *prop* in *pesho***) {  
 **console**.log(***prop***); *//\_name, name, fullName*}

**This&Event (Dancho)**

*Ако кликна на един елемент и не съм казал нищо друго на евента, ?? кликам на елемента и всичко друго под него надолу, т.е. ако моят елемент има някакъв click handler и елемента под него има също евент handler, ще се изпълнят и двата.Затова таргета на евента, може да не е този, който го е пратил,?? бутончето и в първия случай таргета е бутончето, във втория случай дива под него, в третия случай дива под него и т.н.И това е при capture-ing, t.e. от бутончето към това което е под него и при bubbling, от най-долното нещо към бутончето.Това са 2 различни неща. И е.target не е съвсем същото като this. This e sender-a на евента, а target-a е това, на което съм кликнал.*

**This**

**// with new**

**function** *f*() {  
 **console**.log(**this**); *// f {}* **console**.log(arguments.**callee**); *// [Function: f] (самата функция)* **console**.log(**this instanceof** arguments.**callee**); *//true*}  
  
**new** *f*();

**// without new**

**function** *f*() {  
 **console**.log(**this**); *// window or global* **console**.log(arguments.**callee**); *// [Function: f] (самата функция)* **console**.log(**this instanceof** arguments.**callee**); *//false*}  
  
*f*();

**// without new**

**function** *f*() {  
 **console**.log(**this**); *// { name: 'Pesho', introduce: [Function: f] }* **console**.log(arguments.**callee**); *// [Function: f] (самата функция)* **console**.log(**this instanceof** arguments.**callee**); *//false*}  
  
**var *pesho*** = {  
 **name**: **'Pesho'**,  
 **introduce**: *f*}  
  
***pesho***.**introduce**();

**// when call this from object**

**function** *outer*() {  
 **function** *inner*() {  
 **console**.log(**this** === global);*//true* }  
 *inner*();  
 **console**.log(**this**); *//{ f: [Function: outer] }*}  
  
**var *obj*** = {  
 **f**: *outer*};  
  
***obj***.**f**();

*arguments.****callee*** *пази функцията, в която се намираме.‘use strict‘ забранява callee и caller*

**//без this**

**function** *Person*(name) {  
 **console**.log(name);  
}  
  
**var *pesho*** = **new** *Person*(**'Pesho'**); *//Pesho***var *gosho*** = *Person*(**'Gosho'**); *//Gosho***console**.log(***pesho***); *//Person {} -празен обект, т.к. не сме закачили нищо на this-а***console**.log(***gosho***); *// undefined, понеже gosho = функция, която не връща нищо*

**//с this**

**function** *Person*(name) {  
 **this**.**name** = name;  
}  
  
**var *pesho*** = **new** *Person*(**'Pesho'**);   
**var *gosho*** = *Person*(**'Gosho'**);   
**console**.log(***pesho***); *//Person {name: 'Pesho'}***console**.log(***gosho***); *// undefined*

**//имплементация на new (какво всъщност прави new)**

**function** *Person*(name, age) {  
 **this**.**name** = name;  
 **this**.**age** = age;   
}  
  
**function** *newObject*(func) {  
 **var** args = Array.**prototype**.slice.call(arguments, 1);  
 **var** object = Object.create(func.**prototype**);  
 **var** result = func.apply(object, args);  
   
 **return** (**typeof** result === **'object'** && result) || object;  
}  
  
**var *pesho*** = *newObject*(*Person*, **'Pesho'**, 23);  
**var *pesho2*** = **new** *Person*(**'Pesho'**, 23);  
**console**.log(***pesho***); *//Person { name: 'Pesho', age: 23 }*

