**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).

**//ES5**

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:

var box5 = {

color: 'green',

position: 1,

clickMe: function(){

var self = this;

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

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

alert(str);

});

}

}

box5.clickMe();

**ES6**

In case of arrow functions, this keyword in arrow functions is attached to object, so we can use it like this:

const box6 = {

color: 'green',

position: 1,

clickMe: function(){

document.querySelector('.green').addEventListener('click',() => {

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

alert(str);

});

}

}

box6.clickMe();

**Destructuring**

//ES5

var john = ['John',26];

var name = john[0];

var age = john[1];

//ES6

const [name,age] = ['John',26];

**Arrays in ES6**

//ES5

var boxesArr5 = Array.prototype.slice.call(boxes);

boxesArr5.forEach(function(cur){

cur.style.backgroundColor = 'blue';

});

//ES6

const boxesArr6 = Array.from(boxes);

boxesArr6.forEach(cur => cur.style.backgroundColor = 'blue');

//ES5

for(var i = 0; i < boxesArr5.length; i++){

if(boxesArr5[i].className === 'box blue'){

continue;

}

boxesArr5[i].textContent = 'I changed to blue!';

}

//ES6

for(const current of boxesArr6){

if(current.className.includes('blue')){

continue;

}

current.textContent = 'I changed to blue!';

}

//ES5

var ages = [12, 17, 8, 21, 14, 11];

var full = ages.map(function(cur){

return cur >=18;

})

console.log(full);

console.log(full.indexOf(true));

console.log(ages[full.indexOf(true)]);

//ES6

//returns the index of the element's index where the condition is true

console.log(ages.findIndex(cur => cur >= 18));

//returns the element which is true in this condition

console.log(ages.find(cur => cur >= 18));

**The Spread Operator**

Spread operator takes an array and transforms it into single values.

function addFourAges(a, b, c, d){

return a + b + c + d;

}

var sum1 = addFourAges(18, 30, 12, 21);

console.log(sum1);

//ES5

var ages = [18, 30, 12, 21];

var sum2 = addFourAges.apply(null,ages);

console.log(sum2);

//ES6

const sum3 = addFourAges(...ages);

//Joining arrays with spread operator

const familySmith = ['John','Jane','Mark'];

const familyMiller = ['Mary','Bob','Ann'];

const bigFamily = [...familySmith,'Lily',...familyMiller];

**Function Parameters- Rest Parameters**

Rest parameters receive a couple of single values and transforms them into an array.

//ES5

function isFullAges5(){

// console.log(arguments);

var argsArr = Array.prototype.slice.call(arguments);

argsArr.forEach(function(cur){

console.log((2016 -cur) >= 18);

})

}

isFullAges5(1990,1999,1965);

//ES6

//when we give arguments to this function it will transfer the values to an array

function isFullAges6(...years){

console.log(years);

years.forEach(cur => console.log(2016 -cur >= 18));

}

isFullAges6(1990,1999,1965);

**Default Parameters**

//ES5

function SmithPerson(firstName,yearOfBirth,lastName,nationality){

lastName === undefined ? lastName = 'Smith' : lastName = lastName;

nationality == undefined ? nationality = 'American' : nationality = nationality;

this.firstName = firstName;

this.yearOfBirth = yearOfBirth;

this.lastName = lastName;

this.nationality = nationality;

}

var john = new SmithPerson('John',1990);

//ES6

//In ES6 we define default parameters in the perenteces and it of course can be overwritten

function SmithPerson(firstName,yearOfBirth,lastName = 'Smith',nationality = 'American'){

this.firstName = firstName;

this.yearOfBirth = yearOfBirth;

this.lastName = lastName;

this.nationality = nationality;

}

var john = new SmithPerson('John',1990);

**New data structure in ES6/MAPS**

Why maps are better than objects to create hash maps? There are couple of reasons:

* We can use anything as keys
* Maps are iterable, and making it very easy to loop through them and to manipulate data with them.
* It’s really easy to get the size of a map using the size property
* We can easily add and remove data from the map

const question = new Map();

question.set('question','What is the official name of the latest major JS version?');

question.set(1,'ES5');

question.set(2,'ES6');

question.set(3,'ES2015');

question.set(4,'ES7');

question.set('correct',3);

question.set(true,'Correct answer');

question.set(false,'Wrong,please try again');

console.log(question.get('question'));

console.log(question.size);

// if(question.has(4)){

// question.delete(4);

// }

// question.clear();

question.forEach((value,key)=> console.log(`This is ${key}, and it's value is set to: ${value}`));

//Using entries method we can loop through keys and also values

for(let [key, value] of question.entries()){

if(typeof(key) === 'number')

console.log(`This is ${key}, and it's value is set to: ${value}`);

}

const ans = parseInt(prompt('Write the correct answer'));

console.log(question.get(ans ===

question.get('correct')

));

**Classes**

1. Class definitions are not hoisted, so unlike function constructors, we need to first implement a class, and only later in our code we can start using it.
2. We can only add methods to classes, but not properties.

