GRAUMAN

HI-TECH DEV COURSES

Front End

JavaScript | HTML | CSS

JavaScript

Shahar Grauman

www.grauman.co.il | 054-8002004 | info@grauman.co.il

JavaScript History

- Netscape

- The browser wars

- Standardization – ECMAScript

- jQuery

- Node

- SPA

JavaScript in a nutshell

- All browser support JS
    - After all, JS is the language of the browsers
- Client Side
- Structured and syntax inspired by C language
- Dynamic
    - Types, Evaluation at runtime
- Prototype based, as opposed to classical OO with classes
- Function is First-Class
    - Meaning it’s an object! It can be passed around, assigned to variables, create objects

JavaScript on the Server?

- Node.js revolutionized the way JS can be used
- Node is a server-side runtime
    - Allow JS programs on the 'other end'
    - Full-Stack JS apps!

- NPM - The worlds' biggest open-source ecosystem
    - More than 1,000,000 packages!
    - Currently v14
    - Keeps evolving
    - New release every 6 months

ECMAScript

In 2015, the 6th version (referred as ES6 or ES2015) was released

ES6 brought a lot of new features and improvements, most notably

- Block scoped variables (let, const)
- Arrow functions (lambda)
- Enhanced parameters (default values, rest)
- Destructuring
- Modules (Yet to be natively supported)
- Classes syntax (‘Looks like’ OOP)
- Iterators
- Generators
- Promises
- and more...

Variables

- var was used more than 20 years
- Introduced scope issues
- var is scoped to the function
    - regardless of its declaration
    - Hoisting mechanism

- Primitives:
    - number, string, boolean, object, undefined, null, symbol (es6)
    - All the others considered as object types
    - JavaScript is dynamically typed

```
var whoAmI; //undefined
whoAmI = 17;
whoAmI = false;
whoAmI = "Shahar";
whoAmI = { name: 'Shahar', age: 27 };
```

== vs. ===

== checks for value equality

=== behaves as **==** but with type check

```
console.log('== vs. ===');
console.log("0 == ''", 0 == ''); // true
console.log("0 === ''", 0 === ''); // false
console.log("0 == '0'", 0 == '0'); // true
console.log("0 === '0'", 0 === '0'); // false
console.log("false == '0'", false == '0'); // true
console.log("false === '0'", false === '0'); // false
console.log("true == 1", true == 1); // true
console.log("true === 1", true === 1); // false
console.log("null == undefined", null == undefined); // true
console.log("null === undefined", null === undefined); // false
```

typeof operator

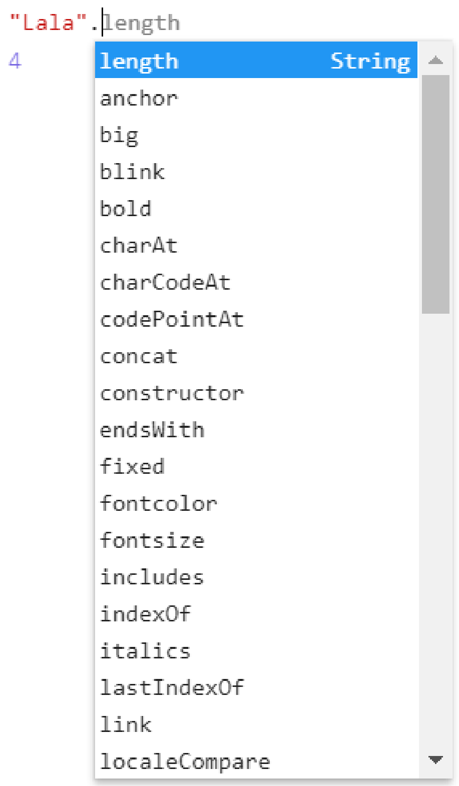
```
console.log('\ntypeof operator');

console.log(typeof "lala");
console.log(typeof '@');

console.log(typeof 3);
console.log(typeof 3.5);
console.log(typeof Infinity);
console.log(typeof NaN);

console.log(typeof true);
console.log(typeof false);

console.log(typeof [1, 2, 3, 4]);
console.log(typeof { name: 'Shahar', age: 34 });
console.log(typeof /^[0-9]$/);
console.log(typeof (new Date()));
console.log(typeof null);

console.log(typeof undefined);

console.log(typeof function () { });

console.log(typeof x === undefined ? 'x is undefined' : 'x is defined');
console.log(x === undefined ? 'x is undefined' : 'x is defined');
```

What can be sent to test for the output to be truthy?

```
function test(x) {
    return x != x;
}
```

Strings

String is a fundamental aspect in any programming language.


```
"Lala".length
4
```

As such, it has lots of methods for our disposal.

```
var text = 'Hi there amigos';
var str = "Ahalan Jama'a";
var firstChar = text.charAt(0);//H
var fourthChar = str[3];//l
str.substring(0, 6);//Ahalan
str.indexOf('J');//7
```

String is immutable, meaning, it can't be changed after initialized.

So always prefer *string interpolation* (ES6) over merging. (Think why!)

```
var interpolation = `${text}! and ${str}. Welcome`;
var chaining = text + '! and ' + str + '. Welcome';
```

Functions

In JavaScript, functions have more, much more capabilities than other languages.

Function can receive arguments, but unsupplied argument is *undefined*

```
function addXtoY(x, y) {
  //We can check if an argument is defined
  if (x == undefined) {
    console.log("x want provided");
  } else if (y == undefined) {
    console.log("y wasnt provided");
  } else {
    //If both x and y are OK, then calculate
    console.log(x + y);
  }
}
addXtoY(2, 7);
addXtoY(2);
```

Ex (functions 1):

1- Make sure addXtoY works here as well: addXtoY(3, ‘two’);

2- If there’s a problem, return NaN. Otherwise return the result.

3- Check the result of addXtoY and print appropriate message.

Ex (functions 2):

Write a function which gets a day and returns:

If the day is Sunday/Monday	return 'Have a good week'
If the day is Tuesday/Wednesday	return 'Ohhh...'
If the day is Thursday	return 'Yalla habayta'
If the day is Friday/Saturday	return 'Yammi Jachnun'

Print the result

Ex (functions 3):

Convert the previous function by using switch statement

Ex (functions 4):

Complete this code

```
function doSomethingExtraOrdiner(number, x) {
    var isDivided = findIfNumberIsDividedByX(number, x);
    var res;
    if(isDivided) {
        res = multiplyNumberByXWithoutMultiplication(number, x);
    }
    if(res){
        printResult(number, x, res);
    }
}
```

Ex (functions 5):

1- Write a function which gets a number and returns the sum of its digits

For example: 745238 -> 29

2- Write a function which gets a number and sum its digits until it's a single digit

For example: 745238 -> 29 -> 11 -> 2

Arrays

Arrays are objects!

They are very dynamic and expose numerous functionalities.

```
var array = [];
array.push(1);
array.push(2);
array.push(3);
//[1, 2, 3]
array.length; // 3
array.pop();
//[1, 2]
array.length; // 2
```

Iterating over an array using index or foreach

```
for(var index = 0; index < arr.length; index++) {
    //arr[index] will give us the value in this position
    var value = arr[index];
    console.log(value);
}

for(var item of arr) {
    console.log(item);
}
```

We can declare the array with initial values

```
var numbers = [11, 5, -2, 3, 14];
var names = ['Shahar', 'Mahmud', 'Lucie', 'Abraham'];
var sortedNames = names.sort();
```

Quick Ex:

Try sorting the above numbers array and figure out the issue.

Ex (Arrays 1):

You have numbers array

Create another array containing the power by 2 of each number from the first array

Ex (Arrays 2):

You have numbers array

Create another array containing the negative number of each number from the first array

After completing Ex1+2, do you see a pattern here?
Can you extract out the similarity between the two above solutions?

Ex (Arrays 3):

Write a function which gets 2 parameters: an array of something and a function. The function returns array of whatever the function parameter does.

