OO JS

jQuery(‘.class_name_input').attr('maxlength', '43'); // limit the input field of element with .class_name_input to 43 chars
webStorm
JavaScript can be interactive with **confirm**(‘this is for example’) boxes.
var operatingSistem = prompt("What is Ubuntu?”);// use **prompt to ask for input from the user**

- > Greater than
- < Less than
- <= Less than or equal to
- >= Greater than or equal to
- **===** Equal to
- **!== Not** equal to

if (**12 / 4 === "Ari".length**) {
    confirm("Will this run the first block?");
} else {
    confirm("Or the second block?");
}

"some word".**substring(x, y) // extracts starting from position x and stopping before position y, at y-1**

chars count in strigs starts from 0, like arrays elements

// This is what a function looks like:

var divideByThree = function (number) {
    var val = number / 3;
    console.log(val);
};
divideByThree(6);

!!!!!You have to put a semi-colon (;) at the end of each line of code in the reusable block and at the end of the entire function! In PHP after the function block ends you do not need semicolon (after closing bracket).

The function runs, and when the **return** keyword is used, the function will i**mmediately stop** running **and return the value**.

The **var** keyword creates a new variable **in the current scope**. That means if var is used outside a function, that variable has a global scope. If var is used inside a function, that variable has a local scope.

```
var my_number = 7; //this has global scope

var timesTwo = function(number) {
   my_number = number * 2;
   console.log("Inside the function my_number is: ");
   console.log(my_number);
};

timesTwo(7);

console.log("Outside the function my_number is: ")
console.log(my_number);
```

//Keep flipping a coin until is tail:

```
var coinFace = Math.floor(Math.random() * 2);

while(coinFace === 0){
      console.log("Heads! Flipping again...");
      var coinFace = Math.floor(Math.random() * 2);
}
console.log("Tails! Done flipping.");

while(condition){
}
```

When you use a number in a condition, as we did earlier, JavaScript understands 1 to mean true and 0 to mean false.

```
var understand = true;

while(understand){
      console.log("I'm learning while loops!");
      understand = false;
}
```

do { }
while;
and change the STEP outside the loop// condition runs at least one time

while(condition){

change the **STEP** inside the loop (incerement/decremet in opposite direction
}
var youHit = **Math.floor(Math.random()*2)**;
//This sets youHit to a random number that's either **0 (which JavaScript reads as false) or 1 (which JavaScript reads as true).**

var damageThisRound = Math.floor(Math.random()*5+1);
//This sets damageThisRound to a random number that's between 1 and 5 (up to and including 5).

isNaN()//isNaN on something, it checks to see if that thing *is not* a number

you cannot call isNaN on a variable if you didn't defined that previously

.**toUpperCase**(); // applied to something that return a string

heterogeneous arrays:
var mix = [42, true, "towel"];

var twoDimensional = [[1, 1], [1, 1]];
This array is **two-dimensional** because it has two rows that each contain two items. If you were to put a new line between the two rows, you could log a 2D object—a square—to the console
also:
var newArray = [[2,3,4],[6,9,4],[1,2,4]];

jagged arrays:

var jaggedArray = [[1,2,3],[2,5,6,7],[3]];
different no. of lines and cols000

Nouns and verbs together
Let's go back to the analogy of computer languages being like regular spoken languages. In English, you have nouns (which you can think of as "things") and verbs (which you can think of as "actions"). Until now, our nouns (data, such as numbers, strings, or variables) and verbs (functions) have been separate.
No longer!
Using **object**s, we can put our information and the functions that use that information *in the same place*.
You can also think of objects as combinations of key-value pairs (like arrays), only their keys don't have to be numbers like 0, 1, or 2: they can be strings and variables.

