Ten JavaScript Theory Questions

These are popular conceptual questions asked in JavaScript interviews.

There are countless questions your interviewers may ask when it comes to how JavaScript works. The idea behind asking these questions is to assess whether you have recent experience in writing JavaScript code.

Some more clueless interviewers tend to ask you for lexical knowledge and edge cases that you could simply look up online. I, personally, think that this signals a lack of competence from the end of my interviewers, and I tend to start getting concerned whether I am in the right place.

Usually, you are not allowed to use Google to find the answer, and you have to answer on the spot.

Question 1

Is JavaScript a “pass by value” or a “pass by reference” type of language when it comes to passing function arguments?

Answer: JavaScript passes function arguments by value. In case we pass an array or an object, the passed value is a reference. This means you can change the contents of the array or the object through that reference.

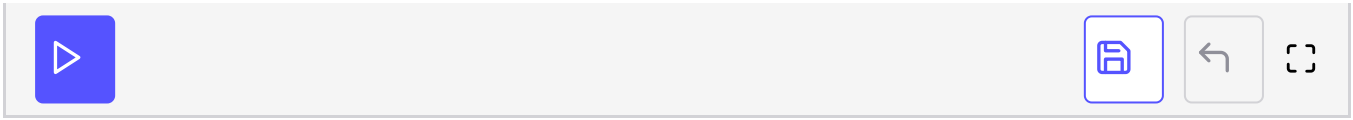
Read this article on value and reference types for more details.

Question 2

Study the following code snippet:

```
let user1 = { name: 'FrontendTroll', email: 'ihatepopups@hatemail.com' };
let user2 = { name: 'ElectroModulator', email: 't2@coolmail.com' };

let users = [ user1, user2 ];

let swapUsers = function( users ) {
    let temp = users[0];
    users[0] = users[1];
```

```
    users[1] = temp;
    return users;
}


let setCredit = function( users, index, credit ) {
    users[ index ].credit = credit;
    return users;
}

console.table( swapUsers( [...users] ));
console.table( setCredit( [...users], 0, 10 ) );
console.table( users );
```



What does `[...users]` do? What is printed to the console?

Answer: `[...users]` makes a *shallow copy* of the `users` array. This means we assemble a brand new array from scratch. The elements of the new array are the same as the elements of the original array.

However, each element is an object in each array. These objects are *reference types*, which means that their content is reachable from both arrays. For instance, modifying `[...users][0].name` results in a modification in `users[0].name`.

Let's see the printed results one by one.

In the first console table, we expect the two elements to be swapped. This change left the `users` array intact, because none of its elements were modified.

```
console.table( swapUsers( [...users] ) );
```

This is printed on the screen:

(index)	name	email
0	“ElectroModulator”	"t2@coolmail.com"
1	“FrontendTroll”	"ihatepopups@hatemail.com"

Let's see the second result. We shallow copied the elements of the `users` array again, and added a credit of `10` to the first user. The order of the users is still `FrontendTroll` before `ElectroModulator`, as the order of the elements of the `users` array were not changed by `swapUsers` due to shallow copying. `FrontendTroll` receives ten credits in the cloned array. As we only shallow copied the `users` array, this credit will make it to the original array as well.

```
console.table( setCredit( [...users], 0, 10 ) );
```

The above line of code results in:

(index)	name	email	credit
0	“FrontendTroll”	"ihatepopups@hatemail.com"	10
1	“ElectroModulator”	"t2@coolmail.com"	

Based on the explanation, the third `console.table` will be identical with the second, including the credit of `10`:

```
console.table( users );
```

0	“FrontendTroll”	"ihatepopups@hatemail.com"	10
1	“ElectroModulator”	"t2@coolmail.com"	

Read more on shallow and deep cloning in my article Cloning Objects in

JavaScript.

You can execute and visualize this code on pythontutor.com.

Question 3

Does the code in Question 2 conform to the principles of pure functional programming?

Answer: No. A function is pure if and only if it is free of side-effects. `setCredit` modifies a field in the global object `users[0]` as a side-effect of the function execution.

Read the first few paragraphs of the article Functional and Object Oriented Programming with Higher Order Functions for more details.

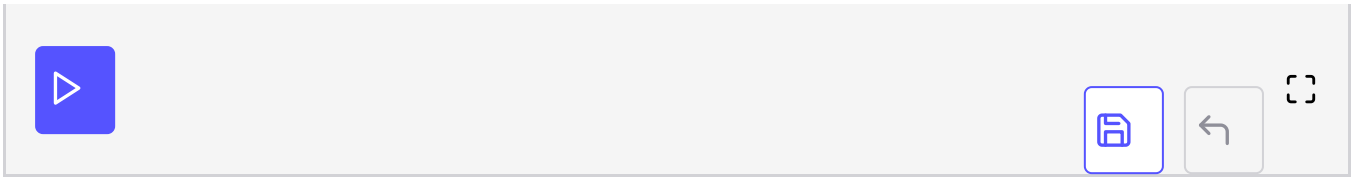
Question 4

How can we prevent `setCredit` from modifying the original array?

