

INTRO

As soon as you start looking for your first job in IT, you should get familiar with the list of most popular Javascript questions you can be asked on the interview for a frontend developer.

In this e-book, you will get the most common and fundamental questions which answer to make your first interviews as easy and smooth as possible.

Get familiar with the topics covered here, and you will feel less stressed during your first interviews for a frontend developer.

List of questions:

1. What is closure?
2. What is hoisting?
3. What is the difference between var, let, and const?
4. What are self invoking functions? (IIFE)
5. What is spread operator (...)?
6. What is prototypal inheritance?
7. What is functional programming?
8. What is Object Oriented Programming (OOP)?
9. What is this?
10. What is the difference between == and ===?
11. What is scope?
12. What is asynchronous programming and what is the advantage of using it in Javascript?
13. How would you improve performance of your Javascript code?
14. What are bind(), call() and apply() methods?
15. What is the difference between Local storage and Session storage?
16. What is "use strict";?
17. What are useful features of ES6 and newer Javascript versions?
18. How can you check if object is an array or not?
19. Explain what is callback hell in Javascript and how to avoid it?
20. What is .map() method?



01. WHAT IS CLOSURE?

Closure definition says that inner function has access to its variables and variables of its outer function.

```
function outer() {  
  var name = 'Mark';  
  function inner() {  
    console.log(name);  
  }  
  inner();  
}  
outer();  
// 'Mark'
```

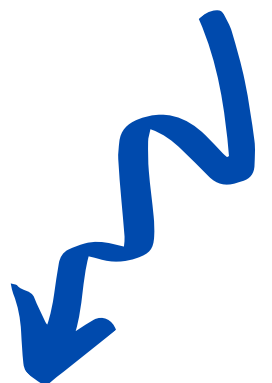
In this example, you can see that **inner** function have access to the parent function variable **name**.

If you will call the **outer()** function, it console.log the value of the name variable 'Mark'.

However, it can access outer function arguments object, but very often, inner function has its own arguments object, which overshadows the outer function arguments object. You can see it, while the closure is used with arrow function, for example:

```
function outer(a, b) {  
  const inner = (a, b) => console.log(a, b);  
  inner(1, 2);  
}  
outer('Alice', 'Mark');  
// returns 1, 2
```

The reason why you should use closures is to create functions that can return other functions.



02. WHAT IS HOISTING?

Hoisting is the mechanism that lifts all declared variables and functions as well, to the top of their local scope or at the top of the global scope if they are placed in the global scope.

In Javascript, it's possible to declare a variable after it's used. Hoisting is used to avoid undefined errors because otherwise, it could happen that code with variable or function is executed, but it's not defined.

To make sure your code won't have any issues with undefined values, remember to declare your variables first.

Here is an example to show you how it works.



```
name = 'Mark';  
console.log(name);  
var name;  
// returns 'Mark'
```

Here's what you see



```
var name;  
name = 'Mark';  
console.log(name);  
// returns 'Mark'
```

Here's what happens in background

Hoisting with let and const

While you will create a variable definition using **var**, it will be initialized in every line as undefined.

It's a little bit different with **let** and **const**. The variable is not initialized until the line where the initialization really happens. So, it doesn't call any undefined in the meantime.

Also, it's important to remember that while you declare **const**, it's necessary to initialize it at the same time because it won't be possible to change it.



03. WHAT IS THE DIFFERENCE BETWEEN VAR, LET, AND CONST?

Var and **let** are used to declare variables, and **const** is used to declare constants.

The difference between var and let is **scope**. Var declarations have a global or local scope.

Let variables scope is a block of code (the part of code between curly braces).

Let variable is only available to use inside the block of code where it's declared.

Var and let can be changed, and const needs to be declared and assigned to any value at the beginning because it can't be changed.

```
// global scope
var number = 10;

function calc() {
  var divider = 5;

  console.log(number)
  // returns 10
  console.log(divider)
  // returns undefined
```

Here an example with var

```
// block scope
if ('name'.length > 10) {
  let nameLength = true;
  console.log(nameLength);
  // returns true
}
console.log(nameLength);
// returns undefined
```

Here an example with let



04. WHAT ARE SELF INVOKING FUNCTIONS? (IIFE)

Self invoking function, or differently called **IIFE** (Immediately Invoked Function Expressions) is a function in Javascript which is invoked immediately after it's declared.