Another thing we can do with classes is to add STATIC methods, and static methods are methods that are simply attached to the class, but not inherited by the class instances, so by the objects that we create through that class.

//ES5

var Person5 = function(name,yearOfBirth,job){

this.name = name;

this.yearOfBirth = yearOfBirth;

this.job = job;

}

Person5.prototype.calculateAge = function(){

var age = new Date().getFullYear() - this.yearOfBirth;

console.log(age);

}

var john5 = new Person5('John',1990,'teacher');

john5.calculateAge();

//ES6

class Person6{

constructor(name,yearOfBirth,job){

this.name = name;

this.yearOfBirth = yearOfBirth;

this.job = job;

}

calculateAge(){

const age = new Date().getFullYear() - this.yearOfBirth;

console.log(age);

}

static greeting(){

console.log('Hey there!');

}

}

const john6 = new Person6('john',1990,'teacher');

john6.calculateAge();

//Static methods are attached to this class definition

Person6.greeting();

class Element {

constructor(name, buildYear) {

this.name = name;

this.buildYear = buildYear;

}

}

class Park extends Element {

constructor(name, buildYear, area, numTrees) {

super(name, buildYear);

this.area = area; //km2

this.numTrees = numTrees;

}

treeDensity() {

const density = this.numTrees / this.area;

console.log(`${this.name} has a tree density of ${density} trees per square km.`);

}

}

class Street extends Element {

constructor(name, buildYear, length, size = 3) {

super(name, buildYear);

this.length = length;

this.size = size;

}

classifyStreet () {

const classification = new Map();

classification.set(1, 'tiny');

classification.set(2, 'small');

classification.set(3, 'normal');

classification.set(4, 'big');

classification.set(5, 'huge');

console.log(`${this.name}, build in ${this.buildYear}, is a ${classification.get(this.size)} street.`);

}

}

const allParks = [new Park('Green Park', 1987, 0.2, 215),

new Park('National Park', 1894, 2.9, 3541),

new Park('Oak Park', 1953, 0.4, 949)];

const allStreets = [new Street('Ocean Avenue', 1999, 1.1, 4),

new Street('Evergreen Street', 2008, 2.7, 2),

new Street('4th Street', 2015, 0.8),

new Street('Sunset Boulevard', 1982, 2.5, 5)];

function calc(arr) {

const sum = arr.reduce((prev, cur, index) => prev + cur, 0);

return [sum, sum / arr.length];

}

function reportParks(p) {

console.log('-----PARKS REPORT-----');

// Density

p.forEach(el => el.treeDensity());

// Average age

const ages = p.map(el => new Date().getFullYear() - el.buildYear);

const [totalAge, avgAge] = calc(ages);

console.log(`Our ${p.length} parks have an average of ${avgAge} years.`);

// Which park has more than 1000 trees

const i = p.map(el => el.numTrees).findIndex(el => el >= 1000);

console.log(`${p[i].name} has more than 1000 trees.`);

}

function reportStreets(s) {

console.log('-----STREETS REPORT-----');

//Total and average length of the town's streets

const [totalLength, avgLength] = calc(s.map(el => el.length));

console.log(`Our ${s.length} streets have a total length of ${totalLength} km, with an average of ${avgLength} km.`);

// CLassify sizes

s.forEach(el => el.classifyStreet());

}

reportParks(allParks);

reportStreets(allStreets);

**Asynchronous JavaScript**

Asynchronous JS is the code that keeps running in the background, while our main code is executing.

We pass in callbacks in asynchronous code that run once the function has finished its work.

Functions like “setTimeout” come from Web APIS, they actually live outside the JavaScript engine itself, and we have access to them because they are also in a JavaScript runtime.

In the case of DOM events our event listeners sit in the Web APIs environment, waiting for a certain event to happen. And as soon as that event then happens, then the callback function is placed on a **Message Queue**, ready to be executed.

The job of the **Event loop** is to constantly monitor the Message Queue and the Execution Stack, and to push the first callback function in line onto the **Execution Stack**, as soon as the stack is empty.

This example is called ‘’Callback hell’’ which can be avoided by **promises**.

function getRecipe(){

setTimeout(() => {

const recipeID = [523, 883, 432, 974];

console.log(recipeID);

setTimeout(id =>{

const recipe = {

title: 'Fresh tomato pasta',

publisher: 'Jonas'

};

console.log(`${id}: ${recipe.title}`);

setTimeout(publisher => {

const recipe2 = {

title: 'Italian Pizza',

publisher: 'Jonas'

}

console.log(recipe2);

},1500,recipe.publisher);

},1500,recipeID[2]);

},1500)

}

getRecipe();

Promises are ES6 features, designed specifically to deal with asynchronous JavaScript.

