

JavaScript

what is JavaScript ?

JavaScript was initially created to “**make web pages alive**”.

The programs in this language are called scripts. They can be written right in a web page's HTML and run automatically as the page loads.

Scripts are provided and executed as plain text. They don't need special preparation or compilation to run.

In this aspect, **JavaScript** is very different from another language called **java**.

Why is it called JavaScript?

When JavaScript was created, it initially had another name: “LiveScript”. But Java was very popular at that time, so it was decided that positioning a new language as a “younger brother” of Java would help.

But as it evolved, JavaScript became a fully independent language with its own specification called ECMAScript, and now it has no relation to Java at all.

Why is it called JavaScript?

Today, JavaScript can execute not only in the browser, but also on the server, or actually on any device that has a special program called the JavaScript engine.

The browser has an embedded engine sometimes called a “JavaScript virtual machine”.

Different engines have different “codenames”. For example:

- V8 – in Chrome, Opera and Edge.
- SpiderMonkey – in Firefox.

How do engines work?

Engines are complicated. But the basics are easy.

1. The engine (embedded if it's a browser) reads ("parses") the script.
2. Then it converts ("compiles") the script to machine code.
3. And then the machine code runs, pretty fast.

The engine applies optimizations at each step of the process. It even watches the compiled script as it runs, analyzes the data that flows through it, and further optimizes the machine code based on that knowledge.

What can in-browser JavaScript do?

Modern JavaScript is a “safe” programming language. It does not provide low-level access to memory or the CPU, because it was initially created for browsers which do not require it.

JavaScript’s capabilities greatly depend on the environment it’s running in. For instance, [Node.js](#) supports functions that allow JavaScript to read/write arbitrary files, perform network requests, etc.

In-browser JavaScript can do everything related to webpage manipulation, interaction with the user, and the webserver.

What makes JavaScript unique?

There are at least three great things about JavaScript:

- Full integration with HTML/CSS.
- Simple things are done simply.
- Supported by all major browsers and enabled by default.

JavaScript is the only browser technology that combines these three things.

That's what makes JavaScript unique. That's why it's the most widespread tool for creating browser interfaces.

That said, JavaScript can be used to create servers, mobile applications, etc.

Summary

- JavaScript was initially created as a browser-only language, but it is now used in many other environments as well.
- Today, JavaScript has a unique position as the most widely-adopted browser language, fully integrated with HTML/CSS.
- There are many languages that get “transpiled” to JavaScript and provide certain features. It is recommended to take a look at them, at least briefly, after mastering JavaScript.

Code editors

A code editor is the place where programmers spend most of their time.

There are two main types of code editors: IDEs and lightweight editors. Many people use one tool of each type.

1.visual studio code

2.notepad++

Developer console

Code is prone to errors. You will quite likely make errors... Oh, what am I talking about? You are absolutely going to make errors, at least if you're a human, not a robot.

But in the browser, users don't see errors by default. So, if something goes wrong in the script, we won't see what's broken and can't fix it.

There's an error in the JavaScript code on it. It's hidden from a regular visitor's eyes, so let's open developer tools to see it.

In browsers:

Press F12 or, if you're on Mac, then Cmd+Opt+J.

The developer tools will open on the Console tab by default.

Fundamentals

we need a working environment to run our scripts

so, we will use script tag

JavaScript programs can be inserted almost anywhere into an HTML document using the `<script>` tag.

```
<!DOCTYPE HTML>
<html>

<body>

  <p>Before the script...</p>

  <script>
    alert( 'Hello, world!' );
  </script>

  <p>...After the script.</p>

</body>

</html>
```

Modern markup

The `<script>` tag has a few attributes that are rarely used nowadays but can still be found in old code:

The type attribute: `<script type=...>`

The old HTML standard, HTML4, required a script to have a type. Usually it was `type="text/javascript"`. It's not required anymore. Also, the modern HTML standard totally changed the meaning of this attribute. Now, it can be used for JavaScript modules. But that's an advanced topic, we'll talk about modules in another part of the tutorial.

Modern markup

The language attribute: `<script language=...>`

This attribute was meant to show the language of the script. This attribute no longer makes sense because JavaScript is the default language. There is no need to use it. **XXXX**

```
<script type="text/javascript"><!--  
    ...  
//--></script>
```

External scripts

If we have a lot of JavaScript code, we can put it into a separate file.

Script files are attached to HTML with the src attribute:

```
<script src="/path/to/script.js"></script>
```

Here, /path/to/script.js is an absolute path to the script from the site root. One can also provide a relative path from the current page. For instance, src="script.js", just like src="./script.js", would mean a file "script.js" in the current folder.

