Front-End v23.0

Dashboard / My courses / FE23.0 / JavaScript v23.0 / Day 1 | Pre-work / JavaScript | Day 1 | Pre-work

Back to 'Day 1 | Pre-work'

JavaScript | Day 1 | Pre-work

JavaScript

Day 1

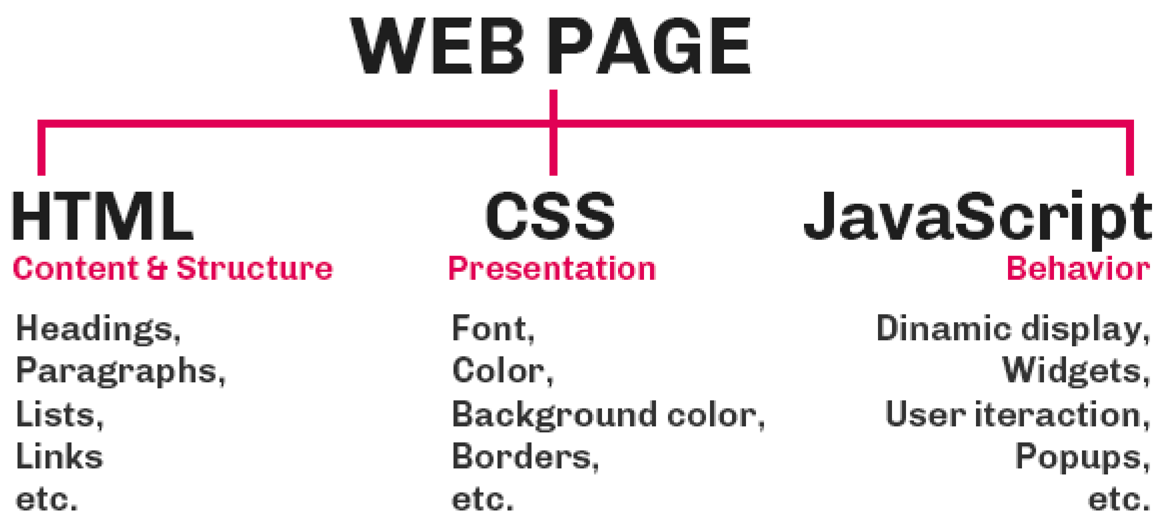
Table of contents:

History of JavaScript

JavaScript was created by Brendan Eich while he was working for a company called Netscape. Back then it was called Mocha, and after a long period of updates and improvements on it, it is now called JavaScript. It is however very different from Java, even if part of the name is the same.

It is an interpreted language, which means it needs a translator to translate this programming language to work on it. In our case, the translator will be the browser. Even so, Javascript is by now used not just for building websites as you will learn.
More than just fancy website-interactivity, it supports useful concepts such as OOP, Object Oriented programming (We will learn more about it later :)).
While in the beginning, most of the usage of Javascript works with the client-side, nowadays it can work on the server-side too using an environment like NodeJS.
All that being said, welcome to the wonderful world of JavaScript (JS).

JavaScript is a programming language used to make web pages interactive. In conjunction with HTML5 and CSS3, you will be able to create a rich, interactive, and dynamic GUI (Graphic User Interface).



What do I need to learn JavaScript?

To start creating interesting things with JavaScript, we need just HTML and CSS. Additionally, we don't need to install anything to start coding with JavaScript.

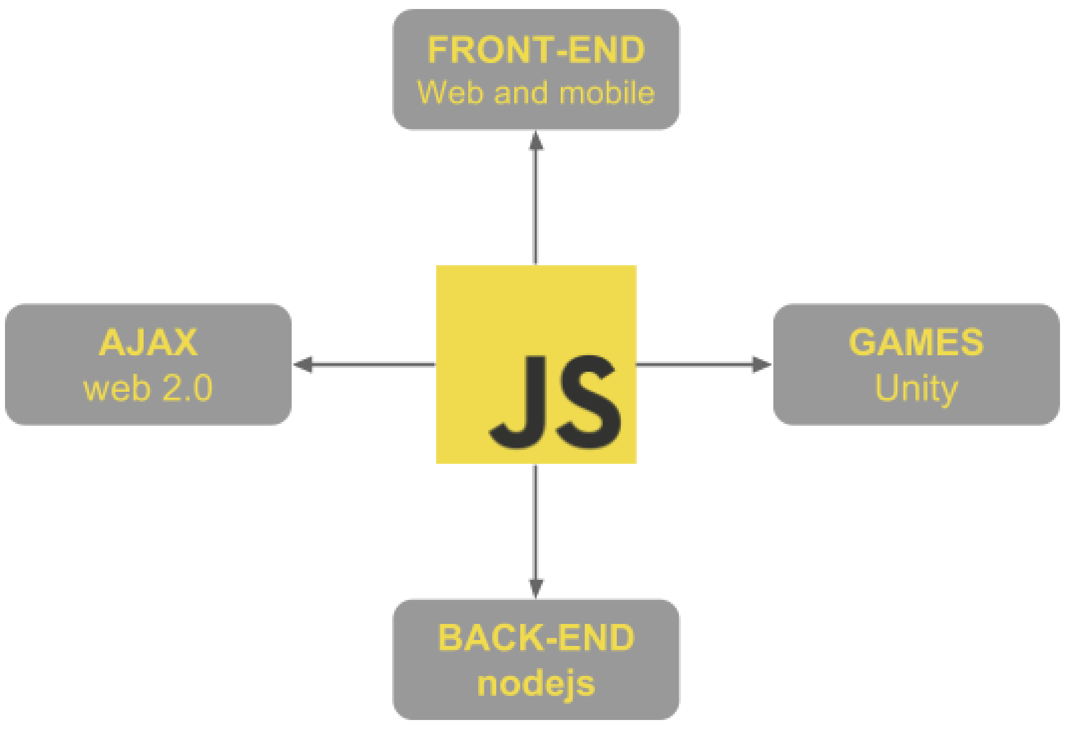
What is special about JavaScript is that it is the only programming language that is supported by ALL browsers. Which means any JavaScript code you write will automatically be recognized and run by the respective browser.

JavaScript brings **dynamic functionality** to the website: every time something pops up in the browser, dragging and dropping objects, dynamic change of text and images, it is JavaScript in action. Because it runs inside the Browser, JavaScript has direct access to all elements on the page.

JavaScript is a high-level language, which means it enables writing programs more or less independent of a particular type of computer. High-level languages are closer to human languages and further from machine languages.

