JavaScript Interview 2022

Questions with Answers

Ianis Triandafilov

Contents

This book is still in BETA. Please send your comments, ideas and mistakes you find to me: janis.sci@gmail.com.

Change History

0.1

- Initial version

0.2

- Added section 3 “Variables and scopes”
- Added section 4 “Operators”

0.3

- Added section 5 “Functions”
- Started section 6 “Elements of FP”

0.4

- Added sections 7 and 8, and 9
- Exercise solutions moved to the end of the book
- Multiple typo fixes, improvements, and links added

0.5

- Added sections 10 (regular expressions), 11 (async), 12 (patterns), 13 (DOM), and 14 (error handling)
- How to read this book
- Multiple typo fixes and improvements

Attributions

The book cover is made by Andrii Bakunovskyi.

Vintage vector created by rawpixel.com - www.freepik.com

How to read this book

This book is meant to be a concise guide that will help anyone quickly prepare for a JavaScript interview.

The material is organized in chapters, each covering a particular JavaScript topic.

You can read this book in order, pick chapters that interest you, and read those.

Questions are either theoretical or practical (exercises).

Exercises are the core of this book. But in every chapter, before you reach any exercise, you should read through theoretical questions first. It will ensure you are prepared to start practicing.

Solutions for every exercise are provided in the last part of the book. At the end of every exercise, there is a link to the solution for easy navigation. I highly recommend trying to solve the exercise on your own before jumping into the solution.

General Questions

What is JavaScript

JavaScript is a popular programming language (PL) commonly used in web pages but also in server-side and mobile development, in areas like Internet of things and to a lesser degree in artificial intelligence.

JavaScript is a multi-paradigm high-level language with dynamic typing, a prototype-based object model, and first-class functions.

Let's define all those things.

- **Multi-paradigm** means that OOP and functional programming concepts can be both easily used (see the next question) as opposed to single-paradigm languages like Smalltalk (OOP) or Haskel (functional programming).
- **Dynamic typing** means that the type checking is performed at run-time (as opposed to compile-time for languages like Java - which is statically typed).
- **Prototype-based** object model is when objects inherit properties from other objects (called prototypes).
- A PL has **first-class functions** means that functions are treated as any other variables (they can be passed to other functions and returned from other functions).

We will explore some of those characteristics in more detail in the following questions.

What paradigm is JavaScript?

JavaScript is usually referred to as a "multi-paradigm" language.

That means that language features allow it to be used efficiently as an object-oriented language, a functional language, and a plain old procedural/imperative language.

In that sense, JavaScript gives us freedom of choice. The flip side of this coin is that every developer and project needs to decide what paradigm they should use or if mixing programming styles is a good choice.

How is JavaScript related to Java?

In short, it's completely unrelated.

The name "JavaScript" is what marketing people at Netscape came up with at the last minute.

When Brendan Eich finished the first version of the language for Netscape Navigator, it was called LiveScript. The codename was 'Mocha'.

The idea was to appeal to Java developers and essentially present JavaScript as Java for the web.

In reality, there are just a few similarities between the languages. The syntax might be similar (we use `{` / `}` to indicate a block of code, etc.), as both languages inherited the C-like syntax. But other than that, those are entirely different languages.

Oracle through Sun still owns the "JavaScript" trademark. That is why the TC39 committee had to come up with a different name, "ECMAScript", which is not that catchy.

Learn more:

- Brendan Eich on Creating JavaScript in 10 Days, and What He'd Do Differently Today
- Brendan Eich: An Introduction to JavaScript (Video)

What is ECMAScript?

The ECMAScript is the official specification of the JavaScript language.

There is a technical committee TC39 comprised of several dozens of members from the big companies (Mozilla, Google, Samsung, etc.).

TC39 meets regularly to propose and discuss changes. Those changes go through several stages and end up in the specification.

Currently, a new version of ECMAScript specification is out every year. The latest release is ECMAScript 2022 (13th edition).

Why it's called "ECMAScript", and not "JavaScript"?

Because the "JavaScript" is a trademark of Oracle (through Sun).

ECMAScript stands for "European Computer Manufacturer's Association".

What major features were introduced as part of ECMAScript 20XX?

Here's a brief history of ECMAScript. We only cover the most prominent changes.

The first two editions of ECMAScript basically cemented the JavaScript as a standard (the difference between the two was mostly editorial).

ECMAScript 3 (2002) introduced regular expressions and try / catch blocks.

ECMAScript 4 was never published.

ECMAScript 5 (2009) added support for JSON, array manipulation functions, and strict mode.

ECMAScript 6 (2015) was a huge update which paved the way to the modern JS. It introduced modules (import / export syntax), classes, lexical scoping with `let` / `const` , promises, destructoring, Maps, Sets, and more.

ECMAScript 7 (2016) added exponential operator and Array.prototype.includes.

ECMAScript 8 (2017) came out with `async / await` , `Array.prototype.includes` , `Object.values` , `Object.entries` , and `Object.getOwnPropertyDescriptors` .

ECMAScript 9 (2018) introduced AsyncIterator protocol, and object rest and spread properties.

ECMAScript 10 (2019) came with `flat` and `flatMap` , `Object.fromEntries` , `trimStart` , `trimEnd` , stability for Array.sort

ECMAScript 11 (2020) added `matchAll` on Strings, dynamic `import()` , `BigInt` data type, `Promise.allSettled` , globalThis, `import.meta`

Data Types

What data types does JavaScript have?

In JavaScript all values are either **primitives**, or **objects**.

There primitives are:

- numbers: `1` , `-3.14` , `0b101`
- strings: `'str'` , `"another string"`
- booleans: `true` and `false`
- symbols: `Symbol('my symbol')`
- `null` represents "emptyness", the absence of a value
- `undefined` is another way to represent an empty value (some people argue that having 2 null-like values is a design mistake).
- `BigInteger` s: `8003239254440991n` (introduced as part of ECMAScript 2020) - see a dedicated question.

Everything else is an object (yes, even arrays and functions).

- Objects: `{ x: 123 }`
- Arrays: `[1, 2, 3]` , `[null, true, 4]` (arrays can hold any kind of value)
- Functions: `function myFunc() { /* some code */ }`

Functions are no exception. This is a perfectly valid code:

```
const myFunc = () => {
  console.log("hey!");
};

myFunct.myKey = 123; // assign a property
console.log(myFunct.myKey) // => 123
```

What are the differences between primitives and objects?

1. All primitives are immutable. They can not be altered.

2. Primitives are passed by value: when you assign a primitive to a variable, or pass to a function, the content is copied.

Objects are passed by "pointers": passing an object to a function results in copying the pointer, but it still refers to the same object (so it can be changed).

```
function change(obj) {
  obj.a = 456;
}

const x = { a: 123 };
change(x);
x; // => { a: 456 }
```

3. Primitives are compared by value, while objects are compared by reference, so

```
// primitives are compared by value
const a = 5;
const b = 5;
a == b; // true

// objects are compared by reference
const arr1 = [1, 2, 3];
const arr2 = [1, 2, 3];
arr1 == arr2; // false

// now the variable holds the reference to the same object
const arr3 = arr1;
arr1 == arr3; // true
```

How to know if a value is of a particular type?

1. The `typeof` operator returns a string that indicates the type of a value.

```javascript
// primitives
typeof 123; // "number"
typeof 123n; // "bigint"
typeof true; // "boolean"
typeof undefined; // "undefined"
typeof "abc"; // "string"

// objects
typeof {}; // "object"
typeof []; // "object"
typeof function fn() {}; // "function"
```

Note that `type of array` is “object”, which makes sense as arrays in JavaScript are objects. But `typeof function` is “function”.

This is a bit confusing, if we remember that functions are also objects in JavaScript.

There are other quirks:

```javascript
typeof null; // "object"
typeof NaN; // "number"
```

The first one is a bug that TC39 tried to fix. Essentially they decided not to change this behavior, as it would break too much code on the web.

The last one is kinda make sense if you think about `NaN` as a very special value which has to do with numbers and hence belongs to type `number` (see What is NaN).

2. `instanceof` checks whether a value is created from a particular class. Using it with primitives doesn’t make any sense and always returns false.

```javascript
{} instanceof Object //=> true

// array is an instance of Array, which inherits from Object
```

```
[] instanceof Array //=> true
[] instanceof Object //=> true

// primitives are not instances of anything
"abc" instanceof String; // false

// but
new String("abc") instanceof String; // true
```

What is the difference between `null` s and `undefined` s?

Both `null` and `undefined` represent special primitive values of a "singleton type" (that means that `null` is a sole member of type `null` , the same goes for `undefined`).

The `null` s can only be created by using the `null` keyword.

`undefined` is a predefined global constant (not a keyword). You can get it when:

- accessing a non-existing property of an object or an array:

```
const obj = {};
obj.myVal; // undefined

const arr = [];
arr[10]; // undefined
```

- accessing a variable that wasn't initialized:

```
let myVar;
myVar; // undefined
```

- getting a value from a function that doesn't explicitly return anything:

```
function fn() {}
fn(); // undefined
```

What is `NaN` ? How to check if a value is `NaN` ?

`NaN` ("not a number") is a special variable in the global scope (you can access it as `window.NaN` , though it is the same as `Number.NaN`) that represents a "not a number".

`NaN` is returned by functions and operators, when there is no way to represent a value as a number.

For example,

```
parseInt("abc"); // => NaN (can't parse this string)
Number(undefined); // => NaN (can't convert)
0 * Infinity; // => NaN (can't determine the result)
Math.sqrt(-1); // => NaN (no complex numbers in JavaScript)
```

It has some interesting properties:

```
// it's not equal to itself
NaN == NaN; // => false
NaN === NaN; // => false

// but!
Object.is(NaN, NaN); // => true

// its type is "number", which may sound confusing
typeof NaN; // => "number", oh well
```

To check if a value is `NaN` , we can't use comparison (`NaN` is not equal to itself as shown earlier). There is a special function on `Number` :

```
Number.isNaN(parseInt("abc")); // => true
```

Note that this function checks for a specific value `NaN` ; everything else will return false:

```
Number.isNaN("definitely not a number"); // => false
```

There is also a global `isNaN` function that behaves very differently. It returns `true` for a whole set of other than NaN values but it's very inconsistent and should be avoided:

```
isNaN(NaN) // => true; nice!
isNaN('abs') // => true; well I guess it's not a number so that's fine
isNaN('') // => false; so is it a number?
isNaN({}) // => true; what is going on here?
```

Another fun thing is that because `NaN` is the only value in JavaScript which is not equal to itself, we can employ this fact as suggested on MDN:

```
function valIsNaN(val) {
  return val !== val;
}
```

Explain `truthy` and `falsy` values.

In JavaScript, we can apply boolean operators (and operators that expect boolean values) to non-boolean values.

That might sounds confusing, but it's very simple.

A value behaves either as `true` (and we call those values "truthy") or `false` ("falsy").

For example,

```
const name = "my name";

if (name) {
  console.log("the name is", name);
}
```

Here `"my name"` is a string. But when used with `if` it behaves as a boolean value `true` .

The condition above is, in fact, the same as:

```
if (Boolean(name) === true) {
  console.log('the name is', name)
}
```

`Boolean(name)` is a way to convert any value to a boolean. If it converts to `true`, we call this value "truthy", otherwise "falsy".

Another way to check if a value is truthy or falsy is to use double negation:

```
// those values are truthy:

!!"my string"; // => true
!!123; // => true
!![]; // => true
!!{}; // => true

// and here are some falsy ones
!!""; // => false
!!null; // => false
!!undefined; // => false
!!NaN; // => false
!!0; // => false
```

Since empty string `''` is falsy, and so are `null` and `undefined`, it is quite common in JavaScript to use `if` statement as a way to check for a "presence of a value".

We should note that these checks are not precise. Depending on the situation, you might not want to count some of the values as empty (for example, `0` or `false`).

Here are some more examples of checking for empty values using boolean operators:

```
const name = ''

// conditional `?:` operator
name ? console.log(`Name is ${name}`) : console.log('No name provided')

// logical short-circuiting (see the next question)
console.log(name || 'no name')
```

When would you use BigInt type?

The `BigInt` is a primitive data type introduced with ES2020.

As its names implies, it represents really big integers (64-bit integers).

The maximum safe integer that can be represented by a `number` is `2^53 - 1` (`Number.MAX_SAFE_INTEGER`).

Here "safe" means to represent precisely and be able to correctly compare.

JavaScript's `number` represents a double-precision floating-point number. That means we can go higher than the max value, but it won't be "safe" (which we we can check using `Number.isSafeInteger` function).

With BigInt we can go beyond that:

```
2n**53n + 5 // 9007199254740997n

123n // by default BigInts are based 10

0b1111n // but you can use binary format (but also octal `0o`, and hexodeci
```

What's the type of `1n` ?

```
typeof 1n // => 'bigint'
```

BigInt'a can be used as arguments to the regular arithmetic operations:

```
1n + 1n // => 2n

3000n * 5n // => 15000n

// Division automatically floors the results
8n / 3n // => 3n

// BUT!
1n + 1 // Uncaught TypeError: Cannot mix BigInt and other types, use explic
```

We can't mix BigInt with other types, but comparison operations work:

```
1 < 2n // => true
0 == 0n  // => true

// BUT!
0 === 0n  // false (because types are different)
```

Explain the Symbol type

A `Symbol` is one of the JavaScript primitives primarily used as a property key (along with the strings).

```
const nameProperty = Symbol("name")

const obj = {
  [nameProperty]: "John"
}

console.log(obj[nameProperty]) //=> "John"
```

Symbols are unique in the sense that newly created symbols are never equal to each other.

```
Symbol('test') === Symbol('test') //=> false
```

Symbols are most useful on a language design level because they allow introduction of new properties for the language without the risk of interfering with already existing ones.

Several “well-known” public symbols play a significant role in JavaScript.

The `Symbol.iterator` must be used to make an object iterable (more on iterables in later chapters).

```
const coords = { x: 1, y: 2, z: 3 }
for (let c of coords) { console.log(c) } //=> TypeError: coord is not itera

// let's make it iterable
coord[Symbol.iterator] = function* (next) {
  yield coord.x;
  yield coord.y;
  yield coord.z;
}

for (let c of coords) { console.log(c) } //=> 1, 2, 3
```

Other useful well-known symbols are `Symbol.toPrimitive` (for converting an object to a primitive), `Symbol.toStringTag` (for altering the string tag of an object when using `toString`), `Symbol.hasInstance` (for making it work with `instanceof` - see the next exercise), and others.

Another way to create a symbol is to use `Symbol.for` property. In that case, a symbol will be stored in a global registry. `Symbol.for` will either try to find an existing symbol in the registry or create a new one.

```
Symbol.for("test") === Symbol.for("test") //=> true
```

We should also note that symbols are not a good means for encapsulation. One can easily find symbol properties using `Object.getOwnPropertySymbols()` . For private fields, see chapter on classes.

Explain template literals

A template literal is another way to create a string by using backticks ‘my string‘.

It is similar to the usual string literals (like `'this'` and `"that"`) with some additional benefits.

1. It allows interpolation using the `${}` syntax.

```
const name = "John"
console.log(`Hello, ${name}!`) //=> Hello, John!
```

2. It can be multi-line

```

const msg = "Dear Santa,
Send me a JavaScript book this year!"

//=> Uncaught SyntaxError: Invalid or unexpected token

// but

const msg = `Dear Santa,
Send me a JavaScript book this year!`

// perfectly legal
```

What are tagged templates, and what are they useful for?

Tagged templates provide a way to convert a string (along with the interpolated text) to practically anything.

This is mostly useful for creating DSL (domain-specific language).

A tagged template is just any function that conform to a special form:

```
const myTag = (literals, ...data) {
  console.log('literals', literals)
  console.log('data', data)
}

const hours = new Date().getHours()
myTag`Hello, John! It's ${hours} already, time to go to school!`

//=> ["Hello, John! It's ", " already, time to go to school!"]
//=> [8]
```

As you can see, our tagged template function gets an array of literals and then a list of interpolations (`11` in this case).

We can easily convert it to another string. For instance, we can replace a number with a word.

```
const HOURS_TO_STR = {
  // ...
  8: "eight",
  9: "nine",
  10: "ten",
  // ...
}

const myTag = (literals, hour) {
  return literals[0] + HOURS_TO_STR[hour] + literals[1]
}

const hours = new Date().getHours()
console.log(myTag`Hello, John! It's ${hours} already, time to go to school!
//=> Hello, John! It's eight already, time to go to school!
```

The returned value doesn't have to be a string. Here's a (simplified) example of a tag that parses a string to a JavaScript object.

```
function([str]) {
  return str
    .split(",")
    .map(s => s.split(":"))
    .reduce((acc, [ name, val ]) => (acc[name] = val, acc))
}

console.log(obj`name:John,surname:Smith`)
//=> { name: 'John', surname: 'Smith' }
```

There are many popular frameworks and libraries heavily rely on this feature: - styled-components (a CSS-in-JS library) - lit (HTML templating) - graphql-tag (GraphQL query builder) - karin (AJAX library) - sql-tag (prepares SQL statements)

, and many others.

What are typed arrays and when are they useful?

Typed arrays is an API that provides means for reading and modifying raw binary data.

There are several typed arrays, each for a specific type, e.g., Int8Array, Uint8Array, Int16Array, all the way to BigUint64Array.

They all act as proxies providing access to the underlying binary data stored as an instance of ArrayBuffer.

ArrayBuffer is a store for binary data, but it provides no way to access it or modify it.

To do this, we need an instance of a typed array. In that sense, typed arrays act as a view for binary data stored in an ArrayBuffer.

Here's an illustration of this.

Let's create an empty array buffer of 2 bytes:

```
const data = new ArrayBuffer(2)
const int8 = new Uint8Array(data)
const int16 = new Uint16Array(data)
```

Note how both Int8 and Int16 arrays act as a view for the same data. The first one reads 1 byte at a time, and the second 2 bytes at a time.

We can set both bytes to decimal 1.

```
int8[0] = 1
int8[1] = 1
```

Note that in the binary format, our data would look something like this now `00000001` and `00000001` .

If we read it now using Int16Array, we get

```
int16[0] //=> 257
```

Variables and scope

How to declare variables? What are the differences between `var`, `let`, and `const`?

We can declare variables and optionally assign them values:

```
// declaring a variable with `var` keyword
var name;

// Declaring a variable and assigning a value
var name = "John";
```

Historically, `var` was the only way to assign a variable, with `let` / `const` introduced later as part of the ECMAScript 2015 (ES6).

The `var` declaration has some confusing properties when it comes to scope. On the other hand, `const` and `let` have cleaner scoping rules and usually should be preferred.

How do they differ?

Variables declared with `var` have a so-called "function-level scoping". They are accessible to any code within the same functions.

```
function testScope() {
  var v = 123;
  console.log(v); //=> 123
}
console.log(v); //=> Uncaught ReferenceError: v is not defined
```

When used outside of any function, it will assign values to the global scope. That means it will be accessible globally in any part of the code and available on the global object (e.g., `window.myVar`).

Function scoping is opposed to **block scoping** when a variable is only visible within a block (that is, within curly braces `{` / `}`). The `let` and `const` keywords declare variables with a block scope.

```
if (true) {
  var name = "Emma";
  let surname = "Stone";
}

console.log("name", name);
console.log("surname", surname); // ReferenceError: surname is not defined
```

Block-scoping is useful because it allows variables to only be visible within a block, thus preventing possible errors with overlapping variable names.

`const` works similar to `let` in terms of scoping. A difference between those two is that `const` prevents re-assigning. You can only assign a value to that variable once, and it must be done at the declaration time.

```
const age = 123;
age = 321; // TypeError: Assignment to constant variable.

let name = "John";
name = "Alex"; // No problem here
```

It's important to note that `const` prevents re-assignment, but it doesn't make it immutable.

```
const person = { name: "John" };
person.name = "Anna"; // perfectly valid
```

Another important difference has to do with the hoisting (we'll talk about it in detail later).

For now, it's important to note that variables declared with `const` and `let` while being hoisted don't get initialized with `undefined` as the variables declared with `var` do.

Consider the following example.

```
function hoist() {
  console.log(a); //=> undefined
  console.log(b); //=> Uncaught ReferenceError: Cannot access 'a' before ir
  var a = 10;
  const b = 10;
}
```

Both `a` and `b` are hoisted because the interpreter already knows about their existence. But `a` also gets initialized with `undefined` , while `b` is not initialized, and thus, we get an error.

In practice, it's good to always default to `const` . If you know that you'll need to re-assign the variable sooner, use `let` . Avoid using `var` (unless you specifically mean "function-level" scoping).

What is the global scope? What is the globalThis object?

The outermost scope in JavaScript is called the "global scope", and variables declared on the global scope are accessible from anywhere.

There are two ways to create a global variable.

First, we could declare a variable using `let` or `const` while in the global scope.

Second, we can create them using `var` or a `function` keyword. In that case, those variables also become properties on the so-called "global object" (`window` in a browser, or `global` in a Node.js context).

Consider this example.

```
var a = 123;
const b = 456;

function main() {
  console.log(a); //=> 123
  console.log(window.a); //=> 123
  console.log(b); //=> 456
  // but
  console.log(window.b); //=> undefined
}

main()
```

There is also a way to create a global variable from any inner scope. It happens when the declaration keyword is omitted and usually is a sign of a mistake.

Consider the following,

```
function test() {
  for(i = 0; i < 10; i++) {
    console.log("text")
  }
}

test()

console.log(window.i) //=> 10
```

We unintentionally introduced a global variable because we forgot to use any kind of a declaration keyword with `i` (either `let` or `var` would do).

That, though, would result in an error when using the strict mode.

In ECMAScript2020, a new global variable was introduced called `globalThis` . It could be used both in the browser and in Node.js, and it refers to the global object in the current context (`window` in a browser, `global` in Node.js).

```
// in browser
window === globalThis //=> true

```

```
// in Node.js
global === globalThis //=> true
```

What is hoisting, and how does it work?

Hoisting is when variable declarations are (seemingly) moved to the top of the function (or global) scope. It works a bit differently for different keywords.

1. Function declarations moved to the top. That allows us to call functions in the code before they are declared.

```
console.log(today()) //=> 03/12/2021

function today() {
  new Date().toLocaleDateString()
}
```

Note that it only works for functions declared using the `function` keyword. The following doesn't work.

```
console.log(today()) //=> Uncaught TypeError: today is not a function

const today = () => new Date().toLocaleDateString()
```

2. For variables declared with `var`, only the declaration is hoisted, but not the initialization. The hoisted variable is accessible, but its value is `undefined`.

```
console.log(name); //=> undefined

var name = 'John';

console.log(name); //=> 'John'
```

3. Variables declared with `let`, `const`, and classes are hoisted in the sense that the interpreter knows about them, but they don't get initialized, which practically means we can't use them before

the initialization.

```
console.log(name); // ReferenceError: Cannot access 'name' before initializ
const name = "John";
```

Ex. 3.1: What will be printed here?

```
console.log(name);

name = "Alex";
console.log(name);

var name = "John";
console.log(name);
```

Go to solution →

Ex. 3.2: What will be the output of the following?

```
var myVar = 1;

function run(){
  console.log(text);
  var myVar = 2;
};

run();
```

Go to solution →.

Ex. 3.3: What will be the output of the following code?

```
function print() {
  console.log(a)
}

print()
setTimeout(print, 1000)

var a = 10;
```

Go to solution →.

Ex. 3.4: What will be the result of the previous exercise code if we declare `a` using `const` ?

Go to solution →.

Ex. 3.5: What will be the output of the following code?

```
for (var i = 0; i < 3; i++) {
  setTimeout(()=> console.log(i), 1);
}

for (let j = 0; j < 3; j++) {
  setTimeout(()=> console.log(j), 1);
}
```

Go to solution →.

Operators

What’s the difference between `==`, `===`, and `Object.is`?

The `==` operator is sometimes referred to as **loose equality operator** compares two values trying to coerce them to a single type (but only primitives!).

```
123 == "123"; // => true
```

While this may seem convenient, in reality it can lead to confusing results.

```
"" == 0; //=> true
null == undefined; //=> true

0 == false; // => true
1 == true; // => true
// but
2 == true; // => false

[1, 2] == "1,2"; // => true
// but
[1, 2] == "1, 2"; // => false
```

Two non-primitive objects are only equal when they refer to the same object:

```
[1, 2] == [1, 2]; // false (different objects)

const a = [1, 2];
const b = a;
a == b; // true (both a and b refer to the same object)
```

The strict equality operator (triple equals `===`) doesn’t try to coerce the types; thus, two values are only equal if their types are the same and their values are the same:

```
"1" === 1; // false
undefined === null; // false
```

Triple equals operator should always be preferred when comparing values as it has much more reliable behavior.

Another way to compare values is `Object.is` function. It compares two values and returns `true` if they are equal. It works almost the same as the triple equals operator `===` but fixes some of the quirks:

```
NaN === NaN; // => false
Object.is(NaN, NaN) - // => true
0 === -0; // => true
Object.is(-0, 0); // => false
```

What are `&&`, `||`, and `??` operators useful for?

The `&&` operator performs a logical conjunction. It returns `true` if and only if both operands are `true`.

```
false && false; // => false
false && true; // => false
true && false; // => false
true && true; // => true
```

We can also use it with non-boolean values. In this case, it returns either the first non-truthy value or the last value if all operands are truthy.

```
false && 1 // => false
true && 1 // => 1
true && 1 // => 1
'test' && 123 && { x: 5 } // => { x: 5 }
```

The `&&` is short-circuited, which means it will only evaluate the following operand if the current operand is truthy.

```
true && console.log('Yes') // => returns undefined, prints 'Yes'
false && console.log('Yes') // => returns false, doesn't print anything
```

`||` it the logical “or” operator. In the same way as `&&`, it can be invoked with booleans and non-boolean values (in which case they are converted to booleans).

```
false || false; // false
false || true; // true
true || false; // true
true || true; // true
```

When used with non-booleans, it returns the first truthy operand.

```
true || "123"; // true
false || "123"; // "123"
```

Sometimes the `||` operator is used to assign a default value to a variable.

```
function printName(maybeName) {
  const name = maybeName || "Unknown";
  console.log(name);
}

printName("John") //=> John
printNamr() //=> "Unknown"
```

It works because both `null` and `undefined` are falsy. But this comes with a catch: there are other falsy values, for example, `0` or `false`.

Consider this example:

```
const maybeTemparature = null;
const temp = maybeTemparature || 20; //=> 20
```

```
const maybeTemp = 0;
const temp = maybeTemp || 20; // 20
```

In the last case, we assign 20 to the variable temp which is incorrect.

Zero is a perfectly valid temperature, so in this case, we don't want it to fall back to the default value.

We can use the **nullish coalescing operator** `??` (introduced in ECMAScript 2020) to save the day.

```
const maybeTemp = 0;
const temp = maybeTemp ?? 20; // 0
```

The `??` works similarly to `||`, but it only evaluates to the right operand when the left is either `null` or `undefined`. It makes it a much better candidate for settings default values.

Essentially, it is equivalent to:

```
const temp = maybeTemp !== null && maybeTemp !== undefined ? maybeTemp : 20
```

Spread syntax `...`

The spread syntax is a way to expand an iterable in places where multiple arguments are expected.

Might be easier to look at an example.

```
function sum(x, y) {
  return x + y;
}

const arr = [1, 2];

sum(...arr); //=> 3
```

The `sum` function expects two arguments, and we pass an array to it, which we expand with the spread syntax (three dots `...`).

The spread syntax has many use-cases.

```
// (shallowly) clone an array
const arr  = [1, 2, 3];
const newArray = [...arr];
```

```
// push items
const arr = [1, 2, 3];
const newArr = [0, ...arr, 4, 5]; //=> [0, 1, 2, 3, 4, 5]

// merge arrays
const a1 = [1, 2, 3];
const a2 = [4, 5, 6];
const a1 = [...a1, ...a3]; //=> [1, 2, 3, 4, 5, 6]
```

The spread syntax is very similar to the `rest parameters` feature (and also the destructoring assignment, which essentially does the opposite - takes a list of arguments and converts them into an array.

```
function test(a, b, ...params) {
  console.log(params);
}

test(1, 2, 3, 4, 5); //=> [3, 4, 5]
```

Since ECMAScript 2018, the spread syntax can also be applied to object literals.

```
// creating a new object with extra properties
const user = { name: 'John', surname: 'Smith' }
const userWithEmail = { ...user, email: 'john@smith.com' }
```

The destructuring assignment

The destructuring assignment syntax is used to decompose an object or an array and assign new variables.

```
const user = { name: "John", surname: "Smith" };
const { surname } = user;
console.log(surname); //=> "Smith"
```

Since we know the object's structure, it is possible to "destruct" it and dig the data that we need.

It works similarly for arrays.

```
const arr = [1, 2, 3, 4, 5];
const [firstEl, ...rest] = arr;
console.log(firstEl); //=> 1;
console.log(rest); //=> [2, 3, 4, 5];
```

We can even set a default value.

```
const { surname = 'Smith' } = {}
console.log(surname); //=> Smith
```

It may be useful in several situations:

```
// swapping values
let a = 1, b = 2;
[a, b] = [b, a];
console.log(a); //=> 2
console.log(b); //=> 1

// creating a new object with some fields dropped
const user = { name: 'John', surname: "Smith", email: "john@smith.com" };
const { email, ...userNoEmail } = user;
console.log(userNoEmail); //=> { name: 'John', surname: 'Smith' }
```

What the comma operator is useful for?

A comma operator `,` is a less-known feature of the JavaScript language that can be useful in some cases.

What does it do? It evaluates its operands and returns the result of the last expression.

```
2 + 2, "hello" //=> "hello"
```

Or we can join multiple operands.

```
const x = (false, "hello", 3 * 3)
console.log(x) //=> 9
```

Compared with short-circuit operators `||` and `&&` , it always executes both operands, no matter the intermidiate results.

Compare,

```
const x = 1 || console.log("executed")
console.log(x)
//=> 1



const x = (1 , console.log("executed"))
console.log(x)
//=> executed
//=> undefined
```

When can it be useful?

Anywhere where we can take advantage of writing several expressions on one line.

Here's a map function implemented using `reduce` .

```
function map(arr, mapper) {
  return arr.reduce((acc, el) => {
    acc.push(mapper(el));
    return acc;
  }, [])
}
```

We're using an arrow function with a body because we need to do two things here: push a new element, and return the accumulator. We can do both with a comma operator. And voila, we have a one-liner.

```
function map(arr, mapper) {
  return arr.reduce((acc, el) => (acc.push(mapper(el)), acc), [])
}
```

Functions

How to declare a function?

There are several ways to create a function in JavaScript, each of which has distinctions from the others.

The first way is a “function declaration”.

```
function sum(a, b) {
  return a + b;
}
```

With function declaration, we write the `function` keyword followed by an obligatory name, zero or more arguments, and a function body within the curly brackets `{ / }` .

The function body can optionally return a value. If the returned value is not specified, it always returns `undefined` .

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


log('hello'); //=> undefined returned
```

A significant advantage of the function declaration is that the function can be used before it was declared.

It is due to `hoisting` (see the question on this topic). In short, functions are seemingly lifted up in code prior to the execution.

```
callme(); //=> 'Hello'


function callme() {
  console.log('Hello')
}
```

The other way to declare a function is `function expression` . We create a function and assign it to a variable. We can do that because functions in JavaScript are first-class citizens (that means that, like any other values, they can be assigned to variables, moved around, passed to, and returned from the other functions).

```
const sum = function(a, b) {
  return a + b;
}
```

Unlike `function declarations` , function expressions don't get hoisted - we can't use them before they are declared.

```
callme();
//=> ReferenceError: Cannot access
//=> 'callme' before initialization

const callme = function () {
  console.log('Hello')
}
```

Another difference is that function expressions can be anonymous.

```
const arr = [1, 2, 3];
const multiplied = arr.map(function(x) { return x * x })
```

We could also create a function using the `Function` constructor.

```
const fn = new Function("return 5")
```

However, similarly to `eval` , it is considered to be not secured and also suffers from performance issues. Creating functions like this should be avoided.

Finally, we can create functions using `the arrow syntax` (`() => {}`). The next question explores it in detail.

What are arrow functions? How do they differ from regular ones?

Arrow functions were introduced in ECMAScript 6 (2015) as a compact alternative to the regular `function` expression.

It has several important differences:

1. It can not be used as a constructor, and as such, it doesn't have the `prototype` attribute.

```
const fn = () => {}
new fn()  // Uncaught TypeError: fn is not a constructor
```

2. Arrow functions do not have their binding to `this` . When executed, it uses `this` from the enclosing scope. Attempt to reassign using `bind` doesn't work:

```
// regular function
function fn1() {
  return this
}
fn1.bind({})() // => {}

// arrow function
const fn2 = () => this
fn2.bind({})() // => Window
```

3. It doesn't have access to `arguments` .

4. Can be used without curly braces. In this case, it returns the last expression (no need to `return` a value explicitly):

```
[1, 2, 3].map(x => x**2) // [1, 4, 9]

// BUT! Don't forget the `return` statement in a case when curly braces are
[1, 2, 3].map(x => { return x**2 }) // [1, 4, 9]
```

5. The parenthesis can be omitted when there's only argument (like

in the last example)

6. Unlike `function declaration` , it can be anonymous, and it doesn't get hoisted.

What is a closure?

Functions in JavaScript (and not only) have a feature called `closure` - they can access variables from the outer scope even when they are executed in a different context.

Have a look at the following code.

```
function counter() {
  let i = 0;
  return function () {
    return i++;
  };
}

const next = counter();
console.log(next()); // 0
console.log(next()); // 1
console.log(next()); // 2
```

When we call `counter` it creates a new binding - a variable `i` . Then it creates and returns a new function that we can use outside of its initial scope.

That function is "closed over" its initial scope (with variable `i`), and continues to have access to this variable.

Here's another example, when a closure is a function parameter:

```
function multiplyBy(num) {
  return (val) => val * num;
}
```

```
const double = multiplyBy(2);
double(5); //=> 10

const quadrouple = multiplyBy(4);
quadrouple(5); //=> 20
```

Ex. 5.1: What is wrong with the following code?

Imagine you have three buttons with ids `btn-1` , `btn-2` , `btn-3` . You want each of them to alert its number when being clicked. So you write a simple loop.

```
for (var i = 1; i <= 3; i++) {
  const btn = document.getElementById(`btn${i}`)
  btn.addEventListener("click", () => alert(`I'm a button #${i}`))
}
```

Why is this code wrong? How to fix it?

Go to solution →

What is an IIFE?

IIFE or "Immediately Invoked Function Expression" is a function that gets called right after its declaration.

~~~
(function () {
  // I'm an IIFE
})()

It is primarily useful for creating a new isolated scope to not introducing

```javascript
(function() {
  // user is created within the functional scope
  var user = getUser()
~~~

```
})()

// user is not available here
```

The other use-case was widespread before the introductions of JavaScript modules.

```
const counter = (function() {

  var count = 0

  function inc() {
    count = count + 1;
    return count;
  }

  return { inc: inc }

})();
```

Here we created a module with just one function without exposing the inner variable `count` (encapsulation!).

How can we use function rest parameters?

Rest parameters (three full-stop symbols `...`) allow us to define functions with an unspecified number of arguments.

The rest syntax converts multiple arguments into an array.

```
function sum(...args) {
  return args.reduce((acc, el) => acc + el, 0)
}

sum(1, 2, 3) //=> 6
sum(1, 2, 3, 5, 6) //=> 17
sum() //=> 0
```

The `sum` function can now be used with any number of arguments.

The rest argument can follow after any number of normal parameters.

```
function says(name, ...words) {
  console.log(`${name} says: ${words.join(" ")}`)
}

says("John", "hello", "there", "!") //=> John says: hello there !
```

Only the last parameter can use the rest syntax.

```
// invalid!
function myFunc(...firstArgs, theLastOne) { }
```

There also can’t be multiple rest params.

```
// invalid!
function myFunc(...firstBatch, ...secondBatch, ...thirdBatch) { }
```

Ex. 5.2: Nested HTML tree

Write a function that takes multiple arguments and builds a nested HTML tree.

Example:

```
tree('a', 'span')
// =>  <a><span></span></a>

tree('div', 'ul', 'li', 'a')
// => "<div><ul><li><a></a></li></ul></div>"
```

Go to solution →

Explain call, apply and bind

There are several ways to invoke a function. The first one is to use the parenthesis:

```
myFunc()
```

`Function.prototype.call()` accepts a `this` reference, and arguments.

```
fn.call(myObj, arg1, arg2, ...)
```

`Function.prototype.apply()` is very similar, but it accepts the arguments as an array.

```
fn.apply(myObj, [arg1, arg2, ...])
```

Why would you use any of those?

One reason is when you have a function that you want to apply with `this` pointing to another object.

```
Array.prototype.map.apply([1, 2, 3], [(x) => x * x])
```

The `bind` method returns a new function which `this` pointer is set to a specified object.

```
const arr = [1, 2, 3]
const bmap = Array.prototype.map.bind(arr)
bmap(x => x * x) //=> [1, 4, 9]
```

Elements of Functional Programming

What does it mean that functions are first-class citizens?

In JavaScript, a function behaves like any other value. We can assign it to a variable, return it from a function and use it as an argument in another function.

This is what it means to be "a first-class citizen".

For example,

```
const add(a) {
  return (b) => a + b
}

const add2 = add(2)
add2(10) //=> 12

const add5 = add(5)
add5(10) //=> 15
```

In this example, the `add` function creates a new function and returns it in its body.

Here's another very familiar example:

```
[1, 2, 3].map((x) => x * x) //=> [1, 4, 9]
```

Here we pass a function `(x) => x * x` as an argument to the `map` function.

That property of a function in JavaScript opens many great opportunities for a developer and makes it possible to use the functional programming paradigm.

What are pure functions, and why are they useful?

A pure function is a function that returns the same result for any given arguments, does not depend on any state, and does not produce any side effects.

A simple example of a pure function:

```
function sum(a, b, c) {
  return a + b + c;
}
```

On the other hand, `Math.random()` is **not** pure, as it returns different results each time.

Here are some example of non-pure functions that produce a side-effect

```
// DOM is an external state
function toggleActive(el) {
  if (el.classList.contains('active')) {
    el.classList.add("active");
  } else {
    el.classList.remove("active");
  };
}

// Network request
function getProducts() {
  return fetch("/api/products.json").then(res => res.json())
}

// invoking console.log is a side effect
function double(a) {
  console.log('double ' + a);
  return a * 2;
}
```

The benefit of using pure functions is that since they are so simple (don't depend on any state) and predictable (always return the same result for the given input), it's very easy to reason about them. It's very easy to test them.

```
function dig(obj, path) {
  // try implementing me
}


it('should dig into the object and return result', () => {
  const input = { a: { b: { c: 1 } } }
  expect(dig(input, 'a.b.c')).toEqual(1)
})
```

As a result, using pure function leads to clean and maintainable code.

We can't eliminate the state from our programs entirely (otherwise, they'd be useless). Still, it's a good approach to minimize accessing the state and break the program into smaller testable pure functions.

Ex. 6.1: Implement currying

A curried function is a function that you can invoke with only some arguments and get a new function back that accepts the rest of the arguments.

Example.

```
// normal invocation
sum(1, 2) // => 3


/// partial invocation
const addOne = sum(1)
addOne(2) // => 3
```

Implement the `curry` function that takes a function as an argument and returns a curried version of the same function.

Go to solution →

Ex. 6.2: Implement flatten

Implement `flatten` - function that takes an array and returns a new array with all the sub-array elements concatenated into it.

Here’s an example,

```
flatten([1, [2], [3, 4]]) //=> [1, 2, 3, 4]
```

Go to solution →

Ex. 6.3: Implement compose

Implement the `compose` function that takes an unspecified number of functions and produces a new function resulting from applying the initial functions sequentially from right to left.

The `compose` function is part of many functional libraries including Ramda, underscore, and even Redux.

Here’s an example,

```
const upcase = (str) => str.toUpperCase()
const takeName = (obj) => obj.name
const greet = (name) => `Hello, ${name}!`

const greetLoudly = compose(greet, upcase, takeName)

const greetLoudly({ name: 'John' }) //=> Hello, JOHN!
```

Go to solution →

Objects

How to create an object?

There are multiple ways to create an object in JavaScript.

1. The simplest way is to use the object literal `{}` like this:

```
const person = {
  name: "Alice",
  age: 27,
}
```

2. We can use **a constructor function** (a special function which intended to be invoked with the `new` keyword).

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

const person = new Person("Alice", age)
```

3. We can also use the `new` keyword with the classes introduced in ES6.

```
class Person {
  constructor(name, age) {
    this.name = name
    this.age = age
  }
}

const person = new Person("Alice", age)
```

4. Finally, we can use `Object.create` method which comes handy if we need to manually set a prototype (more on prototypal inheritance later).

```
const proto = {
  name: "Alice",
  age: 27,
}

const person = Object.create(proto)

console.log(person.name) //=> "Alice"
Object.getOwnPropertyNames(person)
//=> [], no own properties because
// name belong to the prototype
```

What is the diff between class and constructor function (pre-ES6) ?

```
// ES5 Function Constructor
function Student(name, studentId) {
  // Call constructor of superclass to initialize superclass-derived member
  Person.call(this, name);

  // Initialize subclass's own members.
  this.studentId = studentId;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

// ES6 Class
class Student extends Person {
  constructor(name, studentId) {
    super(name);
    this.studentId = studentId;
  }
}
```

What will be the output of this code?

```
var Employee = {
  company: 'xyz'
}
var emp1 = Object.create(Employee);
delete emp1.company
console.log(emp1.company);
```

Explain prototypal inheritance? How is it different from the classical inheritance?

In JavaScript, every object has a special hidden property called a **prototype**. When a property is accessed on an object, it first tries to find it on that object, then on its prototype, then on the prototype of the prototype and so on until it reaches the end of the **prototype chain**.

Have a look at the following code snippet.

```
const arr = [1, 2, 3]
arr.map(x => x * x)
```

Where does this `map` method come from? Well, it is defined on the `Array.prototype` which is a prototype of every array instance.

```
[].map === Array.prototype.map //=> true
```

Every function has a special "prototype" field which refers to the prototype assigned to every object created using the `new` keyword.

```
function A() {
}
A.prototype.foo = "bar"

const a = new A()
```

```
a.foo //=> "bar"
```

While the `foo` property is not found on the object itself, it is found on its prototype.

An alternative way to get an object's prototype, by the way, is to use the `Object.getPrototypeOf` function. The `__proto__` field, even though widely supported, is not defined by the ECMAScript standard.

In "classical" inheritance, properties are inherited by classes. In JavaScript, classes are just a syntax sugar on top of the same prototype model.

The underlying inheritance mechanism is still based on the prototype chain.

```
class A {}
class B extends A {}
const b = new B()
b.__proto__ === B.prototype //=> true
b.__proto__.__proto__ === A.prototype //=> true
b.__proto__.__proto__ .__proto__ === Object.prototype //=> true
b.__proto__.__proto__.__proto__.__proto__ // => null
```

How to clone an object? Explain deep clone vs. shallow clone.

In JavaScript, there are multiple ways to clone an object.

For example, you can use the `slice` method on arrays.

```
const arr = [1, 2, 3]
const cloned = arr.slice()
```

Similarly, we can clone objects

```
const user = { name: "John" }
const cloned = { ...user }
```

Both are examples of so-called **shallow cloning**. It means that the properties of a newly created object still refer to the same objects as properties of the original object.

Consider the following example, where we clone a user and then change his city.

```
const user = { name: "John", address: { city: "London" } }
const cloned = {...user}
cloned.address.city = "Paris"
console.log(cloned.address.city) // => Paris
console.log(user.address.city) // => Paris
```

The city of the original user has changed as well. This is not surprizing considering that the `address` property on both objects refers to the same object.

If we need to clone the object completely along with all of its nested fields no matter how deep, we need **deep cloning**.

Implementing deep cloning in JavaScript is only trivial for simple cases when all the properties are either primitives, objects, or arrays.

In this case, we can write a simple recursive function.

```
function cloneDeep(o) {
  if (Array.isArray(o)) {
    return o.map(cloneDeep);
  } else if (typeof o === 'object') {
    return Object.keys(o).reduce((acc, k) => {
      acc[k] = cloneDeep(o[k]);
      return acc;
    }, {});
  } else {
    // assuming the only other option left is a primitive
    return o;
  }
}
```

An alternative way to achieve the same result is use a simple trick with serializing to a JSON string, and the parse it back.

```
const cloned = JSON.parse(JSON.stringify(user))
```

While this is usually enough, we need to mention that it has some significant drawbacks.

Not all of the types we deal with are serializable by JSON.stringify.

```
// can't serialize maps
JSON.stringify(new Map().set("test", 123)) // => {}

// symbols are invicible
JSON.stringify({[Symbol("test")]: 123}) // => {}

// you loose your key if the value is undefined
JSON.stringify({test: undefined}) // => {}

// and so on
```

Also, note that JSON.stringify/parse can also be slower compared to other implementations (like lodash's `cloneDeep` or jQuery's `extend`).

Finally, there's also a fairly-new global structuredClone function that seems to be very performant. It has the benefit of supporting more types (Dates, RegExps, Maps, etc.).

Just note, that this it is not part of the JavaScript language, but rather part of the Browser's (and Node's!) API.

What are immutable objects?

An immutable object is an object which can't be modified. If you need to change any of its properties, you have to create a new object.

Immutable objects are useful because they allow the writing of more straightforward code - since the objects are immutable it's very easy to track the state of the application at each particular point in time.

In JavaScript objects are mutable. One way to make sure they are immutable is to limit oneself to only using functions that don't modify the current object, but rather return a new one.

Here are some example

```
const obj = { name: "John", surname: "Smith" }
obj.name = "Mary" // mutating the existing object

const mary = {...obj, name: "Mary" } // leaving the original untouched
```

This approach is far from ideal. There are better ways (see the next question).

How does Object.freeze work?

Thankfully, there's a way to make an object immutable (prohibiting changing, adding, or removing its properties) and that's `Object.freeze` .

```
const john = Object.freeze({name: "John"})
john.name = "Mary" // no error is thrown
console.log(john.name) //=> "John", the object didn't change
```

We can also freeze arrays, in which case we wouldn't be able to push new elements, neither to remove, or update.

```
const arr = Object.freeze([1, 2, 3])
arr.push(4)
console.log(arr) //=> [1, 2, 3]
```

In strict mode, both of those examples would throw a `TypeError` exception.

`Object.freeze` works shallowly. That means that if there are nested objects, the properties of those can still be modified.

```
const user = Object.freeze({ name: 'John', phone: { mobile: '123' } })
user.phone.mobile = '456' // that works!
```

Ex. 7.1: Implement a deep freeze

Implement a function that freezes an object deeply (that is including all the nested properties).

```
const user = {
  name: "John",
  address: { city: "Amsterdam" }
}

deepFreeze(user)

user.address.city = "test"
console.log(user.address.city) //=> Amsterdam
```

Go to solution →

Arrays

How to merge two arrays?

Since the introduction of the spread syntax in ES6, the easiest way to merge to arrays is to use it,

```
const arr1 = ['Tom', 'John']
const arr2 = ['Anna', 'Sarah']

const all = [...arr1, ...arr2] //=> ['Tom', 'John', 'Anna', 'Sarah']
```

Alternatively, we could use the `Array.prototype.concat()` method that doesn't change the original array. It also allows us to add multiple values.

```
const all = arr1.concat(arr2)
const all2 = arr1.concat('Anna', 'Sarah')
```

Finally, we can also use `Array.prototype.push()` . It adds multiple elements but modifies the array, so it should be used carefully.

```
arr1.push(...arr2)
arr1 //=> ['Tom', 'John', 'Anna', 'Sarah']
```

What are `slice` and `splice` methods are useful for?

The `slice` method is used to get a shallow copy of an array (or part of it).

The syntax is `slice(start, end)` with both arguments being optional.

```
// get a shallow copy of the whole array
[1, 2, 3, 4, 5].slice() //=> [1, 2, 3, 4, 5]

// get elements 1 through 4 excluding the last one
[1, 2, 3, 4, 5].slice(1, 4) //=> [2, 3, 4]
```

```
// get from the first till the second from the end
[1, 2, 3, 4, 5].slice(1, -2) //=> [2, 3]
```

The `slice` method doesn't modify the original array and returns a new one.

On the other hand, the `splice` method modifies the original array in place.

The `splice` syntax is `splice(start, num_elem_to_remove, item_to_insert1, ite`
, and it returns the removed elements.

```
const arr = [1, 2, 3, 4, 5]

// Remove one element at index 1
arr.splice(1, 1) //=> [2]
arr //=> [1, 3, 4, 5]

// Remove two elements at index 1, and insert 2 more items
arr.splice(1, 2, 'a', 'b') //=> [2, 3]
arr //=> [1, 'a', 'b', 4, 5]

// Do not remove anything, only insert an item at index 2
arr.splice(2, 0, 'c') //=> []
arr //=> [1, 2, 'c', 3, 4, 5]
```

Ex. 8.1: uniq

The `uniq` function accepts an array and returns a new one with all the duplicates removed.

For example,

```
uniq([1, 2, 3, 4, 5, 1, 5]) //=> [1, 2, 3, 4, 5]
```

Go to solution →

Ex. 8.2: Converting an array into object

Suppose, that you have an array of key-value pairs and you need it to convert into an object `{key: value, ...}` .

```
const ages = [ ["john", 5], ["jane", 27], ["tom", 71] ];
fromEntries(ages)
//=> { john: 5, jane: 27, tom: 71 }
```

How would you do that?

Go to solution →

Ex. 8.3: What will be the output of the following code

```
const student1 = { name: "Anna" }
const student2 = { name: "Jane" }

const grades = {}
const grades[student1] = 'A'
const grades[student2] = 'C'

console.log(grades[student1])
```

Go to solution →

Ex. 8.4: Implement a function that rotates an array n times

```
const arr = [1, 2, 3, 4, 5]
rotate(arr, 1) //=> [5, 1, 2, 3, 4]
rotate(arr, 3) //=> [3, 4, 5, 1, 2]
```

Go to solution →

Ex. 8.5: How to check if two strings are anagrams of each other?

Here are some examples,

```
elbow = below
inch = chin
state = taste
```

Go to solution →

Ex. 8.6: How to reverse an array in place?

Reversing an array is simple if you can create a new one and copy elements from one to another (not unsimilar to the previous exercise). But sometimes this is not optimal (say you have a very large array, and you don't want to waste memory).

Write a function that reverses an array in place.

Go to solution →

Ex. 8.7: Implement a filter function using recursion

Recursion is a very powerful technique that allows implementing of many `Array.prototype` methods (like, `map` , `filter` , and even `reduce`).

Use it to implement a filtering function that works like `Array.prototype.filter` , but takes the array as the first argument.

```
filter(
  [1, 2, 3, 4, 5, 6],
  (x) => x % 2 === 0
)

// => [2, 4, 6]
```

Go to solution →

Ex. 8.8: Shuffle an array

Write a function that will shuffle an array.

```
shuffle([1, 2, 3, 4, 5])
//=> (5) [4, 1, 2, 5, 3]
```

Go to solution →

Collections

What are Iterators and Iterables?

An iterable is a value that can be iterated over, a value with a source of data, that can we can work with in a sequential matter.

A good example, is an array.

JavaScript since ES6 standardized iterables by introducing the **Iterable** interface. Any value that implements this interface can be iterated over, which mean it can be used with `for..of` loop, and also with the **"spread" operator**

```
// array is "iterable"
const nums = [1, 2, 3, 4, 5]

for (let n of nums) {
  console.log(n)
}

// ... and so is a string
const chars = [..."hello"] // => ['h', 'e', 'l', 'l', 'o']

// Sets are iterables
for (v of new Set([1, 2, 3])) { console.log(v) }

// and Maps
for (v of new Set([{a: 'value'}, {b: 'another value'}])) { console.log(v) }
```

In case of Maps, we iterate over "entries" which are tuples (arrays of 2 elements) with the first element being a key, and the second — a value associated with that key.

How to make an object iterable?

Any object can become iterable by implementing the Iterable protocol.

Imagine, that we have an object that represents a user with a name, surname, phone number, and an email. We want to be able to iterate over that object and print one piece of data at a line.

```
const user = {
  name: "John",
  surname: "Lennon",
  phone: "111111111",
  email: "john@beatles.com",
};

for (d of user) {
  console.log(d);
}
```

If we run this code like this we get an exception `TypeError: user is not iterable` . That’s because plain objects are non-iterable by default.

In order to make it iterable, we need to add a method that returns *an iterator*.

Here how we can do this:

```
const user = {
  name: "John",
  surname: "Lennon",
  phone: "111111111",
  email: "john@beatles.com",
  [Symbol.iterator]() {
    // list of fields that we want to print out
    // we could have just use Object.entries, but it doesn't guarantee the
    const fields = ["name", "surname", "phone", "email"];

    // local index
    let i = 0;

    // we return an iterator (an object with method "next")
    return {
      next: () => {
        if (i >= fields.length) {
          // we have finished
          return { done: true };
        }

        // take the current field
        const field = fields[i];

        // compute the value we want to return for that field
        // `this` here refers to our user object
        const value = `${field}: ${this[field]}`;

        // increment the value for the future iteration
        i = i + 1;

        return {
          done: false,
          value,
        };
      },
    };
  },
};

for (d of user) {
  console.log(d);
}
```

What is the difference between a Map and an object?

The `Map` class has been introduced into JavaScript with ECMAScript 6.

It allows for storing and retrieving key-value pairs. In that sense, it is quite similar to objects `{}` . Objects are indeed often used as maps.

But there are several key differences.

1. Any value can be used as Map's key (including objects and functions), while object keys can only be strings or Symbols.

```
const map = new Map()
map.set("123", "a")
map.set(123, "b")
map.get("123") //=> "a"
map.get(123) //=> a
```

2. Maps are **iterable** (that is can be used in `for...of`), while objects are not (that is unless we specify the iterator).
3. Maps preserves the insertion order, which can be seen when using the `for...of` loop.
4. Maps don't have any keys by default. Objects inherit from `Object.prototype` , and thus they contain some keys already (for example, `toString` , etc.).

What is Set?

A `Set` class was introduced in ECMAScript 6/2015.

It is a data structure that is used to store unique values (no duplicates). No value can be stored more than once.

```
const s = new Set()
s.add(1) //=> Set(1) {1}
s.add(2) //=> Set(2) {1, 2}
s.add(1) //=> Set(2) {1, 2} (no changes)
```

Values are considered equal according to the semantics of === operator (with the exception of `NaN`).

`Set` provides some standard operations like `add` , `delete` , `has` , `size` (which is a property, not a method), and also `clear` to empty the entire set.

It also implements an iterator, so it’s possible to iterate over its items with `for...of` cycle.

The order of elements is preserved.

```
const s = new Set()
s.add("A")
s.add("B")
s.add("C")

for (el of s) {
  console.log(el)
}

//=> A
//=> B
//=> C
```

-> Learn more about Sets.

Ex. 9.1: Implement Set’s difference.

Implement the `diff` function which takes two `Set` s and returns a new `Set` that contains all the elements that are present in the first Set, but not in the second.

```
// example
diff(
  new Set([1, 2, 3, 4, 5]),
  new Set([[1, 3, 5, 7]])
)
// => { 2, 4 }
```

Go to solution →

Regular Expressions

How to create a regular expressions in JavaScript?

Regular expressions are implemented in many languages and serve as a way to match character combinations in strings.

There are two ways to create a regular expression object - one is to use the `RegExp` constructor, and the other is to use the regular expression literal.

```
// using the constructor
const re = new RegExp("p\d+")

// using the regex literal
const re = /p\d+/
```

Several methods on `String` (`match` , `matchAll` , `search` , `replace` , `replaceAll` , `split`) and `RegExp` itself (`exec` , `test`) are used with regular expressions.

How to check if a specific regex pattern is present in a string?

The easiest way to test if a string matches a regular expression is to use the `String.prototype.test` method.

The following pattern matches any string that has “java” in it, case-insensitive, followed by one or more characters.

```
const re = /java.+/i

re.test("Java") //=> false
re.test("JavaScript") //=> true
```

What RegExp flags do you know?

JavaScript regular expression can optionally include flags that modify its behavior in a certain way.

Here are some of the most commonly used ones.

The `i` flag makes the pattern case-insensitive:

```
"Bear is an animal".match(/bear/i)
// => ['Bear', index: 0, input: 'Bear is an animal', groups: undefined]
```

The `g` flag makes it search for all matches (globally).

```
"bear and bears".match(/bears?/g)
// => ['bear', 'bears']
```

The `m` flag is used when a search is applied to a multiline string. It modifies the behavior of `^` and `$` symbols making them consider each line separately.

Imagine that we want to find an all-digit line in a string.

```
const str = `
test
123
abc
`


str.match(/^\d+$/)
// => null
// because it matches against the whole string

str.match(/^\d+$/m)
// => ['123', index: 6, ...]
// works as expected
```

There are more flags, but they are used less often.

For the whole list refer to MDN.

Ex. 10.1: Remove extra spaces at the end of each line

Sometimes, we developers, forget to clean extra white spaces at the end of each line. It's hard to notice those since by default they are not visible in code editors.

Imagine, that as part of the tooling that you develop you want to clean all the extra spaces at the end of the lines. How do you that?

Go to solution →

Ex. 10.2: Remove extra semicolons at the end of the line

We have a string that looks like this

```
const myVar = 123;
console.log(myVar);;;
console.log(myVar * 5);;
```

As you see, there are multiple excessive semicolons. Using regular expression, how do we leave only one per each line?

Go to solution →

Ex. 10.3: Scrape flight numbers

Imagine that you program a web scraper. You need to extract all the flight numbers from the text. Every flight starts with "FN-" and then is followed by exactly 6 alphanumerical characters. For example, `FN-TE1GD2` , or `FN-0192XY` .

Write a regular expression to extract all the flight numbers from a multi-line string. Leave out the "FN-" part.

Go to solution →

Ex. 10.4: Functions to arrows

In JavaScript there are two ways to create a function - by using the `function` keyword and by using the arrow function syntax.

For some unclear reason you have decided that the later is aesthetically more pleasing then the former. You are now on your way to write a script that will go through all of your files, and replace functions with `arrow syntax`.

Here's an example of a file.

```
const file = `

function blah(arg) {
  // a bunch of stuff
  // all of these
  // I don't care
}

function foobar () {
  console.log("Hi")
}

function anotherOne(a, b) {
  // oops
  return a * 2 + b
}

`
```

Write a regular expression to update it.

Go to solution →

Asynchronicity

Explain how the event loops works.

JavaScript is single-threaded. But we can still run some code asynchronously, and the `event loop` is what makes it happen.

JavaScript can do only one thing at a time. But we can offload some tasks (e.g., `fetch`) to the browser. JavaScript itself doesn't make the fetching itself. Instead, it uses the browser API (`fetch`) to load the data and notify us when it's ready.

But when JavaScript runs code, it never stops until it finishes the task. So how can browser API return the data to the main thread?

Here comes the event loop.

It roughly looks like this.

```
while (true) {
  if (callStack.emoty()) {
    callStack.push(eventQueue.nextTask())
  }
}
```

Where the `call stack` tracks whatever task is currently executing. No new task can be started until the current task is finished. That is, until the call stack is empty.

The event queue is the queue of the tasks that need to be executed. Browser APIs can put tasks in that queue, and the event loop makes sure to put them into the main thread as soon as it has nothing to work on.

For example, let's consider the following code.

```
console.log("start")
fetch("/api/user").then((data) => {
  console.log('name', data.name)
})
```

```
console.log("finish")
```

The output will be:

```
start
finish
data { name: "John" }
```

Let’s see what happens step by step.

1. The call stack is empty, and the event queue is empty
2. `console.log("start")` - the call stack is busy running the main code
3. `fetch` - JavaScript asks the browser to perform a fetch and moves on
4. `console.log("finish")` - JavaScript reaches the end of the tasks, the call stack is empty
5. At some point, the browser finishes the fetching and puts the callback into the `event queue`
6. Event loop sees that the call stack is empty, which means it can take a new task from the event queue.
7. `console.log('name', data.name)` - the callback from the event queue goes to the call stack and gets executed

I recommend watching this video, which is one of the greatest and most straightforward explanations of the event loop.

Ex. 11.1: In which order will these messages be printed out?

```
console.log("1")
setTimeout(() => console.log("2"), 0)
console.log("3")
```

Go to solution →

What is callback hell?

JavaScript asynchronicity for a long time was built on the concept of a callback. A function that is invoked, after an asynchronous task has finished.

For example,

```
loadData(params, (res) => {
  console.log(res)
})
```

The `loadData` accepts a callback as the last argument, and hence let us run some code after the loading data is finished.

It is not uncommon to have many nested callbacks when one async task can only start working after another.

```
loadOrgData((org) => {
  loadEmployees(org.guid, (employees) => {
    const first = employees[0];
    loadPayslipData(first.id, (payslipData) => {
      // more code goes here
    })
  })
})
```

The nested callbacks kinda form a pyramid that tends to go further to the right.

That particular style of coding is very hard to manage and track the logic, hence the name “the callback hell”.

The callback hell can be solved in multiple ways. After the arrival of native Promises (link) and async/await, it’s not a problem anymore.

What is Promise?

A Promise refers to a value that will be available at some point in the future.

Native Promises were first introduced as part of ECMAScript 6, but before that were available in a form of third-party libraries, like q or bluebird.

An instance of `Promise` has a method `then`. It accepts success and error callback and can return a new promise. That makes the promises chainable.

What problem does it solve?

Consider the following example.

```
fetchUserByEmail("john@smith.com", (user) => {
  const { companyId } = user
  fetchCompany(companyId, (company) => {
    const { securityMapId } = company
    fetchSecurityMap(securityMapId, (sm) => {
      checkSomething(user, sm)
    }, (err) => {
      console.log("Error: ", err)
    })
  }, (err) => {
    console.log("Error: ", err)
  })
}, (err) => {
  console.log("Error: ", err)
})
```

For every fetch, we need to provide success and an error handler. It's easy to see how the more asynchronous actions is happening the more code shifts to the right making it harder to follow.

This style is often called "the callback hell" (add a link to the last quest).

The promises streamline the code flow and make it thus much more readable and easy to reason about.

```
fetchUserByEmail("john@smith.com")
  .then(user => {
    const { companyId } = user
    return fetchCompany(companyId)
  })
  .then(company => {
    const { securityMapId } = company
    return fetchSecurityMap(securityMapId)
  })
  .then(sm => {
    checkSomething(user, sm)
  })
  .catch((err) => {
    console.log('Error: ', err)
  })
```

The code doesn't have the tendency to shift to the right and is easily understood. Also notice, how the error case can be only handled once.

It gets even better with `async/await` syntax introduced as part of ECMAScript 2017.

```
  try {
    const user = fetchUserByEmail("john@smith.com")
    const { companyId } = user

    const company = await fetchCompany(companyId)
    const { securityMapId } = company

    const sm = await fetchSecurityMap(securityMapId)
    checkSomething(user, sm)
  } catch (err) {
    console.log('Error: ', err)
  }
```

Ex. 11.2: Promisify

Given an asynchronous function like the following:

```
loadSong(id, (song) => {
  // success callback
}, (err) => {
  // failure callback
})
```

Convert it to the one using a promise.

Go to solution →

How and when would you use Promise.all?

The `Promise.all` function accepts an array of promises and returns a single promise which gets fulfilled as soon as all promises are resolved, or as soon as one of the promises is rejected (in that case all other promises are ignored) - so called “fail-fast” behavior.

A typical use case for Promise.all is when we need to run multiple tasks in parallel (we don’t care in which order they run).

```
const promises = [
  fetch("/user1.json"),
  fetch("/user2.json"),
  fetch("/user3.json"),
]

Promise.all(promises).then(([user1, user2, user3]) => {
  console.log(user1);
  console.log(user2);
  console.log(user3);
})
```

What if one of the promises fails, due to for instance network problem? In that case, the promise is rejected, even if others are resolved successfully.

In case when we don't care if any of the promises rejects, we should use Promise.allSettled instead.

Ex. 11.3: Implement Promise.all

Implement a function `all` that works the same way as Promise.all does.

Go to solution →

What does Promise.allSettled do?

The `Promise.allSettled` function is a bit like Promise.all.

In a similar fashion, It takes an array of promises and returns a new promise which gets resolved as soon as all the input promises are resolved. If some of the promises are rejected (and this is the distinction from `Promise.all`), it still waits until all promises are settled and then resolves with either values or errors.

Compared to `Promise.all` , the values also contain a special `status` key which is either `fulfilled` or `rejected` .

Here's an example,

```
const promises = [
  Promise.resolve(1),
  new Promise((res) => {
    setTimeout(() => res(2), 1000);
  }),
  Promise.reject("ooh-oh"),
];

Promise.allSettled(promises)
  .then((ps) => console.log(ps));
```

Results in,

```
[
  { status: 'fulfilled', value: 1 },
  { status: 'fulfilled', value: 2 },
  { status: 'rejected', reason: 'ooh-oh' }
]
```

When to use Promise.any

`Promise.any` accepts an array of promises and returns a new promise. It fulfills as soon as any of the promises fulfills. If all of the supplied promises get rejected, then it gets rejected as well. Errors from all promises are provided in that case.

```
Promise.any([
  Promise.resolve(1),
  Promise.resolve(2),
  Promise.reject(3)]
)
  .then((res) => {
    console.log('res', res);
  })
  .catch((err) => {
    console.log('error', err);
  });

// => res 1
```

```
Promise.any([
  Promise.reject(1),
  Promise.reject(2),
  Promise.reject(3)]
)
  .then((res) => {
    console.log('res', res);
  })
  .catch((err) => {
    console.log('error', err);
  });

// => error [All promises were rejected]
// => { [errors]: [ 1, 2, 3 ] }
```

A typical use case for `Promise.any` is when you need to load a resource that has multiple mirrors. You don't know which one will gets loaded faster, so you start loading all of them.

```
const urls = [
  "https://example1.mywebsite.com/userdb.json",
  "https://example2.mywebsite.co.uk/userdb.json",
  "https://example3.mywebsite.co.jp/userdb.json",
]

const promise = Promise.any(
  urls.map(url => fetch(url))
).then(
  (res) => res.json()
).then(
  (users) => console.log(users)
)
```

In this case, it would be nice to also abort the fetches that came back later. See the corresponding question.

What does Promise.race do?

Similarly to `Promise.any`, `Promise.race` takes an array of promises. But unlike it, it tracks which of the promises settles first (either gets resolved or rejected).

```
Promise.any([
  Promise.reject(1),
  Promise.reject(2),
  Promise.reject(3)]
)
  .then((res) => {
    console.log('res', res);
  })
  .catch((err) => {
    console.log('error', err);
  });

// => error 1
```

Ex. 11.4: Longer then expected

Imagine you have functionality to load some data. When the data takes too long to load, you want to show a user a friendly UI banner saying “It takes a bit longer than usual, please bear with us...”.

Assume, that you have a function `fetchData` that returns a promise and a function `showBanner` .

Go to solution →

Async functions

The `async/await` syntax (introduced in ECMAScript 2017) allows working with asynchronous code, as it was synchronous.

An `async` function is declared with the `async` keyword. Within its body, we can use `await` keyword to wait for the result of a promise. Since ECMAScript 2022, it is also allowed to use `await` at the top level of modules.

A typical asynchronous code that looks like this

```
function main() {
  fetch("/api/blogs")
```

```
    .then((res) => res.json())
    .then((items) => console.log(items))
    .catch((err) => console.log("Error occurred: ", err))
}
```

becomes

```
async function main() {
  try {
    const res = await fetch("/api/blogs")
    const items = await res.json()
    console.log(items)
  } catch (err) {
    console.log(console.log("Error occurred: ", err))
  }
}
```

It greatly improves readability, as we don't deal with callbacks and promise chains, but rather have a normal declarative-looking code flow.

Another advantage of `async` functions is the better stack trace.

Consider this example,

```
function bar() {
    return Promise.resolve().then(() => { throw new Error("error!") });
}

function foo() {
  return bar()
}

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

// Error: error!
//    at <anonymous>:2:49
```

The stack trace is lost! Now consider the `async/await` version.

```
 function bar() {
   return Promise.resolve().then(() => { throw new Error("error!") });
 }

 async function foo() {
   bar();
 }

 await foo();

 //=> Uncaught (in promise) Error!
 // (anonymous) @ VM6497:2
 // bar @ VM6497:2
 // foo @ VM6497:6
```

In the last case, the stack trace is preserved.

How to abort an already fired `fetch` request?

Imagine a user downloading a very large file.

But the user changed her mind, she wants to click on a button to stop the loading. How to this?

The answer is to use the `AbortController` .

`AbortController` allows us to pass a special parameter to `fetch` called `signal` , and then use it afterward to cancel the request.

```
let controller;

function download() {
  controller = new AbortController();
  fetch(url, { signle: controller.signal; })
    .then((resp) => {
      console.log("Finished: ", resp);
    })
    .catch((error) => {
      console.error("Error: ", error.message);
    });
}

function cancel() {
  if (!controller) return;
  controller.abort();
}
```

This API is very simple and powerful.

Another userful example is that we can automatically abort a `fetch` request after a timeout using `AbortSignal.timeout` .

```
fetch(url, { signal: AbortSignal.timeout(2000) });

// in 2 sec
// => Uncaught (in promise) DOMException: The user aborted a request.
```

Various patterns and techniques

Ex. 12.1: Fibonacci numbers

The fibonacci numbers are a sequence of numbers where nth number is the sum of nth-1 and nth-2.

Here’s a recursive function to calculate those:

```
function fib(n) {
  console.log("calculating fib for ", n)
  if (n === 0) return 0
  if (n === 1 || n === 2) return 1
  return fib(n - 1) + fib(n - 2)
}
```

If you run it even for some small values like 20 or 30, you will notice that it runs terribly slow. Can you spot why and improve it?

Go to solution →

Ex. 12.2: memoize

Similarly, to the previous exercise, quite often it’s required to remember an output of a function so that the next time it is invoked with the same arguments we could reuse the value without going through the potentially expensive computation.

An example of this is React.memo which is used to remember the result of a component render.

Write a simple function that will take any arbitrary function and return a new one which will memoize its return values.

For simplicity, assume that the input function is a pure one (that is you don’t need to worry about `this` , and given the same arguments it always returns the same value, no side effects… link?).

Go to solution →

What is debounce and when to use it?

Debouncing is a simple technique that delays a particular action until it is stopped from being requested by the user.

Imagine a user typing a query into the search bar.

```
input.addEventListener("keydown", (event) => {
  const query = event.target.value;
  invokeSearchApi(query)
})
```

The problem with this approach is that every time a user types a character a network request is triggered. But if user wants to search for "xbox" he is not interested in searched for "x", "xb", "xbo". All of those are wasteful network calls.

So how do we make it so that only one network call is fired?

The answer is to delay the request until the user stops typing. And this is where debounce enters the stage.

The `debounce` function takes any function and a delay, and returns a new function, which we can call multiple times, but it will only call the original function once the delay time is passed from the last function call.

```
function test() { console.log("invoked!") }
const testDb = debounce(test, 500)

testDb()
testDb()
testDb()
// 500ms passes
// => invoked!
```

Ex. 12.3: Implement debounce

Implement the debounce function described in the previous question.

Go to solution →

Ex. 12.4: Implement throttle

The `throttle` function only allows executing a function at most once in a given time window (for example, at most once every 100ms).

Go to solution →

Ex. 12.5: Dig

Implement a `dig` function which takes an object with a nested structure and a path, and returns a value located in that position.

Example:

```
const user: {
  name: "John",
  phones: [
    { type: "mobile", num: "555-333-222" },
    { type: "mobile", num: "555-111-000" },
  ]
}

const secondPhone = dig(user, "phones[1].num")
// => "555-111-000"
```

Go to solution →

Ex. 12.6: A function that is only invoked once

Write a function that converts any function into another one, which is only invoked once (the first time), and all the consequent times it just returns the same value.

Go to solution →

Document Object Model

What is DOM?

The DOM (Document Object Model) is one of the Web APIs, a collection of programming interfaces that allows accessing and modifying a web page.

A web document in DOM is represented as a tree of elements (where each element is represented as an object with its properties and methods). The root of that tree is `document` .

How to access/query an element on a page?

The DOM API provides several ways to access an element on a page depending on what parameters we want to query.

Here are some of the most useful ones.

```
// by id - there can only be one
document.getElementById("myId");

// by class name
document.getElementsByClassName("btn");

// by tag name
document.getElementsByTagName("a")

// accessing by a CSS selector

// first match
document.querySelector("header a.logo")

// all matches
document.querySelectorAll(".main a")
```

How to modify elements on a page?

There are a whole bunch of small questions when an interviewer asks to perform a particular change on a page, most of which end up being changing a property or invoking a method on a corresponding DOM node.

The most popular questions are the following.

```
// change the title of the page
document.title = "My New Title"

// add a class to an element
const el = document.getElementById("myBtn")
el.classList.add("myClass")

// disable a button
const el = document.getElementById("myBtn")
el.disabled = true

// submit a form
const form = document.getElementById("myForm")
form.submit()

// create an element and append it to the page
const container = document.getElementById("cont")
const btn = document.createElement("button")
btn.textContent = "Click me"
div.append(btn)
```

How to hide/show an element

There are multiple ways to hide an element on a page.

First, we could toggle the `display` CSS property.

```
const btn = document.getElementById("myBtn")
btn.style.display = 'none'

// to show it back
btn.style.display = 'block'
```

Another way is to use the `visibility` property.

```
btn.style.visibility = "hidden"
btn.style.visibility = "visible"
```

However, in this case, the element continues to occupy space on the screen. It's just not visible anymore.

Finally, we can remove the element from the DOM altogether.

```
btn.remove()
```

Ex. 13.1: Replace links

You are given a web page with multiple links on it. Those URLs are relative, for example `/posts/my-post` , `/help/aboutus` .

Your task is to replace all those links with absolute URLs, so `/posts/my-post` becomes `https://mywebsite.io/posts/my-post` .

Go to solution →

What is event bubbling?

In modern browsers when we click on a button the event is fired on a button itself, but also on all the ancestors in a specific order.

First, it is fired on the outermost element (`body`) and then on every child as it goes down until it reaches the target element (a button that was clicked). This is called a `capture phase` .

Then, it is fired on the target itself, on its parent, on its parent's parent, and so on until it reaches the outermost element again. This is called a `bubbling phase` .

Note that while bubbling phase event listeners are triggered by default, to listen to the capture phase, we need to set the third parameter (`useCapture`) to `true` .

```
document.body.addEventListener("onclick", handler, true)
```

What are DOM events? Give some examples

The DOM API provides a way for us as a programmer to react to certain events happening in the system so that we could provide a good user experience.

It does this by allowing us to register callbacks for certain events.

There are many types of events, but the most popular are `click` (element clicked), `load` (page loaded), `keypress` (key pressed), etc.

In order to register a listener, we invoke the `addEventListener` method.

```
button.addEventListener('click', function() {
  console.log("Button clicked");
});
```

We can also remove the listener.

```
element.removeEventListener(event, callback)
```

Note, that in this case callback must be the same function that we use to add the event listener in the first place.

The `callback` function also accepts the `event` object which contains some useful information about the event.

For example, the `keypress` event contains information about the key pressed.

```
textarea.addEventListener("keypress", (event) => {
  console.log("You pressed", event.key)
})
```

It also contains some useful methods. For example, here's how we can cancel the form submission (if we want to send a request using ajax).

```
form.addEventListener("submit", (event) => {
  event.preventDefault()
  // now process the form and sen
})
```

What is the difference between the window `load` and the document's `load` events?

The `window.onload` event is fired when the whole page is loaded including images, styles, fonts, etc.

The `document.onload` only fired when the page itself loaded (the document), prior to images, styles, and other resources.

Error Handling

Explain `try..catch` statement

The `try...catch` statement is used when executing code that potentially can throw an exception. It allows a programmer to make a decision in each case depending on what kind of exception was thrown.

A typical `try...catch` statement is comprised of the obligatory `try` block, and then optional `catch` and `finally` blocks (though at least of them should be present).

```
try {
  // code that can throw
} catch (e) {
  if (e instanceof MyException) {
    // handle MyException
  } else if (...) {
    // ...
  }
} finally {
  // run in any case
}
```

The `try` block contains regular code that can potentially throw an exception.

The `catch` block contains the code that specifies what to do in case an exception has been thrown.

The `finally` block always executes before the last control flow statement in either `try` or `catch` block, no matter what.

How to throw an exception?

There's a special `throw` statement that allows a user to throw an exception.

We can throw `Error` objects

```
try {
  throw new Error("my_error");
} catch (e) {
  console.log("err:", e);
}

//=> err: Error: my_error
//=>    at Object.<anonymous> (/.../index.js:2:9)
//=>    at Module._compile (node:internal/modules/cjs/loader:1159:14)
```

But we also throw plain strings…

```
try {
  throw "my_err";
} catch (e) {
  console.log("err:", e);
}

//=> err: my_err
```

…numbers…

```
try {
  throw 42;
} catch (e) {
  console.log("err:", e);
}

//=> err: 42
```

…and even simple objects

```
try {
  throw { my: "obj" };
} catch (e) {
  console.log("err:", e);
}

//=> err: { my: 'obj' }
```

Ex. 14.1: What will be printed on the screen?

```
const res = run();
console.log("res", res);

function run() {
  try {
    return 1;
  } finally {
    console.log("finally");
  }
}
```

Go to solution →

Ex. 14.2: What will be printed on the screen - 2

```
try {
  throw new Error("my_error");
} catch (e) {
  console.log(e.message);
  throw e;
} finally {
  console.log("finally");
}
```

Go to solution →

Ex. 14.3: What will be printed on the screen - 3

```
function crash() {
  throw new Error("booom");
}

function getRes() {
  try {
    crash();
    return 1;
  } catch (e) {
    return 2;
  } finally {
    return 3;
  }
}

const res = getRes();
console.log("res", res);
```

Go to solution →

Solutions

Ex. 3.1

The output will be: `undefined, Alex, John` .

Because of the hoisting the declaration of that variable goes to the top, and thus we can rewrite it like this

```
var name;
console.log(name);

name = "Alex";
console.log(name);

name = "John";
console.log(name);
```

← Go back.

Ex. 3.2

It will print `undefined` .

First of all, within the `run` function, the global `myVar` is shadowed by the local variable, so the global value doesn't matter at all.

The declaration of the variables comes after accessing it. But due to hoisting, it is hoisted up to the top of the function and initialized with `undefined` .

← Go back.

Ex. 3.3

Output:

```
undefined
10
```

Explanation: Before JavaScript starts executing the code, the `a` variable is hosted and initialized with `undefined` .

The first time `print` is called, the value is still `undefined` .

Then JavaScript adds a timeout function and continued executing the code setting `a` to `10` .

Next, in 1 second, `print` is executed again and this time the variable is already set 10, hence the result.

← Go back.

Ex. 3.4

This time we get an exception `Uncaught ReferenceError: a is not defined` .

This happens because `const` declarations are not initialized, the attempt to access it before the initialization results in an exception.

← Go back.

Ex. 3.5

Output:

```
3
3
3

0
1
2
```

Explanation.

JavaScript is single-threaded. It never stops until it finishes executing a given chunk of work.

So the cycle finishes first, leaving the value of `i` equal to `3`, and only after that `console.log` starts.

All three anonymous functions are now closed on the same value of i which is by the time of invocation is `3`.

The second loop is different. This time we used `let` to declare the variable. Because, `let` declarations provide a block-scoped variable, every anonymous function we put in the timeout is now bound to its own variable `j`, hence the result.

← Go back.

Ex. 5.1

When you click any button, it says "I'm a button #4".

We created those three handlers, closing over the same variable `i`. By the time we click the button, the cycle has already finished, and the value of `i` is `4`.

So how do we fix it?

The easiest way is to change `var i` to `let i`. The `let` keyword creates a variable with a block scope (as opposed to function scope). That means that every iteration gets its own new block-scoped `i` variable and thus will alert a correct number.

Before the introduction of block scoping, the way to fix it was to wrap the listener in an anonymous function that gets immediately executed, the so-called immediately-invoked function expression (see the IIFE question).

```
for (var i = 1; i <= 3; i++) {
  const btn = document.getElementById(`btn${i}`);
  (function(num) {
    btn.addEventListener("click", () => alert(`I'm button #${num}`));
  })(i)
}
```

Introducing a function creates its own scope and thus eliminates the problem.

← Go back.

Ex. 5.2

```
function tree(...args) { // rest syntax to capture all arguments in an arra
  return args
    .reverse() // reverse array, because last args should be inside
    .reduce((acc, tag) => { // using reduce to build the string
      return `<${tag}>${acc}</${tag}>`
    }, "")
}
```

← Go back.

Ex. 6.1

```
function curry(fn, arity) {
  // arity is the number of arguments the function accepts
  arity = arity || fn.length

  // return a new function
  return function(...args) {

    // check if we have all arguments...
    if (args.length < arity) {

      // not enough
      // return a curried version (recursion!)
      // of a function which accepts the rest
      // of parameters
      function f(...rest) {
        return fn(...args, ...rest)
      }
      return curry(f, arity - args.length)
    } else {
      // enough arguments
      // return the result
      return fn(...args)
    }
  }
}

// How to use it
const sum = curry((a, b) => a + b)
const add1 = sum(1)
add1(10) //=> 11
```

← Go back.

Ex. 6.2

We can easily use `reduce` if we only want to flatten the array one level deep.

```
function flatten(arr) {
  return arr.reduce((acc, el) => {
    if (Array.isArray(el)) {
      return [...acc, ...el]
    } else {
      return [...acc, el]
    }
  }, [])
}
```

This solution only works for one level deep. If we use it on arrays with multiple nesting, it will fail.

```
flatten([1, [2, [3]]]) //=> [1, 2, [3]]
```

But we can quickly transform our functions to support deep nesting if we add a bit of recursion.

```
function flatten(arr) {
  return arr.reduce((acc, el) => {
    if (Array.isArray(el)) {
      // using recursion here!
      return [...acc, ...flatten(el)]
    } else {
      return [...acc, el]
    }
  }, [])
}

flatten([1, 2, [3, [4, 5]]]) //=> [1, 2, 3, 4, 5]
```

Note that the `Array.prototype.flat()` function takes an optional argument - how many levels deep we want that array to be flatten.

← Go back.

Ex. 6.3

Again, we will use the almighty `reduce` in our solution.

```
function compose(...fns) {
  return (arg) => {
    return fns
      // functions apply from right to left,
      // hence reverse
      .reverse()
      // the last result become an input arg
      // for the next function
      .reduce((res, fn) => fn(res), arg)
  }
}
```

← Go back.

Ex. 7.1

A deep freeze is a regular freeze invoked recursively for every nested property.

```
function deepFreeze(obj) {
  const keys = Object.getOwnPropertyNames(obj);
  for (const k of keys) {
    const v = obj[k]
    if (v !== null && typeof v === "object") {
      deepFreeze(v)
    }
  }
  return Object.freeze(obj)
}
```

← Go back.

Ex. 8.1

There are multiple ways to implement this function.

```
function uniq(arr) {
  return arr.reduce((acc, el) => {
    if (acc.indexOf(el) > -1) {
      return acc
    } else {
      return [...acc, el]
    }
  }, [])
}
```

Here we are using `reduce` to iterate through elements of the array, and collect the items into an new array, but only if it doesn't already have that element (using `indexOf`).

Another way to do the same is to use **Set**. Because a Set by definition can't contain equal items, we can use it to filter our array.

```
function uniq(arr) {
  const set = new Set(arr)
  return [...set]
}
```

← Go back.

Ex. 8.2

There are multiple ways to do this. The most elegant would be to use `reduce` .

```
function fromEntries(pairs) {
  return pairs.reduce((acc, el) => {
    const [key, value] = el;
    acc[key] = value;
    return acc;
  }, {})
}
```

And here's a slightly less readable one-line solution.

```
function fromEntries(pairs) {
  return pairs.reduce((acc, [k, v]) => (acc[k] = v, acc), {})
}
```

In fact, Object.fromEntries was introduced into the spec as part of EC-MAScript 2019.

← Go back.

Ex. 8.3

Output: `C`

This unexpected result is in fact quite logical if we remember that object keys can only be strings (and Symbols, but that's not relevant).

JavaScript will silently convert any object passed as a key to a string using `toString()` method, which gives us `[object Object]` .

This is why both "student" objects used as a key resolve to the same key `[object Object]` , and hence the latest assignment of `C` overwrites the previous one.

← Go back.

Ex. 8.4

```
function rotate(arr, n) {
  let res = [...arr]; // shallow copy
  for (let i = 0; i < n; i++) {
    const last = res.pop()
    res.unshift(last)
  }
  return res
}
```

← Go back.

Ex. 8.5

There are multiple ways to solve it. The most straightforward is to revert a string and check if they are the same.

```
function anagram(str1, str2) {
  return str1 === reverse(str2)
}
```

Now, how to implement a `reverse` function?

One way is to go through each character and add it into a new array of characters, and then join it to build a string.

```
function reverse(str) {
  const chrs = []
  for (let i = str.length - 1; i >= 0; i--) {
    chrs.push(str.charAt(i))
  }
  return chrs.join("")
}
```

← Go back.

Ex. 8.6

We will iterate over the elements of an array, and swap them.

```
function reverse(arr) {
  const len = arr.length
  for (let i = 0; i < len / 2; i++ ) {
    const tmp = arr[i]
    arr[i] = arr[len - i - 1]
    arr[len - i - 1] = tmp
  }
}
```

```
const b = [1, 2, 3, 4, 5]
reverse(b)
console.log(b) //=> [5, 4, 3, 2, 1]
```

← Go back.

Ex. 8.7

```
function filter(arr, f) {
  // base case
  if (arr.length === 0) return [];
  // let's split the array taking the first
  // and the rest elements
  const [head, ...tail] = arr;
  if (f(head)) {
      // head should be in the result
      return [head, ...filter(tail, f)]
  } else {
      // we only need to filter
      // the rest
      return filter(tail, f)
  }
}
```

← Go back.

Ex. 8.8

We are going to use a very basic algorithm: iterate over each element of an array and swap it with another random element.

```
function shuffle(arr) {
  // create a shallow copy
  const newArr = [...arr]

  const length = newArr.length

  for (let i = 0; i < length - 1; i++) {

      // random index
      const rand = randIndex(i + 1, length - 1)

      // swap elements
      const val = newArr[rand]
      newArr[rand] = newArr[i]
      newArr[i] = val
  }

  return newArr
}

// a helper function that returns a random number
// between lower and upper (including both ends)
function randIndex(lower, upper) {
    return lower + Math.floor(Math.random() * (upper - lower + 1))
}
```

← Go back.

Ex. 10.1

Let's say we have a multi-line string.

```
const str = `
  function sum(a, b) {<s><s><s>
    return a + b;<s><s><s>
  }<s>
`
```

I use `<s>` to refer to a space here, because otherwise you would not see it.

In order to match one or more whitespaces at the end of the line, we will need a regex like `/\s+$/` , where

- `\s` is a meta character refered to any whitespace character (that is space, tab, and others).
- `+` is the same as `{1,}` which means we expect this pattern to appear one or more times.
- `$` refers to the end of the line

Finally, same as in the last exercise we'll need `gm` flags for multi-line global match.

```
console.log(str.replaceAll(/\s+$/gm, ""))
```

← Go back.

Ex. 10.2

To match 2 or more semicolons at the end of the line we need a regexp like this `/;{2,}$/` . Here `{2,}` means “two or more”, and `$` refers to the end of the string.

Now because we are dealing with multi-string here, we need to add the `m` flag. Otherwise it will only watch semicolons at the end of the whole string (ignoring the fact that our string consists of multiple lines).

Finally, we will need the `g` flag as there are more than one occurrences.

```
const str = `
  const myVar = 123;
  console.log(myVar);;;
  console.log(myVar * 5);;
`


console.log(str.replaceAll(/;{2,}$/gm, ";"))
```

← Go back.

Ex. 10.3

Assuming, we have a string like this,

```
const str = `The flight number FL-76AXB2 is 20 minutes late being late. The
```

Let’s extract the flight numbers using capturing groups with `matchAll`
.

```
const re = /FL-([\d\w]{6,6})/g;
const match = str.matchAll(re);
for (m of match) {
  console.log(m[1])
}

//=> "76AXB2"
//=> "GHADE3"
```

Here,

- `[\d\w]` means any numerical or alphabetical character
- `{6,6}` exactly 6 characters
- `()` used to capture a group of symbols (in our case everything after "FL-")

← Go back.

Ex. 10.4

If you thikn about it, all we have to do here is to replace the first line of each function.

For example, we want to replace `function blah(arg)` with `const blah = (arg) =>` .

Let's do it step by step.

1. It starts with `function` keyword
2. One or more spaces can follow it `\s+`
3. Then goes the name of the function that we must remember `(\w+)` . We put it into the rounds brackets `()` - that's a capturing group - so that we can then use that value in the replace string as `$1`
4. Then go zero or more spaces `\s*`
5. Then go the arguments starting with `\(` and ending with `\)` (brackets are escaped).

6. Everything inside the brackets we need to remember so we put it into the round brackets again `([^(^)])` and the value will be available to us as `$2`

And that's really it. Putting it all together we have

```
const re = /function\s+(\w+)\s*\(([^(^)]*)\)/g
```

We also use the `g` flag, cause we want to replace all occurences, and not just the first one.

Now we are just going to use `replace` .

```
const result = text.replace(re, 'const $1 = ($2) => ');
console.log(result);
```

← Go back.

Ex. 11.1

The messages will appear in the following order:

```
1
3
2
```

`setTimeout` takes a task and puts it into the task queue after a specified period. Since we used 0 as a timeout period, it does this without any delay.

Still, the event loop now needs to wait until the current task is finished executing (that is running the code till the end of the file), and only after that it can pick up this task from the task queue and execute it. Hence, "3" prints out the last.

← Go back.

Ex. 11.2

```
function promisify(fn) => {
  return function(...args) {
    return new Promise((resolve, reject) => {
      fn(...args, (data) => {
        resolve(data)
      }, (err) => {
        reject(err)
      })
    })
  }
}

// Examples of usage
const timeout = promisify((ms, onSuccess) => {
  setTimeout(ms, onSuccess)
})

await timeout(5000)
```

← Go back.

Ex.11.3

In order to implement it, we will simply need to attach a resolver to each promise and count the number of successful promises. As soon as all are resolved, we can resolve our big promise with the values. As soon as one of them rejects, we reject the promise.

```
// accept an array or promises
function all(promises) {
  // return a new single promise
  return new Promise((res, rej) => {
    const length = promises.length;
    // tracking the number of resolved promises
    let resovledCount = 0;

    // tracking the resolved values
    const values = [];

    promises.forEach((p, ind) => {
      p.then(
        (val) => {
          // it's important to put the result
          // into corresponding slot in the array
          values[ind] = val;
          resovledCount++;
          // all good - resolve
          if (resovledCount === length) {
            res(values);
          }
        },
        (err) => {
          // there's an error - reject
          rej(err);
        }
      );
    });
  });
}
```

And we can now use it like this:

```
  const promises = [
    Promise.resolve(1),
    new Promise((res) => {
      setTimeout(() => res(2), 1000);
    }),
    Promise.reject("ooh-oh"),
  ];

  all(promises)
    .then((ps) => console.log(ps))
    .catch((err) => {
      console.log("error", err);
    });

    // error, ooh-oh
```

← Go back.

Ex.11.4

The idea is that we can use a timeout of some time and use `Promise.race` to track if the request takes longer.

```
const timeout = (val, period) => {
  return new Promise((res, rej) => {
    setTimeout(() => res(val), period)
  })
}

Promice.race([
  fetchData(),
  timeout("TOO_LONG", 5000)
]).then((res) => {
  if (res === "TOO_LONG") {
    showBanner();
  }
})
```

← Go back.

Ex. 12.1

It’s easy to see (with the help of that console log statement) that fib function takes this long because it calculates the same value over and over again.

In order to calculate fib(10), it needs to calculate fib(9) and fib(8), and to calculate fib(9) it needs fib(8) and fib(7). Already we see that fib(8) is calculated twice, and gets worse the further down that pyramid we go.

The solution is simply to remember the fiboncacci numbers we already calculated.

```
const cache = {}

function fib(n) {
  console.log("calculating fib for ", n)
  if (n === 0) return 0
  if (n === 1 || n === 2) return 1

  // checking if we already calculated it
  if (cache[n] !== undefined) {
    return cache[n]
  }

  // if not, calculate
  const val = fib(n - 1) + fib(n - 2)

  // and put to cache
  cache[n] = val
  return cache[n]
}
```

← Go back.

Ex. 12.2

```
function memoize(fn) {
  const cache = {}

  function cacheKey(args) {
      return args.join("///")
  }

  return (...args) => {
      const key = cacheKey(args)
      if (cache[key] !== undefined) {
          return cache[key]
      }
      const val = fn(...args)
      cache[key] = val
      return val
  }
}
```

That function can be improved further by being able to accept a custom cacheKey function. For example, see the lodash implemntation.

← Go back.

Ex. 12.3

```
const debounce = (fn, delay) => {
    // store the timer
    let timer
    return function(...args) {
        // remember "this"
        const context = this;

        // clear the old timer if any
        if (timer) clearTimeout(timer);
        // and add a new one
        timer = setTimeout(() => fn.apply(context, args), delay)
    }
}
```

← Go back.

Ex. 12.4

```
function throttle (fn, limit) {
    // allow initially
    let wait = false;

    // return a new function
    return function (...args) {
        // if we don't wait, let's run it
        if (!wait) {
            const ctx = this
            const val = fn.apply(ctx, args);

            // now let's wait
            wait = true;
            setTimeout(() =>. {
                // now we can run again
                wait = false;
            }, limit);

            // don't forget to return result
            return val;
        }
    }
}
```

← Go back.

Ex. 12.5

To solve it, we will first break the path using `.` , `[` , and `]` delimiters. And after that iterate over it with `reduce` , digging one level deeper with each iteration.

```
function dig(source, path) {
  if (!path) {
    return undefined;
  }

  // split path: "param[3].test" => ["param", 3, "test"]
  const parts = path
    .split(/\.|\]|\[/)
    .filter(x => !!x);

  return parts.reduce((acc, el) => {
    if (acc === undefined) {
      // return early
      return undefined;
    }

    return acc[el]
  }, source);
}
```

← Go back.

Ex. 12.6

Here we are going to use the clousure. Our function will create a new closure (`value`), and will only call the original function if the value is not set yet.

```
function once(fn) {
  // a closure which is always
  // accessible from our function
  let value = undefined
  return function(...args) {
      if (value !== undefined) {
        return value
      }
      const ctx = this
      value = fn.apply(ctx, args)
      return value
  }
}
```

← Go back.

Ex. 13.1

```
// lets find all the anchors
const links = document.getElementsByTagName("a")

// iterate over elements
for (link of links) {
  // update the href
  link.href = `https://mywebsite.io${link.href}`
}
```

← Go back.

Ex. 14.1

It will print:

```
finally
res 1
```

because `finally` blocks always runs before whatever the last control flow statement runs in the `try` or `catch` block.

← Go back.

Ex. 14.2

The output:

```
my_error
finally
.../index.js:5
  throw e;
  ^


Error: my_error
    at Object.<anonymous> (.../index.js:2:9)
    ...
```

Similarly to the previous example, `finally` block will always run, and will always run before the last control statement in either `try` , or `catch` - in this case before re-throwing the exception in the catch block.

← Go back.

Ex. 14.3

The output:

```
res 3
```

If there's a `return` statement in `finally` , it will always get priority over `returns` in `try` and `catch` .

← Go back.