We can give a full URL as well. For instance:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"></script>
```

Task

Show an alert in sandbox in same page and in external page

```
<!DOCTYPE html>
<html>

<body>

  <script>
    alert( "I'm JavaScript!" );
  </script>

</body>

</html>
```

The HTML code:

```
1  <!DOCTYPE html>
2  <html>
3
4  <body>
5
6    <script src="alert.js"></script>
7
8  </body>
9
10 </html>
```

For the file `alert.js` in the same folder:

```
1  alert("I'm JavaScript!");
```

The modern mode

"use strict"

For a long time, JavaScript evolved without compatibility issues. New features were added to the language while old functionality didn't change.

That had the benefit of never breaking existing code. But the downside was that any mistake or an imperfect decision made by JavaScript's creators got stuck in the language forever.

This was the case until 2009 when ECMAScript 5 (ES5) appeared. It added new features to the language and modified some of the existing ones. To keep the old code working, most such modifications are off by default. You need to explicitly enable them with a special directive: "use strict".

The modern mode

“use strict”

The directive looks like a string: "use strict" or 'use strict'. When it is located at the top of a script, the whole script works the “modern” way.

For example:

```
"use strict";  
  
// this code works the modern way  
...
```

Ensure that “use strict” is at the top

Please make sure that "use strict" is at the top of your scripts, otherwise strict mode may not be enabled.

Comments

As time goes on, programs become more and more complex. It becomes necessary to add comments which describe what the code does and why.

Comments can be put into any place of a script. They don't affect its execution because the engine simply ignores them.

One-line comments start with two forward slash characters `//`.

```
// This comment occupies a line of its own  
alert('Hello');  
  
alert('World'); // This comment follows the statement
```

Comments

Multiline comments start with a forward slash and an asterisk `/*` and end with an asterisk and a forward slash `*/`.

Like this:

```
/* An example with two messages.  
This is a multiline comment.  
*/  
alert('Hello');  
alert('World');
```

Variables

Most of the time, a JavaScript application needs to work with information. Here are two examples:

1. An online shop – the information might include goods being sold and a shopping cart.
2. A chat application – the information might include users, messages, and much more.

Variables are used to store this information.

A variable is a “named storage” for data. We can use variables to store goodies, visitors, and other data.

Variables

To create a variable in JavaScript, use the let keyword.(first one)

Example:

```
let message;
```

```
message = 'Hello!';
```

```
alert(message);
```

we can combine the variable declaration and assignment into a single line:

```
let message = 'Hello!';
```

```
alert(message);
```

We can also declare multiple variables in one line:

```
let user = 'John', age = 25, message = 'Hello';
```

Variables

var instead of let(second one)

In older scripts, you may also find another keyword: var instead of let:

```
var message = 'Hello';
```

The var keyword is almost the same as let. It also declares a variable but in a slightly different, “old-school” way.

Variables Naming

There are two limitations on variable names in JavaScript:

1. The name must contain only letters, digits, or the symbols \$ and _.
2. The first character must not be a digit.

Examples of valid names:

```
let userName;
```

```
let test123;
```

Variables Naming

There is a list of reserved words, which cannot be used as variable names because they are used by the language itself.

For example: let, class, return, and function are reserved.

The code below gives a syntax error:

```
let let = 5;
```

```
let return = 5;
```

Constants

To declare a constant (unchanging) variable(third), use `const` instead of `let`:

```
const myBirthday = '18.04.1982';
```

Variables declared using `const` are called “constants”. They cannot be reassigned.

An attempt to do so would cause an error:

```
const myBirthday = '18.04.1982';
```

```
myBirthday = '01.01.2001'; // error, can't reassign the constant!
```

Interaction Function

Interaction: alert, prompt, confirm

As we'll be using the browser as our demo environment, let's see a couple of functions to interact with the user.

alert, prompt and confirm.

alert

This one we've seen already. It shows a message and waits for the user to press "OK".

For example:

```
alert("Hello");
```

Interaction Function

prompt

The function prompt accepts two arguments:

```
result = prompt(title, [default]);
```

It shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel.

title

The text to show the visitor.

default

An optional second parameter, the initial value for the input field.

Interaction Function

prompt

The call to prompt returns the text from the input field or null if the input was canceled.

```
let age = prompt('How old are you?', 100);  
alert(`You are ${age} years old!`); // You are 100 years old!
```

Interaction Function

confirm

The syntax:

```
result = confirm(question);
```

The function confirm shows a modal window with a question and two buttons: OK and Cancel.

The result is true if OK is pressed and false otherwise.

For example:

```
let isBoss = confirm("Are you the boss?");
```

```
alert( isBoss ); // true   if OK is pressed
```

Task

Create a web-page that asks for a name and outputs it.