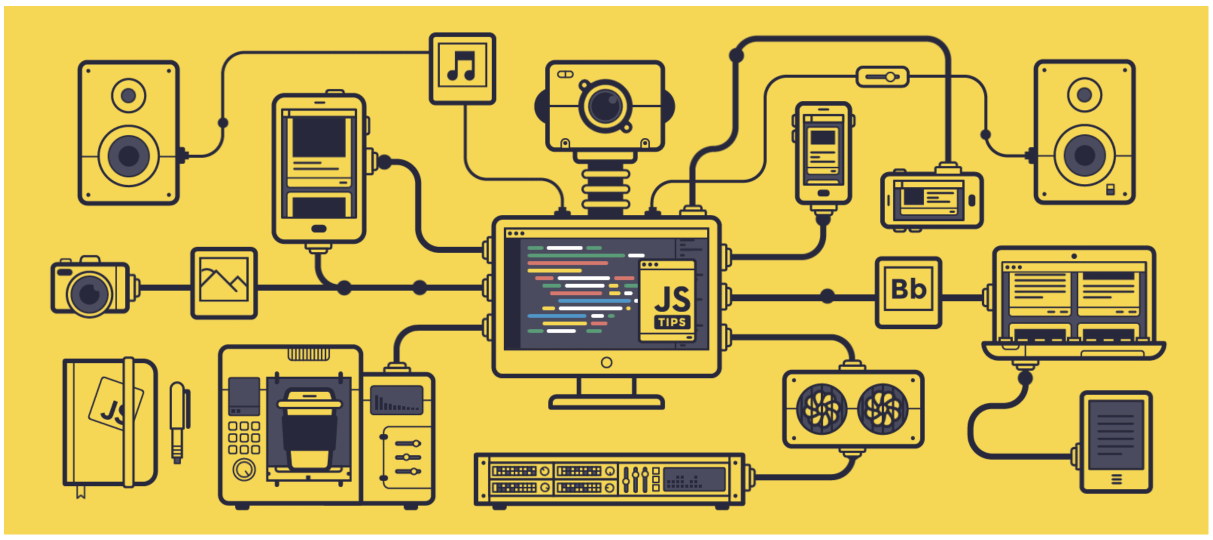
JavaScript supports much of the structured programming syntax used by the C programming language. Therefore once you have learned JavaScript, learning other programming languages that use the C syntax, such as PHP, C#, Java, etc. will be much easier.

The following figures show where JavaScript is used nowadays:



Note that the support and state of technologies is always in flux. New JavaScript game frameworks and interactive graphics libraries are created regularly, for example:

- Phaser
- Three.js (animating 3D computer graphics, 3D games for web browsers)
- PixiJS



What is a script

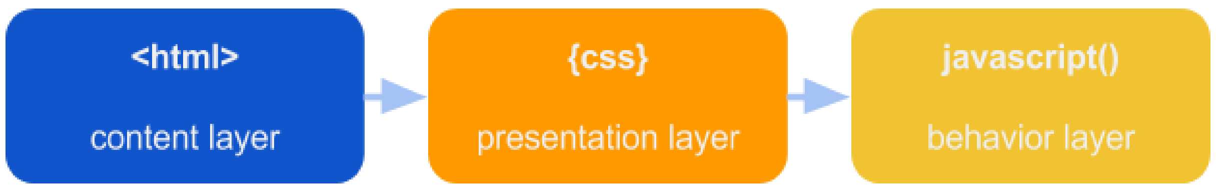
What we call a **script is a series of instructions** that a computer can follow to **complete a task**. You might know similar things in your non-digital life, such as:

1. **User Handbooks**
   Information for things like evacuation procedures or training instructions.
2. **Manuals**
   User manuals i.e. for cars, motorbikes or even computer games, that give instructions on both the use as well as how to find and fix common errors.
3. **Recipes**
   Following a recipe, anyone can cook a delicious meal even if they have never seen or done it before.

JavaScript scripts will be executed by your computer, i.e. by the browser, line by line and step by step. Depending on how the program is written, as well as the user input, different parts may be used at different times. To write such a code, you need to first **state your goal** and then **list the tasks** that need to be completed **to achieve it**. It is best to start with the big picture of what you want to achieve, and break that down into smaller steps. If the task still seems overly complicated, use even smaller steps to move towards the goal.

How HTML, CSS, & JavaScript fit together

Before we go into the usage of JavaScript, we need to understand how it interacts with HTML and CSS. Web developers usually talk about three technologies that are commonly used to create web pages: HTML, CSS, and JavaScript. Following the principle of "separation of concerns", wherever possible, **aim to keep them in separate files**. When you do this, the HTML page will then link to the CSS and JavaScript files. Each technology forms a separate part with a different purpose, building on each other.



HTML - This is where the **content** of the page is described. HTML is what gives the page structure and adds semantics.

CSS - We use CSS to style and enhance the HTML page with rules that describe how the HTML content is **presented** (colors, distances, placement, opacity, etc.).

JavaScript - Here we define how our code **behaves**, adding interactivity. To keep a good overview of our project, it is best to keep JavaScript in separate files.

Placing and Including JavaScript

Internal JS

To write some JavaScript code we place it between opening **<script> and closing </script> tags** inside the <body> tag (most common practice is **before the closing </body> tag**).

External JS

We can include **external JS code** either from our website or from anywhere on the internet, and this is the recommended method because it is better organized and easier to read when you split your html and JavaScript code. For such a purpose, we use the following syntax before the closing body tag:

```
<script type="text/javascript" src="script.js"></script>
```

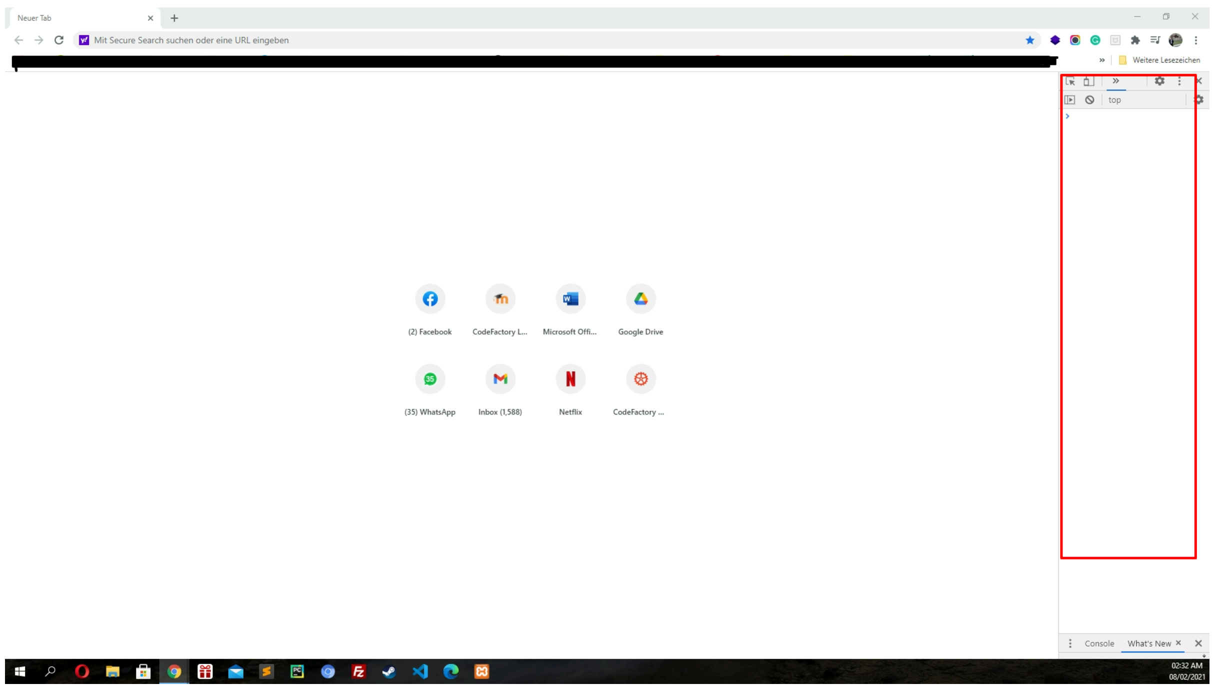
Note: External JS scripts should **not include any <script> or </script> tags**, as they are an HTML tag and not JavaScript. The browser already knows that a JavaScript file ".js" is being loaded. Putting them in the JavaScript files will cause an error.