**console**.log(***pesho2***);*//Person { name: 'Pesho', age: 23 }*

*//in browser console*

Function instanceof Object *//true*

Object instanceof Function *//true*

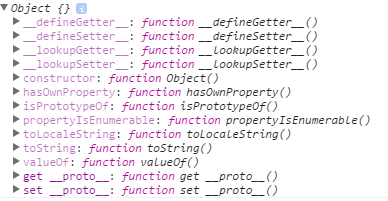
String instanceof Function *//true*

String instanceof Object *//true*

Math instanceof Object *//true*

Math instanceof Function *//false*

Number instanceof Object *//true*

**Number instanceof Function *//true*

Function.prototype *//function() {}*

Function.\_\_proto\_\_ *//function() {}*

Object.prototype *//Object {}*

Object.\_\_proto\_\_ *//function() {}*

String.\_\_proto\_\_ *//function() {}*

String.prototype *//String {length: 0,[[PrimitiveValue]]: “”}*

Number.\_\_proto\_\_ *//function() {}*

Number.prototype *//Number {[[PrimitiveValue]]: 0}*

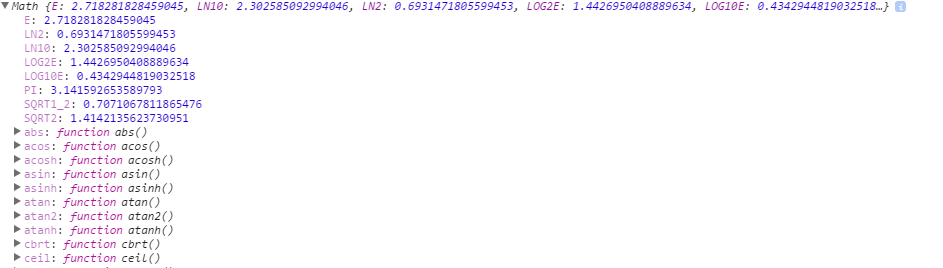
Math.\_\_proto\_\_ *// Object {}*

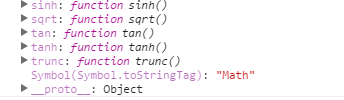
Math.prototype *//undefined*

Function *//function Function() {[native code]}*

Object *// Object() {[native code]}*

Math *//* *{E: 2.718281828459045, LN10: 2.302585092994046, LN2: 0.6931471805599453, LOG2E: 1.4426950408889634, LOG10E: 0.4342944819032518…}*



*******.....................................*

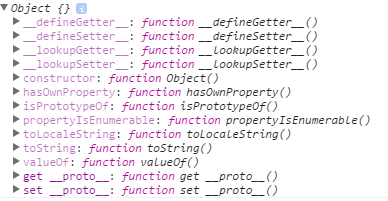
Function.prototype *//function() {}*

Function.\_\_proto\_\_ *//function() {}*

Object.\_\_proto\_\_ *//function() {}*

Array.\_\_proto\_\_ *//function() {}*

String.\_\_proto\_\_ *//function() {}*

***В тези всички има всичко от обекта, демек това, без последните 2 реда:*

*и още: apply, arguments, bind, call,*

*caller, length, name –*

*нещата, които ги има всяка*

*функция*

*В* ***prototype*** *(с изключение на Function.prototype) освен, че има всичко отгоре(това , което се отнася за обектите, не за фукциите – т.е. без оранжевкото) има и неща , специфични за конкретния function обект:*

***String*** *– anchor, big, blink, bold, charAt, indexOf и т.н.*

***Array*** *– concat, every, fill, filter, join, map и т.н.*

***Number*** *– toFixed, toPrecision, toExponential*

**Function Constructor Fix**

**John Resig(jQuery) designed a simple way to check if the function is not used as constructor:**

**function** *Person*(name,age) {  
 **if**(!(**this instanceof** arguments.**callee**)) {   
 **return new** arguments.**callee**(name, age);  
 }  
 **this**.**\_name** = name;  
 **this**.**\_age** = age;  
}

*//Ако го извикаме без new или без Object.create създай нов обект*

**Function Constructor in Module**

**var *Person*** = (**function**() {  
 **function** Person(name, age) {  
 **this**.**\_name** = name;  
 **this**.**\_age** = age;  
 }  
   
 Person.**prototype**.sayHello = **function**() {  
 **return 'Hello, I\'m '** + **this**.\_name + **' '** + **this**.\_age;   
 }  
   
 **return** Person;  
})();  
  
**var *pesho*** = **new *Person***(**'Pesho'**, 19);  
**console**.log(***pesho***.sayHello()); *//Hello, I'm Pesho 19*

**Prototypal OOP**

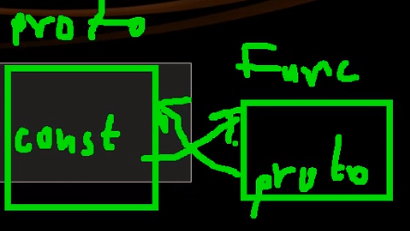
**Prototypal OOP uses the prototype nature of JS to produce objects.**