```
<!DOCTYPE html>
<html>
<body>

  <script>
    'use strict';

    let name = prompt("What is your name?", "");
    alert(name);
  </script>

</body>
</html>
```

Data types

A value in JavaScript is always of a certain type. For example, a string or a number.

We can put any type in a variable. For example, a variable can at one moment be a string and then store a number:

```
// no error
```

```
let message = "hello";
```

```
message = 123456;
```

Programming languages that allow such things, such as JavaScript, are called “**dynamically typed**”, meaning that there exist data types, but variables are not bound to any of them.

Data types

Number

```
let n = 123;
```

```
n = 12.345;
```

The number type represents both **integer** and **floating** point numbers.

There are many operations for numbers, e.g. multiplication *, division /, addition +, subtraction -, and so on.

Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: Infinity, -Infinity and NaN.

Data types

Number

Infinity represents the mathematical Infinity ∞ . It is a special value that's greater than any number.

We can get it as a result of division by zero:

```
alert( 1 / 0 ); // Infinity
```

NaN represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:

```
alert( "not a number" / 2 ); // NaN, such division is erroneous
```

Data types

Number

Rounding

One of the most used operations when working with numbers is rounding.

There are several built-in functions for rounding:

Math.floor: Rounds down: 3.1 becomes 3, and -1.1 becomes -2.

Math.ceil: Rounds up: 3.1 becomes 4, and -1.1 becomes -1.

Math.round: Rounds to the nearest integer: 3.1 becomes 3, 3.6 becomes 4. In the middle cases 3.5 rounds up to 4, and -3.5 rounds up to -3.

Example:

```
let num = 1.23456;
```

```
alert( Math.round(num * 100) / 100 );
```

```
// 1.23456 -> 123.456 -> 123 -> 1.23
```

Data types

Number

parseInt and parseFloat

The sole exception is spaces at the beginning or at the end of the string, as they are ignored.

But in real life, we often have values in units, like "100px" or "12pt" in CSS. Also in many countries, the currency symbol goes after the amount, so we have "19€" and would like to extract a numeric value out of that.

That's what parseInt and parseFloat are for.

Data types

Number

parseInt and parseFloat

They “read” a number from a string until they can’t. In case of an error, the gathered number is returned. The function `parseInt` returns an integer, whilst `parseFloat` will return a floating-point number:

```
alert( parseInt('100px') ); // 100
```

```
alert( parseFloat('12.5em') ); // 12.5
```

```
alert( parseInt('12.3') ); // 12, only the integer part is returned
```

```
alert( parseFloat('12.3.4') ); // 12.3, the second point stops the reading
```

Data types

String

Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in `${...}`, **for example:**

```
let name = "John";// embed a variable
```

```
alert( `Hello, ${name}!` ); // Hello, John!// embed an expression
```

```
alert( `the result is ${1 + 2}` ); // the result is 3
```

The expression inside `${...}` is evaluated and the result becomes a part of the string. We can put anything in there:

a variable like `name` or an arithmetical expression like `1 + 2` or something more complex.

Please note that this can only be done in backticks. Other quotes don't have this embedding functionality!

Data types

String

To get a character at position pos, use square brackets [pos] or call the method str.at(pos). The first character starts from the zero position

```
let str = `Hello`; // the first character
```

```
alert( str[0] ); // H
```

```
alert( str.at(0) ); // H // the first character
```

```
alert( str[str.length - 1] ); // o
```

```
alert( str.at(-1) ); //??????????????
```

Data types

String

The first method is **str.indexOf(substr, pos)**.

It looks for the substr in str, starting from the given position pos, and returns the position where the match was found or -1 if nothing can be found.

For instance:

```
let str = 'Widget with id';
```

```
alert( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning
```

```
alert( str.indexOf('widget') ); // -1, not found, the search is case-sensitive
```

Data types

String

includes, startsWith, endsWith

The more modern method str.includes(substr, pos) returns true/false depending on whether str contains substr within.

It's the right choice if we need to test for the match, but don't need its position:

```
alert( "Widget with id".includes("Widget") ); // true
```

```
alert( "Hello".includes("Bye") ); // false
```

```
alert( "Widget".startsWith("Wid") ); // true, "Widget" starts with "Wid"
```

```
alert( "Widget".endsWith("get") ); // true, "Widget" ends with "get"
```

Data types

String

Getting a substring

There is method in JavaScript to get a substring: slice.

```
str.slice(start [, end])
```

Returns the part of the string from start to (but not including) end.

For instance:

```
let str = "stringify";
```

```
alert( str.slice(0, 5) ); // 'strin', the substring from 0 to 5 (not including 5)
```

```
alert( str.slice(0, 1) ); // 's', from 0 to 1, but not including 1, so only character at 0
```

Data types

Arrays

Objects allow you to store keyed collections of values. That's fine.

But quite often we find that we need an ordered collection, where we have a 1st, a 2nd, a 3rd element and so on.

For example, we need that to store a list of something: users, goods, HTML elements etc.

It is not convenient to use an object here, because it provides no methods to manage the order of elements. We can't insert a new property "between" the existing ones. Objects are just not meant for such use.

There exists a special data structure named Array, to store ordered collections.

Declaration

Data types

Arrays

There are two syntaxes for creating an empty array:

```
let arr = new Array();
```

```
let arr = [];
```

the second syntax is used. We can supply initial elements in the brackets:

```
let fruits = ["Apple", "Orange", "Plum"];
```

Array elements are numbered, starting with zero. We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
alert( fruits[0] ); // Apple
```

```
alert( fruits[1] ); // Orange
```

```
alert( fruits[2] ); // Plum
```

Data types

Arrays

...Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

The total count of the elements in the array is its length:

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
alert( fruits.length ); // 3
```

We can also use alert to show the whole array.

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
alert( fruits ); // Apple,Orange,Plum
```

Data types

Arrays

Methods pop/push, shift/unshift

A queue is one of the most common uses of an array. In computer science, this means an ordered collection of elements which supports two operations:

- push appends an element to the end.
- shift get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st.