Object syntax
Did you see that? The phonebookEntry object handled data (a name and a telephone number) as well as a procedure (the function that printed who it was

calling).
In that example, we gave the **key** name the **value** 'Oxnard Montalvo' and the key number the value '(555) 555-5555'. An object is like an array in this way, except its keys can be variables and strings, not just numbers.
Objects are just collections of information (keys and values) between curly braces, like this:

```
var myObject = {
   key: value,
   key: value,
   key: value
};
```

Creating a new object
Great work! You just created your very first object.
There are two ways to create an object: using **object literal notation** (which is what you just did) and using the **object constructor**.
Literal notation is just creating an object with curly braces, like this:

```
var myObj = {
   type: 'fancy',
   disposition: 'sunny'
};
```

Properties are like variables that belong to an object, and are used to hold pieces of information. Properties can be accessed in two ways:

- **Dot notation**, with ObjectName**.PropertyName**
- **Bracket** notation, with ObjectName**["PropertyName"]** (don't forget the quotes!)

```
var james = {
   job: "programmer",
   married: false
};

// set to the first property name of "james"
var aProperty = "job";

// print the value of the first property of "james"
// using the variable "aProperty"
console.log(james[aProperty]);
```

You wouldn't know it, but **every object in JavaScript comes with some baggage** (stay tuned for more on this!). Part of this baggage includes a method called **hasOwnProperty**. This lets us know if an object has a particular property.
var **suitcase** = {

```
    shirt: "Hawaiian"
};

if (suitcase.hasOwnProperty("shirt")){
console.log(suitcase.shorts);
}
```

```
var objectName = {
    fullName: "New York City",
    mayor: "Bill de Blasio",
    population: 8000000,
    boroughs: 5
};
let all properties of object like this
for (var property in objectName){
   console.log(property);
}
// write a for-in loop to print the value of objects's properties
for (var property in objectName){
    console.log(objectName[property]);
}
```

```
var emptyObj = {};
```

When you use the constructor, the syntax looks like this:

var myObj = new Object();

This tells JavaScript: "I want you to make me a new thing, and I want that thing to be an Object.

You can add keys to your object after you've created it in two ways:

myObj["name"] = "Charlie";

myObj.name = "Charlie";

Both are correct, and the second is shorthand for the first. See how this is sort of similar to arrays?

```
var myArray = [1,true, '2', me = {age:9}];

var object = {
   keyObj1:{
key1:val1,
key2:val2},
  keyObject2:{
      key:val,
       key:val}
    };
```

```
for (var key in object) {
  // Access that key's value
  // with object[key]
}
```

The "key" bit can be any placeholder name you like. It's sort of like when you put a placeholder parameter name in a function that takes arguments.

```
   var search = function(name){
      for (var key in object){

      if ( name === obect[key].key_name){
         console.log(friends[key]);
      }
      }
   }
```

Methods:

1. They can be used to change object property values. The method setAge on line 4 allows us to update bob.age.
2. They can be used to make calculations based on object properties. Functions can only use parameters as an input, but methods can make calculations with object properties. For example, we can calculate the year bob was born based on his age with our getYearOfBirth method (line 8).

```
object.methodName = function(newValue){
this.object = newValue;
}
or
square.calcArea = function(sideLength){
   return this.sideLength*this.sideLength;
}

// here is bob again, with his usual properties
var bob = new Object();
bob.name = "Bob Smith";
bob.age = 30;
// this time we have added a method, setAge
bob.setAge = function (newAge){
  bob.age = newAge; // use instead ——> this.age = newAge;
};
// here we set bob's age to 40
bob.setAge(40);
```

new Object( ); we are using a built-in constructor called Object. This constructor is already defined by the JavaScript language and just makes an object with no properties or methods.