**Objects are created from objects, instead from functions**

**var *Person*** = {  
 init: **function**(name) {  
 **this**.**\_name** = name;  
 **return this**;  
 },  
 introduce: **function**() {  
 **return 'Hello I\'m '** + **this**.**\_name**;  
 }  
};  
  
**var *pesho*** = Object.create(***Person***).init(**'Pesho'**);  
**console**.log(***pesho***.introduce());

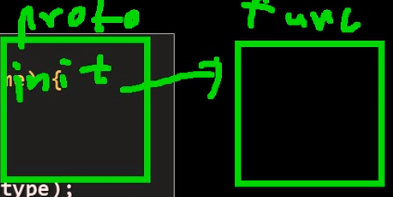
*//Object.create e метод, който от даден обект, създава друг обект, като  
//прототипите им са вързани след това***console**.log(***pesho***.\_\_proto\_\_ == ***Person***); *//true*

***//classical OOP in JS*****function** *Person*(name) {  
 **this**.**\_name** = name;  
}  
**var *pesho*** = **new** *Person*(**'Pesho'**);  
**console**.log(***pesho***);*// Person { \_name: 'Pesho' }****//prototypal OOP in JS*****var *PersonPrototype*** = {  
 init: **function**(name) {  
 **this**.**\_name** = name;  
 }  
}  
**var *pesho*** = Object.create(***PersonPrototype***);  
***pesho***.init(**'Pesho'**);  
**console**.log(***pesho***); *//{ \_name: 'Pesho' }*

** ***При класическото ООП се създава функция(Func),***

***която си създава Prototype, който си има поле***

***Constructor, който сочи наобратно към същата***

*** функция. Т.е. Func си има прототип, който сочи към прото , а прото си има конструктор, който сочи към Func.***

***При прототипното ООП първо се създава прототипа,***

***който си създава вътре Function, примерно init, който***

***създава функцията като горе.Разликата е, че горното***

***има в прототипа един конструктор, който сочи обратно към функцията, а долу нямаме такъв конструктор и нещата се създават наобратно.***

**Object.prototype**

**All JavaScript objects inherit methods / properties from their prototype**

**Object.prototype.constructor**

**Specifies the function that creates an object’s prototype**

**var *obj*** = { **name**: **'Pesho'** };  
**var *arr*** = [5, 7];  
**var *num*** = **new** Number(3);  
**console**.log(***obj***.**constructor** === Object); *// true***console**.log(***arr***.**constructor** === Array); *//true***console**.log(***num***.**constructor** === Number); *//true*

**function** *Person*(name){ **this**.**name** = name}  
**var *pesho*** = **new** *Person*(**'Pesho'**);  
**console**.log(***pesho***.**constructor** === *Person*); *//true*

**console**.log(***pesho***.**constructor** === *Person*.**prototype**.**constructor**); *//true***console**.log(*Person* === *Person*.**prototype**.**constructor**); *//true*

**Object.prototype.\_\_proto\_\_**

**Points to the object which was used as prototype(on instantiation) .** *Означава прототипа на обекта(функцията), който е бил извикан при инстанциране.Това не се използва, вместо него* ***Object.getPrototypeOf()***

***pesho***.\_\_proto\_\_ === *Person*.**prototype** *//true*

**Object.prototype.toString**

***Това са 3 неща, които всяко нещо в JavaScript има:***

* ***конструктор – функцията, която се вика когато кажем new***
* ***прототип - \_\_proto\_\_***
* ***toString***

**hasOwnProperty()**

***Every object inherits the hasOwnProperty.***

***Does not traverse the prototype chain***

**var *obj*** = {**name**: **'Pesho'**};  
**console**.log(***obj***.hasOwnProperty(**'name'**)); *//true***console**.log(***obj***.hasOwnProperty(**'age'**)); *//false***console**.log(***obj***.hasOwnProperty(**'toString'**)); *//false***console**.log(***obj***.\_\_proto\_\_.hasOwnProperty(**'toString'**)); *//true***console**.log(Object.getPrototypeOf(***obj***).hasOwnProperty(**'toString'**)); *//true*