For example:

- You send it ['Avi', 'Shahar', 'Gila', 'Omer', 'Sara', 'Ilan', 'Shani'] and a function (think of it) - and gets back [3,6,4,4,4,4,5]

- You send it [1,2,3,4,5] and a function (think of it) and gets back [1,4,9,16,25]

- You send it ['Avi', 'Shahar', 'Gila', 'Omer', 'Sara', 'Ilan', 'Shani'] and a function (think of it) - and gets back ['A', 'S', 'G', 'O', 'S', 'I',' S']

Arrays have may useful functions we can use.

Most notably is *map*. *map* does very similar thing to what we did in previous exercises.

Using our convert functions, we can ease the pain:

```
var names = ['Avi', 'Shahar', 'Gila', 'Omer', 'Sara', 'Ilan', 'Shani'];
var namesLength = names.map(getLength);
var numbers = [1, 2, 3, 4, 5, 6];
var powersBy2 = numbers.map(powerBy2);
var firstLetters = names.map(firstLetter);
```

<u>Ex (Arrays 4):</u>
Using map:
1- Convert [7.9, 1.2, 3.47, 10.69] to [8, 1, 3, 11]
2- Convert ['Sha', 'lab', 'Of', 'Dok'] to ['a', 'b', 'f', 'k']

Another useful function is *filter* which screens values by a condition we provide.

<u>Ex (Arrays 5):</u>
Using filter:
Create array of 10 random numbers between 24 and 43.
Create another array of the numbers starting with 3 (from the first array).

You see? We handle a single case and let map/filter do the hard job for us.

There are more array functions we'll investigate as we proceed.

var scope

What is printed here (Or do we have an error)?

```
for (var i = 1; i < 5; i++) {
  console.log(i);
}

console.log(i); //?
```

var scope – **hoisting**

What is printed here?

```
function hoist() {
    var ok = true;

    if (ok) {
        var notOK = false;
        console.log(notOK);
    }

    console.log(ok, notOK);
}

hoist();
```

JS interpreter reorganizes the code and hoist notOK to the top of the function:

```
function hoist(){
    var notOK;
    var ok = true;

        if (ok) {
        var notOK = false;
        console.log(notOK);
    }

    console.log(ok, notOK);
}

hoist();
```

This is the reason it's available to the last console.log...

So now that you’re more familiar with hoisting, what is printed here?

```
function hoist2(){
    console.log(i);
    for (var i = 0; i < 5; i++) {
        console.log(i);
    }
    console.log(i);
}

hoist2();
```

This is a tough one.
Although you're not familiar yet with some features, you can still understand what's going on.

What is printed here?

```
function loopsi(){
    var numbers = [];

    for(var i = 0; i < 5; i++){
        numbers.push(function () { return i; });
    }

    numbers.forEach(item => console.log(item()));
}

loopsi();
```

How can we fix that?

Because *var* is hoisted, all functions refer to the same variable!

There are several techniques, but ES6 brought us the *let* keyword

```
function loopsi2(){
    var numbers = [];

    for(let i = 0; i < 5; i++){
        numbers.push(function () { return i; });
    }

    numbers.forEach(item => console.log(item()));
}

loopsi2();
```

let and const

- *let* and *const* was introduced in es 2015 (ES6)

- *let* behaves similarly to declarations in other languages

- It's available **ONLY** inside the immediate surrounding block

    - From the declaration line until the end of the block

- *const* is like *let*, but needs initialization right away

- Can't be modified later on

- If *const* is reference, it can't be assigned a new one

    - But we can modify the data it points to

```
for (let j = 0; j < 5; j++) {
    console.log(j);
}
console.log(j);// Error

const num = 100;
num++; //Error
```

A thumb of rule: Use *const*. If the variable may be reassigned, use *let*.

Functions - Continued

- Functions are objects
- Function constructors
- Can be assigned to a variable
- Can be passed as an argument
- Closure - Can be nested inside another function
- Can get more/less arguments than declared
- Can invoke itself (uh?!)
- Lambda (Arrow functions)

Functions are objects

```
function imAnObject(){
    console.log('Hi there' + imAnObject.whatsup);
}

imAnObject.whatsup = 'Whasssaaaa?';
imAnObject();
```

Function Expressions

- JS functions are flexible structure, for example:

  You can name a function as usual and assign it to a variable like so:

```
function calc(num) {
    return num * 2;
}

//Function expression
let calcF = calc;

console.log(calcF(3));
```

Anonymous function

```
const myFunc = function(x){
    console.log(isNaN(x) ? 'Not a number' : `x is ${x}`);
}

myFunc();
myFunc(8);
myFunc('6t');
myFunc(0x32);
```

function can be passed as argument

```
function calcThis(x, func){
    func(x);
    console.log(`Raising x by 2: ${Math.pow(x,2)}`);
}

calcThis(8, myFunc);
```

* Heavily used for callbacks (e.g. when a button is clicked)

There is no overloading in JS

```
function studentDetails(name, age, email) {
    let details = 'No details!';

    if (name != undefined) {
        details = `Student Details:\nName: ${name}`;
    }

    if (age != undefined) {
        details += `\nAge: ${age}`;
    }

    if (email != undefined) {
        details += `\nEmail: ${email}`;
    }

    return details;
}

console.log(studentDetails());
console.log(studentDetails('Shahar'));
console.log(studentDetails('Shahar', 27));
console.log(studentDetails('Shahar', 27, 'info@grauman.co.il'));
```

It is common to pass an object containing the parameters

As with configuration objects

```javascript
function studentDetails2(props) {

    if (!props) return 'No details!';

    let details = '';

    if (props.name != undefined) {
        details = `Student Details:\nName: ${props.name}`;
    }

    if (props.age != undefined) {
        details += `\nAge: ${props.age}`;
    }

    if (props.email != undefined) {
        details += `\nEmail: ${props.email}`;
    }

    return details;
}

console.log(studentDetails2());
console.log(studentDetails2({ name: 'Shahar' }));
console.log(studentDetails2({ name: 'Shahar', age: 27 }));
console.log(studentDetails2({ name: 'Shahar', age: 27, email:
'info@grauman.co.il' }));
```

Destructuring is a new feature of es6.

(more on that later) but let's see a bit of it

```
function studentDetails3({ name, age, email }) {

    let details = 'No details!';

    if (name != undefined) {
        details = `Student Details:\nName: ${name}`;
    }

    if (age != undefined) {
        details += `\nAge: ${age}`;
    }

    if (email != undefined) {
        details += `\nEmail: ${email}`;
    }

    return details;
}

console.log(studentDetails3({ name: 'Shahar' }));
console.log(studentDetails3({ age: 27, name: 'Shahar' }));
console.log(studentDetails3({ email: 'info@grauman.co.il', age:
27, name: 'Shahar' }));
```

functions can be nested, so the inner has the scope of the outer.

