BaseFunctionsJS_EN

Introduction

In this guide, you can get basic knowledge of JavaScript. The reason for writing the guide was the complexity and confusion that I have noticed in other guides and guides. Of course, I'm not an expert in JavaScript, but at the same time I have some experience and I think that I can help others with this knowledge :)

Useful links

Since this guide is quite subjective, I'm not sure that I will cover all the topics and I'm not sure that we won't have questions after reading, so in this block I will attach good links that can be used in such cases.

First of all, you can email me and I will try to help you:

boyarkin.gleb@gmail.com

source.boar@gmail.com

The mail is also duplicated in my profile.

Besides me, you can use:

https://developer.mozilla.org/en/docs/Web/JavaScript/Guide

https://developer.mozilla.org/en/docs/Web/JavaScript/Reference

These sources I guarantee that they are verified, I studied them myself, but as for me, many things are painted there with too complicated words.

Briefly about JavaScript

JavaScript (JS) is a fairly young and well-known dynamic programming language. The main industry of application is primarily Web development, as well as application development.

Syntax

The syntax of JS is mostly C-like, but with its own minor differences that we will encounter along the way.

If you go through the basic rules of syntax then:

- `;` – line ending character
- `{}` - characters for opening and closing code blocks
- `camelCase` - naming style for variables and functions
- `PascalCase` - class naming style
- `//` – character for single line comment
- `/* */` - characters for commenting a block of code

Small important note

`consoe.log(x);` is a function to output some `x` to the console. More specifically about this will be further, but this is necessary to understand the nearest section of the code.

Variables

As I think you know, or will become known now, in fact the entire programming language is based on variables and operations on them.

In JS, a variable is a pointer to an area or a memory cell in which some value is stored. For now, the fact about memory can simply be put into your head, we will come to it later.

The keywords `var` , `let` and `const` are used to declare variables.

Declaration example:

```
var a1 = "Hello";
let a2 = 1;
const A3 = true;
```

When naming variables, as mentioned above, the camelCase style is used.

What does it include?

camelCase:

- first letter is lowercase
- each first letter of the next word is uppercase

Also, if specifically about JS, then in it the name of a variable can begin with the symbol `_` or a letter of any case, subsequent characters can already be the same plus numbers (0-9). It is also worth clarifying that JS is case sensitive, that is, in it `a` and `A` are different names.

Difference between `var` , `let` and `const`

- `var` - a way to declare a variable with context scope, does not require a value when declaring, the default is `undefined` , the variable can be redeclared. This way of declaring is deprecated and it is recommended to use `let` instead
- `let` is a way to declare a variable with a block scope, does not require a value when declaring, the default is `undefined` , the variable cannot be redeclared.
- `const` - a way to declare a variable with block scope, but further access to the variable after the declaration is read-only, which assumes that the value at the declaration is mandatory. It is also customary to name constant variables with uppercase letters, but this is not required.
- `Without keyword` - a way to declare a variable with a global scope

And now in simple words

Scope - a section of the program where your variable, function, class will be visible to other program objects.

Global scope - scope is the entire program

Block scope – scope within a program block

Program block - function, class, whole program

Context scope - scope depending on the context of the declaration: when declaring in a function, the scope will be this function, when declaring in the entire program, the scope will be the entire program, etc.

Examples

```
var a1 = 1;
let a2 = 2;
const A3 = 3;
a4 = 4;

function f(){
    var f1 = 1;
    let f2 = 2;
    const F3 = 3;
    f4 = 4;
}

f();

function g(){
     console.log(a1); // will output
     console.log(a2); // will output
     console.log(A3); // will output
     console.log(a4); // will output
     console.log(f1); // won't output
     console.log(f2); // won't output
     console.log(F3); // won't output
     console.log(f4); // will output
}

g();

function h(){
     var a1 = 0; // will work
     let a2 = 0; // "but" will work
     const A3 = 0; // "but" will work
     a4 = 0; // will work
     f4 = 0; // will work
}

h();

var a1 = 0; // will work
let a2 = 0; // won't work
const A3 = 0; // won't work
a4 = 0; // will work
```