Data types

Arrays

Methods pop/push, shift/unshift

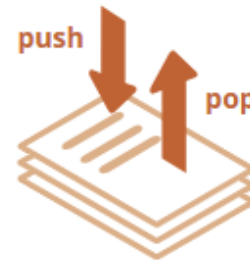
Arrays support both operations.

In practice we need it very often. For example, a queue of messages that need to be shown on-screen.

There's another use case for arrays – the data structure named stack.

It supports two operations:

- push adds an element to the end.
- pop takes an element from the end.



A stack is usually illustrated as a pack of cards: new cards are added to the top or taken from the top:

Data types

Arrays

Methods that work with the end of the array:

pop

Extracts the last element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];
```

```
alert( fruits.pop() ); // remove "Pear" and alert it
```

```
alert( fruits ); // Apple, Orange
```

Both `fruits.pop()` and `fruits.at(-1)` return the last element of the array, but `fruits.pop()` also modifies the array by removing it.

Data types

Arrays

Loops

One of the oldest ways to cycle array items is the for loop over indexes:

```
let arr = ["Apple", "Orange", "Pear"];
```

```
for (let i = 0; i < arr.length; i++)
```

```
{alert( arr[i] );}
```

But for arrays there is another form of loop, for..of:

```
let fruits = ["Apple", "Orange", "Plum"];// iterates over array elements
```

```
for (let fruit of fruits)
```

```
{alert( fruit );}
```

Data types

Arrays

toString

Arrays have their own implementation of toString method that returns a comma-separated list of elements.

For instance:

```
let arr = [1, 2, 3];
```

```
alert(String(arr) ); // 1,2,3
```

```
alert( String(arr) === '1,2,3' ); // true
```

Data types

Arrays

map

The arr.map method is one of the most useful and often used.

It calls the function for each element of the array and returns the array of results.

For instance, here we transform each element into its length:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);  
alert(lengths); // 5,7,6
```

sort(fn)

The call to arr.sort() sorts the array in place, changing its element order.

```
let arr = [ 1, 15 ,2]; // the method reorders the content of arr  
arr.sort();           alert( arr ); // 1, 2,15
```

Data types

Arrays

split and join

Here's the situation from real life. We are writing a messaging app, and the person enters the comma-delimited list of receivers: John, Pete, Mary. But for us an array of names would be much more comfortable than a single string. How to get it?

The `str.split(delim)` method does exactly that. It splits the string into an array by the given delimiter `delim`. In the example below, we split by a comma:

```
let names = 'Bilbo, Gandalf, Nazgul';
```

```
let arr = names.split(',');
```

```
for (let name of arr)
```

```
{alert( `A message to ${name}.` ); // A message to Bilbo (and other names)}
```

Data types

Arrays

split and join

The call arr.join(glue) does the reverse to split. It creates a string of arr items joined by glue between them.

For instance:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
```

```
let str = arr.join(';'); // glue the array into a string using ;
```

```
alert( str ); // Bilbo;Gandalf;Nazgul
```

Data types

Boolean (logical type)

The boolean type has only two values: true and false.

This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”.

For instance:

```
let nameFieldChecked = true; // yes, name field is checked
```

```
let ageFieldChecked = false; // no, age field is not checked
```

Data types

The “null” value

The special null value does not belong to any of the types described above.

It forms a separate type of its own which contains only the null value:

```
let age = null;
```

In JavaScript, null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.

It’s just a special value which represents “nothing”, “empty” or “value unknown”.

The code above states that age is unknown.

Data types

The “undefined” value

The special value undefined also stands apart. It makes a type of its own, just like null.

The meaning of undefined is “value is not assigned”.

If a variable is declared, but not assigned, then its value is undefined:

```
let age;
```

```
alert(age); // shows "undefined"
```

Data types

Objects

As we know from the chapter Data types, there are eight data types in JavaScript. Seven of them are called “primitive”, because their values contain only a single thing (be it a string or a number or whatever).

In contrast, objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

Data types

Objects

We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It's easy to find a file by its name or add/remove a file.

An empty object ("empty cabinet") can be created using one of two syntaxes:

`let user = new Object();` // "object constructor" syntax

`let user = {};` // "object literal" syntax



Data types

Objects

Literals and properties

We can immediately put some properties into {...} as “key: value” pairs:

```
let user = { // an object
```

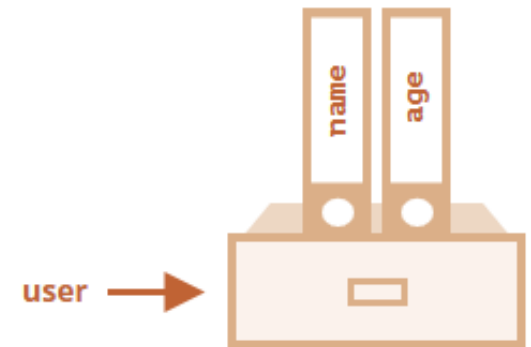
```
  name: "John", // by key "name" store value "John"
```

```
  age: 30 // by key "age" store value 30};
```

A property has a key (also known as “name” or “identifier”) before the colon ":" and a value to the right of it.

In the user object, there are two properties:

1. The first property has the name "name" and the value "John".
2. The second one has the name "age" and the value 30.



Data types

Objects

Literals and properties

We can add, remove and read files from it at any time.

Property values are accessible using the dot notation:

// get property values of the object:

```
alert( user.name ); // John
```

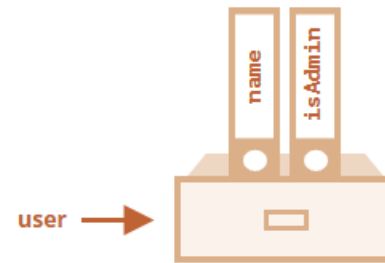
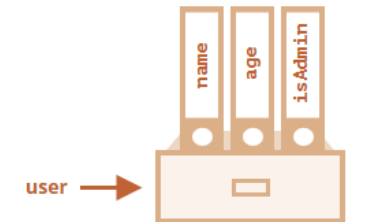
```
alert( user.age ); // 30
```

The value can be of any type. Let's add a boolean one:

```
user.isAdmin = true;
```

To remove a property, we can use the delete operator:

```
delete user.age;
```



Task

Objects

Write the code, one line for each action:

1. Create an empty object user.
2. Add the property name with the value John.
3. Add the property surname with the value Smith.
4. Change the value of the name to Pete.
5. Remove the property name from the object.

Task

Objects

Create a function `multiplyNumeric(obj)` that multiplies all numeric property values of `obj` by 2.

For instance:

// before the call

```
let menu = {width: 200,height: 300,title: "My menu"};
```

```
multiplyNumeric(menu);
```

// after the call

```
menu = {width: 400,height: 600,title: "My menu"};
```

Task

Objects

Create a function `multiplyNumeric(obj)` that multiplies all numeric property values of `obj` by 2.

```
function multiplyNumeric(obj) {  
  for (let key in obj) {  
    if (typeof obj[key] == 'number') {  
      obj[key] *= 2;  
    }  
  }  
}
```

Data types

Objects

If you've just started to read the tutorial and learn JavaScript, maybe the problem hasn't touched you yet, but it's quite common.

As an example, let's say we have user objects that hold the information about our users.

Most of our users have addresses in user.address property, with the street user.address.street, but some did not provide them.

In such case, when we attempt to get user.address.street, and the user happens to be without an address, we get an error:

```
let user = {}; // a user without "address" property
```

```
alert(user.address.street); // Error!
```

Data types

Objects

The optional chaining `?.` stops the evaluation if the value before `?.` is undefined or null and returns undefined.

Further in this article, for brevity, we'll be saying that something “exists” if it's not null and not undefined.

In other words, `value?.prop`:

- works as `value.prop`, if value exists,
- otherwise (when value is undefined/null) it returns undefined.

Here's the safe way to access `user.address.street` using `?.`:

```
let user = {}; // user has no address
```

```
alert( user?.address?.street ); // undefined (no error)
```

Task

What is the output of the script?

```
let name = "Ilya";
```

```
alert( `hello ${1}` );      // ?
```

```
alert( `hello ${"name"}` ); // ?
```

```
alert( `hello ${name}` );   // ?
```

Date and time

Let's meet a new built-in object: Date. It stores the date, time and provides methods for date/time management. For instance, we can use it to store creation/modification times, to measure time, or just to print out the current date.

Creation

To create a new Date object call new Date() with one of the following arguments:

`new Date()`

Without arguments – create a Date object for the current date and time:

```
let now = new Date();
```

```
alert( now ); // shows current date/time
```

Date and time

Access date components

There are methods to access the year, month and so on from the Date object:

getFullYear()

Get the year (4 digits)

getMonth()

Get the month, from 0 to 11.

getDate()

Get the day of month, from 1 to 31, the name of the method does look a little bit strange.

getHours(), getMinutes(), getSeconds(), getMilliseconds()

Get the corresponding time components.

Date and time

Examples:

// current date

let date = new Date();// the hour in your current time zone

alert(date.getHours());

let today = new Date();

today.setHours(0);

alert(today); // still today, but the hour is changed to 0

today.setHours(0, 0, 0, 0);

alert(today); // still today, now 00:00:00 sharp.

JSON

The JSON (JavaScript Object Notation) is a general format to represent values and objects. Initially it was made for JavaScript, but many other languages have libraries to handle it as well. So it's easy to use JSON for data exchange when the client uses JavaScript and the server is written on Ruby/PHP/Java/Whatever.

JavaScript provides methods:

- `JSON.stringify` to convert objects into JSON.
- `JSON.parse` to convert JSON back into an object.

JSON

For instance, here we **JSON.stringify** a student:

```
let student = {name: 'John',age: 30,isAdmin: false,courses: ['html', 'css', 'js'],spouse: null};
```

```
let json = JSON.stringify(student);
```

```
alert(typeof json); // we've got a string!
```

```
alert(json); /* JSON-encoded object:
```

```
{  
  
  "name": "John","age": 30,"isAdmin": false,  
  
  "courses": ["html", "css", "js"],"spouse": null  
  
}  
*/
```

JSON

JSON.parse

To decode a JSON-string, we need another method named JSON.parse.

The syntax:

```
let value = JSON.parse(str[, reviver]);
```

str:JSON-string to parse.

reviver:Optional function(key,value) that will be called for each (key, value) pair and can transform the value.

```
// stringified array
```

```
let numbers = "[0, 1, 2, 3]";
```

```
numbers = JSON.parse(numbers);
```

```
alert( numbers[1] ); // 1
```

JSON

JSON.parse

examples:

```
let userData = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';
```

```
let user = JSON.parse(userData);
```

```
alert( user.friends[1] ); // 1
```

Operators

Mathematical Operators:

The following math operations are supported:

- Addition +,
- Subtraction -,
- Multiplication *,
- Division /,
- Remainder %,
- Exponentiation **.

The first four are straightforward, while % and ** need a few words about them.

Operators

Mathematical Operators:

Remainder %

The remainder operator %, despite its appearance, is not related to percents.

The result of `a % b` is the remainder of the integer division of `a` by `b`.

For instance:

`alert(5 % 2);` // 1, the remainder of 5 divided by 2

`alert(8 % 3);` // 2, the remainder of 8 divided by 3

`alert(8 % 4);` // 0, the remainder of 8 divided by 4

Operators

Mathematical Operators:

Exponentiation `**`

The exponentiation operator `a ** b` raises `a` to the power of `b`.

In school maths, we write that as ab .

For instance:

```
alert( 2 ** 2 ); // 22 = 4
```

```
alert( 2 ** 3 ); // 23 = 8
```

```
alert( 2 ** 4 ); // 24 = 16
```

Operators

Assignment Operators

Let's note that an assignment `=` is also an operator. It is listed in the precedence table with the very low priority of 2.

That's why, when we assign a variable, like `x = 2 * 2 + 1`, the calculations are done first and then the `=` is evaluated, storing the result in `x`.

```
let x = 2 * 2 + 1;    alert( x ); // 5
```

Assignment `=` returns a value

The fact of `=` being an operator, not a “magical” language construct has an interesting implication.

All operators in JavaScript return a value. That's obvious for `+` and `-`, but also true for `=`. The call `x = value` writes the value into `x` and then returns it.

Operators

String concatenation with binary +

Let's meet the features of JavaScript operators that are beyond school arithmetics.

Usually, the plus operator + sums numbers.

But, if the binary + is applied to strings, it merges (concatenates) them:

```
let s = "my" + "string"; alert(s); // mystring
```

Note that if any of the operands is a string, then the other one is converted to a string too.

For example:

```
alert( '1' + 2 ); // "12"
```

```
alert( 2 + '1' ); // "21"
```

Operators

Increment/decrement

Increasing or decreasing a number by one is among the most common numerical operations.

So, there are special operators for it:

Increment ++ increases a variable by 1:

```
let counter = 2;
```

```
counter++;    // works the same as counter = counter + 1, but is shorter
```

```
alert( counter ); // 3
```

Decrement -- decreases a variable by 1:

```
let counter = 2;
```

```
counter--;    // works the same as counter = counter - 1, but is shorter
```

```
alert( counter ); // 1
```

Operators

The operators ++ and -- can be placed either before or after a variable.

- When the operator goes after the variable, it is in “postfix form”: `counter++`.
- The “prefix form” is when the operator goes before the variable: `++counter`.

Both of these statements do the same thing: increase counter by 1.

Is there any difference? Yes, but we can only see it if we use the returned value of ++/--.

Let's clarify. As we know, all operators return a value. Increment/decrement is no exception. The prefix form returns the new value while the postfix form returns the old value (prior to increment/decrement).

To see the difference,

Operators

here's an example:

```
let counter = 1;
```

```
let a = ++counter; // (*)
```

```
alert(a); // 2
```

In the line (*), the prefix form ++counter increments counter and returns the new value, 2. So, the alert shows 2.

Now, let's use the postfix form:

```
let counter = 1;
```

```
let a = counter++; // (*) changed ++counter to counter++
```

```
alert(a); // 1
```

Operators

Comparisons

We know many comparison operators from maths.

In JavaScript they are written like this:

- Greater/less than: $a > b$, $a < b$.
- Greater/less than or equals: $a \geq b$, $a \leq b$.
- Equals: $a == b$, please note the double equality sign $==$ means the equality test, while a single one $a = b$ means an assignment.
- Not equals: In maths the notation is \neq , but in JavaScript it's written as $a != b$.

Operators

Boolean is the result

All comparison operators return a boolean value:

- true – means “yes”, “correct” or “the truth”.
- false – means “no”, “wrong” or “not the truth”.

For example:

```
alert( 2 > 1 ); // true (correct)
```

```
alert( 2 == 1 ); // false (wrong)
```

```
alert( 2 != 1 ); // true (correct)
```

Operators

String comparison

To see whether a string is greater than another, JavaScript uses the so-called “dictionary” or “lexicographical” order.

In other words, strings are compared letter-by-letter.

For example:

```
alert( 'Z' > 'A' ); // true
```

```
alert( 'Glow' > 'Glee' ); // true
```

```
alert( 'Bee' > 'Be' ); // true
```

Operators

The algorithm to compare two strings is simple:

1. Compare the first character of both strings.
2. If the first character from the first string is greater (or less) than the other string's, then the first string is greater (or less) than the second. We're done.
3. Otherwise, if both strings' first characters are the same, compare the second characters the same way.
4. Repeat until the end of either string.
5. If both strings end at the same length, then they are equal. Otherwise, the longer string is greater.

Operators

In the first example above, the comparison 'Z' > 'A' gets to a result at the first step.

The second comparison 'Glow' and 'Glee' needs more steps as strings are compared character-by-character:

1. G is the same as G.
2. l is the same as l.
3. o is greater than e. Stop here. The first string is greater.

Operators

Strict equality

A regular equality check `==` has a problem. It cannot differentiate 0 from false:

```
alert( 0 == false ); // true
```

The same thing happens with an empty string

```
alert( "" == false ); // true
```

Operators

Strict equality

This happens because operands of different types are converted to numbers by the equality operator `==`. An empty string, just like `false`, becomes a zero.

What to do if we'd like to differentiate 0 from `false`?

A strict equality operator `===` checks the equality without type conversion.

In other words, if `a` and `b` are of different types, then `a === b` immediately returns `false` without an attempt to convert them.

Let's try it:

```
alert( 0 === false ); // false, because the types are different
```

There is also a “strict non-equality” operator `!==` analogous to `!=`.

Task

What will be the result for these expressions?

- `5 > 4`
- `"apple" > "pineapple"`
- `"2" > "12"`
- `undefined == null`
- `undefined === null`
- `null == "\n0\n"`
- `null === +"\n0\n"`

Conditional Operators

Conditional branching: if, '?'

Sometimes, we need to perform different actions based on different conditions.

To do that, we can use the if statement and the conditional operator ?, that's also called a "question mark" operator.

The "if" statement

The if(...) statement evaluates a condition in parentheses and, if the result is true, executes a block of code.**For example:**

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');
```

```
if (year == 2015) alert( 'You are right!' );
```

Conditional Operators

If we want to execute more than one statement, we have to wrap our code block inside curly braces:

```
if (year == 2015) {alert( "That's correct!" );  
alert( "You're so smart!" );}
```

The “else” clause

The if statement may contain an optional else block. It executes when the condition is falsy.**For example:**

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');  
if (year == 2015) {alert( 'You guessed it right!' );}  
else {alert( 'How can you be so wrong?' ); }// any value except 2015
```

Conditional Operators

Several conditions: “else if”

Sometimes, we'd like to test several variants of a condition. The else if clause lets us do that.**For example:**

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');  
if (year < 2015) {alert( 'Too early...' );}  
else if (year > 2015) {alert( 'Too late' );}  
else {alert( 'Exactly!' );}
```

In the code above, JavaScript first checks `year < 2015`. If that is falsy, it goes to the next condition `year > 2015`. If that is also falsy, it shows the last alert.

There can be more else if blocks. The final else is optional.

Conditional Operators

Conditional operator '?'

Sometimes, we need to assign a variable depending on a condition.

For instance:

```
let accessAllowed;
```

```
let age = prompt('How old are you?', '');
```

```
if (age > 18) {
```

```
    accessAllowed = true;}
```

```
else {
```

```
    accessAllowed = false;}
```

```
alert(accessAllowed);
```

Conditional Operators

Conditional operator '?'

The operator is represented by a question mark ?. Sometimes it's called "ternary", because the operator has three operands. It is actually the one and only operator in JavaScript which has that many.

The syntax is:

```
let accessAllowed = (age > 18) ? true : false;
```

Conditional Operators

Conditional operator '?'

Multiple '?'