- Meaning, the inner has access to outer variables/parameters
- This is called *Closure*

```
function outer(num) {
    let num1 = 9;

    function inner(x) {
        console.log('inner', num + x + num1);
        return x;
    }

    console.log('outer', inner(7));
}

outer(2);
```

Functions maintain a lexical scoping of variables upon declaration

- It enables encapsulation

```
function func2(){
    let num = 12;

    function square(){
        return num * num;
    }

    return square;
}

let innerF = func2();

console.log(innerF());
```

```
function F(f) {
  function G(g) {
    function H(h) {
      console.log(f + g + h);
    }
    H(3);
  }
  G(2);
}
F(1);
```

How do we use this code?

```
function sumNumbers() {
    var sum = 0;
    function addNumber(x) {
        sum += x;
        return sum;
    }
    return addNumber;
}
```

Ex (Closure 1):

Create an outer function which returns inner function

This function receives character

It returns all previous characters, but reversed

For example:

You call the outer and receive the inner

Next you call the inner with 'a' - > 'a'

Next you call the inner with 'b' - > 'ba'

Next you call the inner with 'c' - > 'cba'

Function's arguments object

Functions can be passed less/more arguments than declared

```
function concat(separator) {
   var result = '', i;

   for (i = 1; i < arguments.length; i++) {
      result += arguments[i] + separator;
   }

   return result;
}

concat(', ', 'me', 'you', 'her');
concat('; ', 'X', 'R', 'S', 'U');
concat('. ', 'js', 'node', 'es6', 'web', 'fun');
```

Prior to es6, you might stumble upon validation code like this

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;

  return a * b;
}

multiply(5);
```

But now - default value to the rescue

```
function multiply (a, b = 2) {
  return a * b;
}
multiply(5);
```

```
function multiply (a, b = 2) {
  return a * b;
}

function foo (num = 1, mul = multiply(num)) {
  return [num, mul];
}
foo(); // [1, 2]
foo(6); // [6, 12]
```

Immediate Invoked Function Expression

Immediate **I**nvoked **F**unction **E**xpression (IIFE)

```
(function(){
    console.log('Shahar');
})();
```

```
(function(str){
    console.log(str);
})('Shahar');
```

```
var res = (function () {
    var name = 'Shahar';
    return name;
})();
console.log(res);
```

So, function can be self-invoked! Yap.

But Why?

It can create closure out of it, making private scopes and modules possible

```
var addFunc = (function () {
    var sum = 0;
    function addNumber(x) {
        sum += x;
        return sum;
    }
    return addNumber;
})();

var result = addFunc(1); //1
result = addFunc(2); //3
result = addFunc(3); //6
```

```
var config = (function (){
    //Get data from somewhere
    var server = 'server-address',
        database = 'database-name',
        userid = 'username',
        pwd = 'password';

    return {
        getSQLConnectionString() {
            return `Server=${server};
               Database=${database}; User Id=${userid};
               Password=${pwd}`;
        }
        //more config stuff...
    };
})();
```

```
var images = (function countImgs(element) {
    if (element.nodeName.toLowerCase() === 'img') return 1;

    var count = 0;
    for (var i = 0, child; child = element.childNodes[i]; i++) {
        count += countImgs(child);
    }
    return count;
})(document.body);
```

- Invoking IIFE with argument

- The name, countImgs, is necessary here

    - To be able to call it recursively

    - Notice - the name can be called ONLY within the scope of the IIFE

Remember this?

```
function func(){
    let funcs = [];

    for(var i = 0; i < 5; i++) {
        var res = function(){
            console.log(i);
        }
        funcs.push(res);
    }

    funcs.forEach(f => f());
}
func();
```

Fix this by

1- Using Closure

2- Using IIFE

Closure

```
function func(){
    let funcs = [];

    for(var i = 0; i < 5; i++) {
        function outer(){
            let num = i;
            function res(){
                console.log(num);
            }
            return res;
        }
        funcs.push(outer());
    }

    funcs.forEach(f => f());
}
func();
```

IIFE

```
function func(){
    let funcs = [];

    for(var i = 0; i < 5; i++) {
        funcs.push((function(){
            let num = i;
            function res(){
                console.log(num);
            }
            return res;
        })());
    }

    funcs.forEach(f => f());
}
func();
```

The same, but with parameter

```
function func(){
    var numbers = [];

    for(var i = 0; i < 5; i++){
        numbers.push((function(n){
            return function () { console.log(n); }
        })(i));
    }

    numbers.forEach(f => f());
}

func();
```

The same but with lambda

```
function func(){
    var numbers = [];

    for(var i = 0; i < 5; i++){
        numbers.push(((n) => () => console.log(n))(i));
    }

    numbers.forEach(f => f());
}

func();
```

Or… just use *let* 😊

```
function func(){
    let funcs = [];

    for(let i = 0; i < 5; i++) {
        funcs.push(() => console.log(i));
    }

    funcs.forEach(f => f());
}
```

So, let's see what you've got for me:

(1) What is the output of this?

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }
    var b;
    b = a + doSomethingElse(a * 2);
    console.log(b * 3);
}
doSomething(2);
```

(2) What is the output of this?

```
function foo() {
    function bar(a) {
        i = 3;

        console.log(a + i);
    }
    for (var i = 0; i < 10; i++) {
        bar(i * 2);
    }
}
foo();
```

(3) What is the output of this?

```
var a = 2;

(function func(obj) {
    var a = 3;
    console.log(a);
    console.log(obj.a);
})(window);

console.log(a);
```

(4) What is the output of this?

```
var a = 2;
(function func(def) {
    def(window);
})(function (obj) {
    var a = 3;
    console.log(a);
    console.log(obj.a);
});
```

Object Oriented

Until ES6, JavaScript didn't have classes at all and (actually) it still hasn't, but we'll see it later on.

JavaScript is a prototypal language. Each object is linked to a parent object (The Prototype) and can access it's properties.

So how can we group few properties together like in other languages?

Good you asked.

There're several ways we can create objects.

The very simple way is like that:

```
const person = {
    name: 'Shahar',
    age: 27,
    address: {
        city: 'Karkur',
        street: 'Neta'
    },
    phone: '054-8002004'
};
```

Just use curly braces and add properties as key:value pairs.

Use dot (.) to access object's properties

```
console.log(`Person ${person.name} lives in ${person.address.city}`);
person.age++;
```

Ex (OO 1):

Create array of 4 student objects with name, course and city.

Print those who live in Haifa.

Object can have functions. We call object.function a *method*.

```
const person = {
    name: 'Shahar',
    age: 27,
    address: {
        city: 'Karkur',
        street: 'Neta'
    },
    phone: '054-8002004',
    toString: function(){
        return `${this.name} aged ${this.age} with phone ${this.phone}
lives in ${this.address.street}, ${this.address.city}`;
    },
    happyBirthday: function(){
        this.age++;
    }
};
```

Notice we use *this* to access object's properties.

Why do we need *this* inside methods?

Consider this example:

```
function myNameIs(){
    console.log(this.name);
}
```

What will be printed if we call myNameIs?

Whom does *this* refer to?

```
function myNameIs(){
    console.log(this.name);
}
name = 'lala';
myNameIs();
myNameIs.call(person);
```

Shahar Grauman
www.grauman.co.il | 054-8002004 | info@grauman.co.il

We can *call* a function and tell it what *this* should be.
So when we're inside a *method*, *this* is the object we're working on.

toString is a special method:

```
console.log(`Person Details: ${person}`);

// Person Details: Shahar aged 27 with phone 054-8002004 lives in
Neta, Karkur
```

When we created person, JavaScript set its prototype to Object, which is the base prototype of all objects.

***console.log* needs to print person as a string. It can't figure out how to do it, so it seeks a *toString* method. Not found? It searches up in the *prototype chain* until it meets *Object.toString* which prints a default [Object object] string. But we <u>*override*</u> *toString*, so console.log need not search further.**

Attaching outer function to behave like methods

```
function sayYourName() {
    console.log(this.myName);
}
const person = {
    myName: 'Shahar',
    sayYourName
}

person.sayYourName();
```

Or binding the *this* context for later use

```
function sayYourName() {
    console.log(this.myName);
}

const person = {
    myName: 'Shahar'
}

const saySomething = sayYourName.bind(person);
saySomething();
```

Objects are fine, but in this way, each object is independent of the others.

In order to create objects of the same footprint, we can use functions.

Yes, functions again (and all over 😊)

Function Constructors

```
function Car(model) {
  this.model = model;
  this.wheels = 4;
}

var myCar = new Car("Volvo");
console.log(`Car ${myCar.model} has ${myCar.wheels} wheels`);

myCar.constructor.name; //'Car'
```

Using the *new* keyword, the function behaves as a *constructor*, creating an object and using the newly created object with the *this* keyword to set up key:values, and finally – returns the new object.