The Chrome JavaScript console

For us, as aspiring developers, it is great to know that there is a console and it exists almost in all browsers. It is a **DevTools** feature that gives you many ways to troubleshoot HTML, CSS, and JavaScript problems.

The **JavaScript console** is perfect for helping you find errors in your code. It not only describes the errors it finds but also **identifies the line** in your code where each error occurred.

To **access** the console in the **browser**, press **F12** or **CTRL+SHIFT+I (Windows) / CMD+OPTION+J (MAC)** or **right-click on the browser, Inspect and choose the console tab**.



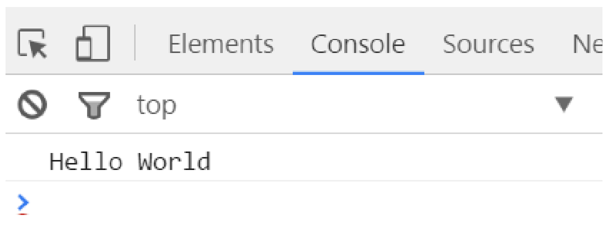
It is incredibly helpful and efficient to test your code in the console and to see if you have any error in your code, including the file and line it is in, in case you have one ;).

Hello World Example

Let's create a Hello World program using Javascript. In order to print something in the console we need to learn this command (console.log):

```
<html>
<head><title>Hello World</title></head>
<body>
  <script type="text/javascript">
     console.log("Hello World") //prints output on the console
  </script>
</body>
</html>
```

On the 4th line, we are opening the <script> tag and closing it on line 6. Between the script tags we write the **console.log function that outputs a message "Hello World" to the web console**. Click **F12** on your keyboard while you are inside the Web Browser and navigate to the console. Here you can see the output of the console.log function.



```
Hello World
```

As we now know, the console is mostly used for debugging, logging, and testing purposes. From here on out, we will use the Chrome JavaScript console to do these things.

Within the console, we will often get error messages that try to give us information on the type and source of the bug we encountered. The reasons for bugs are very varied, but here are a few of the most common:

1. **Missing punctuation**: JavaScript programming often relies on clear structures that show when certain elements begin and end, such as closing parentheses and brackets. Should you happen to leave out the closing quotation marks for example when writing console.log("Hello World"; you will receive the: "**Unexpected token**;" message, meaning that Chrome was expecting something other than the semicolon after the closing quotation marks.
2. **Missing quotation marks:** As you will learn later in more detail, JavaScript is designed to work with different data types. One of those is text, or what programmers call a "string". It is defined by the set of quotation marks, one at the beginning and one at the end of the string. Should you forget one of them, or inadvertently mix up double and single quotes, you might receive an "**Uncaught SyntaxError: Unexpected token ILLEGAL**" error.
3. **Typos in commands:** JavaScript will do its best to follow your commands to the letter. This also means however, that if you mix up or forget a letter, it will not know what you are commanding it to do. If you want to write onto your website and, instead of "document.write" you mistakenly type "docment.write" forgetting the "u" in it, you will receive an "**Uncaught ReferenceError: document is not defined**" error since you misspelled the document object.
4. **Syntax error:** If JavaScript can not identify exactly where the mistake in your code is, it will provide you this **generic error message**. You might have to take a closer look into the structure and immediate surroundings of the link where this error occurred. While this type of error requires a bit more digging to find the cause of, it will also give you excellent experience in understanding the intricacies of JavaScript when you have solved it.

Comments

It's easy to forget how a program works and why you wrote your code the way you did. Fortunately, most programming languages provide a way for programmers to leave notes for themselves or other programmers who might look through their code. A comment is simply a line or more of notes rather than actually parsed code: The JavaScript interpreter ignores them, but they can provide valuable information on how your program works.

When writing comments, we should keep to some guidelines and best practices, such as:

- Keep your comments short, relevant and precise
- Use comments comments liberally
- Ask for or define a standard of using comments in your projects, especially if you are working in a team.

There are also different places where you can put a comment. So-called “Header block” comments can give information and notes relevant to the entirety of the file they are placed in, or to the things right at the top of the file (for example a group of defined variables).
Alternatively, “In line” comments can be placed anywhere within the code that you find useful. Still, we would recommend sticking to a standard, i.e. when commenting functions to do so in the line right above the function declaration.

Now that we have talked about why and where we write comments, this is the “how”:

Single line comment

```
// This is a comment
```

Multiline comment

```
/* This is a section
of multiline comments
that will not be
interpreted */
```

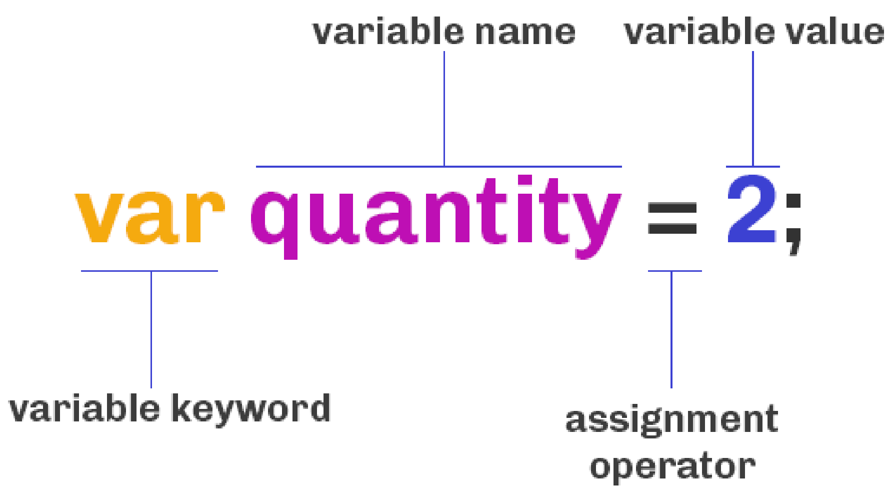
Storing Data

Let’s consider two ways of storing data: storing single pieces of data in variables and storing multiple entries using arrays.

Variables

A script will have to temporarily store the bits of information it needs to do its job. It can **store this data in variables**. A variable is a good name for this concept because the data stored in a variable can change (or vary) each time a script runs. You could, for example, want to save a value somewhere, and later on, you change or add value to it. To do this, you need to save the value somewhere. To achieve this, we use variables.

We are declaring a variable in the following way:

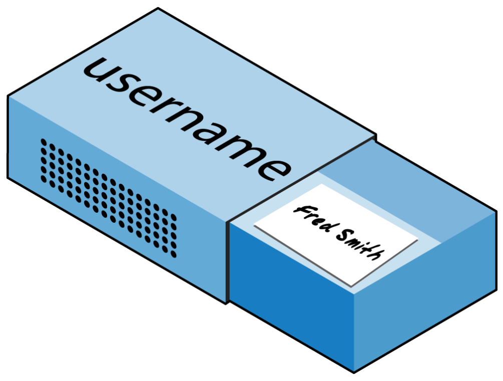


1. Set the keyword **var**
2. Define the **name** of the variable
3. Assign a **value** to it

Above we set the variable called “quantity” and assign the value of 2 to it.