Custom Constructors - the construct defines = **the prototype of the class**

```
var buddy = new Dog("Golden Retriever");
buddy.bark = function() {
  console.log("Woof");
};
```

Instead of always using the boring Object constructor, we can make our own constructors.

```
function Person(name,age) {
  this.name = name;
  this.age = age;
this.species= “Homo Sapiens”; // all the objects of type Person will have the same species
}
// Let's make bob and susan again, using our constructor
var bob = new Person("Bob Smith", 30);
```

```
function Rectangle(height, width) {
  this.height = height;
  this.width = width;
  this.calcArea = function() {
     return this.height * this.width;
  };
  } // our rectangle constructor defines a height and width property and calcArea method

var rex = new Rectangle(7,3);
var area = rex.calcArea();
```

functions taking objects as arguments:

```
// Our person constructor
function Person (name, age) {
    this.name = name;
    this.age = age;
}

// We can make a function which takes persons as arguments
// This one computes the difference in ages between two people
var ageDifference = function(person1, person2) {
   return person1.age - person2.age;
}
```

```
//We must be careful not to pass anything but Person objects into ageDifference.
var alice = new Person("Alice", 30);
var billy = new Person("Billy", 25);
// get the difference in age between alice and billy using our function
var diff = ageDifference(alice, billy);
```

```
function Circle (radius) {
   this.radius = radius;
   this.area = function () {
       return Math.PI * this.radius * this.radius;

   };
   // define a perimeter method here
  this.perimeter = function(){
     return 2*Math.PI*this.radius*this.radius;
  }
};
```

```
var bob = {
   firstName: "Bob",
      email: "bob.jones@example.com"
};
var mary = {
   firstName: "Mary",
     email: "mary.johnson@example.com"
};
var contacts = [bob, mary];

function printPerson(person) {
   console.log(person.firstName + " " + person.lastName);
}

function list() {
       var contactsLength = contacts.length;
       for (var i = 0; i < contactsLength; i++) {
              printPerson(contacts[i]);
       }
}

list();
```

```
var james = {
   job: job,
   married: false,
```

```
    sayJob: function() {
        // complete this method
        console.log("Hi, I work as a "+ this.job);
    }
};
this.job = "programmer";
// james' first job
james.sayJob();
this.job = "super programmer";

// james' second job
james.sayJob();
```

```
// complete these definitions so that they will have the appropriate types
var anObj = { job: "I'm an object!" };
var aNumber = 42;
var aString = "I'm a string!";

console.log( typeof anObj); // should print "object"
console.log( typeof aNumber ); // should print "number"
console.log( typeof aString ); // should print "string"
```

Dog.prototype.bark.
Click run this time, and both buddy and snoopy can bark just fine! Snoopy can bark too even though we haven't added a bark method to that object. How is this so? Because we have now changed the *prototype* for the class Dog. **This immediately teaches *all* Dogs the new method.**
In general, if you want to add a method to a class such that all members of the class can use it, we use the following syntax to *extend the prototype*:

```
className.prototype.newMethod = function() {
statements;
};

function Dog (breed) {
  this.breed = breed;
};

// here we make buddy and teach him how to bark
var buddy = new Dog("golden Retriever");
Dog.prototype.bark = function() {
  console.log("Woof");
};
buddy.bark();
```

```
// here we make snoopy
var snoopy = new Dog("Beagle");
/// this time it works!
snoopy.bark();

//both buddy and snoopy ( because they are dogs) can bark
```

```
// the original Animal class and sayName method
function Animal(name, numLegs) {
    this.name = name;
    this.numLegs = numLegs;
}
Animal.prototype.sayName = function() {
    console.log("Hi my name is " + this.name);
};

// define a Penguin class
function Penguin(name){
    this.numLegs = 2;
    this.name =  name;
}
Penguin.prototype =  new Animal();
// set its prototype to be a new instance of Animal
//Penguin inherits from Animal
var penguin = new Penguin();
penguin.sayName();
```

```
function Penguin(name) {
    this.name = name;
    this.numLegs = 2;
}

// create your Emperor class here and make it inherit from Penguin
function Emperor(name){
    this.name = name;
}
Emperor.prototype = new Penguin(name);

// create an "emperor" object and print the number of legs it has
var emperor = new Emperor();
console.log(emperor.numLegs);
```