Ex (OO 2):
Create a function constructor of a Point with x and y (1,1 if no values supplied)
Add a method which gets another point and returns the biggest point by x
Create 3 Point objects.
Using only the method - find the biggest point.

Now, the bad news are that the method you've added is copied for each point object!
We need a way in which all objects have the same method.

Prototypes

All JavaScript objects inherit properties and methods from a prototype

The *Object.prototype* is on the top of the prototype inheritance chain:

- Array objects inherit from *Array.prototype*

- Date objects inherit from *Date.prototype*

- Car objects inherit from *Car.prototype*

- All the above inherit from *Object.prototype*

```
function sayYourName() {
    console.log(this.myName);
}
const person = {
    myName: 'Shahar'
}
//Set the prototype of person
Object.setPrototypeOf(person, {saySomething: sayYourName});
person.saySomething();//Shahar
console.log(person);//{ myName: 'Shahar' }
```

So, by setting the prototype, we're actually delegating the functionality to a higher object.

If the current object doesn't have the method, it's being looked up the *prototype chain*

```
const person = { sayYourName }
const shahar = { myName: 'Shahar' }

Object.setPrototypeOf(shahar, person);
shahar.sayYourName();
```

We can chain prototypes

```
const person = {
    sayYourName
}
const shahar = {
    myName: 'Shahar'
}
const son = {
    myName: 'Hanoch'
}

Object.setPrototypeOf(shahar, person);
Object.setPrototypeOf(son, shahar);
shahar.sayYourName();
son.sayYourName();
```

Take a minute and examine the following code

```
const person = { };

const shahar = {
    myName: 'Shahar',
    sayYourName
};

Object.setPrototypeOf(shahar, person);

//Now shahar has it too!
person.stateYourName = function(){
    console.log(`Please state your name: ${this.myName}`);
}

shahar.sayYourName();
shahar.stateYourName();

//Override by implement the same function down the chain

person.sayYourName = function(){
    console.log(`My name is: ${this.myName}`);
}

delete shahar.sayYourName;
shahar.sayYourName();
```

Notice we can use *delete* to remove properties from an object.

Still, properties lookup always occur until it reached somewhere higher, or not found at all (which triggers an exception).

A common practice is to use *function constructors* for setting properties and outside of it – settings *prototype* methods

```javascript
function Person(name, age){
    this.name = name;
    this.age = age;
}

let me = new Person('Shahar', 27);
console.log(me.name, me.age);

let him = new Person('Yeled', 4);

Person.prototype.details = function(){
    console.log(`Name: ${this.name}, Age: ${this.age}`);
}
him.details();
me.details();
```

Create a new object, using an existing object as the prototype of the newly created object

```javascript
function Student(name, age, course){
    Person.call(this, name, age);
    this.course = course;
}
Student.prototype = Object.create(Person.prototype);
```

Then we 'reset back' the constructor because now it's Person

```javascript
Student.prototype.constructor = Student;
```

Student has *details* method from its prototype

```javascript
let student = new Student('Ace', 25, 'JavaScript');
student.details();
```

We can override *details* method and if needed, call ‘parent’ *details* method

```
Student.prototype.details = function(){
    //Using delegation to parent
    Person.prototype.details.call(this);
    console.log(`Studying ${this.course}`);
};

student.details();
```

Ex (OO 3):

Make this work

```
const arr = ['a','b','c','d','e','f'];
arr.shuffle();
console.log(arr); //['b','d','a','c','f', 'e']
```

Ex (OO 4):

Write a Shape with x and y

Add to the prototype 2 methods: toString and area

Write a Rectangle (inheriting from Shape) with width and length

Override both methods

Write a Box (inheriting from Rectangle) adding height

Override both methods

Create Shape, Rectangle and Box objects and print their details and area

Output should look like this

```
Shape (12,23)
Shape area is 0
Rectangle (34,55) width 11 length 20
Rectangle area is 220
Box (66,21) width 10 length 20 Height 30
Box area is 2200
```

The operator *Instanceof* checks if constructor.prototype exists in the object's prototype chain.

Ex (OO 4 – continued):

Validate that

```
box.__proto__.constructor.name
```

is Box and not Shape!

EX (OO 5):

What is the output here?

```
const test = {
    foo: function() {
        console.log(test === this);

        function bar(){
            console.log(test === this);
        }

        bar();
    }
}
test.foo();
```

This is a common mistake to assume nested function has *this* as methods have. Actually, it gets the global object as the context.

To get around that, 'cache' *this* as local variable and use it onwards as the context.

Ex (OO 6):

Use *assignTo* for the following questions

```
function assignTo(x, y, msg) {
    this.summary = `${msg}: ${x + y}`;
}
```

1- Use it with global object and print the summary

2- Define your own object with *assignTo* and print the summary

```
const stamObj = {};
```

3- Without modifying *stamObj*, use *call* and print the summary

4- Without modifying *stamObj*, use *apply* and print the summary

5- Use *assignTo* as function constructor and print the summary

Ex (OO 7):

We're going to investigate (at a glance) JS in HTML page:

Create a simple HTML page with input of type color.

User selects a color and triggers our code.

Our code delays for 8 seconds and present alert with this data:

- Current selected color
- Previous selected color
- Index number of current selection
- Total number of selections

```html
<!DOCTYPE html>
<head>
    <meta charset="utf-8" />
    <title>JavaScript Ex</title>
</head>

<body>
    <input type="color" id="btnColor" />

    <script>
        const btnColor = document.getElementById('btnColor');

        btnColor.addEventListener('change', function() {
            setTimeout(() => {
                alert(btnColor.value);
            }, 8000);
        });
    </script>
</body>

</html>
```

For example: User chose green and then red.

First alert (After 8 sec): color: #00ff00, prev: #000000, index: 1, total: 2

Second alert (After 8 sec): color: #ff0000, prev: #00ff00, index: 2, total: 2

We saw how prototype works – the ability to inherit from some object.

We can check if object own a property or got it from prototype:

```
const obj = {
    num: 42,
    details: function(){
        console.log(this.num);
    }
}

console.log('num' in obj);//true
console.log('details' in obj);//true
console.log('toString' in obj);//true
console.log('age' in obj);//false
```

As you can see, *toString* is inherited from *Object*.

But obj doesn't have this method.

We can check if an object owns a method by using *hasOwnProperty*

```
console.log(obj.hasOwnProperty('num'));//true
console.log(obj.hasOwnProperty('details'));//true
console.log(obj.hasOwnProperty('toString'));//false
console.log(obj.hasOwnProperty('age'));//false
```

In order to iterate over object properties, we can use for-in loop

```
for(const prop in obj){
    console.log(prop);
}
// num
// details

console.log(obj.propertyIsEnumerable('details'));//true
console.log(obj.propertyIsEnumerable('toString'));//false
```

Another interesting Object inherited method is *valueOf.*

It allows you to convert an object to primitive.

<u>Ex (OO 8):</u>

Create *Grade* function constructor having student name and grade.

Create 3 instances:

- 'Shahar' and 88,
- 'Yusuf' and 79
- 'Suha' and 91.

Make this work:

```
console.log(`Average: ${(shahar + yusuf + suha) / 3}`);
//Average: 86
```

ECMAScript Enhancements

We’ve seen some es6 features along the way until here (e.g. string interpolation).

- getter/setter
- Class
- Fat Arrow (Lambda) Functions
- Rest & Spread operators
- Destructuring
- New Object Methods
- New Collections
- Promise

getter/setter

```
const employee = {
    name: 'Shahar',
    salary: 3500,
    phones: [],
    get vat(){
        return this.salary * 0.17;
    },
    set phone(p) {
        this.phones.push(p);
    }
}

console.log(`Emp name: ${employee.name}.
Salary ${employee.salary}.
VAT: ${employee.vat} NIS`);

employee.phone = '0548962002';
employee.phone = '0521010447';
console.log(`Phones: ${employee.phones}`);

// Emp name: Shahar.
// Salary 3500.
// VAT: 595 NIS
// Phones: 0548962002,0521010447
```

Class

The first one is of course *class*!
For many years, developers who came from OOP languages such as C# or Java, struggled with prototypes and tried to make it ‘behave’.