Now let's see what happens:

1. Variables `a1`, `a2`, `A3`, `a4` are declared at the program level, and `f1`, `f2`, `F3`, `f4` at the function level.
2. What will happen to `var`:
    1. Due to contextual typing, `a1` will become global, and `f1` local variables, due to which `g` `a1` will be visible in the function, but `f1` will not
    2. In the function `h` or in the program itself, the variable `a1` can be declared anew
3. What will happen to `let`:
    1. Due to block typing, `a2` will be visible to the entire program, and `f2` only to the function in which it is declared, which is why `g` `a2` will be visible in the function, but `f2` will not
    2. In the `h` function, the `a2` variable can be declared anew, "but" now `a2` for the entire program block will be new and the old one will not be available
    3. At the block level at which `a2` was set, it will no longer be possible to re-declare it
4. What will happen to `const`:
    1. Due to block typing, `A3` will be visible to the entire program, and `F3` only to the function in which it is declared, which is why in the `g` function `A3` will be visible, but `F3` is not
    2. In the `h` function, the `A3` variable can be declared again, "but" now `A3` for the entire block of the program will be new and the old one will not be available
    3. At the block level at which `A3` was set, it will no longer be possible to re-declare it
    4. It will be impossible to change the value of any of the variables of this type
5. What will happen to `no keyword`:
    1. Both variables will become global due to the global scope
    2. Redeclaration is no different from reassignment, so technically it can be changed in any way in any block of the program.

Why is `var` deprecated?

As we saw above: variables declared through `var` can be redeclared, which can cause a lot of errors and inconsistencies in the code. To solve this problem, `let` was created, which in turn does not suffer from such problems.

Data types

There are 7 primitive data types in JS.

Data type - types of what type of value the variable can store.

Primitives (primitive data types) are data types that are not objects and do not have methods.

In simple words: The simplest system data types.

Primitives in JS:

- `Boolean` – boolean data type, stores 2 types of values: `true` or `false`
- `Number` - numeric data type, stores numbers in the range approximately from *-2*10^307* to *2*10^307*, in case of passing a value less than `-Infinity` will be written, and more than `Infinity`
- `String` - string data type, stores a string of text of almost unlimited size
- `null` is a keyword that means null or "empty" value
- `undefined` - a keyword that means that the variable does not store any value, by default it is assigned to all variables when declared without a value, except for `const`

Typing

JS typing is dynamic, weak, implicit.

Now let's try to understand:

- Dynamic - means that the variable is not bound to a single type when declared, the variable can store different types as the program is used
- Weak - means that when doing actions with a variable of one type, actions of another type will not cause an error. *Famous example*: `1 + "2" == "12"`, that is, that the number was tried to be summed with a string, but this did not cause an error and worked
- Implicit - means that when declaring it is not necessary to "explicitly" indicate the type of the variable. *For example, like this in C++ with explicit typing:* `string a = "a"`, when `let a = "a"` is enough in JS, without specifying the type

What do these knowledge give us?

From JS typing, we can extract the facts that:

- We don't need to keep track of types at the program level
- We do not need to declare a variable by specifying its type
- We can assign values of different types to one variable

Let's look specifically at the different types

Boolean

This is a boolean data type that is used in places where you need to store 2 options for a variable: YES (`true`) or NO (`false`)

Declarations of variables of this type may look like this:

```
let t = true;
let f = false;
```

They are used mainly in conditionals, which we will discuss later.

####Number

Numeric data type. Can store different numbers: positive, negative, zero, integers, non-integers...

Declarations of variables of this type may look like this:

```
let n = 2;
let p = 3.14;
let m = -4;
let o = 0;
```

You can also use the `ParseInt(string)`, `ParseFloat(string)` functions to convert variables of another type to `Number`, or simply by creating an object `Number()`

String

Sink data type, stores text.

Declarations of variables of this type may look like this:

```
let txt1 = "Hello";
let txt2 = 'World';
let txt3 = `!`;
```

There is no difference between quotes (with the exception of ``, but not about that now), the main thing is that they open and close the string at the same time.