Answer: Instead of `[...users]`, use deep cloning. As we only work with data that can be represented using a finite JSON string, we can stringify, then we can parse our original object to get a deep copy.

For example:

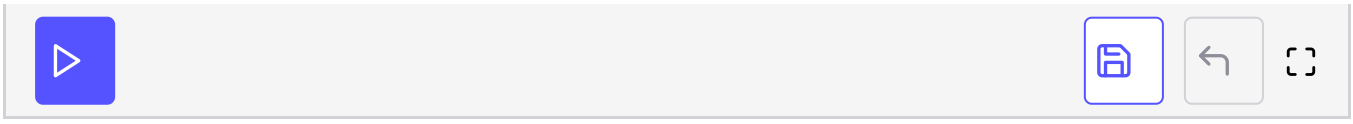
```
const user1 = { name: 'FrontendTroll', email: 'ihatepopups@hatemail.com' };
const user2 = { name: 'ElectroModulator', email: 't2@coolmail.com' };

const users = [ user1, user2 ];

const deepClone = function( o ) {
    return JSON.parse( JSON.stringify( o ) );
}

let swapUsers = function( users ) {
    let temp = users[0];
    users[0] = users[1];
    users[1] = temp;
    return users;
}

let setCredit = function( users, index, credit ) {
    users[ index ].credit = credit;
    return users;
}

console.table (swapUsers(deepClone(users)));
console.table (setCredit(deepClone(users), 0, 10));
console.table(users);
```



Question 5

What is wrong with the following code?

```
let sum = (...args) => args.reduce( (a,b) => a+b, 0 );
let oneTwoThree = [1, 2, 3];

let moreNumbers = [ ...oneTwoThree, 4 ];
console.log( sum( ...moreNumbers, 5 ) );

let [...lessNumbers, ] = oneTwoThree;
console.log( sum( ...lessNumbers ) );
```



Answer: In ES6, `...` denotes both the Spread operator and rest parameters.

In the first line, `...args` is a rest parameter. The rest parameter has to be the last parameter of the argument list, symbolizing all the remaining arguments of the function. Given there are no more arguments left after `...args`, the rest parameter is in its correct place.

The Spread operator spreads its elements into comma separated values. Therefore:

```
moreNumbers = [...[1, 2, 3], 4] = [1, 2, 3, 4]
```

because `...[1, 2, 3]` becomes `1, 2, 3`.

In a function call, the spread operator can also be used:

```
sum(...[1, 2, 3, 4], 5) = sum(1, 2, 3, 4, 5)
```

because `...[1, 2, 3, 4]` becomes `1, 2, 3, 4`.

Inside the *destructuring* assignment,

```
let [...lessNumbers, ] = oneTwoThree;
```

`...lessNumbers` is a rest parameter. It has to stand at the very end of the array. Given there is a comma after the rest parameter, we expect the code to throw a `SyntaxError`, because the rest parameter has to be the last element of the array. Due to the syntax error, the last line of the code cannot be executed.

Question 6

Consider the following function:

```
let printArity = function() {
    console.log( typeof arguments, arguments.length );
    console.log( arguments.pop() );
}
printArity( 1, 2, 3, 4, 5 );
```

Determine the output without running the code!

> **Header**
>
> *[1,2,3].pop()*
>
> *3*

Solution:

`arguments` is an object. It is not an array! See the Mozilla documentation for more details.

For some reason, `arguments` has a `length` property, and it equals the number of arguments of the function, which is `5`.

Given that `arguments` is not an array, the `Array` prototype method `pop` is not available. Therefore, after printing `object 5`, the code throws a `TypeError`, because `arguments.pop` is not a function.

Whenever you can, use rest parameters instead of the `arguments` array. You can read more details on the relationship between the arguments array and rest parameters in ES6 in Practice.

Question 7

Why isn't `0.1 + 0.2` equal to `0.3`?

Answer: This is strictly speaking a computer science question and not a JavaScript question. All you need to know is that JavaScript uses floating point arithmetics, where a number is represented using a finite number of bits.

In this specific example, `0.1 + 0.2` adds up to `0.30000000000000004` due to floating point arithmetics.

Question 8

How can we retrieve a DOM node collection of all `div` elements on a website? How can we retrieve a DOM node collection of all `div` elements having the class `row-fluid` on a website?

Answer: It is important that we do not need jQuery for this purpose. If your answer is based on jQuery, please think again, because reliance on jQuery in 2017 is not always optimal.