**How to Inherit the prototype Object**

**function** *Person*(name) {  
 **this**.**name** = name;  
}  
  
**function** *Student*(name, grade) {

***Person***.call(**this**, name)

**this**.**grade** = grade;  
}  
  
*Student*.**prototype** = Object.create(*Person*.**prototype**);  
**console**.log(*Student*.**prototype**.**constructor** === *Person*);*//true  
Student*.**prototype**.**constructor** = *Student*;*//когато кажем new Student, да не се извика Person*

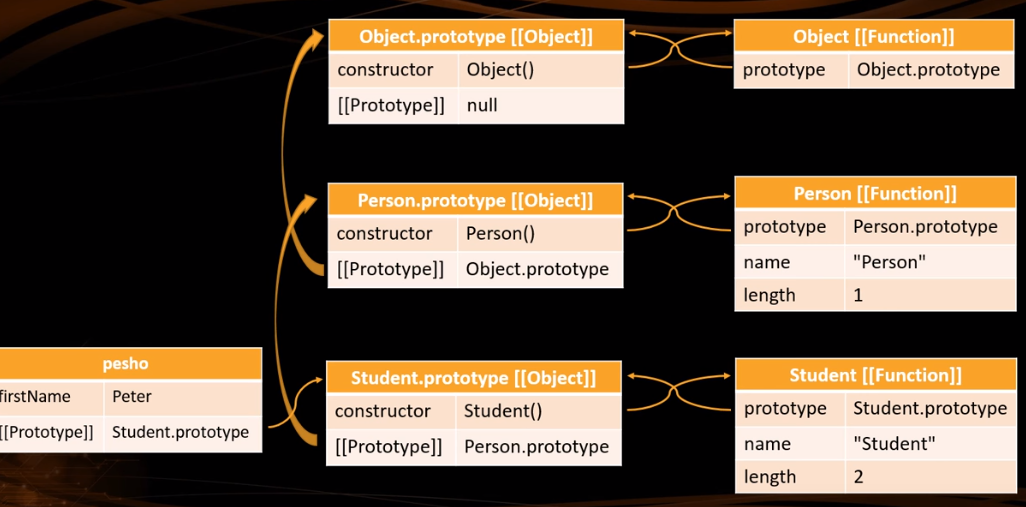
//extends method

Function.**prototype**.extends = **function**(parent) {  
 **this**.**prototype** = Object.create(parent.**prototype**);  
 **this**.**prototype**.**constructor** = **this**;  
}

*Student*.extends(*Person*);

**Fixing Missing Object create (old browsers)**

**if**(!Object.create) {  
 Object.create = **function**(proto) {  
 **function** F() {};  
 F.**prototype** = proto;  
 **return new** F();  
 };  
}

******

**// Prototypal inheritance**

**var *Animal*** = {  
 init: **function** (name, species) {  
 **this**.**\_name** = name;  
 **this**.**\_species** = species;  
 **return this**;  
 },  
 produceSound: **function**() {  
 **throw new** Error(**'Cannot call the abstract method Animal.produceSound'**);   
 },  
 walk: **function** () {  
 **console**.log(**this**.**\_name** + **' is walking...'**);  
 }  
};  
  
**var *animal*** = Object.create(***Animal***).init(**'Pesho'**, **'cat'**);*//правим копие на Animal***var *animal1*** = ***Animal***.init(**'Gosho'**, **'cat'**); *//animal1 e самия обект, ако променим нещо по animal1, ще се  
 //отрази на Animal, t.k. това е един и същи обект***console**.log(***animal1*** == ***Animal***); *//true*

**var *Tiger*** = Object.create(***Animal***);  
***Tiger***.init = **function** (name, species, isSaberTooth) {  
 ***Animal***.init.call(**this**, name, species) ;  
 **this**.**\_isSaberTooth** = isSaberTooth;  
};

***Tiger***.produceSound = **function**() { //ако не го презапишем, вика базовия и гърми  
 **return 'What does the tiger say?? Rmmmmmm'** ;   
}  
**var *gosho*** = Object.create(***Tiger***).init(**'Gosho'**, **'cat'**, **true**);  
**console**.log(***gosho***); *//{ \_name: 'Gosho', \_species: 'cat', \_isSaberTooth: true }***console**.log(***gosho***.produceSound()); *// What does the tiger say?? Rmmmmmm*

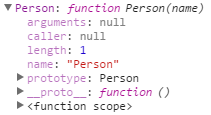
***Object.create*** *- когато закачаме нещо директно за обекта(по-скоро функцията), правим нещо като статик метод, който не може да се ползва от инстанциите.*

***Object.prototype.extends*** *– когато закачаме нещо за прототипа на функцията, то се ползва от инстанциите. Тук this е това, което трябва да бъде (Данчо ☺) Когато извикаме new, ние правим копие на самия обект, обаче прототипът му не се копира, той остава същия и това позволява методите, които са в него да не се копират всеки път при създаване на нова инстанция.Ще бъде копирано само това, което е в конструктора.(name, age, ..), т.е. нещата (пропъртитата), които са различни за всяка инстанция. Методите, закачени за прототипа са едни и същи.*

**-------когато след време чета това---- оттук да започна----Владимир обяснява най-добре прототипите и наследяването---------------------------------------------------------------**

**Prototype Chain and Inheritance in JavaScript (Владо)**

**Object.prototype - *всяка една функция има prototype обект. Той ни позволява да добавяме разни полета и функции към инстанциите на този обект. Object, Array, Function и т.н. са функции и когато ги извикваме с new създаваме нова инстанция от тях. Tricky-то тук е, че обектът се създава от prototype-a на фукцията.***

*****Въпрос: Дали prototype-а е като конструктор? Владо: Конструкторът си е функцията, която си създал. Този prototype обект се създава от всяка фунцкия, за да ни помага да създаваме обекти от тази функция, защото от функцията не може да създадем директно обект, обаче може да създадем обект от друг обект.***

**function** *Person*(name) {  
 **this**.**\_name** = name;  
}  
  
*Person*.**prototype**.introduce = **function** () {  
 **return this**.**\_name**;  
}

****

***\_\_proto\_\_ създава prototype chain-a, а prototype-a***

***ни помага да си съсздаваме обекти от даден Function.***

***Прототипът е различен от prototype обекта. В JS всеки обект има прототип, докато prototype object имат само фукциите. Той по принцип е скрит. Досега няма имплементация на ЕcmaScript, която да казва, че дадено пропърти ще държи този прототип. Затова някои браузъри имплементират \_\_proto\_\_.***

***Ако направим инстанция на Object, \_\_proto\_\_ - то на инстанцията е Object.prototype. На Object.prototype.\_\_proto\_\_ = null***

**Important 1!! This says a lot and I understand it!!**

**function** *Person*(name) {  
 **this**.**\_name** = name;  
}  
  
*Person*.**prototype**.introduce = **function** () {  
 **return 'I\'m '** + **this**.**\_name**;  
}  
  
**function** *Student*(name, grade) {  
 *Person*.call(**this**, name);  
 **this**.**\_grade** = grade;  
}  
  
*Student*.**prototype** = Object.create(*Person*.**prototype**);  
*Student*.**prototype**.**constructor** = *Student*;  
  
*Student*.**prototype**.introduce = **function**() {  
 **return** *Person*.**prototype**.introduce.call(**this**) + **' Grade: '** + **this**.**\_grade**;  
}

**var *pesho*** = **new** *Person*(**'Pesho'**);  
**var *gosho*** = **new** *Student*(**'Gosho'**, 6);

**console**.log(Object.**prototype**.\_\_proto\_\_ == **null**); *// true***console**.log(*Person*.**prototype**.\_\_proto\_\_== Object.**prototype**); *// true***console**.log(*Student*.**prototype**.\_\_proto\_\_ == *Person*.**prototype**); *//true***console**.log(***pesho***.\_\_proto\_\_ == *Person*.**prototype**); *//true*

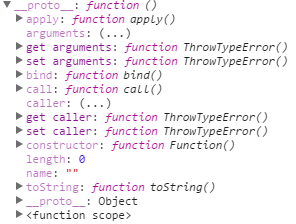
**|console**.log(***gosho***.\_\_proto\_\_ == *Student*.**prototype**); *//true*