You can also use the `String()` or `.toString()` functions to convert variables of another type to `String`

There are also a very large number of methods for strings, a list of which and instructions for them can be found at the link: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/String

Operators

Assignment operators

`=` - assignment

`+=` - addition assignment

`-=` - assignment with subtraction

`*=` - multiplication assignment

`/=` - assignment with division

`%=` - division assignment with remainder

`**=` - assignment with exponentiation

`++` is an increment, same as `+= 1`

`--` - decrement, same as `-= 1`

`<<=` - left shift assignment

`>>=` - right shift assignment

`>>>=` - unsigned right shift assignment

`&=` - assignment with bitwise `AND`

`|=` - assignment with bitwise `OR`

`^=` - assignment with bitwise exclusive `OR`

Boolean operators

`==` - equality

`===` - equality with type checking

`!=` - inequality

`!==` - type-checked inequality

`>` - more

`>=` - greater than or equal to

`<` - less than

`<=` - less than or equal to

`&&` is a logical `AND`, i.e. both conditions in the construction `(condition 1)&&(condition 2)` must be `true`

`||` is a logical `OR`, i.e. at least one condition in the construction `(condition 1)||(condition 2)` must be `true`

Arithmetic operators

`+` - plus

`-` - minus

`*` - multiplication

`/` - division

`%` - division with remainder

`**` - exponentiation

Bitwise operators

`&` - bitwise `AND`

`|` - bitwise `OR`

`^` - bitwise exclusive `OR`

`~` - bitwise `NOT`

`<<` - shift left

`>>` - right shift

`>>>` - right shift padded with zeros

Conditional constructs

Variables are, of course, good, but you have to work with them somehow, and conditional constructions are one of the ways to write program logic.

if...else

The most famous and commonly used design.

A condition is passed to `if`, if the condition is true, then what is in the `if` block is executed, otherwise what is in the `else` block

Syntax looks something like this:

```
if(/*condition: either a logical expression or a variable of type Boolean*/){
    // action if condition == true
}
else{
    // action if condition == false
}
```

You can also add `else if` to this construction

Application example:

```
if(/*condition 1: either a logical expression or a variable of type Boolean*/){
    // action if condition 1 == true
}
else if(/*condition 2: either a logical expression or a variable of type Boolean*/){
    // action if condition 1 == false, but condition 2 == true
}
...
else{
    // action if condition 1 == false and condition 2 == false
}
```

Such `else if` constructs can be added any number of times

switch..case...default

Also a conditional construction that shortens the code that can be written through `if...else if...else`

`switch...case...default` is different in that we pass some variable to switch, and then the construction compares this variable with each value of `case` and if no match with `case` is found, the `default` block is executed

Syntax looks something like this:

```
switch(/*variable to compare*/){
    case /*value 1*/:
        // action if value 1 matches
        break;
    case /*value 2*/:
        // action if value 2 matches
        break;
    ...
    default:
        // action if no value matches
        break;
}
```

`break` is a mandatory command at the end of blocks so that the comparison of variables does not go further

try...catch...finally

Conditional block, in which the execution condition will be the presence or absence of an error in the `try` code block

Syntax looks something like this:

```
try{
     // block of code that will reproduce to search for an error
}
catch(/*error object of type Error*/){
     // action if error
}
finally{
     // action after executing the block, which will not reproduce on the search for
an error
}
```

More about Error objects: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Error

Loops

while

Loop in which the action is performed while the condition is true

Syntax looks something like this:

```
while(/*condition: either a logical expression or a variable of type Boolean*/){
     // action if condition is true
}
```

###do...while

You can say a kind of `while` loop, only in `while` the condition is checked first and then the action, and in `do...while` vice versa - first the action, and then the check

Syntax looks something like this:

```
do{
     // action if condition is true
} while(/*condition: either a logical expression or a variable of type Boolean*/)
```

for

Type of loop with an action that is performed before the start of the bypass, an execution condition and an action that is performed at the end of each pass

Syntax looks something like this:

```
for(/*action at the beginning*/; /*execution condition*/; /*action at the end of each pass*/){
    // action while condition is true
}
```