A sequence of question mark operators ? can return a value that depends on more than one condition.

For instance:

```
let age = prompt('age?', 18);
```

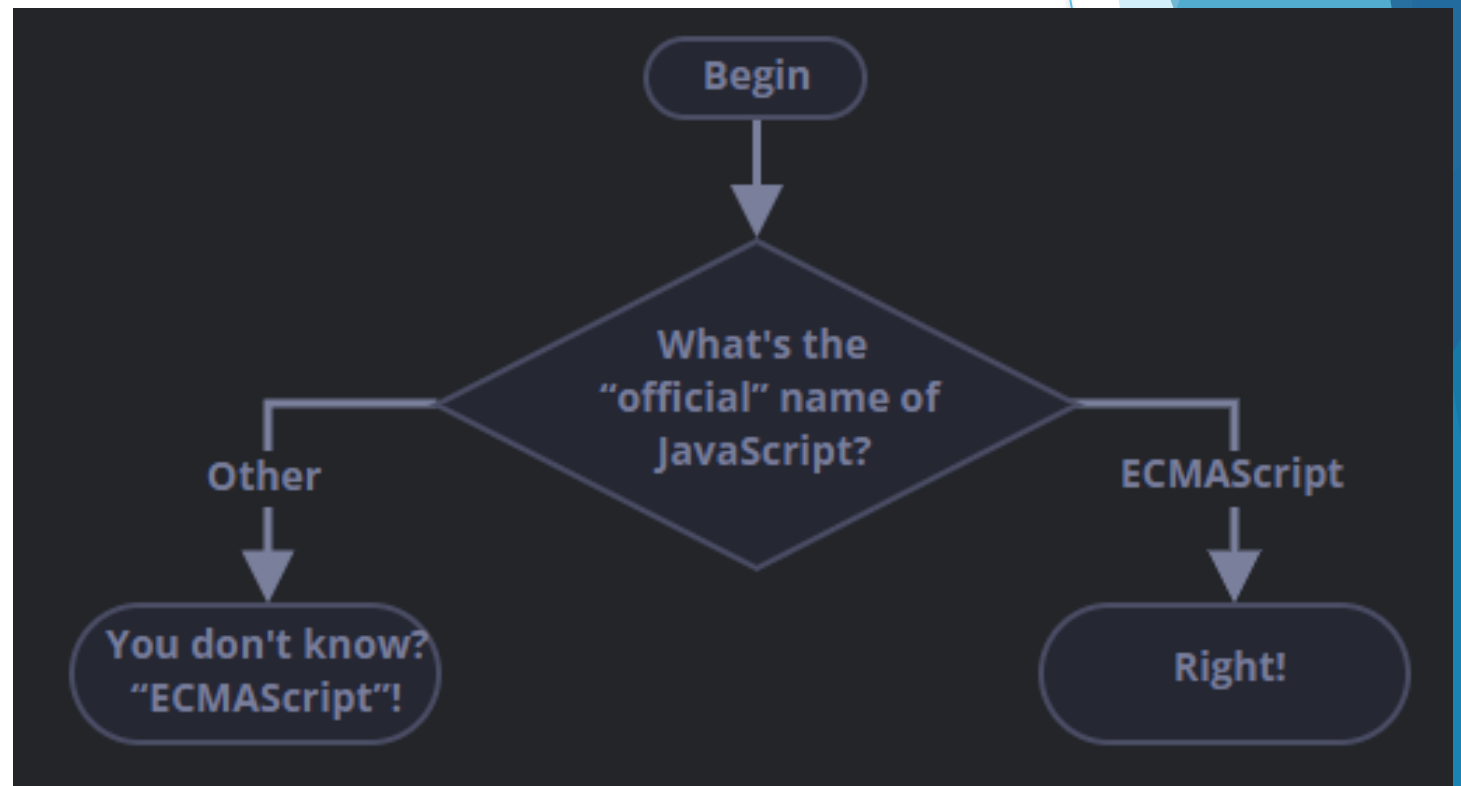
```
let message = (age < 3) ? 'Hi, baby!' :(age < 18) ? 'Hello!' :(age < 100) ? 'Greetings!' :'What an unusual  
age!';alert( message );
```

Task

Using the if..else construct, write the code which asks: 'What is the "official" name of JavaScript?'

If the visitor enters "ECMAScript", then output "Right!", otherwise – output: "You don't know? ECMAScript!"

an `_` need it as conditional operator



Task solution

Using the if..else construct, write the code which asks: 'What is the "official" name of JavaScript?'

If the visitor enters "ECMAScript", then output "Right!", otherwise – output: "You don't know? ECMAScript!"

```
<!DOCTYPE html>
<html>

<body>
  <script>
    'use strict';

    let value = prompt('What is the "official" name of JavaScript?', '');

    if (value == 'ECMAScript') {
      alert('Right!');
    } else {
      alert("You don't know? ECMAScript!");
    }
  </script>

</body>

</html>
```

Thank you