The "prototype chain" in JavaScript knows this as well. **If JavaScript encounters something it can't find in the current class's methods or**

properties, it looks up the *prototype chain* to see if it's defined in a class that it inherits from. This keeps going upwards until it stops all the way at the top: the mighty Object.prototype (more on this later). **By default, all classes inherit directly from Object,** unless we change the class's prototype, like we've been doing for Penguin and Emperor.

```
// original classes
function Animal(name, numLegs) {
    this.name = name;
    this.numLegs = numLegs;
    this.isAlive = true;
}
function Penguin(name) {
    this.name = name;
    this.numLegs = 2;
}
function Emperor(name) {
    this.name = name;
    this.saying = "Waddle waddle";
}

// set up the prototype chain
Penguin.prototype = new Animal();
Emperor.prototype = new Penguin();

var myEmperor = new Emperor("Jules");

console.log(myEmperor.saying); // should print "Waddle waddle"
console.log(myEmperor.numLegs); // should print 2
console.log(myEmperor.isAlive); // should print true
```

```
function Dog (breed) {
    this.breed = breed;

};

// add the sayHello method to the Dog class
// so all dogs now can say hello
Dog.prototype.sayHello = function(){
    console.log("Hello this is a "+this.breed+" dog");
}

var yourDog = new Dog("golden retriever");
yourDog.sayHello();
```

```
var myDog = new Dog("dachshund");
myDog.sayHello();
```

Private Variables

Good! But what if **an object wants to keep some information hidden**?
Just as functions can have local variables which can only be accessed from within that function, objects can have private variables. **Private** variables are pieces of information you do not want to publicly share, and they can only be directly **accessed from within the class**.

```
function Person(first,last,age) {
  this.firstname = first;
  this.lastname = last;
  this.age = age;
  var bankBalance = 7500; // use var instead this
}
```

Accessing Private Variables

Although we cannot directly access private variables from outside the class, there is a way to get around this. We can **define a public method that returns the value of a private variable**.

```
function Person(first,last,age) {
  this.firstname = first;
  this.lastname = last;
  this.age = age;
  var bankBalance = 7500;

  this.getBalance = function() {
    return bankBalance;
    };

 var returnBalance = function() {
    return bankBalance; //private method cannot be accesed directly from outside the method
  };

 this.askTeller = function(){
  return returnBalance;
  }
}
var john = new Person('John','Smith',30);
```

```
console.log(john.bankBalance);

// create a new variable myBalance that calls getBalance()
var myBalance1 = john.getBalance();
console.log(myBalance1);


var john = new Person('John','Smith',30);
console.log(john.returnBalance); // this returns undefined
var myBalanceMethod = john.askTeller();
var myBalance = myBalanceMethod();
console.log(myBalance);
```

The way **to access a private method** is similar to accessing a private variable. **You must create a public method for the class that returns the private method.**

askTeller within the Person class that returns the returnBalance method. This means that it returns the method itself and **NOT** the result of calling that method. So you should **NOT** have parentheses after returnBalance.
Because askTeller returns a method, we need to call it to make it any use. This is what var **myBalance = myBalanceMethod();** does.

Using constructor notation, a property declared as this.property = "someValue;" will be public, whereas a property declared with var property = "hiddenValue;" will be private.

```
var languages = {
   english: "Hello!",
   french: "Bonjour!",
   notALanguage: 4,
   spanish: "Hola!"
};

// print hello in the 3 different languages
for (var language in languages){
   if (typeof languages[language] === "string"){ // verification ca sa afiseze
numb stringuri
console.log(languages[language]);
}
}
```