Until you have assigned a value to a variable, we say **the value is undefined.**

There’s a simple metaphor that will help understand what variables are all about. Imagine you have a matchbox on which you have written the word username. You then write Fred Smith on a piece of paper and place it into the box. Well, that’s the same process as assigning a string value to a variable, like this:



```
var username = "Fred Smith";
```

You can think of variables as matchboxes containing items. All JavaScript **variables** must be identified with **unique names**. These unique names are also called identifiers.

Note:
some names are already reserved in JavaScript like function, location, and more. You will learn more about them in the upcoming days.

Variable’s declaration: var, let, and const

As seen before, declaring and using a variable isn't that hard. In the beginning of JavaScript, there was only one way to declare variables, and that was using the **var** keyword. Since there were some issues with variables declared using the var keyword it was necessary to evolve new ways for variable declaration. With the introduction of ES6 version of JavaScript, two new ways of creating variables were introduced: **let** and **const**. They were introduced to have a better control over the scope of the variable and to help with hoisting issues. To understand their differences and when to use them, we first need to understand how var, let and const behave.

Var

Keyword var originated from the very first days of JavaScript. It is generally not used in modern JavaScript, but still lurks in the old ones.

```
var myVar = 'I like coding in JavaScript';
console.log(myVar);
```

Above we have declared a variable called “**myVar**” and assigned a value of string to it.

Let

The **let** keyword is used to create or declare a variable. Variables that are declared with let can be reassigned to a different value, like in the example below.

```
let color = 'red';
color = 'blue';
console.log(color); // blue
```

With **let** we can declare a variable without assigning a value at all. We can reassign the value of the variable.

```
let age;
console.log(age); // undefined
age = 32;
console.log(age); // 32
```

Const

We use **const** keyword to declare a variable that can store any value. The structure of **const** remains the same as **let** and **var**.

```
const url = "https://www.google.com/";
console.log(url); // https://www.google.com/
```

However, if you try to reassign a **const** variable you will get a **Type Error** because constant variables cannot be changed once they've been declared.

```
const url = "https://www.google.com/";
url = "https://www.youtube.com/";
console.log(url); // Type Error: Assignment to constant variable
```

If you declare a **const** variable without a value, you will get a **SyntaxError**.

```
const url;
url = "https://www.google.com/";
console.log(url); // Syntax Error: Missing initializer in const declaration
```

var, **let** and **const** also differ in aspect of scope, for which we will learn in the upcoming lessons.

Data Types

To work with information, computers, just like humans, need to classify them into categories. These categories, or types of data, determine how JavaScript will treat the information and what we can do with it. Using different data types allows our program to e.g. keep track of the total of a bill, decide if a user is allowed to proceed, or greet the user with a friendly individualized message.

The set of types in JavaScript consists of:

- Primitive data types
  - **Boolean** - This data type only has two possible values - either true or false.
  - **Number** - Any number, including decimal and negative numbers.
  - **BigInt** - Any number, greater than $2^{53}-1$ or less than $-(2^{53}-1)$ with **n** appended to the end. They are rarely needed, so we don't cover them here.
  - **String -** Any grouping characters on your keyboard surrounded by single or double quotes.
  - **Null** - This data type represents the intentional absence of any value
  - **Undefined -** This data type also represents the absence of a value though it has a different use than null.
  - **Symbol -** is a unique identifier, useful in more complex coding. No need to worry about this for now
- Objects

The first 7 of those are considered primitive and are the most basic data types in JavaScript. Objects on the other hand are a little more complex, we will discuss them in the upcoming days.

Boolean

The Boolean data type can only take one of two values, **true or false**. Boolean data types are necessary to create software with decision-making capabilities. For example, to check if the customer of your webshop is of a legal age to purchase your products, you can **add logic** to your page by working with the information: “Is my user 18 or older?” The answer is a Boolean value: either they are (true) or they are not (false).

```
let isOlderThen18 = true;
console.log(isOlderThen18); // true

console.log(5 == 6); // false
```

Number

In JavaScript, a number is represented by a **numeric character**; 10 is the number ten. You can also use decimals like 6.5 or 10.3333333. JavaScript even lets you use negative numbers, like –130 or -6.9.

Other programming languages usually categorize numbers into different types and treat each type in a different way. E.g. Java categorizes numbers into int (integers) and float (with decimals) and each has also subcategories. In JS there is only one variable for a number and we don’t need to declare it. We can just write the number.

```
const firstValue = 3;
console.log(firstValue); // 3
console.log(10 + 5); // 15
```

String

To display any kind of text, long or short, we use the data type of **string**. A string is a **series of characters** that are enclosed **in quotation marks**. For example, “Hello World”, and "I am hungry" are both examples of strings. So is “21”, since it is enclosed in quotation marks and therefore not recognized as a number.
A string opening quote mark tells the JavaScript interpreter that what follows is a **string**. When the second quotation mark is encountered, JavaScript knows that it has reached the end of the string.

```
console.log("This is a string");
console.log('This is also a string');
```

Strings are simply a group of characters like **'JavaScript'**, **'Hello world!'**, **'http://www.codefactory.wien'** or even **'14'** (the number fourteen is in single quotes). When writing strings in JavaScript, always write them within **single or double quotes**. This informs the browser that it is dealing with a string. **Don't mix up your quotes**, if you start a string with a single quote and end it with a double quote, JavaScript doesn't understand what you mean.

Null

The Null type has exactly one value: null. It represents the intentional absence of a value, meaning a variable has been explicitly set as (null = no value) or has been initialized and defined to be nothing.

```
const number = null;
console.log(number); // null
```

Undefined

The **undefined** data type represents a value that is not assigned. If a variable is declared but is not assigned to any value, then the value of that variable will be **undefined**.

```
let x;
console.log(x); //undefined
```

Difference between **null** and **undefined**?

Null and **undefined** are both for the absence of a value, but they have different meanings.

```
// Undefined means there should be some values, but it is undefined now
let  finishCourseTime = undefined;

// Null means there is no value here
let finishStudyingDate = null;
```

Type Conversions

Many types of values are rather flexible in JavaScript. Some values (“truthy” values) are considered equivalent to true and others (“falsy” values) are considered equal to false. Similar situations are true for other types: if JavaScript expects a string, it will convert whatever value you give it to a string. If JavaScript expects a number, it will try to try to do the same and convert it to a number (or to NaN if it cannot perform a meaningful conversion). Using the **typeof operator, we can find out the data type of a JavaScript variable**, as shown below:

```
console.log(typeof isLateToWork); // Returns "boolean"

let result = "4" - "hello";
console.log(result); // Returns "NaN"
console.log(typeof result); // Returns "number"
```

Explicit Conversions

In case we want to strictly define the variable type, we use JavaScript Type Conversion built-in methods:

1. **String()** - converts **numbers to strings**

```
//Converting "number"  to "string".

let result = 10;
console.log(typeof result)      //Returns "number"

let newResult = String(result);
console.log(typeof newResult);  // Returns "string"
```

2. **Number()** - converts **strings to number**

```
// Converting "string" to "number".
let firstValue = "123";
console.log(typeof firstValue)  //Returns "string"

let newValue = Number(firstValue);
console.log(typeof newValue);   // Returns "number"
```

Concatenation

Combining multiple strings to make a single new string is a common programming task. For example, you can take a visitor's name and address to give him a personalized welcoming message such as “Welcome Bob, we are not delivering to Bobtown!”
Combining strings is what we call concatenation, and it can be accomplished in different ways.