`break` and `continue` keywords

`break` is used to exit the loop early, i.e. to terminate the loop regardless of the condition

Usage example:

```
for(let i = 0; i < 10; i++){
    if(i % 3 == 0){ // if the number is divisible by 3 without a remainder
        break;
    }
    consoe.log(i);
}
```

That is, the loop was executed until it came across the first number that is divisible by 3

`continue` is used to terminate the loop pass early and move to the next pass, that is, it is used when you need everything after its call not to be executed in this pass, but to be executed immediately making the pass

```
for(let i = 0; i < 10; i++){
    if(i % 3 == 0){ // if the number is divisible by 3 without a remainder
        continue;
    }
    consoe.log(i);
}
```

That is, for numbers that are divisible by 3, `console.log()` was not executed, but the next loop run was immediately performed

Practical tasks on the given knowledge

For better consolidation, I propose to perform a few simple tasks that will check how much you understand this level of material, all the knowledge that may be needed to complete the tasks is above.

All problem solutions will be in the "Problems" folder and will be numbered, but I do not recommend you use them until you yourself succeed

1. Add such variable declarations to the beginning of the code so that the rest of the code works as written in the comment. Not all places to write code must be filled

```
// put your declarations here
func1(){
    // or here
}

func2(){
    try{
        A1 = 0;
    }
    catch(e){
        console.log("You cannot change variable A1");
    }
    console log(a2);
}

func1();
func2();
```

   (All console output must occur)

2. Write a loop that will print to the console all numbers from 1 to 100 inclusive that are divisible by 5 or 7 (`n % /*number*/ == 0` - the condition for divisibility of `n` by any number)

3. Write a block of code that will display the phrase "An error occurred" when the action `throw ""` is performed, and after that the phrase "Errors processed" will be displayed

Functions

Functions are another important programming tool. If in simple words, this is the union of any section of code for its further call. In JS functions, this is an object and you can work with them as with an object, that is, write to variables, pass as an argument, write methods, and so on.

Every function in any programming language has:

- Name
- List of arguments (parameters) – variables whose values are set when calling the function. They are declared in `()` separated by commas. Also, when calling, their values are specified when calling in the same way in `()`. In the code block of our function, they play the role of ordinary block-scoped variables, but they do not need to be declared, since they are declared when the function is called.
- A block of code that will be executed when called. Written to `{}`

Syntax looks something like this:

```
function /*function name*/(/*argument1*/, /*argument2*/.../*argumentN*/){
    // block of executable code
}
```

`function` is a keyword that means function declaration

To call a function, it is enough to write its name and `()` in which you can list the arguments if they were declared

Function example:

```
function f(a1, a2, a3){
    console.log(a1);
    console.log(a2);
    console.log(a3);
}

f(1, 2, 3);
```

Functions and procedures

If we deviate from the topic of JS, where there is in fact no syntactic difference between these concepts, then this is a very important issue.

What's the difference?

Function is a block of code that will return some value as a result of its execution.

Procedure is a function that does not return a value

The `return` keyword is responsible for returning a value in JS

Let's see the difference with examples:

```
function func(a1, a2, a3, a4){ // function
    console log(a1);
    console log(a2);
    console log(a3);
    return a4;
}

function proc(b1, b2, b3, b4){ // procedure
    console log(b1);
    console log(b2);
    console log(b3);
}

let a = func(1, 2, 3, 4) // == 4(a4)
let b = proc(1, 2, 3, 4) // == undefined
```

What happened?

When declaring a **function**, we used `return`, so the function call can be assigned to a variable and it will have the value that we specified in `return`

And when declaring **procedure** `return` we did not use and because of which our call does not matter, that is, `undefined`

Arrow functions and function assignment to a variable

Since a function in JS is an object, it can be assigned to a variable.

That is, declare for example like this:

```
let f = function(a1, a2, a3){
    console.log(a1);
    console.log(a2);
    console.log(a3);
}
```

This method is not not particularly correct from a logical point of view, but at the same time it is possible.