**Promise** is an object that keeps track about whether a certain event has happened already or not. It determines what happens after the event has happened.

Promise implements the concept of a future value that we’re expecting.

const getIDs = new Promise((resolve,reject) =>{

setTimeout(() =>{

resolve([523, 883, 432, 974]);

},1500);

});

const getRecipe = recID => {

return new Promise((resolve,reject =>{

setTimeout(ID =>{

const recipe = {

title: 'Fresh tomato pasta',

publisher: 'Jonas'

};

resolve(`${ID}: ${recipe.title}`);

},1500,recID);

}));

};

getIDs

//if successful

.then(IDs => {

console.log(IDs);

return getRecipe(IDs[2]);

})

.then(recipe => {

console.log(recipe);

})

//catching the error

.catch(error => {

console.log("error");

});

**Async / Await**

Instead of then and catch which we used in the above code, we can use async function and await promises inside this function.

const getIDs = new Promise((resolve, reject) => {

setTimeout(() => {

resolve([523, 883, 432, 974]);

}, 1500);

});

const getRecipe = recID => {

return new Promise((resolve, reject) => {

setTimeout(ID => {

const recipe = {title: 'Fresh tomato pasta', publisher: 'Jonas'};

resolve(`${ID}: ${recipe.title}`);

}, 1500, recID);

});

};

const getRelated = publisher => {

return new Promise((resolve, reject) => {

setTimeout(pub => {

const recipe = {title: 'Italian Pizza', publisher: 'Jonas'};

resolve(`${pub}: ${recipe.title}`);

}, 1500, publisher);

});

};

async function getRecipesAW(){

const IDs = await getIDs;

console.log(IDs);

const recipe = await getRecipe(IDs[2]);

console.log(recipe);

const related = await getRelated('Jonas');

console.log(related);

return recipe;

}

getRecipesAW().then(result => console.log(`${result} is the best ever`));

**AJAX and API**

**AJAX- A**synchronous **J**avaScript **A**nd **X**ML

API-**A**pplication **P**rogramming **I**nterface-It’s a piece of software that can be used by another piece of software in order to allow applications to talk to each other.

In referring to JavaScript API is part of the server, which receive the requests and sends back responses.

We can use 2 types of APIs,

* Our own APIs, for data coming from our own server.
* 3rd-party APIs:

1. Google maps
2. Embed YouTube videos
3. Weather data
4. Movies data
5. Send email or SMS
6. Thousands of possibilities

**Making AJAX Calls with Fetch and Promises**

Same origin policy in JavaScript prevents us from making AJAX requests to a domain different than our own domain.

In order to make requests to different domains, **C**ross **O**rigin **R**esource **S**haring, or **CORS**, was developed.

So, in order to send request for example to metaweather.com, the developers of the API, they need to implement CORS on their server.

There is other option too, what developers usually do is to proxy or to channel the request through their own server, like doing the AJAX request on our own server, where the same policy doesn’t exist and then send the data to the browser( crossorigin.me proxy).

function getWeather(woeid){

//fetch API requests the data and returns the promise

fetch(`https://crossorigin.me/https://www.metaweather.com/api/location/${woeid}/`)

.then(result => {

// console.log(result);

return result.json();

})

.then(data => {

// console.log(data))

const today = data.consolidated\_weather[0];

console.log(`Temperature in ${data.title} stay between ${today.min\_temp} and ${today.max\_temp}. `)

})

.catch(error => console.log(error));

}

//Weather in San francisco

getWeather(2487956);

//for London

getWeather(44418);

**Making AJAX calls with Fetch and ASYNC/Await**

async function getWeatherAW(woeid){

const result = await fetch(`https://crossorigin.me/https://www.metaweather.com/api/location/${woeid}/`);

const data = await result.json();

const today = data.consolidated\_weather[0];

console.log(`Temperature in ${data.title} stay between ${today.min\_temp} and ${today.max\_temp}. `)

}

getWeatherAW(2487956);

getWeatherAW(44418);

**Handling errors with try catch**

async function getWeatherAW(woeid){

try{

const result = await fetch(`https://crossorigin.me/https://www.metaweather.com/api/location/${woeid}/`);

const data = await result.json();

const today = data.consolidated\_weather[0];

console.log(`Temperature in ${data.title} stay between ${today.min\_temp} and ${today.max\_temp}. `);

}catch(error){

console.log(error);

}

}