**|console**.log(Object.getPrototypeOf(***gosho***) == *Student*.**prototype**); *//the same //true*

**console**.log(***pesho***.introduce()); *//I'm Pesho***console**.log(***gosho***.introduce()); *//I'm Gosho Grade: 6*

**var *obj*** = {**key**: **'value'**}; *//the same as new Object({key: 'value'});***console**.log(***obj***.\_\_proto\_\_ == Object.**prototype**);

**а) \_\_proto\_\_ на инстанция на функция (на обект) – pesho, gosho, obj и т.н. е винаги равно на prototype na функцията (function конструктора), от която е криетнат обекта(инстанцията).**

**б)\_\_proto\_\_ на функция/Function constructor – Object, Person, Student, String и т.н. е винаги [Function: Empty], демек това: 🡪**

а)***pesho* instanceof** Person *//true*

**проверява дали**

***pesho***.\_\_proto\_\_ == *Person*.prototype *//true*

б)String **instanceof** Function *//true*

**проверява дали**

String.\_\_proto\_\_ == **Function**.\_\_proto\_\_*//true*

**instanceOf приема само функции отдясно**

***Prototypal Inheritance***

*Като цяло, псевдо-класическото наследяване работи отдолу по този начин, защото new прави Object.create(Person.prototype). Тук правим един (простичък☺) обект, който има конструктор, който е функция (init,construct, create, constructor ) и вътре в него закачаме в/у this нещата. По същия начин стават нещата, само че всичко е wrap-нато в обект, а не във функция. Имаме person обект, който си има init, който е все едно нашия конструктор. И после казваме Object.create от person, и след това, върху този обект, който ни е създаден си извикваме init метода, за да ни се закачат name-а и agе-а към this.(student)*

**var *person*** = {};  
**var *student*** = Object.create(***person***);  
**console**.log(***student***.\_\_proto\_\_ == ***person***); *//true*

**Important 2!! This says a lot and I understand it!!**

**var *person*** = {  
 init: **function**(name, age) {  
 **this**.**\_name** = name;  
 **this**.**\_age** = age;  
 **return this**;  
 },  
 introduce: **function**() {  
 **return 'Hello, I\'m '** + **this**.**\_name** + **', '** + **this**.**\_age** + **' years old'**;  
 }  
};  
**var *student*** = Object.create(***person***);  
***student***.init = **function**(name, age, grade) {  
 ***person***.init.call(**this**, name, age);  
 **this**.**\_grade** = grade;  
 **return this**;  
};  
***student***.introduce = **function**() {  
 **return *person***.introduce.call(**this**) + **', grade '** + **this**.**\_grade**;   
};  
**console**.log(***student***.\_\_proto\_\_ == ***person***);*//true***var *pesho*** = Object.create(***student***).init(**'Pesho'**, 23, 6); *//Hello, I'm Pesho, 23 years old, grade 6***console**.log(***pesho***.introduce());  
**console**.log(***pesho***.\_\_proto\_\_ == ***student***);*//true*

*With* ***extends*** *method:*

Object.**prototype**.extends = **function**(properties) {  
 **function** *Child*(){}  
 *Child*.**prototype** = Object.create(**this**);  
 **for**(**var** prop **in** properties) {  
 *Child*.**prototype**[prop] = properties[prop];  
 }  
 *Child*.**prototype**.**\_parent** = **this**;  
  
 **return new** *Child*();  
};  
**var *person*** = {  
 init: **function**(name, age) {  
 **this**.**\_name** = name;  
 **this**.**\_age** = age;  
 **return this**;  
 },  
 introduce: **function**() {  
 **return 'Hello, I\'m '** + **this**.**\_name** + **', '** + **this**.**\_age** + **' years old'**;  
 }  
};  
**var *student*** = ***person***.extends({  
 init: **function** *init*(name, age, grade) {  
 **this**.**\_parent**.init.call(**this**, name, age);  
 **this**.**\_grade** = grade;  
 **return this**;  
 },  
 introduce: **function**() {  
 **return this**.**\_parent**.introduce.call(**this**) + **' grade '** + **this**.**\_grade**;  
 }  
});  
**var *pesho*** = Object.create(***student***).init(**'Pesho'**, 23, 6); *//Hello, I'm Pesho, 23 years old, grade 6***console**.log(***pesho***.introduce()); *//Hello, I'm Pesho, 23 years old grade 6***console**.log(***pesho***.\_\_proto\_\_ == ***student***);*//true*