To simplify the recording of the above, the so-called arrow functions were invented

Syntax looks something like this:

```
let /*function name*/ = (/*argument1*/, /*argument2*/.../*argumentN*/) => {
    // block of executable code
}
```

There are several important aspects:

- If there is only one command in the code block `{}` are not required
- If there is only one command in the code block, then it will automatically return through the `return` block without even specifying it
- If there is only one argument, then `()` are not required

```
let func = a1 => a1;
let a = func(2); // == 2(a1)
```

That is, `a1` will be automatically returned and `()` is not required since there is only one argument

Ordering data types

Variables are certainly good, but sometimes you have to store a lot of any data by combining them into one structure. For such things, arrays and dictionaries are invented.

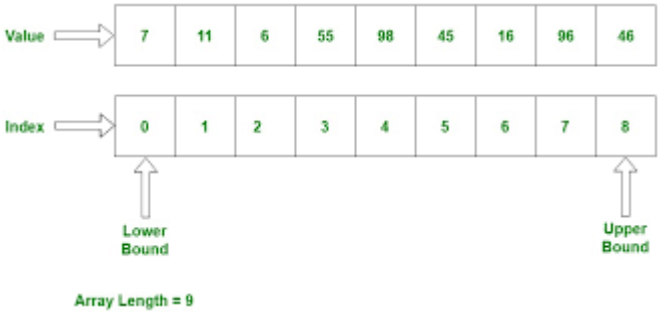
Arrays

Arrays are an ordered set of values that can be obtained by referring to the index.

An array in JS is an object, so it has all the properties of an object.

*** We also consider several ways to declare it: ***

```
let arr = new Array(); // empty array via object
let arr = new Array(/*element 1*/, /*element 2*/, ... /*element N*/); // through
object creation, for filling
let arr = new Array(/*array length*/); // via object creation, to create with
default length
let arr = []; // empty array explicitly
let arr = [/*element 1*/, /*element 2*/, ... /*element N*/]; // just explicitly
```



For a better understanding, let's use the picture above. As you can see, an array is, roughly speaking, a set of "cells" with values to which indices are attached in parallel, by which you can get the value to which they correspond.

To get a value or assign by index, it is enough to specify the index after the array name in `[]`. You can call the `.length` property to get the length of an array.

```
let a = arr[0]; // get
arr[0] = 1; // assignment
letlen = arr.length; // getting the length
```

!!!Indexing in programming languages starts from 0!!!

Bad and good feature of JS:

If JS is accessed by an undeclared index, then `undefined` will be returned when getting a value, and when assigned, an index with this number will be created.

Example:

```
let arr = [1, 2, 3];
let a = arr[4]; // undefined
arr[4] = 4; // arr == [1, 2, 3, 4]
```

This usage is not particularly correct from a logical point of view, but it is present and actually appears due to the fact that an array in JS is an object.

Basic array methods

- `.concat()` - array concatenation

```
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let arr3 = arr1.concat(arr2); // == [1, 2, 3, 4, 5, 6]
```

- `.join(/*separator*/)` - joins the array into a string and inserts a separator between the elements that we specify in the argument

```
let arr = [1, 2, 3];
let s = arr.join("-"); // == "1-2-3"
```

- `.push()` - adding a value to the end of an array

```
let arr = [1, 2, 3];
arr.push(4);
// arr == [1, 2, 3, 4]
```

- `.pop()` - removes the last element of the array

```
let arr = [1, 2, 3];
arr.pop();
// arr == [1, 2]
```

- `.unshift()` - adds values to the beginning of the array

```
let arr = [1, 2, 3];
arr.unshift(0);
// arr == [0, 1, 2, 3]
```

- `.shift()` - removes the first element of the array

```
let arr = [1, 2, 3];
arr.shift();
// arr == [2, 3]
```

- `.indexOf()` - returns the first index of the element in the array

```
let arr = [1, 2, 2, 3];
let i = arr.indexOf(2); // one
```

- `.lastIndexOf()` - returns the last index of an element in an array

```
let arr = [1, 2, 2, 3];
let i = arr.lastIndexOf(2); // 2
```

- `.slice(/*from*/, /*to*/)` – creating a slice

```
let arr1 = [1, 2, 3, 4, 5, 6];
let arr2 = arr1.slice(1, 4); // [2, 3, 4]
```

