# Getting started with JAVASCRIPT

## A high-level definition

[JavaScript](https://developer.mozilla.org/en-US/docs/Glossary/JavaScript) ("JS" for short) is a full-fledged [dynamic programming language](https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_programming_language) that can add interactivity to a website. It was invented by Brendan Eich (co-founder of the Mozilla project), the Mozilla Foundation, and the Mozilla Corporation.

JavaScript is a scripting or programming language that allows you to implement complex features on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc.

Combination of HTML, CSS and Javascript is also called three layer of cake which is combined nicely to produce a beautiful website.

Lets Start with an example –

<p>Player 1: Sachin</p>

Then we can add some CSS into the mix to get it looking nice:

p {

font-family: 'helvetica neue', helvetica, sans-serif;

letter-spacing: 1px;

text-transform: uppercase;

text-align: center;

border: 2px solid rgba(0,0,200,0.6);

background: rgba(0,0,200,0.3);

color: rgba(0,0,200,0.6);

box-shadow: 1px 1px 2px rgba(0,0,200,0.4);

border-radius: 10px;

padding: 3px 10px;

display: inline-block;

cursor: pointer;

}

And finally, we can add some JavaScript to implement dynamic behaviour:

const para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {

let name = prompt('Enter a new name');

para.textContent = 'Player 1: ' + name;

}

## So what can it really do?

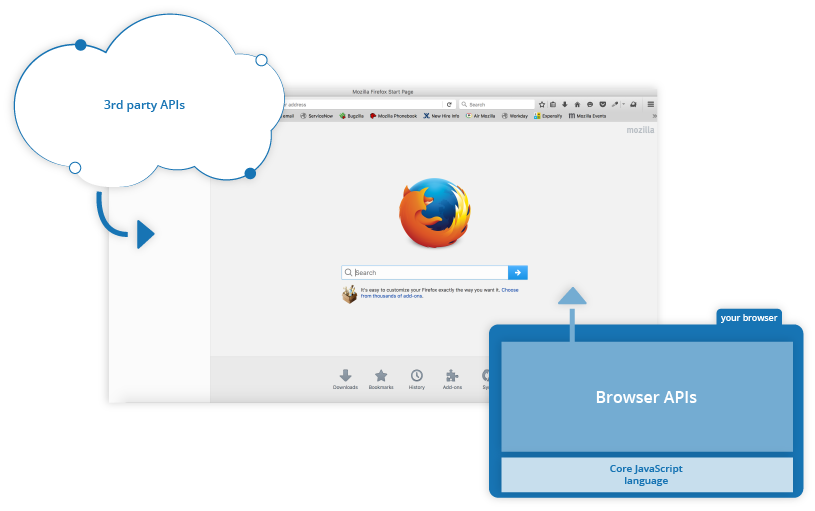
The core client-side JavaScript language consists of some common programming features that allow you to do things like:

* Store useful values inside variables. In the above example for instance, we ask for a new name to be entered then store that name in a variable called name.
* Operations on pieces of text (known as "strings" in programming). In the above example we take the string "Player 1: " and join it to the name variable to create the complete text label, e.g. ''Player 1: Sachin".
* Running code in response to certain events occurring on a web page. We used a [click](https://developer.mozilla.org/en-US/docs/Web/Events/click) event in our example above to detect when the button is clicked and then run the code that updates the text label.
* And much more!

What is even more exciting however is the functionality built on top of the client-side JavaScript language. So-called **Application Programming Interfaces** (**APIs**) provide you with extra superpowers to use in your JavaScript code.

APIs are ready-made sets of code building blocks that allow a developer to implement programs that would otherwise be hard or impossible to implement.

They generally fall into two categories.