Regarding the first question, you can either use `document.getElementsByTagName` or `document.querySelectorAll`. The latter solution uses the same selectors as jQuery does.

```
document.getElementsByTagName( 'div' )
//HTMLCollection(274) [...]

document.querySelectorAll( 'div' )
//HTMLCollection(274) [...]
```

Regarding the second question, you *could* filter the DOM node collection obtained using `getElementsByTagName`:

```
Array.from(
    document.getElementsByTagName( 'div' )
).filter(
    x => x.className.split( ' ' ).indexOf( 'row-fluid' ) >= 0
);
```

We have to know how to convert a DOM node collection to an array. We have to understand how to access the class list of a DOM node, and how `indexOf` works in case of arrays. High risk, low reward solution. Let's figure out

something simpler.

As you have just read, `document.querySelectorAll` can process complex selectors. `div.row-fluid` will do all the filtering for you:

```
document.querySelectorAll( 'div.row-fluid' )
```

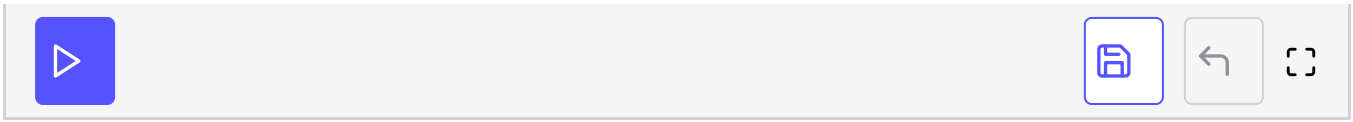
Strictly speaking, the first solution is wrong, because the result is not a `NodeList`, but an array of two nodes.

Question 9

Swap the contents of two variables without introducing a third variable!

Solution: We can use destructuring to accomplish the desired result. Example:

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



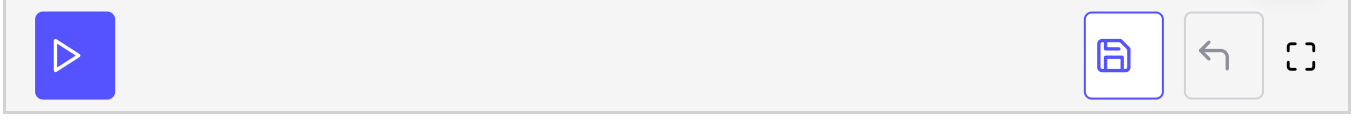
See this blog post for more exercises on destructuring.

Question 10

Write a JavaScript function that determines if a string consists of hexadecimal digits only. The digits `A`-`F` can either be in lower case or upper case.

Solution: The easiest way is to formulate a *regular expression*.

```
hexString => /^[0-9a-fA-F]+$/.test( hexString );
```



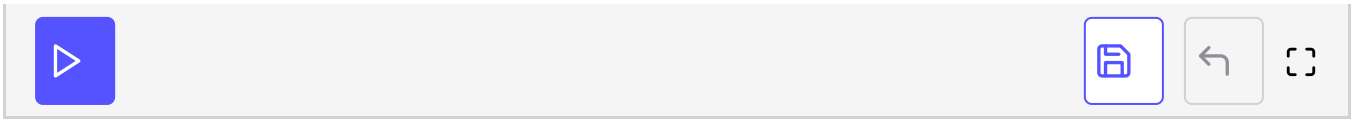
Explanation:

- `^` at the beginning specifies that the string has to start with the specified sequence `[0-9a-fA-F]+`

- `$` at the end specifies that the string has to end with the specified sequence `[0-9a-fA-F]+`
- `[0-9a-fA-F]` is one arbitrary character, which is either a digit or a letter between `a` and `f`, or a letter between `A` and `F`. Note that the solution `[0123456789abcdefABCDEF]` is equally acceptable, just longer.
- `+` specifies that you can repeat the character `[0-9a-fA-F]` as many times as you want, given that you have provided at least one character.

Common sense dictates that a problem with the above solution is that it allows the first digit to be `0`, which is not possible. Notice the task description didn't ask us to take care of this case, so we can simply omit it. However, if we want to go the extra mile, we could write:

```
let checkHexNum = hexString =>
    /^[1-9a-fA-F][0-9a-fA-F]*$/.test( hexString ) ||
    hexString == '0';
```



For more details, check out my article on Regular Expressions in JavaScript.

If you want to avoid using regular expressions, you can write a simple loop. Pay attention to the boolean condition though.

```
let checkHexNum = hexString => {
    for ( let ch of hexString ) {
        if ( '0123456789abcdefABCDEF'.indexOf( ch ) === -1 )
            return false;
    }
    return true;
}
```