(+) operator

You can use the + operator to concatenate strings together. This is the same + operator you use to add number values, but with strings, it behaves a little differently.

Here’s an example:

```
let firstName = 'Bob';
let lastName = 'Boston';
let fullName = firstName + ' ' + lastName;
console.log(fullName);   //Returns "Bob Boston"
```

you can write the code in another way :

```
let firstName = 'Bob',
  lastName = 'Boston',
  fullName = firstName + ' ' + lastName;
console.log(fullName);   //Returns "Bob Boston"
```

The ' ' in the last line is a single quote, followed by a space, followed by another closing single quote. When we place this between the two variables containing the names, we add an empty space between them. This is one example on how to add multiple strings together.
Now let's consider the following scenario:

```
console.log("I'm Bob and I am hungry");
```

This string has a single quote in it, so JavaScript thinks that the string ends at this point, therefore what comes after it will be misunderstood by the JavaScript interpreter and will create an error message. So you need to **escape the single quote**, **telling the browser to treat** it **as a character**, instead of a command to end the string. This is done by placing a **backslash "\"** before it:

```
console.log("I\'m Bob and I am hungry");
```

Escape characters, as seen above when we inserted the single quotation mark, can also **be used to insert various special characters** such as tabs, newlines, and other often used elements. Below you will see a list of all JavaScript escape characters available:

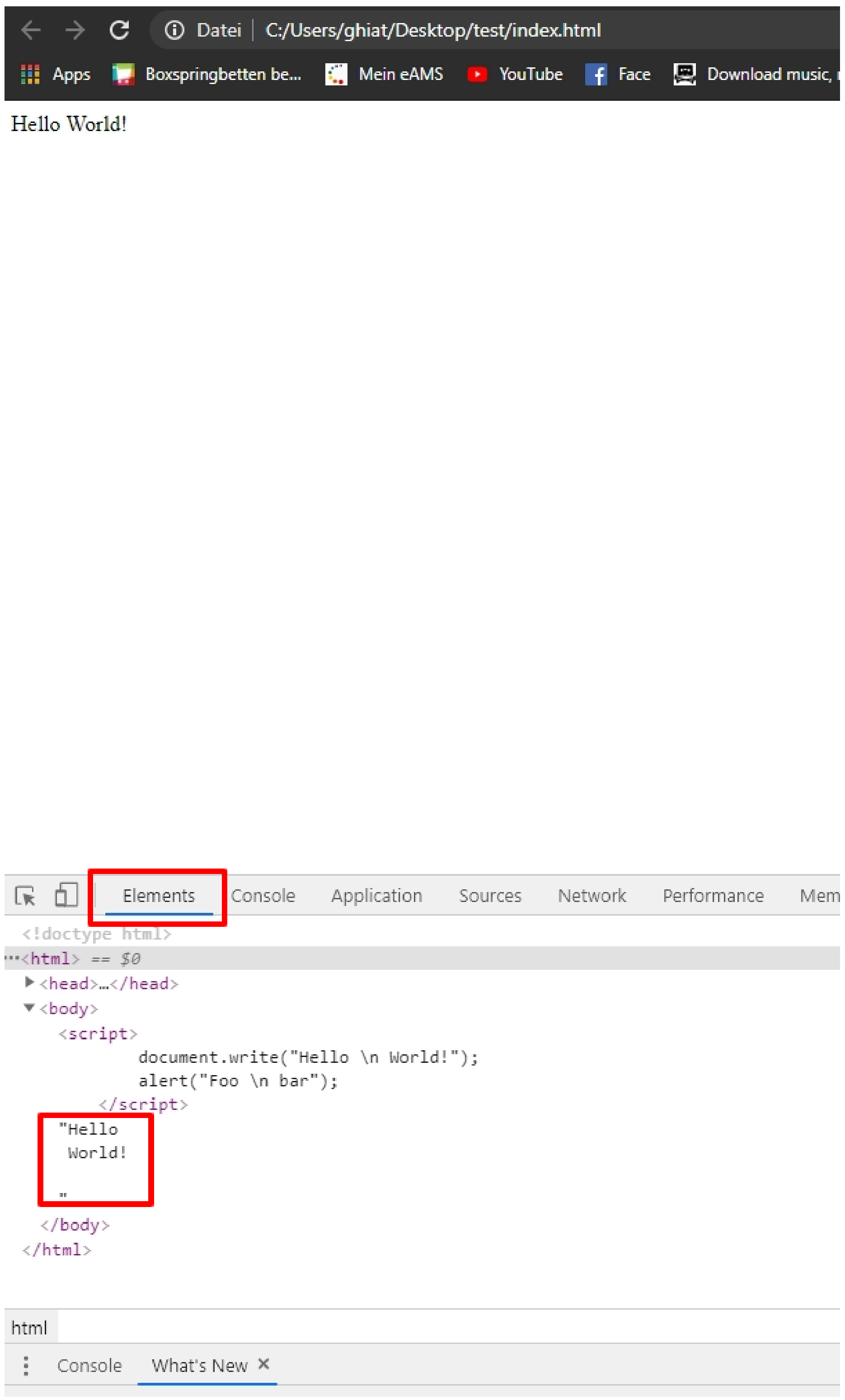
Character	Meaning
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\'`	Single quote (or apostrophe)
`\"`	Double quote
`\\`	Backslash
`\XXX`	An octal number between `000` and `377` that represents the Latin-1 character equivalent (such as `\251` for the © symbol)
`\xXX`	A hexadecimal number between `00` and `FF` that represents the Latin-1 character equivalent (such as `\xA9` for the © symbol)
`\uXXXX`	A hexadecimal number between `0000` and `FFFF` that represents the Unicode character equivalent (such as `\u00A9` for the © symbol)

Note:
JavaScript is not limited to dealing with HTML, it's a general-purpose language. When dealing with other things than HTML, \n functions as a line break. For instance, alert("Foo \n bar"); shows two lines in the alert box. Even though the text we gave to alert isn't HTML, we can see the changes in the element from the console, because the escape character is not supported for HTML.

Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>
     test
  </title>
</head>
<body>
  <script>
     document.write("Hello \n World!");
     alert("Foo \n bar");
  </script>
</body>
</html>
```

In the browser, we will see both words displayed in one line, but if we check the alert box we will see it is displayed in two separate lines as indicated by the \n element. This is because JavaScript is a general language, not all the features are supported for HTML, but you can see a new line if you check the console.



Hello World!

```
<!doctype html>
…<html> == $0
  ▶<head>…</head>
  ▼<body>
     <script>
             document.write("Hello \n World!");
             alert("Foo \n bar");
         </script>
     "Hello
      World!

     "
   </body>
</html>
```

Template literals

In the Es6 version of JavaScript, we can insert or interpolate variables into strings using template literals. The string

interpolation in JavaScript is performed by template literals which are wrapped by backticks ` (the key is usually located on the top of your keyboard, left of the 1 key) and **${expression}** as a placeholder.
In the example below, inside the template literal, you'll see placeholders like **${myName}** and **${favoriteCity}**. The values of myName and favoriteCity variables are inserted into the template literal.