Self invoking functions are nameless (anonymous).

It's possible to pass parameters to self invoking functions.

```
(function(name) {  
  console.log('Hello' + name);  
})('Mark');  
// returns Hello Mark
```

If there are any variables or methods inside the self invoking function, they cannot be accessed from outside of the functions, unless they are assigned to a window object.

But, to access window object in the IIFE function, it has to be passed as a parameter.

```
(function(window) {  
  var num = 10;  
  return num*2;  
  
  window.num = num  
})(window);
```

To find out how to pass variables and methods from the IIFE see the code on the left.



05. WHAT IS SPREAD OPERATOR (...)?

Spread operator (...) is a new feature of Javascript, appeared in ES6 and it takes array or object or other iterable (like string) and expands it into a set of elements, mostly needed in places like function calls, where more arguments are expected.

Let's take a look at the code for better understanding:

```
// Array
var myNum = [1, 3, 5, 7];
var yourNum = [2, 4, 6, 8];

var ourNum = [...myNum, ...yourNum];
// returns [1, 3, 5, 7, 2, 4, 6, 8];

// String
var greeting = 'Hello';
var greetingArr = [...greeting];
// returns ['H', 'e', 'l', 'l', 'o'];

// Object
var person = {
  name: 'Peter',
  age: 24
};
var newPerson = {
  ...person,
  lastName: 'Smith'
};
// returns {name: 'Peter', age: 24, lastName: 'Smith'}
```

In the image above, you can see how the spread operator behaves with an array, string, and object. In every case, it spreads the element on smaller pieces and allows to perform operations on it.

Easy, right?



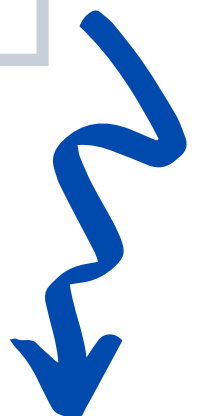
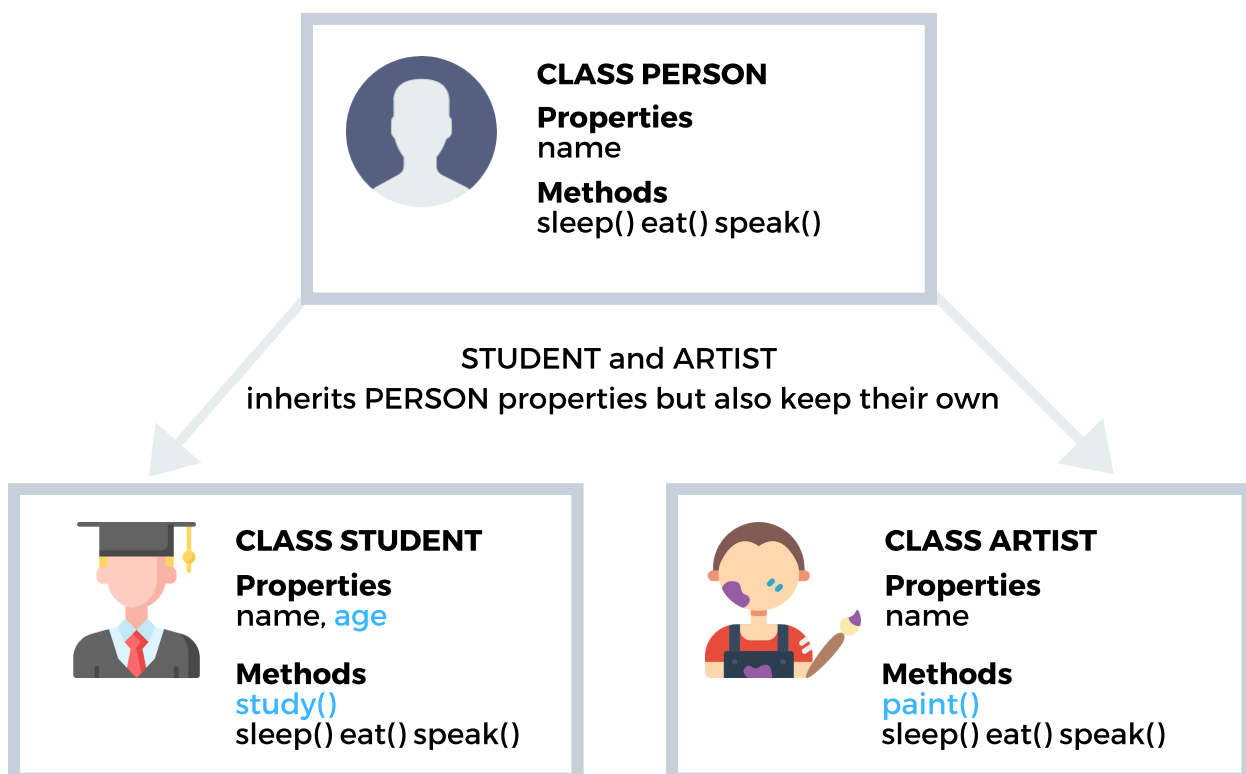
06. WHAT IS PROTOTYPAL INHERITANCE?

In Javascript, objects can inherit methods and prototypes from other objects.

Since ES6 came with classes, you can understand prototypal inheritance as a situation when the prototype of the new object is created based on its constructor **prototype** property.

Every object has a connection to its prototype, and each prototype has a connection to another object, which is its prototype, and its chains until null.

Let's try to see the logic in the prototype chain on the image example.



07. WHAT IS FUNCTIONAL PROGRAMMING?

Functional programming rule says that the output of the function depends only on the input.

So every time we pass x argument to the function, the result should be the same, and there are no side effects.

This kind of function is called pure function in JS.

Let's take a look at the example of pure and impure functions:

```
// Impure function
var greeting = 'Hello';
function sayHi(name) {
  return greeting + ' ' + name;
}

// Pure function
function sayHi(name) {
  return 'Hello ' + name;
}
```

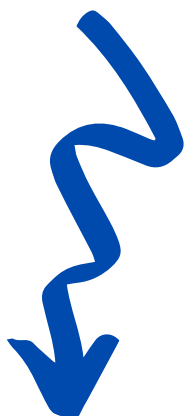
The first function is not a pure function because it uses a global scope variable greeting.

The second one is a pure function because the result depends only on the name parameter.

The main assumption of functional programming is that we are operating mainly on immutable values, not on state or mutable data.

Another important thing to mention in the context of functional programming is that functions are the first-class citizens here.

Functions are also considered values, which makes it easy to pass them to another function.



08. WHAT IS OBJECT ORIENTED PROGRAMMING (OOP)?

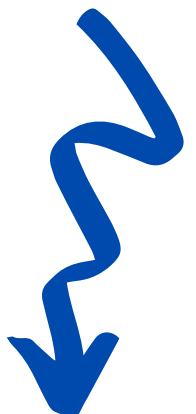
Object Oriented Programming is a paradigm in Javascript where objects take the main role.

They are treated like building blocks. Application is a kind of collection of objects which communicate with each other.

It's worth mentioning that our objects often inherit properties and methods from other objects, so it makes our code reusable.

Another advantage of OOP is that code is very easy to understand.

Objects in Javascript may have properties and methods which can be inherited by the child objects.



09. WHAT IS THIS?

this keyword refers to the object which called the current piece of code.

The value of **this** depends on where a function is placed and how it's called.

In other words, you may say **this** refers to the scope of a function that was called.

In JS, the global object/scope and value of this keyword depends on the mode, the code is called.

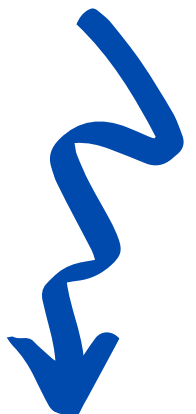
In strict mode, the value of this is undefined, but in case of strict mode, it's a window object.

Here is some code example:

```
// Strict mode
function foo() {
  'use strict';
  console.log(this);
}

// No strict mode
function boo() {
  console.log(this);
}

foo(); // undefined
boo(); // window
```




10. WHAT IS THE DIFFERENCE BETWEEN == AND ===?

Comparing values in Javascript can be done with type coercion or without it.

Variables in Javascript may have different types like number, string, object, undefined, etc.

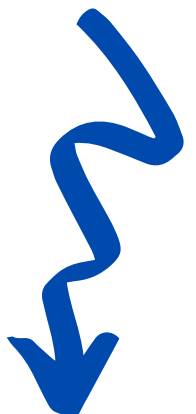
Let's take a look at some code:



```
"2" == 2 // true  
"2" === 2 // false
```

If you compare string 2 with number 2 and use == for comparison (like in the first example), the code will ignore the type of values and will return true.

If you use === (like in the second example), it will return false, as string 2 will be different than number 2.



11. WHAT IS SCOPE?

The **scope** is connected to functions (like context to objects) and blocks of code. It determines the visibility of the variables.

Javascript has three scopes, global, functional (called local as well), and block.

Variables declared out of the function or block of code belongs to the global scope and the ones declared in the function, then they belong to local scope.

Variables defined in the block of code belong to block scope.

It means that variables from inside the function or block of code are not accessible out of it.

But global variables are available everywhere in the global scope.

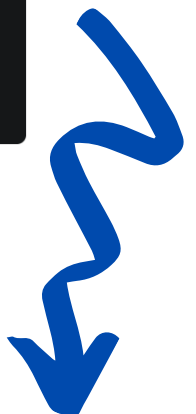
Each function creates a new scope. Let's take a look at some code:

```
// Local scope - variable from the function is not available out of it
function foo() {
  var number = 10;
}
console.log(number); // undefined

// Block scope - variable from the block of code is not available out of it
function foo(a) {
  if (a > 10) {
    let b = 5;
  }
  console.log(b); // undefined
}

// Global scope - global variable can be used anywhere
var number = 10;

function foo() {
  console.log(number); // 10
}
```



12. WHAT IS ASYNCHRONOUS PROGRAMMING AND WHAT IS THE ADVANTAGE OF USING IT IN JAVASCRIPT?

Asynchronous programming means that while Javascript code is executed in some sequence, it can meet a blocking operation.

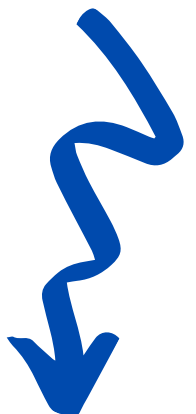
In synchronous programming, the rest of the code needs to wait until the blocking operation finishes, but in asynchronous, the request is started and doesn't have to wait until it finishes.

The rest of the code keeps running, and the result of the request is returned while it is ready.

The biggest advantage of asynchronous programming in Javascript is that it helps developers to improve the performance of code, and it prevents long time loading.

User experience is much better if we can load all the UI and the API calls happen just in the background.

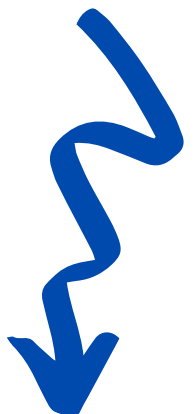
To implement asynchronous functions in Javascript, you can use **callbacks**, **Promises**, or **async/await**.



13. HOW WOULD YOU IMPROVE PERFORMANCE OF YOUR JAVASCRIPT CODE?

Performance of the application can be improved in many ways, let's take a look at some examples:

- avoid no needed, especially nested loops
- use asynchronous programming concept
- try to create reusable code and keep it simple
- cache objects
- minimize HTTP requests
- limit dependencies
- use local scope
- avoid using too many global variables



14. WHAT ARE BIND(), CALL() AND APPLY() METHODS?

All these methods are connected to **this** keyword.

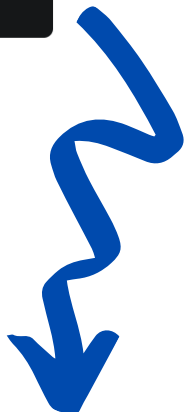
They allow us to change or assign this keyword a different value. **call()** and **apply()** are almost the same, they execute the function immediately with the context passed as the first argument, the only difference is that in **apply()** the arguments have to be passed as an array.

```
function foo(a, b) {  
  console.log(this);  
}  
  
foo.call(this, 1, 2);  
foo.apply(this, [1, 2]);
```

bind() method is very similar to **call()** and **apply()**, it also allows us to call a function with a different context, but it doesn't call the function immediately. It allows us to pass the context and call the function another time.

Simply, it creates the function copy with the new context, which is the **this** we passed.

```
function foo(a, b) {  
  console.log(this);  
}  
  
const newFoo = foo.bind(this, 1, 2);  
newFoo();
```



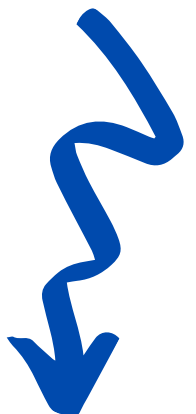
15. WHAT IS THE DIFFERENCE BETWEEN LOCAL STORAGE AND SESSION STORAGE?

Session and local storage are very similar; the main difference between them is that data saved in the session storage expire after the session ends.

The session in the browser lasts until the browser is open.
The session storage data are available per tab, not in the browser.

Data saved in local storage doesn't have an expiration date; it needs to be deleted.

Data saved in local storage is available for the current and next visits on the website.



16. WHAT IS "USE STRICT";?

Use strict is a directive, which came with ES5, and it says that the code should be executed in strict mode.

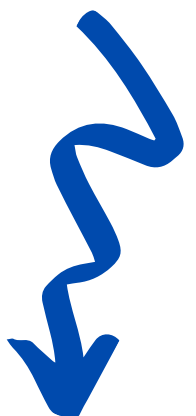
In strict mode, some of the functionality doesn't work, like **with** or **arguments.callee**.

Strict mode requires developers to write better quality code because some of the "quite" errors become visible in this mode, for example, the wrong usage of **delete** method: ,

```
function foo() {  
  'use strict';  
  var x = {};  
  delete x;  
}  
  
foo(); // SyntaxError
```

To start strict mode, we need to put use the strict expression at the top of the script or the function.

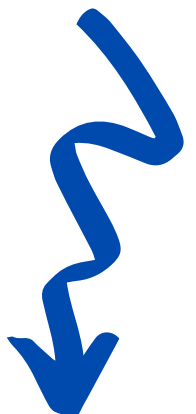
ES modules have forced strict mode inside.



17. WHAT ARE USEFUL FEATURES OF ES6 AND NEWER JAVASCRIPT VERSIONS?

- block scope for let and const
- new methods like includes()
- arrow functions (=>)
- Promises
- spread operators (...)
- object methods like values() or entries()
- async/await

These are some of the useful functions which came with Javascript ES6 and newer, of course, you can place here some of the features which are the most useful for you.



18. HOW CAN YOU CHECK IF OBJECT IS AN ARRAY OR NOT?

You can use `isArray()` method which returns true if the variable is an array and false if it's not.

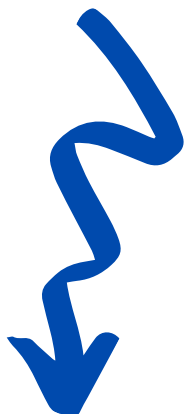
But it's also possible to create a custom function which can check it. Let's take a look at code examples:

```
// After ES5
var myArray = [1, 2, 3, 4, 5];
myArray.isArray(); // true

// Before ES5
var myArray = [1, 2, 3, 4, 5];

function isArray(value) {
    return !!value && value.constructor === Array;
}

isArray(myArray); // true
```



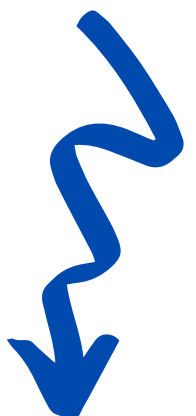
19. EXPLAIN WHAT IS CALLBACK HELL IN JAVASCRIPT AND HOW TO AVOID IT?

Before Promises and async/await methods came into Javascript, most of the asynchronous code was done using callbacks.

Sometimes when developers used it too much, creating lots of nested functions with **callback hell** happened.

It's also known as the Pyramid of Doom. It's an anti-pattern.

To avoid callback hell, we can use Promises, or async/await instead of nesting multiple callbacks.



20. WHAT IS .MAP() METHOD?

.map() is a new array method which creates another array by iterating through the original array and calling the function on every element of the array.

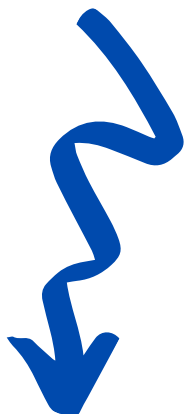
The function modifies the item and saves it as a new array.

Let's take a look at a code example:

```
var numbers = [2, 4, 6, 8, 10];  
var doubleNumbers = numbers.map((item) => {item * 2});  
  
console.log(doubleNumbers); // [4, 8, 12, 16, 20];
```

Here you can see, it iterates through numbers array and doubles every element.

As a result, you have a new array.



SUMMARY

You just went through the most common questions at Javascript interviews.

It'll help you to understand essential Javascript topics and will be an excellent base to learn more.

Good luck!

Congratulations!