**Browser APIs** are built into your web browser, and are able to expose data from the surrounding computer environment, or do useful complex things. For example:

* The [DOM (Document Object Model) API](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) allows you to manipulate HTML and CSS, creating, removing and changing HTML, dynamically applying new styles to your page, etc. Every time you see a popup window appear on a page, or some new content displayed (as we saw above in our simple demo) for example, that's the DOM in action.
* The [Geolocation API](https://developer.mozilla.org/en-US/docs/Web/API/Geolocation) retrieves geographical information. This is how [Google Maps](https://www.google.com/maps) is able to find your location and plot it on a map.
* The [Canvas](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API) and [WebGL](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API) APIs allow you to create animated 2D and 3D graphics. People are doing some amazing things using these web technologies —see [Chrome Experiments](https://www.chromeexperiments.com/) and [webglsamples](http://webglsamples.org/).
* [Audio and Video APIs](https://developer.mozilla.org/en-US/Apps/Fundamentals/Audio_and_video_delivery) like [HTMLMediaElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement) and [WebRTC](https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API) allow you to do really interesting things with multimedia, such as play audio and video right in a web page, or grab video from your web camera and display it on someone else's computer.

**Third party APIs** are not built into the browser by default, and you generally have to grab their code and information from somewhere on the Web. For example:

* The [Twitter API](https://dev.twitter.com/overview/documentation) allows you to do things like displaying your latest tweets on your website.
* The [Google Maps API](https://developers.google.com/maps/) and [OpenStreetMap API](https://wiki.openstreetmap.org/wiki/API) allows you to embed custom maps into your website, and other such functionality.

## What is JavaScript doing on your page?

Let's briefly recap the story of what happens when you load a web page in a browser When you load a web page in your browser, you are running your code (the HTML, CSS, and JavaScript) inside an execution environment (the browser tab). This is like a factory that takes in raw materials (the code) and outputs a product (the web page).

A very common use of JavaScript is to dynamically modify HTML and CSS to update a user interface, via the Document Object Model API (as mentioned above). Note that the code in your web documents is generally loaded and executed in the order it appears on the page. If the JavaScript loads and tries to run before the HTML and CSS it is affecting has been loaded, errors can occur.

Browser security

Each browser tab has its own separate bucket for running code in (these buckets are called "execution environments" in technical terms) — this means that in most cases the code in each tab is run completely separately, and the code in one tab cannot directly affect the code in another tab — or on another website. This is a good security measure — if this were not the case, then hackers could start writing code to steal information from other websites, and other such bad things.

JavaScript running order

When the browser encounters a block of JavaScript, it generally runs it in order, from top to bottom. This means that you need to be careful what order you put things in. For example, let's return to the block of JavaScript we saw in our first example:

const para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {

let name = prompt('Enter a new name');

para.textContent = 'Player 1: ' + name;

}

Interpreted versus compiled code

You might hear the terms **interpreted** and **compiled** in the context of programming. In interpreted languages, the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it. The code is received in its programmer-friendly text form and processed directly from that.

Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer. For example, C/C++ are compiled into machine code that is then run by the computer. The program is executed from a binary format, which was generated from the original program source code.

JavaScript is a lightweight interpreted programming language. The web browser receives the JavaScript code in its original text form and runs the script from that. From a technical standpoint, most modern JavaScript interpreters actually use a technique called **just-in-time compiling** to improve performance; the JavaScript source code gets compiled into a faster, binary format while the script is being used, so that it can be run as quickly as possible. However, JavaScript is still considered an interpreted language, since the compilation is handled at run time, rather than ahead of time.

Server-side versus client-side code

You might also hear the terms **server-side** and **client-side** code, especially in the context of web development. Client-side code is code that is run on the user's computer — when a web page is viewed, the page's client-side code is downloaded, then run and displayed by the browser. In this module we are explicitly talking about **client-side JavaScript**.