```
let myName='John';
let favoriteCity= 'Vienna';
console.log(`My name is ${myName}. My favorite city is ${favoriteCity}`);
//  Output: "My name is John. My favorite city is Vienna"
```

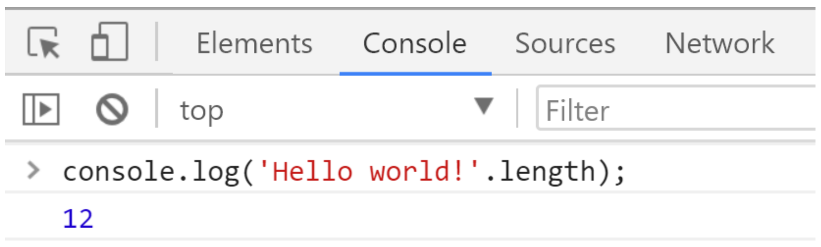
One of the biggest benefits to using template literals is that you can insert values into string literals in a concise and readable manner and avoid the clumsy string concatenation approach.

String Methods

Now that we've defined the strings, we can start using them. For instance, we can concatenate one string to another, we can take the pieces of a string between the second to the fourth character contained in a variable "b" and put them in the middle of the string inside the variable "a". We can read out what the twelfth character of "a" is; how many characters "b" has; if there's a letter 'q' in them and many more things.

To do this, we can use some **automatic predefined properties** that JavaScript assigns to each string. One of them is **.length**, a property that gives us the length of the string. So if we want to get the length of 'Hello world!' we can do:

```
console.log('Hello world!'.length);
// 12
// The length of the string is 12
```



```
> console.log('Hello world!'.length);
  12
```

The important thing is that **we can use .length on any string**: it's an automatic feature. We can read out the length of any string, whether JavaScript created it for us or we have declared it ourselves.

Below is a list of commonly used **built-in String functions:**

1. concatenating strings (+)
2. indexOf
3. lastIndexOf
4. charAt
5. length
6. split
7. toLowerCase and toUpperCase

All these functions can be used on any string simply by doing stringVariableName.command().

indexOf

One of the most widely used built-in methods is indexOf. Each character has an index number, giving its position in the string. Note that the first character has index 0, the second one index 1, etc. Using indexOf we can read out the index number of a character. Put the .indexOf(' ') behind the string name and put the character you are looking for between the quotes

.

```
let a = 'Hello world!';
console.log(a.indexOf('w')); //Output: 6
```

In this example, the output will be 6, because "w" is located in the 7th position. **If the same character occurs more than once**, this method outputs only the **first occurrence**.

```
let firstString = 'Hello world, waldo!';
console.log(firstString .indexOf('w'));
// The index is 6 (will show the first occurrence)

let secondString = 'Hello orld, waldo!';
console.log(secondString.indexOf('w'));
// The index is 12

let anotherString = 'Hello orld, Waldo!';
console.log(anotherString.indexOf('w'));
/* The index is -1 (indexOf is case sensitive. That means it can recognize the difference between uppercase
and lowercase. In this case, there is no lowercase "w"*/
```

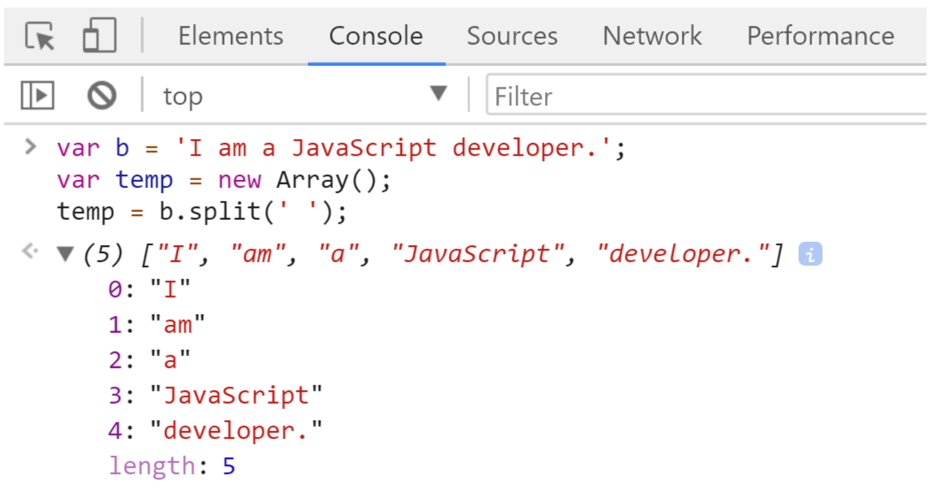
split

split() is another very useful built-in function. It allows you to **split a string at the place of a certain character**. You must put the result in an array, not in a simple variable. As an example, let's split b at the spaces inside the string.

```
let b = 'I am a JavaScript developer.';
let temp = new Array();
temp = b.split(' ');
console.log(temp);  // Output ["I", "am", "a", "JavaScript", "developer."]
```

First, we store the string ‘I am a JavaScript developer’ into the variable called b. Next, we define a new **empty array** called **temp**. Inside the temp array (because the split() function returns an array) **we store the result of the split function** that is called on the b variable. As a parameter to the split() function we pass an empty space. This tells the function to divide the string we encounter at every empty space. Now the previously single string has been split into 5 strings that are placed in the array temp. The spaces themselves are gone. The **output (console)** should look like this:

```
temp[0] = 'I';
temp[1] = 'am';
temp[2] = 'a';
temp[3] = 'JavaScript';
temp[4] = 'developer.';
```



```
> var b = 'I am a JavaScript developer.';
  var temp = new Array();
  temp = b.split(' ');
< ▼(5) ["I", "am", "a", "JavaScript", "developer."]
     0: "I"
     1: "am"
     2: "a"
     3: "JavaScript"
     4: "developer."
     length: 5
```

toLowerCase and toUpperCase

Two other useful methods are: **toLowerCase(),** which puts the whole string in lowercase, and **toUpperCase()** which puts the whole string in uppercase. For example:

```
let myTraining = 'I am becoming a Web Developer.';
console.log(myTraining.toUpperCase());
// "I AM BECOMING A WEB DEVELOPER."
```

outputs 'I AM A JAVASCRIPT HACKER.'

Naming Rules

There are a few general rules for naming variables:

- Variable names cannot start with numbers. The first character of a variable name can be only “**a-z**”, **“A-Z”**, **“$”**, or **“_”** .
- Variable names can only contain letters, numbers, underscores, or dollar signs; no other characters, such as spaces or punctuation may be used in a variable name.

- Variable names are case sensitive! Lowercase and uppercase letters are distinct. For example, **myName** and **myname** would be two different variables.
- Variable names cannot be the same as keywords. Keywords are reserved words that are part of the syntax in JavaScript.

You can use either single or double quotations marks:

```
greeting = "Hello there";
or
warning = 'General Kenobi';
```

You can also assign the value of one variable to another:

```
let meal = "Burrito";
let mainDish = "Chicken";
mainDish = meal;
console.log(meal); // Output: Burrito
console.log(mainDish); // Output: Burrito
```

Here we assign the value of meal to the value of mainDish, so they both have the same value.
Numeric variables are as simple as assigning a value in the Strings:

