**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**.

**EVENT BUBLING, TARGET ELEMENT, EVENT DELEGATION**

Event delegation is to not set up the event handler on the original element that we’re interested in, but to attach it to a parent element and, catch the event there because it bubbles up.

Use cases for event delegation:

1. When we have an element with lots of child elements that we are interested in;
2. When we want an event handler attached to an element that is not yet in the DOM when our page is loaded.

When we want to move up from target element to parent, we write:

var ctrlDeleteItem = function(event){

console.log(event.target.parentNode.parentNode.parentNode.parentNode);

};

**ES6/ES2015**

VAR variables are function-scoped and LET and CONST are block-scoped.

BLOCK- it’s simply all the code that is wrapped between the curly braces, so each time we have an if statement or a for block, or a while block, we’re actually creating a new block, and variables declared with “let” and “const” are only valid, are only accessible by the code that are inside of the same block.

**IIFE**

**//ES5**

(function(){

var c = 3;

})();

**//ES6**

{

const a = 1;

let b = 2;

}

**STRINGS**

**//ES5**

console.log('This is ' + variable + " " + variable + "some text again.")

**//ES6**

console.log(`This is ${variable} ${variable} some text again`);

**ARROW FUNCTIONS**

// ES5

var ages5 = years.map(function(el) {

return 2016 - el;

});

// ES6

let ages6 = years.map(el => 2016 - el);

Unlike other functions arrow functions don’t have this keyword, they use this keyword of the function they are written in, and so we say they have a lexical this variable.

The code below returns undefined values, because the method which is called has callback function inside which refers to global this variable(in browser it’s window object).

var box5 = {

color: 'green',

position: 1,

clickMe: function(){

document.querySelector('.green').addEventListener('click',function(){

var str = 'This is box number ' + this.position + ' and it is' + ' ' + this.color;

alert(str);

});

}

}

box5.clickMe();

To avoid this issue, we can define variable in the first function, like this: