JavaScript Interview 2022

Questions with Answers

Ianis Triandafilov

Contents

Functions

How to declare a function?

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 returns `undefined` .

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

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

Thanks to `hoisting` , a function declared that way can be invoked in code before the declaration.

```
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. Functions in JavaScript are first-class citizens (like any other values, they can be passed around).

```
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')
}
```

Function expressions also 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, this is not secure (dynamical code execution) and not fast. Declaring functions this way 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.

They have several important properties:

1. They can not be used as a constructor, and as such, they don't have the `prototype` attribute.

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

2. They don't track **this** context, and are un-bind-able:

```
// regular function
function fn1() {
  return this
}
const obj = { hello: 'world' }
fn1.bind(obj)() // => {hello: 'world'}

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

3. They don't have access to special `arguments` parameter.
4. They can be used without curly braces, in which case the last expression is returned (no need to `return` explicitly):

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

// BUT! return still required when used with curly braces
[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`, they can be anonymous, and they don't get hoisted.

What is a closure?

Then a function is executed in a different context, it can still access variables from the initial scope (declaration scope). This is called a **closure**.

```
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 - `i` . Then it creates and returns a new anonymous function. That function can be used outside of its initial scope, and still have access to `i` .

Ex. 5.1: Multiply by

Create a **higher-order function** `multiplyBy` , which takes a number and returns a new function.

For example,

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

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

Go to solution →

Ex. 5.2: What is wrong with the following code?

There are 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 →

Ex. 5.3: Field reader

Create a `fieldReader` function that takes a field name, and return a new function that can be applied to an object.

```
const getName = fieldReader('name')
getName({ name: "Alice" }) //=> "Alice"
```

Go to solution →

What is an IIFE?

An **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 any global variables (see scoping).

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

// user is not available here
```

The other use-case was popular 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 dots `...`) 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) { }
```

It's also impossible to have multiple rest params.

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

Ex. 5.4: 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

Normally we use parenthesis to invoke a function.

```
myFunc()
```

But this is not the only way.

`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]
```