```
let count = 42;
let temperature = 98.4;
```

JavaScript is a very loosely typed language, usually, you don't have to worry about the type of data you are currently using. JavaScript figures out what you want and just does it.

Reserved words in JavaScript

There are words in JavaScript that are reserved for specific purposes: function, for example, is used to define a function, so you can't name a variable function. More than just that, many words, like alert, document, and window, are considered special properties of the web browser and the DOM within it. Using any of those reserved words in an inappropriate context will create a JavaScript error**.** Below you will see a list of some of those reserved words and it is a good idea to avoid them when choosing names for your variables.

Naming conventions

It is also good practice to make your variable names as easy to read as possible. Therefore when you use **more than one word in a variable name**, you should either utilize an **underscore "_"** between them (e.g. first_name) or capitalize the first letter of each word after the first (e.g. firstName) which is also known as "**camelCase"** notation. Generally, camelCase is the more commonly used notation.
This way the **first word is all lowercase and each subsequent word starts with an uppercase character**.
To demonstrate, you will find userfirstname much harder to read than user_first_name or userFirstName.
To learn more about reserved JavaScript terms, take a look at these resources: https://www.w3schools.com/js/js_reserved.asp

Arrays

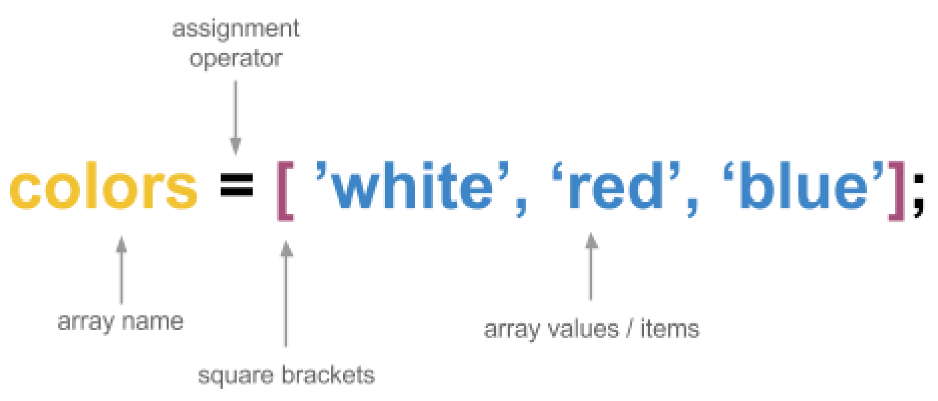
Organizing and storing data is a foundational concept of programming. One way we organize data in real life is by making lists for example.
Arrays are JavaScript's way of making lists. An array is a special type of variable that is capable of storing not just one single value, but a list of different values. It is especially useful when we deal with multiple values that are connected to each other in their use.
They are especially helpful when the exact number of entries is not known or might change in the future.
An Array can be declared in the following three ways:

1. Array literal notation

2. Array() constructor
3. Array.of() method

Array literal notation

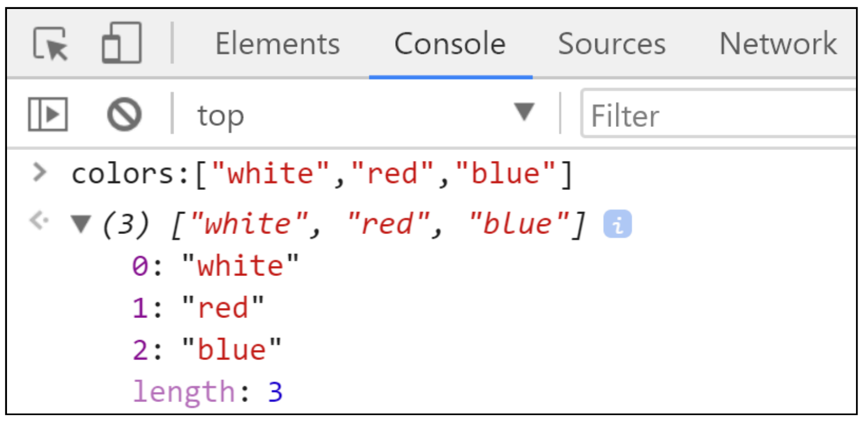
An array literal creates an array by wrapping items in square brackets [ ], in this following way:



Here we create an array called colors and assign three items to it (the strings 'white', 'red' and 'blue'). **Each item in an array is automatically given a reference number called an index**.

The index always starts from "zero", that is why color [1] will output the second item ('red') to the console. Visually it looks like this:

INDEX	VALUE
0	'white'
1	'red'
2	blue'



```
> colors:["white","red","blue"]
< ▼ (3) ["white", "red", "blue"]
      0: "white"
      1: "red"
      2: "blue"
      length: 3
```

Array() constructor

The new keyword used along with an Array() constructor function will initialize a new array. As a parameter you can pass the number of elements you want to create. For example:

```
let fruitArray = new Array(5);
console.log(fruitArray.length); //Output: 5
```

Array.of()

This method creates a new array with the arguments passed to it as the new array's values.

```
let another = Array.of(1, 2, 3);
console.log(another) // Output: [ 1, 2, 3 ]
```

Note: Declaring arrays with the array literal syntax is more popular!

```
let colors = ['red', 'white', 'blue'];
```

We mentioned that arrays in JavaScript are zero-indexed, meaning the positions start counting from 0 rather than 1. We can access individual items using their index.

```
let colors = ['red', 'white', 'blue'];
console.log(colors[1]); //Output: white
```

Once you have access to the elements in the array, you can update their values as in the following example:

```
let colors = ['red', 'white', 'blue'];
colors[1] = 'orange';
console.log(colors); //Output: ['red', 'orange', 'blue']
```

We have already mentioned that variables can be declared using the **let** and **const** keyword. Variables declared with the const keyword cannot be reassigned, while variables with let can be reassigned. However, the items in an array declared with **const** are mutable. We can change the content of a **const** array, but cannot reassign a different value.

```
const myHouse = ['living-room', 'bathroom', 'kitchen'];
myHouse[1] = 'dishwasher'; // we can change an existing element
myHouse.push("bedroom");  // we can add new elements
console.log(myHouse);
//Output: ["living-room","dishwasher","kitchen","bedroom"]
```

We saw that we can change values that exist, add or delete existing values in an array, but we cannot change it entirely.

```
const myHouse = ['living-room', 'bathroom', 'kitchen'];
myHouse = ['bedroom']; // TypeError: Assignment to constant variable.
```

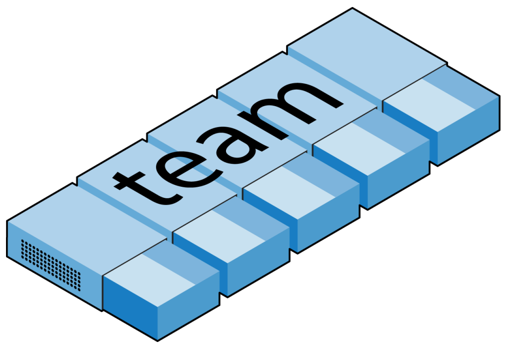
Note:
If you try to check the type of a variable that has a list of values (an array) you will see it is a type of JavaScript object