JavaScript object has some baggage associated with it? Part of this baggage was the hasOwnProperty method available to all objects. Now let's see where

this came from...
If we have just a plain object (i.e., not created from a class constructor), recall that it automatically inherits from Object.prototype. Could this be where we get hasOwnProperty from? How can we check?

```
// what is this "Object.prototype" anyway...?
var prototypeType = typeof Object.prototype; //returns object
console.log(prototypeType);

// now let's examine it!
var hasOwn = Object.prototype.hasOwnProperty('hasOwnProperty'); //
returns true
console.log(hasOwn);
```

In general, a += b; means "add b to a and put the result of that addition back into a.

```
var cashRegister = {
    total:0,
    lastTransactionAmount:0,
    //Dont forget to add your property
    add: function(itemCost) {
        this.total +=  itemCost;
        this.lastTransactionAmount = itemCost;
    },
    scan: function(item,quantity) {
        switch (item) {
        case "eggs": this.add(0.98 * quantity);

        break;
        case "milk": this.add(1.23 * quantity); break;
        case "magazine": this.add(4.99 * quantity); break;
        case "chocolate": this.add(0.45 * quantity);
        break;
        }
        return true;
    },
    //Add the voidLastTransaction Method here
    voidLastTranzaction: function(){
      this.total = this.total - this.lastTransactionAmount;
    }
 };

cashRegister.scan('eggs',1);
```

```
cashRegister.scan('milk',1);
cashRegister.scan('magazine',1);
cashRegister.scan('chocolate',4);

//if by accident we scan to many times,Void the last transaction and then add 3 instead
cashRegister.voidLastTranzaction();
cashRegister.scan('chocolate',3);
//Show the total bill
console.log('Your bill is '+cashRegister.total);
```

```
function StaffMember(name,discountPercent){
    this.name = name;
    this.discountPercent = discountPercent;
}

var sally = new StaffMember("Sally",5);
var bob = new StaffMember("Bob",10);
var me  = new StaffMember("Meee", 20);
// Create yourself again as 'me' with a staff discount of 20%


var cashRegister = {
    total:0,
    lastTransactionAmount: 0,
    discountPercent:0,
    add: function(itemCost){
        this.total += (itemCost || 0);
        this.lastTransactionAmount = itemCost;
    },
    scan: function(item,quantity){
        switch (item){
        case "eggs": this.add(0.98 * quantity); break;
        case "milk": this.add(1.23 * quantity); break;
        case "magazine": this.add(4.99 * quantity); break;
        case "chocolate": this.add(0.45 * quantity); break;
        }
        return true;
    },
    voidLastTransaction : function(){
        this.total -= this.lastTransactionAmount;
        this.lastTransactionAmount = 0;
    },
    // Create a new method applyStaffDiscount here
    applyStaffDiscount: function(employee){
        this.discountPercent = employee.discountPercent;
```

```
        this.total -= this.total*this.discountPercent/100;
    }

};

cashRegister.scan('eggs',1);
cashRegister.scan('milk',1);
cashRegister.scan('magazine',3);
// Apply your staff discount by passing the 'me' object
// to applyStaffDiscount
cashRegister.applyStaffDiscount(me);

// Show the total bill
console.log('Your bill is '+cashRegister.total.toFixed(2));
```

---

targetarea clickului into-un iframe in cazul in care am amble:

```
 window.focus(); //force focus on the currenct window;
            window.addEventListener('blur', function(e){
               if(document.activeElement ==
document.querySelector('#carrier_frame'))
               {
                  $('.bull.active').removeClass('active').next().addClass('active');
                  if (imgIndex <= nr) {
                    if (nr < 2) {
                      $('.player_image').attr('src', images[initial].src);
                      $('.top_title').html(images[initial].title);
                    } else {
                      $('.player_image').attr('src', images[imgIndex].src);
                      $('.top_title').html(images[imgIndex].title);
                    }
                    mImg($('.player_image'));
                    imgIndex++;
                    if (nr < 2) {nr = 2;}
                    $('.sildes').html('>> Slideshow ' + imgIndex + '/' + (nr + 1) + ' <<');
                  }
               }
            });
         }
```