**JavaScript Design Pattern**

**Prototype Pattern**

**var SomeFunctionConstructor = function(param1, param2,…){**

**this.prop = param1;**

**……………………**

**}**

**SomeFunctionConstructor.prototype = {**

**method1: function(){},**

**method2: function(){},…………**

**}**

**var** *Person* = **function**(name, age) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
};  
  
*Person*.**prototype** = {  
 introduce: **function**() {  
 **return 'Hello, I\'m '** + **this**.**name** + **', '** + **this**.**age** + **' years old.'**;  
 },  
 calcSum: **function**(a, b) {  
 **return** a + b;  
 }  
};  
  
**var** *Student* = **function**(name, age, grade) {  
 *Person*.call(**this**, name, age);  
 **this**.**grade** = grade;  
};  
  
*Student*.**prototype** = Object.create(*Person*.**prototype**);  
*Student*.**prototype**.**constructor** = *Student*;  
  
*Student*.**prototype** = {  
 *//Всички методи, наследени от Person.prototype изчезват*};  
  
**var *pesho*** = **new** *Student*(**'Pesho'**, 23, 6);  
**console**.log(***pesho***.introduce()); *//гръм :) pesho.introduce is not a function*

**var** *Calculator* = **function**(name) {  
 **this**.**name** = name  
};  
  
*Calculator*.**prototype** = {  
 add: **function**(a, b) {  
 **return** a + b;  
 },  
 subtract: **function**(a, b) {  
 **return** a - b;  
 }  
};  
  
**var *calc*** = **new** *Calculator*(**'Pesho'**);  
**console**.log(***calc***.add(4, 5));*//9*

**Module Pattern**

**var obj = (function(){**

**function(){}**

**……………………**

**return {**

**f1: function(){},**

**f2: function(){}**

**}**

**})();**

**var *calculator*** = (**function**() {  
 **function** *divisibleByZero*(a) { *//нещо като private метод :)* **if**(a === 0) {  
 **throw new** Error(**'Division by zero exception'**);  
 }  
 };  
  
 **return** {  
 add: **function**(x, y) {  
 **return** x + y;  
 },  
 divide: **function**(x, y) {  
 *divisibleByZero*(y);  
 **return** x / y;  
 }  
 }  
})();  
  
**console**.log(***calculator***.add(5, 6)); *//11***console**.log(***calculator***.divide(6, 12)); *//0.5***try**{  
 **console**.log(***calculator***.divide(5, 0));  
} **catch**(e) {  
 **console**.error(e.**message**);  
}

**Revealing Module Pattern**

**var obj = (function(){**

**function f1(){}**

**function f2(){}**

**return {**

**f1: f1,**

**f2: f2,……**

**}**

**})();**

*Може да return-ва FunctionConstructor, {…}, new FunctionConstructor,………*

**//Classical OOP**

**var *person*** = (**function**() {  
 **function** *Person*(name, age) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 }  
 *Person*.**prototype**.introduce = **function**() {  
 **return 'Name: '** + **this**.**name** + **', Age: '** + **this**.**age**;  
 };  
   
 **return** {  
 getInstance: **function**(name, age) {  
 **return new** *Person*(name, age);  
 }  
 }  
})();  
  
**var *pesho*** = ***person***.getInstance(**'Peter'**, 23);  
**console**.log(***pesho***.introduce()); *//Name: Peter, Age: 23*

**//Prototypal OOP**

**var *person*** = (**function**() {  
 **var** person = {  
 init: **function**(name, age) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 **return this**;  
 },  
 introduce: **function**() {  
 **return 'Name: '** + **this**.**name** + **', Age: '** + **this**.**age**;  
 }  
 }  
  
 **return** person;  
})();  
  
**var *pesho*** = Object.create(***person***).init(**'Peter'**, 23)  
**console**.log(***pesho***.introduce()); *//Name: Peter, Age: 23*