But the thing is – JavaScript is the really ***Object****-Oriented* language around!

Nevertheless, ES6 added the *class* feature as a **syntactic sugar** around prototypes:

```
class Person {
  //Only 1 constructor is allowed
  constructor(name, age) {
    this.name = name;
    this.age = age;
    Person.counter++;
  }
  //getter
  get details(){
    return `${this.name}, ${this.age}`;
  }
  //Static methods can be defined inside the class
  static Counter() {
    return Person.counter;
  }
}
//Static properties - must be defined outside
Person.counter = 0;
```

*Read the comments carefully.

We can inherit from parent class

```
class Student extends Person {
  constructor(name, age, course){
    //First line should call super ctor
    super(name, age);
    //Before accessing 'this'
    this.course = course;
  }

  //Override super method
  get details() {
    return `${super.details}, studying ${this.course}`;
  }
}
```

Ex (ES6 1):

Implement the previous Shapes exercise using classes.

- Add new shape – Circle.
- Add the ability to know how much shapes instances were created.
- Add to Box volume getter.
- Create array of shapes (Use the shapes you've already constructed)
  Print the shapes details
  Print the shapes area
- If the shape is Box, print it's volume

Fat Arrow Functions

Allow cleaner syntax

Preserve the 'this' context in which they operate in

```
let add1 = function(x, y) {
    return x + y;
}

console.log(add1(3,4));


let add2 = (x, y) => x + y;

console.log(add2(3,4));
```

```
var strings = [
  'Lala',
  'Wawa',
  'Babala',
  'Sababa'
];

var arr = strings.map(function(str) {
   return str.length;
});
console.log(arr);


var arr2 = strings.map(str => str.length);
console.log(arr2);
```

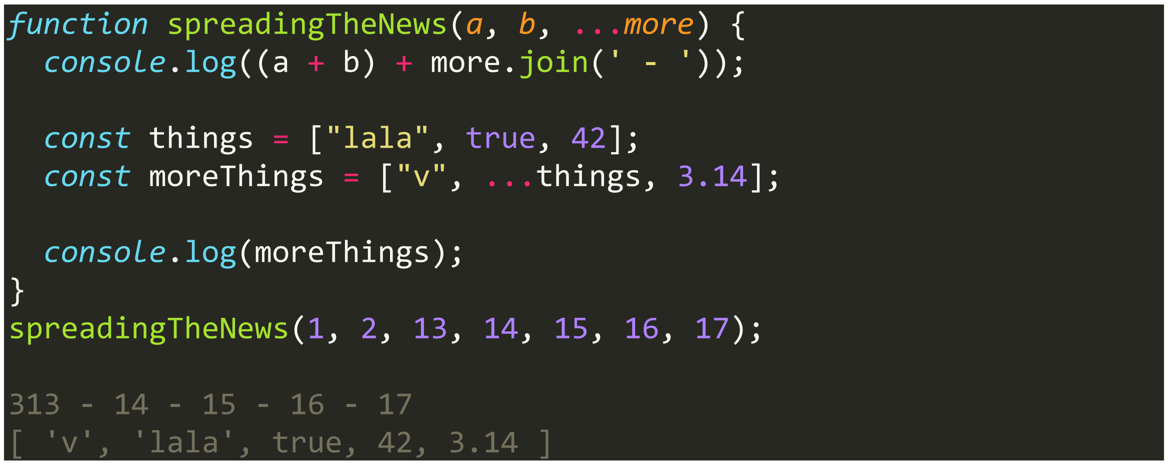
Remember this?

```
const test = {
    foo: function() {
        var self = this;
        console.log(test === this);
        console.log(test === self);

        function bar(){
            console.log(test == this);
            console.log(test === self);
        }

        bar();
    }
}

test.foo();
```

Now fix this using arrow function!

rest & spread operators

rest - the ability to send parameters which are grouped together as array
spread – the ability to enumerate at a glance



```
function spreadingTheNews(a, b, ...more) {
  console.log((a + b) + more.join(' - '));

  const things = ["lala", true, 42];
  const moreThings = ["v", ...things, 3.14];

  console.log(moreThings);
}
spreadingTheNews(1, 2, 13, 14, 15, 16, 17);

313 - 14 - 15 - 16 - 17
[ 'v', 'lala', true, 42, 3.14 ]
```

<u>Ex (ES6 2):</u>
Write a function that gets 2 arrays and returns them as 1 array.
Don't use Array.concat

<u>Ex (ES6 3):</u>
Write a function which gets 2d array and returns it flattened as 1 array. Use the previous function.
For Example: For [[1, 2, 3], [4, 5], [6]] you get [1, 2, 3, 4, 5, 6]

Using spread we can add/rename properties to an object like so

```
const person = {
    id: 12345,
    name: 'Shahar',
    age: 27
};

const cloned = {...person};

const her = {...person, name: 'Hadas'};

const him = {...person, car: 'Volvo'};

console.log(person, cloned, her, him);
{ id: 12345, name: 'Shahar', age: 27 }
{ id: 12345, name: 'Hadas', age: 27 }
{ id: 12345, name: 'Shahar', age: 27 }
{ id: 12345, name: 'Shahar', age: 27, car: 'Volvo' }
```

Destructuring

The ability to extract values from array or object.

When destructuring from array – use [ ] syntax.

When destructuring from object – use { } syntax.

```
const numbers = [1, 2, 3, 4, 5];

let [x, y] = numbers;
// let x = numbers[0],
//     y = numbers[1];

const [first, , , fourth] = numbers;

// Destructuring safely, even with defaults
const list = [17, 42];
const [a = -1, b = -1, c = -1, d] = list;
console.log(a, b, c, d);

const justAnOrdinaryLongNameObject = {
  firstName: 'Shahar',
  ln: 'Grauman',
  luckyNumber: 17
};

const { firstName:fn, ln: lastName } = justAnOrdinaryLongNameObject;
// let fn = justAnOrdinaryLongNameObject.firstName,
//     lastName = justAnOrdinaryLongNameObject.ln;

console.log(lastName);
```

Destructuring with function arguments.

When destructuring from array – use [ ] syntax.

When destructuring from object – use { } syntax.

```
function f1([myName, myVal]) {
  console.log(myName, myVal);
}

function f2({ myName: n, myVal: v }) {
  console.log(n, v);
}

function f3({ myName, myVal }) {
  console.log(myName, myVal);
}

f1(['bar', 42]);
f2({ myName: 'foo', myVal: 7 });
f2({ myVal: 7, myName: 'foo' });
f3({ myName: 'bar', myVal: 42 });
```

Ex (ES6 4):

1- You have object with id, name, and another object with phone, address. Merge them to 1 object with id, name, phone and address.

2- Move the id property to be last in this object

Ex (ES6 5):

Make this work by completing the removeProp function using only destructuring and spreading

```
let remove = removeProp('id');

const person1 = {
    id:454,
    name:'Sha',
    age: 45,
    phone: '0547410147'
};

const student1 = {
    course: 'Angular',
    grade: 77,
    name:'Sha',
    id:654,
    age: 19,
    phone: '0506532650'
}

let p = remove(person1);
//{ name: 'Sha', age: 45, phone: '0547410147' }
let s = remove(student1);
// { course: 'Angular',  grade: 77, name: 'Sha',  age: 19, phone:
'0506532650' }
remove = removeProp('name');
p = remove(person1);
//{ id: 454, age: 45, phone: '0547410147' }
s = remove(student1);
// { course: 'Angular',  grade: 77, id: 654,  age: 19, phone: '0506532650'
}
```

New Object Methods

We already met ***Object.setPrototypeOf*** which is ES6 addition.