- `.splice(/*index*/, /*how many elements to remove*/, /*elements to be inserted*/)` - removes any number of elements from the array by index and inserts the elements listed later in their place

```
let arr = [1, 2, 3, 4, 5, 6];
arr.splice(1, 2, 7, 8);
// arr == [1, 7, 8, 4, 5, 6]
```

- `.reverse()` - rearranges elements in reverse order

```
let arr = [1, 2, 3];
arr.reverse();
// arr == [3, 2, 1]
```

For the rest you can find out at the link:
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array

Array iteration

for

```
for(let i = 0; i < arr.length; i ++){
     // array element arr[i]
}
```

for...in

```
for(let i in arr){
     // array element arr[i]
}
```

for...of

```
for(let a of arr){
     // array element a
}
```

.forEach()

```
arr.forEach(a => {
     // array element a
});
```

Dictionaries

Sometimes there are situations when it is necessary that the indices were not numbers, but something of their own. For this, dictionaries were created.

Dictionaries is an array with self-specified indices.

Declaration syntax:

```
let dic = {
     "/*index name 1*/": /*value 1*/,
     "/*index name 2*/": /*value 2*/,
     ...
     "/*name of index N*/": /*value N*/,};
```

Reversal and assignment is the same, only indices are not necessarily digits:

```
let dic = {"name": "Sam", "age": 17}
let d = dic["name"] // "Sam"
dic["age"] = 18;
```

Working with strings

After we got acquainted with arrays, we can talk about strings.

In fact: **string** is an array of characters. Its elements can be accessed as in an array, but cannot be changed as elements in an array.

```
let s = "abc";
let c = s[0]; // "a"
s[0] = "d"; // impossible
```

String Methods

- `.charCodeAt()` - get element code

  ```
  let c = "a".charCodeAt(); // 97
  ```

- `String.fromCharCode()` - getting an element by code

  ```
  let c = String.fromCharCode(97); // "a"
  ```

- `.indexOf()` - returns the first index of a character in a string

  ```
  let s = "abca";
  let i = s.indexOf("a"); // 0
  ```

- `.lastIndexOf()` - returns the last index of a character in a string

```
let s = "abca";
let i = s.lastIndexOf("a"); // 3
```

- `.slice(/*from*/, /*to*/)` – creating a slice

```
let s1 = "abcdef";
let s2 = s1.slice(1, 4); // "bcd"
```

- `.split(/*separator*/)` - splits a string into an array of strings by separator

```
let s = "a-b-c";
let arr = s.split("-"); // ["a", "b", "c"]
```

- `.toLowerCase()` - returns a lowercase string

```
let s1 = "abC";
let s2 = s1.toLowerCase(); // "abc"
```

- `.toUpperCase()` - returns a string in upper case

```
let s1 = "abC";
let s2 = s1.toUpperCase(); // "ABC"
```

- `.repeat()` - repeats the string the number of times specified in the argument

```
let s1 = "abc";
let s2 = s1.repeat(3); // "abcabcabc"
```

- `.trim()` - removes spaces at the beginning and end of a string

```
let s1 = "abc";
let s2 = s1.trim(); // "abc"
```

Practical tasks on the given knowledge

4. Figure out how to reverse a string
5. Find a way to turn the string `"abcdef"` into `"cdefgh"`
6. Write a function that will return a string created according to the `camelCase` rule (the separator must be passed as the second argument)

   Example:

   `hello world` -> `helloWorld`
7. Create a function to populate an array of dictionaries that contain information about a person

Structure:

```
let people = []; // main array
/* dictionary structure
{
    "firstName": "",
    "secondName": "",
    "age": "",
    "country": "",
    "city": ""
};
*/
```

8. *Think about how you can sort the array

"*" - difficult task

Everything!

It's all about the prince! I gave you everything I wanted. Hope this guide helped you.

I also have a more difficult guide with more complex topics, but it's more just a collection of solutions, but for those who are interested, you can read: https://github.com/s0urcedev/AdditionalFunctions

Thanks to all! Hope to see you again :)