**First class functions (Passing functions as arguments)**

* A function is an instance of the Object type;
* A function behaves like any other object;
* We can store functions in a variable;
* We can pass a function as an argument to another function;
* We can return a function from a function;

// Passing functions as arguments

var years = [1990,1965,1937,2005,1998];

function arrayCalc(arr,fn){

var arrRes = [];

for(var i = 0; i < arr.length; i++){

arrRes.push(fn(arr[i]));

}

return arrRes;

}

function calculateAge(el){

return 2016 - el;

}

var ages = arrayCalc(years,calculateAge);

console.log(ages);

//Functions returning functions

function interviewQuestion(job){

if(job === 'designer'){

return function(name){

console.log(name + ', can you please explain what UX design is?');

}

}else if(job === 'teacher'){

return function(name){

console.log('What subject do you teach, ' + name + '?');

}

}else{

return function(name){

console.log('Hello ' + name + ', what do you do?');

}

}

}

var teacherQuestion = interviewQuestion('teacher');

teacherQuestion('John');

var designerQuestion = interviewQuestion('designer');

designerQuestion('John');

console.log(designerQuestion);

var teacherQuestion = interviewQuestion('programmer')('Mike');

**Immediately invoked function expressions (IIFE)**

With IIFE we obtain data privacy and also we don’t interfere with other variables in Global execution context.

function game(){

var score = Math.random() \* 10;

console.log(score >= 5);

}

game();

//IIFE

(function (){

var score = Math.random() \* 10;

console.log(score >= 5);

})();

//IIFE with argument

(function (goodLuck){

var score = Math.random() \* 10;

console.log(score >= 5 - goodLuck);

})(5);

**Closures**

An inner function has always access to the variables and parameters of its outer function, even after the outer function has returned.

Ex (return function has access to ‘’a’’ variable):

function retirement(retirementAge){

var a = ' years left until retirement';

return function(yearOfBirth){

var age = 2016 - yearOfBirth;

console.log((retirementAge - age) + a);

}

}

var retirementUS = retirement(66);

retirementUS(1990);

retirement(66)(1992);

**Bind, call and apply**

var john = {

name: 'John',

age: 26,

job: 'teacher',

presentation: function(style,timeOfDay){

if(style === 'formal'){

console.log('Good ' + timeOfDay + ', Ladies and Gentlemen! I\'m ' + this.name + ', I\'m a ' + this.job + ', I\'m ' + this.age + ' years old');

}else if (style === 'friendly'){

console.log('Hey whats up?' + 'I\'m ' + this.name + ', I\'m a ' + this.job + ', I\'m ' + this.age + ' years old, ' + 'have a nice ' + timeOfDay + '.');

}

}

}

var emily = {

name: 'Emily',

age: 30,

job: 'designer'

};

john.presentation('formal','morning');

//method borrowing

john.presentation.call(emily,'friendly','afternoon');

// john.presentation.apply is the same as call but apply accepts array after object

//Bind method generates the copy of the function and we can store it somewhere

var johnFriendly = john.presentation.bind(john,'friendly');

johnFriendly('morning');

johnFriendly('night');

**Module Pattern in projects**

* Keeping pieces of code that are related to one another together inside of separate, independent and organized units.
* Each of these modules, we’ll have variables and functions that are private, which means that they are only accessible inside of the module. We want that no other code override our data. This is called data encapsulation, which allows us to hide the implementation details of a specific module from the outside scope, so that we only expose a public interface which is sometimes called **an API**.