```
let colors = ["red","blue"]
console.log(typeof(colors));
// the result will be "object"
```

This object has keys and values. The key is the index of the element [0] and the corresponding value is the value of the element “red” for example.

One-dimensional array

Do you remember the metaphor about the matchboxes that represent a variable? Similarly, you can think of **Arrays** as several matchboxes stuck together.



Imagine that we should store players' names for a five-person football team in an array called ”team”. This is easier done when we use one Array instead of five different variables.

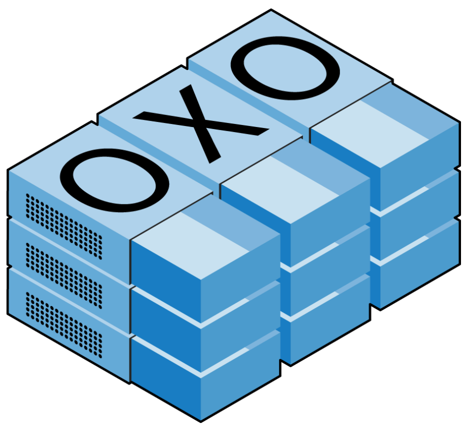
```
let team = ['Bill', 'Mary', 'Mike', 'Chris', 'Anne'];
```

If we want to display the name of the third player, we will use this command:

```
console.log(team[2]); // Output: Mike
```

Two-dimensional array (2D)

Multidimensional arrays enable us to create arrays of more than one dimension. In fact, they can have as many dimensions as we want (or can keep track of when programming).



Let’s say we want to create a tic-tac-toe game in JavaScript, which requires a data structure of nine cells arranged in a 3x3 square. To represent this with matchboxes, we need a matrix of three rows by three columns.

This is how we can write it using the JavaScript syntax:

```
let tic-tac-toe = [['1', '2', '3'], ['4', '5', '6'], ['7', '8', '9']];
```

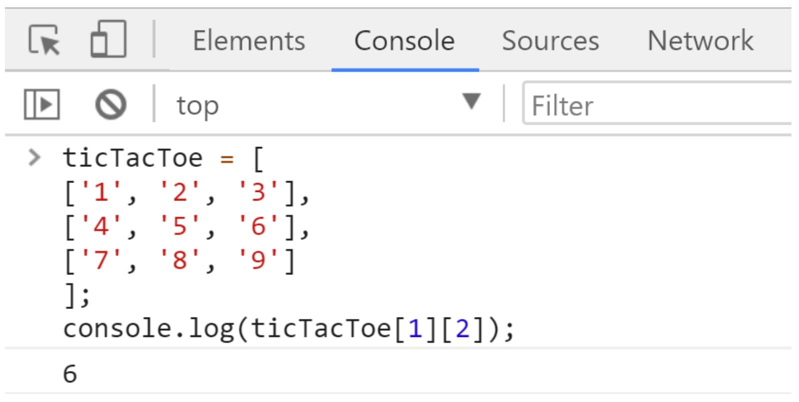
Or like this:

```
let tic-tac-toe = [
['1', '2', '3'],
['4', '5', '6'],
['7', '8', '9']
];
```

In order to access the number six, which is two down and three along in this matrix (because array elements start at position 0), we use the following command:

```
console.log(tic-tac-toe[1][2]);
```

The statement will output the number 6 (six).
JavaScript arrays are powerful storage structures for dealing with data.



```
> ticTacToe = [
  ['1', '2', '3'],
  ['4', '5', '6'],
  ['7', '8', '9']
  ];
  console.log(ticTacToe[1][2]);
  6
```

Array Methods

Push Method

The push() method can **append one or more elements to the <u>end</u> of an array**.

```
let data = ["X"];
data.push("A");
data.push("B", "C");
console.log(data); // Output: ["X","A","B","C"]
```

Notice that the push() method changes or mutates the current array. We also can refer to push() as a destructive array method since it changes the initial array.

Pop Method

The pop() method **pulls the last element off** of the given array and returns it.

```
const letters = ["A", "B", "C"];
const removed = letters.pop();
console.log(letters); //Output: ["A", "B"]
console.log(removed); //Output: C
```

pop() method as well changes the initial array.

Unshift method

The unshift() method is **like the push()** method, only it works at the **beginning of the array**. The unshift() method can prepend one or more elements to the beginning of an array. This alters the array on which the method was called.

```
let data = ["X"];
data.unshift("A");
data.unshift("B", "C");
console.log(data);      // Output: ["B", "C", "A","X"]
```

Shift Method

The shift() method is **like the pop()** method, only it works **at the beginning of the array**. The shift() method pulls the first element off of the given array and returns it. This alters the array on which the method was called.

```
const letters = ["A", "B", "C"];
const removed = letters.shift();
console.log(letters);    //Output: ["B", "C"]
console.log(removed);    //Output: A
```

Length Method

The length property sets or returns the number of elements in an array. Remember that the length of an array is always equal to the number of its last index + 1 (since the first index of an array is always 0).

```
let fruits = ["Apple", "Banana", "Kiwi", "Watermelon"];
console.log(fruits.length);   // Output: 4
```

Sort Method

The sort() method is used to sort the items of an array. Unless given other instructions, the sort() method sorts the values as strings in **alphabetical and ascending order**.

```
let fruits = ["Orange", "Kiwi", "Grape", "Watermelon"];
console.log(fruits.sort());

// ["Grape", "Kiwi", "Orange", "Watermelon"]
```

Slice Method

The slice() method takes two arguments, delineating where it starts and ends at. The returned array starts the first argument and ends at, but does not include, the latter given argument. The original array will not be changed.

```
var fruits = ["Banana", "Kiwi", "Apple", "Watermelon"];
console.log(fruits.slice(0, 2));
// ["Banana", "Kiwi"]

var fruits = ["Banana", "Kiwi", "Apple", "Watermelon"];
console.log(fruits.slice(2));
// ["Apple", "Watermelon"]

var fruits = ["Banana", "Kiwi", "Apple", "Watermelon"];
console.log(fruits.slice(1, 2));
// ["Kiwi"]
```

Splice method

The splice() method can be used **to change an array by adding, removing or replacing items**:

```
let cities = ["Vienna", "London", "Paris", "Madrid"];
cities.splice(2, 0, "Moscow", "Berlin");
console.log(cities);

// ["Vienna", "London", "Moscow", "Berlin", "Paris", "Madrid"]

cities.splice(1,1);
console.log(cities);
// ["Vienna", "Moscow", "Berlin", "Paris", "Madrid"]

cities.splice(2, 2, "Rome");
console.log(cities);
// ["Vienna", "Moscow", "Rome", "Madrid"]
```

The **first parameter** (with the value of 2) in *cities.splice(**2**,0,"Moscow","Berlin")*; defines the **position** where new elements should be added. The **second parameter** (with the value of 0) defines **how many** elements should be **removed** (none in this case). The rest of the parameters ("Moscow" and "Berlin") are the **new elements** to be added.

The first parameter in cities.splice(1,1) (with the value of 1) defines the position in which changes will be made, and the second one refers to how many elements will be deleted ("London" in this case). No elements have been added.

The first parameter in cities.splice(2, 2, "Rome") (with the value of 2) defines the position in which changes will be made, and the second one refers to how many elements will be deleted ("Berlin" and "Paris" in this case). "Rome" will also be added at that position.

Join method

The join() method is used **to return the array as a string**.

```
let subjects = ["English","Chemistry","Math","Physics"];
console.log(subjects.join());
// English,Chemistry,Math,Physics

console.log(subjects.join(" and "));
// English and Chemistry and Math and Physics
```

The default separator is comma (,).

There are many more array methods which you can find on the MDN array documentation.

◄ Bootstrap | Day 2 | Exercises

Jump to...

Javascript | Day 1 | Quiz ►