Object.assign merges properties from source(s) into the target.
Object.assign won't merge inherited properties.

```
const source = { num: 123 };
Object.assign(source, { bool: true });
console.log(source);//{ num: 123, bool: true }
```

```
const source = { num: 123 };
Object.assign(source,
              { bool: true },
              { num: 456, bool: false, text: 'lala' });
console.log(source);// { num: 456, bool: false, text: 'lala' }
```

We can assign properties in constructor like this

```
class Point {
    constructor(x, y) {
        Object.assign(this, {x, y});
    }
}
```

Object.keys returns an array of own properties (non-inherited) as strings

```
Object.keys(source)
      .forEach(prop => console.log(prop, source[prop]));
// num 456
// bool false
// text lala
```

Notice that only enumerated properties are considered

Object.getOwnPropertyNames behaves similar to *Object.keys*, except in considers non-enumerable properties as well:

```
const arr = ['a', 'b', 'c'];
console.log(Object.keys(arr).join(','));
// 0,1,2

console.log(Object.getOwnPropertyNames(arr).join(','));
// 0,1,2,length
```

Ex (Es6 6):

Make this work

```
const person = {
    name: 'Shahar',
    age: 30,
    phones: [1,2,3],
    address: {
        city: 'Haifa',
        street: 5
    }
}

console.log(`object props count: ${Object.propsCount(person)}`);
// object props count: 4
```

Ex (OO 8):

Create a function for copy properties from one JSON object to the other, by flattening the other object

You have this object

```
const otherPerson = {
    fullName: {
        firstName: 'Shahar',
        lastName: 'Grauman'
    },
    age: 80,
    email: 'info@grauman.co.il'
}
```

And you should copy its values to the corresponding properties of this object

```
const person = {
    firstName: '',.
    lastName: '',
    age: 0,
    email: ''
}
```

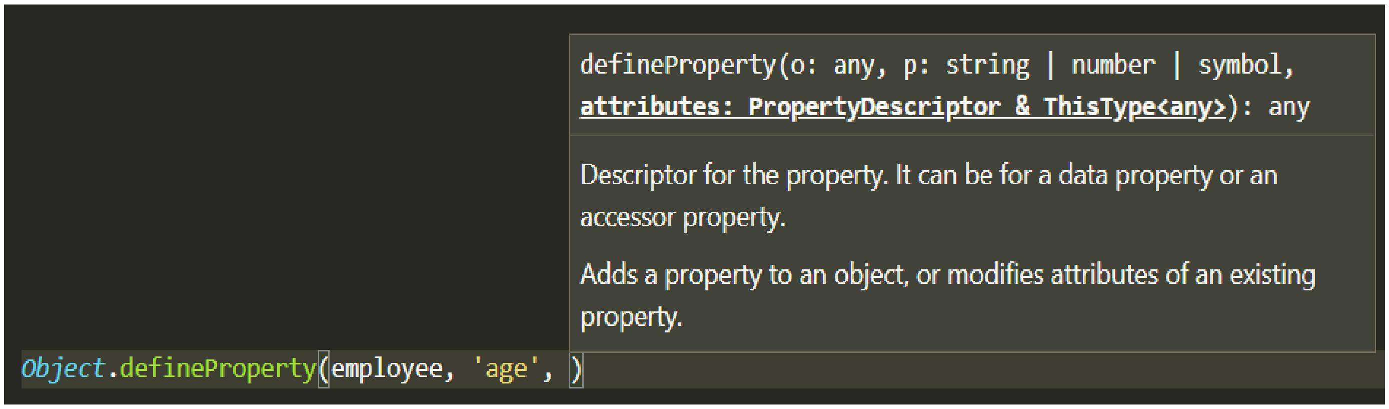
Object.getOwnPropertyDescriptor returns an object represents the property attributes:

For instance, let’s examine the employee name property

```
console.log(Object.getOwnPropertyDescriptor(employee, 'name'));
// { value: 'Shahar',
//   writable: true,
//   enumerable: true,
//   configurable: true }
```

Object.defineProperty is a way to define properties with 5 characteristics:

- name
- value
- enumerable
- writable
- configurable



```
defineProperty(o: any, p: string | number | symbol,
attributes: PropertyDescriptor & ThisType<any>): any
```

Descriptor for the property. It can be for a data property or an accessor property.

Adds a property to an object, or modifies attributes of an existing property.

```
Object.defineProperty(employee, 'age', )
```

So here I'm defining non-enumerable member

```
Object.defineProperty(employee, '_age', {
    value: 31,
    enumerable: false,
    writable: true,
    configurable: false
});

console.log(employee);
// { name: 'Shahar',
//   salary: 3500,
//   phones: [ '0548962002', '0521010447' ],
//   vat: [Getter],
//   phone: [Setter] }

for(let prop in employee){
    console.log(prop);
}
// name
// salary
// phones
// vat
// phone
```

<u>Ex (ES6 7):</u>

Using defineProperty – make this work:

```
console.log(employee.name);
// Shahar
employee.name = 'Yusuf';
console.log(employee.name);
// Shahar
console.log(employee.age);
// 31
employee.age = 32;
console.log(employee.age);
// 32
```

New Collections – Map & Set

Map is a key:value pairs container

```
const citizens = new Map();

citizens.set(128005852, 'Shachar');
citizens.set(205009006, 'Shulamit');
citizens.set(257410000, 'Smirf');
citizens.set(690055222, 'Abed');

for (const [key, value] of citizens /* .entries() */) {
    console.log(key, value);
}

if(citizens.has(128005852)){
    console.log(`Citizens has ${128005852} ->
                               ${citizens.get(128005852)}`)
}

console.log(`Citizens names: ${[...citizens.values()]}`);

128005852 'Shachar'
205009006 'Shulamit'
257410000 'Smirf'
690055222 'Abed'
Citizens has 128005852 -> Shachar
Those are citizens names: Shachar,Shulamit,Smirf,Abed
```

Set is a collection which eliminates duplicates

```
const set = new Set();

set
  .add("Shachar")
  .add(42)
  .add("Lala")
  .add(42)
  .add("Shachar");

console.log(set.size);
console.log(set.has(42));
console.log([...set]);

3
true
[ 'Shachar', 42, 'Lala' ]
```

Ex (ES6 8):

You have a very long text string.

Print how many each letter appears.

```
class Statistics {
   //...
}

const myStats = new Statistics();
myStats.analyze('Very long text...');

const stats = myStats.summary;
//stats = [['e', 2], ['g', 1], ['l', 1], ['n', 1], ['o', 1], ['r', 1],
['t', 2], ['v', 1], ['x', 1], ['y', 1]]
```

Asynchronous JavaScript

For 15 years JavaScript main purpose was interacting with HTML.

For example, when a user clicks a button, it triggers something like showing a popup with a message or send a request to a remote server and so on.

The mechanism is callbacks.

Callbacks are functions we pass to the triggering source to be later invoked.

Think of the button: We ‘tell’ the button what we want to do when it is being clicked. We do that by passing it our function.

Timer gets a callback and invoke it after a period of a given time:

```
function showMessage(){
    console.log('Hi there from callback');
}

setTimeout(showMessage, 1000);
// Will print 'Hi there from callback' after 1 second

setTimeout(() => {
   console.log('Ahalan')
}, 1000);
// Will print 'Ahalan' after 1 second
```

For many years the interaction with HTTP requests and responses was very tedious using *XMLHttpRequest*, so many libraries, such as *jQuery*, helped us with smooth API.

```
function callback() {
     var asJson = JSON.parse(this.responseText);
    var body = document.getElementById('body');
     //Interact with body...
}

var req = new XMLHttpRequest();
req.addEventListener('load', callback);
req.open('GET', 'some url');
req.send();
```

jQuery syntax:

```
$.get('Some url', function(data) {
    $('body')
        .append(`Name: ${data.name}`)
        .append(`Time: ${data.time}`);
}, 'json');
```

So JavaScript added wrapper around XMLHttpRequest – ***fetch***.
fetch uses ***Promises*** which are objects responsible for holding our callbacks and notifying us upon completion of our request, or failure. Promise gets an executor which is our callback. We *then* 'wait' for it to complete with our callback:

```
const promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve('Boo!');
    }, 1000);
});

promise.then(function (value) {
    console.log(value);// 'Boo!'
});
console.dir(promise);// Promise { <pending> }
```

We create a promise which is immediately returned! Notice it’s in *pending* mode.