Server-side code on the other hand is run on the server, then its results are downloaded and displayed in the browser. Examples of popular server-side web languages include PHP, Python, Ruby, ASP.NET and... JavaScript! JavaScript can also be used as a server-side language, for example in the popular Node.js environment

Dynamic versus static code

The word **dynamic** is used to describe both client-side JavaScript, and server-side languages — it refers to the ability to update the display of a web page/app to show different things in different circumstances, generating new content as required. Server-side code dynamically generates new content on the server, e.g. pulling data from a database, whereas client-side JavaScript dynamically generates new content inside the browser on the client, e.g. creating a new HTML table, filling it with data requested from the server, then displaying the table in a web page shown to the user.

A web page with no dynamically updating content is referred to as **static** — it just shows the same content all the time.

## How do you add JavaScript to your page?

JavaScript is applied to your HTML page in a similar manner to CSS. Whereas CSS uses [<link>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link) elements to apply external stylesheets and [<style>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/style) elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML — the [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script)

Internal Javascript – when we add js code just before the </head> it is called internal javascript

<script>

// JavaScript goes here

</script>

External Javascript –

<script src="script.js" defer></script>

Inline JavaScript handlers

Note that sometimes you'll come across bits of actual JavaScript code living inside HTML. It might look something like this:

function createParagraph() {

let para = document.createElement('p');

para.textContent = 'You clicked the button!';

document.body.appendChild(para);

}

<button onclick="createParagraph()">Click me!</button>

Here [<button>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button) element includes an inline onclick handler to make the function run when the button is pressed.

Script loading strategies

There are a number of issues involved with getting scripts to load at the right time. Nothing is as simple as it seems! A common problem is that all the HTML on a page is loaded in the order in which it appears. If you are using JavaScript to manipulate elements on the page (or more accurately, the Document Object Model), your code won't work if the JavaScript is loaded and parsed before the HTML you are trying to do something to.

In the internal example, you can see this structure around the code:

document.addEventListener("DOMContentLoaded", function() {

...

});

This is an event listener, which listens for the browser's "DOMContentLoaded" event, which signifies that the HTML body is completely loaded and parsed. The JavaScript inside this block will not run until after that event is fired,

In the external example, we use a more modern JavaScript feature to solve the problem, the **defer** attribute, which tells the browser to continue downloading the HTML content once the <script> tag element has been reached.

<script src="main.js" defer></script>

In this case both the script and the HTML will load simultaneously and the code will work.

An old-fashioned solution to this problem used to be to put your script element right at the bottom of the body (e.g. just before the </body> tag), so that it would load after all the HTML has been parsed. The problem with this solution is that loading/parsing of the script is completely blocked until the HTML DOM has been loaded. On larger sites with lots of JavaScript, this can cause a major performance issue, slowing down your site.

async and defer

There are actually two modern features we can use to bypass the problem of the blocking script — async and defer (which we saw above). Let's look at the difference between these two.

Scripts loaded using the async attribute (see below) will download the script without blocking rendering the page and will execute it as soon as the script finishes downloading. You get no guarantee that scripts will run in any specific order, only that they will not stop the rest of the page from displaying. It is best to use async when the scripts in the page run independently from each other and depend on no other script on the page.

For example, if you have the following script elements:

<script async src="js/vendor/jquery.js"></script>

<script async src="js/script2.js"></script>

<script async src="js/script3.js"></script>

All the scripts with the defer attribute will load in the order they appear on the page. So in the second example, we can be sure that jquery.js will load before script2.js and script3.js and that script2.js will load before script3.js. They won't run until the page content has all loaded, which is useful if your scripts depend on the DOM being in place (e.g. they modify one of more elements on the page).

To summarize:

* async and defer both instruct the browser to download the script(s) in a separate thread, while the rest of the page (the DOM, etc.) is downloading, so the page loading is not blocked by the scripts.
* If your scripts should be run immediately and they don't have any dependencies, then use async.
* If your scripts need to wait for parsing and depend on other scripts and/or the DOM being in place, load them using defer and put their corresponding <script> elements in the order you want the browser to execute them.

## Comments

A single line comment is written after a double forward slash (//), e.g.