**Revealing Prototype Pattern**

**var SomeFunctionConstructor = function(param1, param2,…){**

**this.prop = param1;**

**……………………**

**}**

**SomeFunctionConstructor.prototype = (function(){**

**function f1(){},**

**function f2(){},…………**

**return {**

**someFunc: pointerTo f1,**

**someFunc2: pointerTo f2,……………**

**}**

**})();**

**Augmenting Modules**

**var *world*** = ***world*** || {};  
  
(**function**(app) {  
 **function** *Person*(name, age) {  
 **this**.**\_name** = name;  
 **this**.**\_age** = age;  
 *//return this;* }  
  
 *Person*.**prototype**.introduce = **function**() {  
 **return 'Name: '** + **this**.**\_name** + **', Age: '** + **this**.**\_age**;  
 };  
  
 app.**person** = *Person*;  
})(***world***);

**var *world*** = ***world*** || {};  
  
(**function**(app) {  
 **function** *Student*(name, age, grade) {  
 app.**person**.call(**this**, name, age);  
 **this**.**\_grade** = grade;  
 }  
 *Student*.extends(app.**person**);  
  
 *Student*.**prototype**.introduce = **function**() {  
 app.**person**.**prototype**.introduce.call(**this**) + **', Grade: '** + **this**.**\_grade**;  
 };  
  
 app.**student** = *Student*;  
})(***world***);

**Method Chaining**

*Student*.**prototype**.introduce = **function**() {  
 app.**person**.**prototype.introduce.call(this) + ', Grade: ' + this.\_grade;**

**return this;**  
};

**This** *– The this object has a different value depending on the scope:*

**Function scope**

**this is window/global object, when is called without new**

**this is created object, when is called with new**

**Object scope**

**this is current object**

**Global scope**

**this is window(in browser)/global(in Node.js)**

**Event handlers**

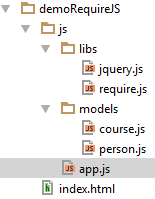
**this holds the event source**

**REQUIRE.JS**

**AMD (Asynchronous Module Definition)**

**JavaScript API for defining modules. AMD allows to create dependent modules**

**Require.js е една от най-използваните библиотеки за AMD. Работи и с Node.js и client-side в браузъра.**

*index.html*

<**body**>  
 <**div id="wrapper"**></**div**>  
 <**script src="js/libs/require.js" data-main="js/app"**></**script**>  
</**body**>

*app.js*

***require***.config({  
 **baseUrl**: **'js'**,  
 **paths**: {  
 **'jQuery'**: **'libs/jquery'**,  
 **'person'**: **'models/person'**,  
 **'course'**: **'models/course'** }  
});

***require***([**'person'**, **'course'**, **'jQuery'**], **function**(Person, Course) {*//модулите, които не връщат нищо, се зареждат накрая* **var** jichka = **new** Person(**'Jichka'**),  
 pesho = **new** Person(**'Pesho'**);  
 jichka.addCourse(**'Java Script'**);  
 jichka.addCourse(**'OOP'**);  
 **document**.getElementById(**'wrapper'**).**innerText** = jichka.**name** + **' '** + pesho.**name**;  
 **for** (**var** i = 0; i < jichka.getCourses().**length**; i += 1) {  
 **document**.getElementById(**'wrapper'**).**innerText** += **' '** + jichka.getCourses()[i].toString();  
 }  
});

*person.js*

***define***([**'course'**], **function**(Course){  
 **var** Person = (**function**() {  
 **function** *Person*(name) {  
 **this**.**name** = name;  
 **this**.**courses** = [];  
 **return this**;  
 }  
  
 *Person*.**prototype**.sayHello = **function**() {  
 **console**.log(**'Hello'** + **this**.name);  
 };  
  
 *Person*.**prototype**.addCourse = **function**(name) {  
 **this**.courses.push(**new** Course(name));  
 };  
  
 *Person*.**prototype**.getCourses = **function**() {  
 **return this**.courses;  
 };  
  
 **return** *Person*;  
 })();  
  
 **return** Person;  
});

*course.js*

***define***([], **function**() {  
 **var** Course = (**function**() {  
 **function** *Course*(name) {  
 **this**.**name** = name;  
 }  
  
 *Course*.**prototype**.toString = **function**() {  
 **return this**.name;  
 };  
  
 **return** *Course*;  
 })();  
  
 **return** Course;  
});