Then we use ***then*** to get the result or error.

```
const promise = new Promise(function (resolve, reject) {
    // doing our work, probably async, then…

    if (/* everything’s good */) {
        resolve(‘OK!’);
    }
    else {
        reject(Error(‘Oh Oh!’));
    }
});

promise.then(function (result) {
    console.log(result); // ‘OK’
}, function (err) {
    console.log(err); // Error: ‘Oh Oh!’
});
```

We can chain promises, and at the end, on any failure, we can chain a catch promise

```
get('some remote json')//Returns a Promise
.then(function (response) {
    return JSON.parse(response);
}).then(function (data) {
    console.log('Here is JSON', data);
}).catch(function(error) {
    console.log('Failed!', error);
});
```

You see, using callbacks leads to something like this

```
getData(function(data) {
  manipulateData(data, function(manipulated) {
    finalize(manipulated, function(result) {
      console.log('Result: ' + result);
    }, failedCallback);
  }, failedCallback);
}, failedCallback);
```

We can refactor this to be read more easily

```
try {
  const data = getData();
  const manipulated = manipulateData(data);
  const result = finalize(manipulated);
  console.log('Result: ' + result);
} catch(err) {
  failedCallback(err);
}
```

So, if those functions return promises, we can chain those functions

```
getData().then(function(data) {
  return manipulateData(data);
})
.then(function(manipulated) {
  return finalize(manipulated);
})
.then(function(result) {
  console.log('Result: ' + result);
})
.catch(failedCallback);
```

Or even shorten this by using lambdas

```
getData().then(data => manipulateData(data))
   .then(manipulated => finalize(manipulated))
   .then(result => console.log('Result: ' + result))
   .catch(faileCallback);
```

Ex (Async 1):

You have a class which contains data to be used with *setTimeout* timer

```
class RandomizePeriod {
    constructor(){
        const periods = new Map();
        periods.set(1000, { payload: 'Whats Up? (after 1sec)' });
        periods.set(2000, { err: 'No Soup For You! (after 2sec)'});
        periods.set(3000, { payload: 'Ahalan (after 3sec)' });
        periods.set(4000, { payload: 'Yoffi Toffi (after 4sec)' });
        this.periods = periods;
    }

    get period() {

    }
}
```

When you call period() it should return you 1 random entry from periods.

Example [3000, { payload: 'Ahalan' }] or [2000, { err: 'No Soup For You!' }].

You have this function which creates and returns promise like so:

If the data contains error, just reject it with the error.

If the data doesn't contain error, resolve it with data.payload after a period stated in timeout (using *setTimeout*)

```
function createPromise([timeout, data]) {
    return new Promise((resolve, reject) => {
        if(!data.err){

        }else{

        }
    });
}
```

Now, use this function to create 3 chained promises like so

```
createPromise(periods.period)
.then(/*
    log what the promise returned
    return a new promise with random period
    */
}).then(/*
    log what the promise returned
    return a new promise with random period
    */
}).then(/*
    log what the promise returned
    return a new promise with random period
    */
}).catch(/*
    log what the error is
*/);
```

Run it few times, you should print results like this:

```
First promise returned Yoffi Toffi (after 4sec)
Second promise returned Whats Up? (after 1sec)
Third promise returned Yoffi Toffi (after 4sec)
```

Or

```
First promise returned Ahalan (after 3sec)
Second promise returned Ahalan (after 3sec)
Rejected with Error: No Soup For You! (after 2sec)
```

ES6 brought us with ***async/await*** syntax to ease the chaining of promises.

Remember this?

```
getData().then(data => manipulateData(data))
    .then(manipulated => finalize(manipulated))
    .then(result => console.log('Result: ' + result))
    .catch(faileCallback);
```

We can refactor the above code to use *async/await* like so

```
async () => {
  try {
    const data = await getData();
    const manipulated = await manipulateData(data);
    const result = await finalize(manipulated);
    console.log('Result: ' + result);
  } catch(err) {
    failedCallback(err);
  }
}
```

Whenever we deal with promises we can *await* for it to be resolved.

Notice that the surrounding function is marked as *async*.

The code suddenly seems like a regular procedural one… but it's not 😊

REST API

Bottom line: ***Client*** make request to a ***Server***.

Most of the time those are regular requests such as *HTML*, *CSS* and *JavaScript* resources.

Since ***AJAX*** came out back then in early 2000, we no longer need to request the entire page all over again, but a chunk of data.

AJAX made asynchronous programming a must and as such, many frameworks arouse (jQuery, Backbone.js, Knockout.js, Angular, React and many more).

Now we call it ***SPA*** – Single Page Applications.

Those frameworks made client developing structured and concise.

At their very heart the ability to structure client UI effectively and fetching data asynchronously.

So, the client sees a page and clicks a button. A background request is made to the server. The client still sees the page, probably with a loading message until the data is fetched. Then, we continue by showing her the data or something else.

SPA changed the way we program to the web and consume it as clients.

A server exposes *REST API* for (mostly) plain JSON data.

Those APIs are for get/set data using HTTP Methods:

GET, POST, PUT, DELETE

POST – **C**reate new record on server

Get – **R**ead data from server

PUT – **U**pdate existing record on server

DELETE – **D**elete existing record on server

And in short – ***CRUD***.

Examples:

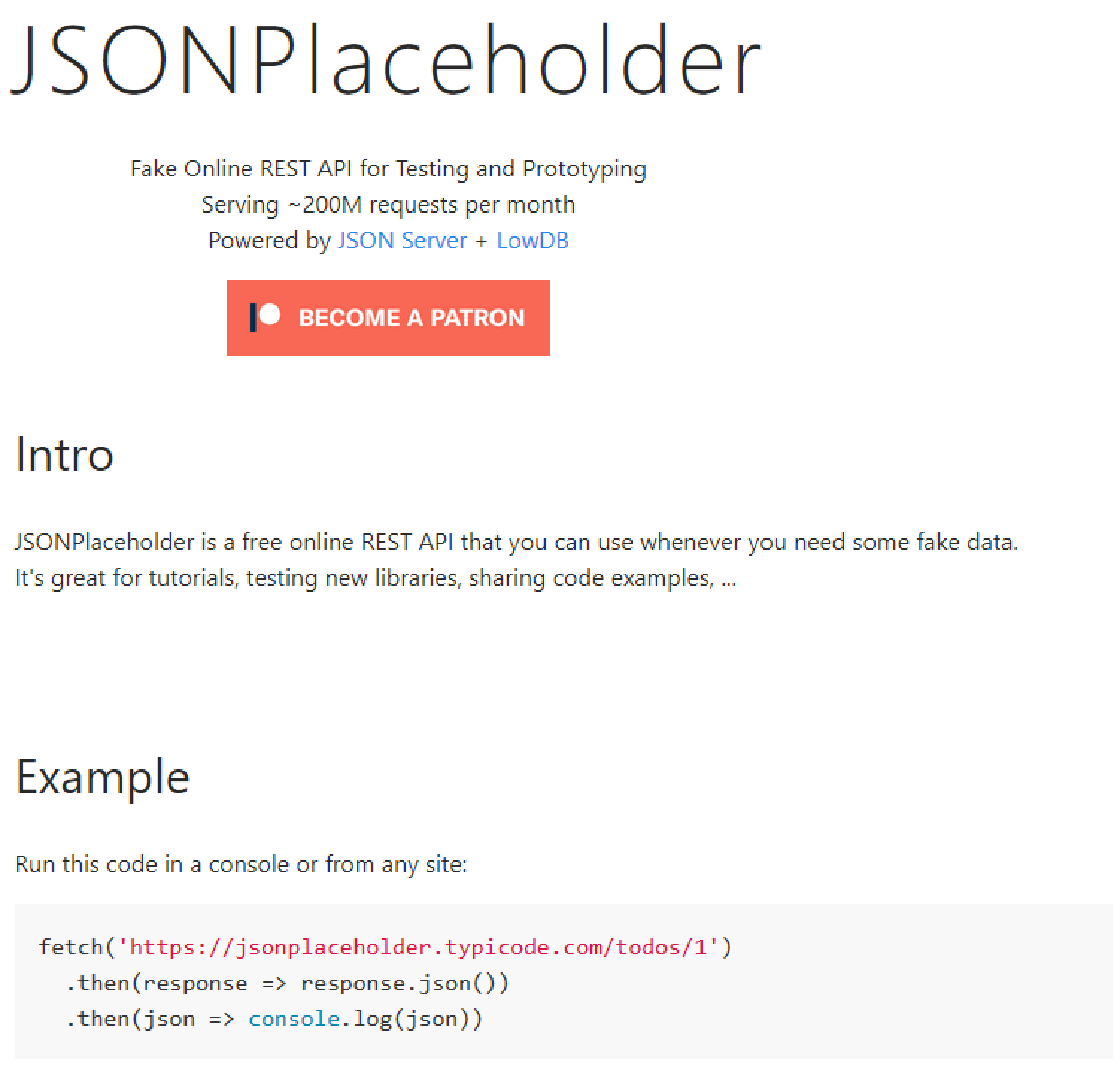
1- GET request to list all students
 http://mysite.com/api/students
2- GET request to get a specific student with id 2332
 http://mysite.com/api/student/2332
3- GET request to get the courses a specific student with id 2332 learn
 http://mysite.com/api/student/2332/courses
4- GET request to get a specific course with id s3a11 a specific student with id 2332 learn
 http://mysite.com/api/student/2332/courses/s3a11
5- POST request to create new student
 http://mysite.com/api/student

And inside the request's body we send a JSON student object data
{ "name": "Shahar", "course": "JS",...}

6- Delete some resource http://mysite.com/api/student/2332

Take for example JSONPlaceholder – a fake REST API

http://jsonplaceholder.typicode.com/



JSONPlaceholder

Fake Online REST API for Testing and Prototyping
Serving ~200M requests per month
Powered by JSON Server + LowDB

BECOME A PATRON

Intro

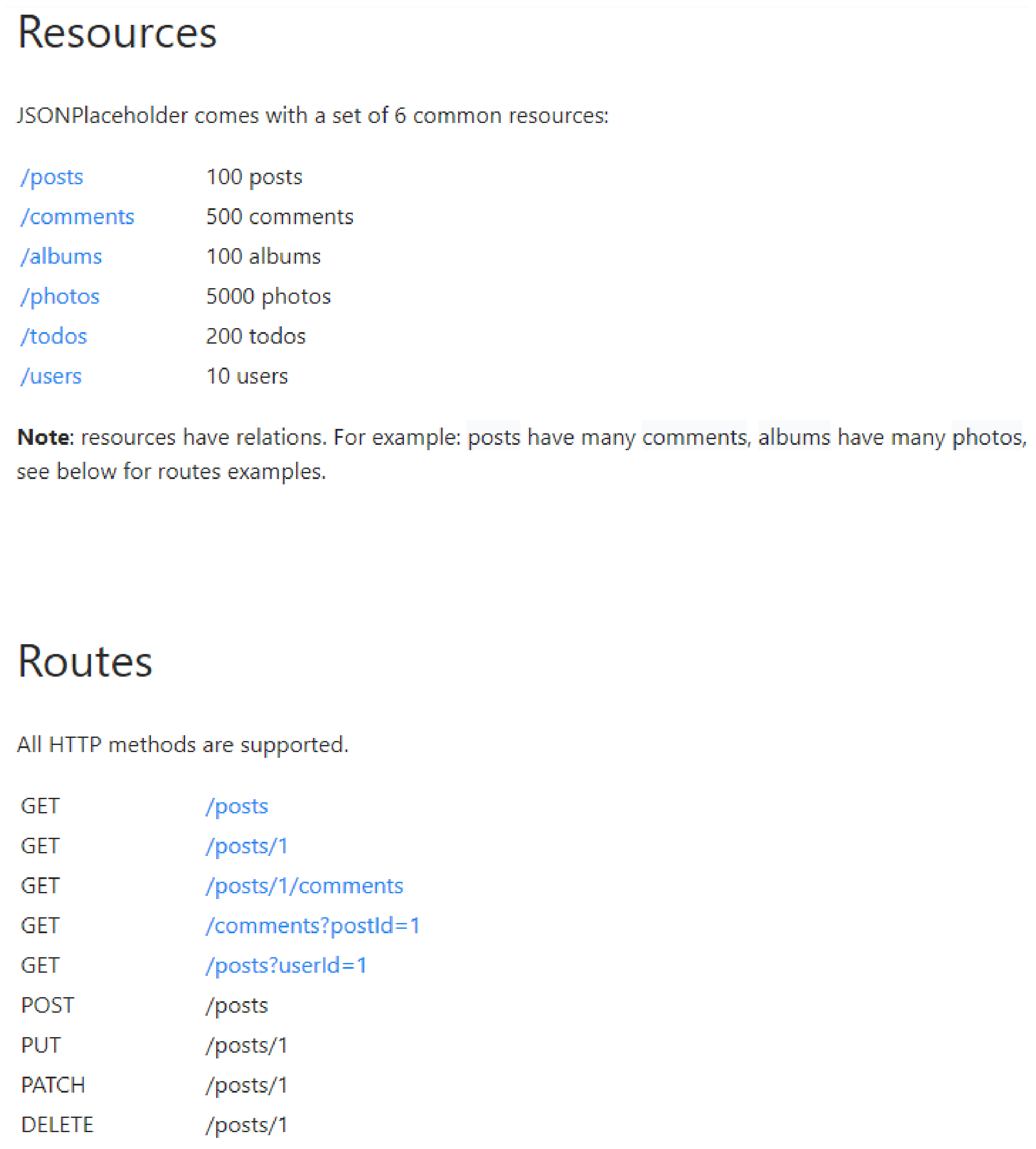
JSONPlaceholder is a free online REST API that you can use whenever you need some fake data.
It's great for tutorials, testing new libraries, sharing code examples, ...

Example

Run this code in a console or from any site:

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(json => console.log(json))
```

As you can see, fetch is used to make requests to this REST API.
fetch uses promises, so we use then to get the results.



Resources

JSONPlaceholder comes with a set of 6 common resources:

/posts	100 posts
/comments	500 comments
/albums	100 albums
/photos	5000 photos
/todos	200 todos
/users	10 users

Note: resources have relations. For example: posts have many comments, albums have many photos, see below for routes examples.

Routes

All HTTP methods are supported.

GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
GET	/posts?userId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Explore JSONPlaceholder API by making CRUD requests

Shahar Grauman
www.grauman.co.il | 054-8002004 | info@grauman.co.il

EX (REST 1):

Extract data from Github public repositories

https://api.github.com/repositories

Let's grab some repositories from github and print for each one (For example):

```
{
 id: 210, //repository id
 full_name: 'polanski/fred',
 avatar_url: 'https://avatars0.githubusercontent.com/u/75?v=4',
 html_url: 'https://github.com/fred'
}
```

Use this function to show users images on the page

```
function showAvatar({name, url}) {
  let img = document.createElement(`img`);
  img.alt = `${name} avatar`;
  img.src = url;
  document.body.append(img);
}
```