// I am a comment

A multi-line comment is written between the strings /\* and \*/, e.g.

/\*

I am also

a comment

\*/

## What is a variable?

A variable is a container for a value, like a number we might use in a sum, or a string that we might use as part of a sentence. But one special thing about variables is that their contained values can change. Let's look at a simple example:

<button>Press me</button>

const button = document.querySelector('button');

button.onclick = function() {

let name = prompt('What is your name?');

alert('Hello ' + name + ', nice to see you!');

}

Another special thing about variables is that they can contain just about anything — not just strings and numbers. Variables can also contain complex data and even entire functions to do amazing things.

## Declaring a variable

To use a variable, you've first got to create it — more accurately, we call this declaring the variable. To do this, we type the keyword var or let followed by the name you want to call your variable:

let myName;

let myAge;

Here we're creating two variables called myName and myAge. Try typing these lines into your web browser's console

You can test whether these values now exist in the execution environment by typing just the variable's name

They currently have no value; they are empty containers. When you enter the variable names, you should get a value of undefined returned. If they don't exist, you'll get an error message

## Initializing a variable

Once you've declared a variable, you can initialize it with a value.

myName = 'Chris';

myAge = 37;

You can declare and initialize a variable at the same time, like this:

let myDog = 'Rover';

## The difference between var and let

For a start, if you write a multiline JavaScript program that declares and initializes a variable, you can actually declare a variable with var after you initialize it and it will still work.

myName = 'Chris';

function logName() {

console.log(myName);

}

logName();

var myName;

This works because of **hoisting**

Hoisting no longer works with let. If we changed var to let in the above example, it would fail with an error. This is a good thing — declaring a variable after you initialize it results in confusing, harder to understand code.

Secondly, when you use var, you can declare the same variable as many times as you like, but with let you can't. The following would work:

var myName = 'Chris';

var myName = 'Bob';

But the following would throw an error on the second line:

let myName = 'Chris';

let myName = 'Bob';

You'd have to do this instead:

let myName = 'Chris';

myName = 'Bob';

Again, this is a sensible language decision. There is no reason to redeclare variables — it just makes things more confusing.

## Updating a variable

Once a variable has been initialized with a value, you can change (or update) that value by simply giving it a different value

An aside on variable naming rules

You can call a variable pretty much anything you like, but there are limitations. Generally, you should stick to just using Latin characters (0-9, a-z, A-Z) and the underscore character.

* You shouldn't use other characters because they may cause errors or be hard to understand for an international audience.
* Don't use underscores at the start of variable names — this is used in certain JavaScript constructs to mean specific things, so may get confusing.
* Don't use numbers at the start of variables. This isn't allowed and causes an error.
* A safe convention to stick to is so-called ["lower camel case"](https://en.wikipedia.org/wiki/CamelCase#Variations_and_synonyms), where you stick together multiple words, using lower case for the whole first word and then capitalize subsequent words. We've been using this for our variable names in the article so far.
* Make variable names intuitive, so they describe the data they contain. Don't just use single letters/numbers, or big long phrases.
* Variables are case sensitive — so myage is a different variable from myAge.
* One last point: you also need to avoid using JavaScript reserved words as your variable names — by this, we mean the words that make up the actual syntax of JavaScript! So, you can't use words like var, function, let, and for as variable names. Browsers recognize them as different code items, and so you'll get errors.

Good name examples:

age

myAge

init

initialColor

finalOutputValue

audio1

audio2

Bad name examples:

1

a

\_12

myage

MYAGE

var

Document

skjfndskjfnbdskjfb

thisisareallylongstupidvariablenameman

## Variable types

There are a few different types of data we can store in variables. In this section we'll describe these in brief

You don't need to declare variable types in JavaScript, unlike some other programming languages

Numbers

You can store numbers in variables, either whole numbers like 30 (also called integers) or decimal numbers like 2.456

let myAge = 17;

Strings

Strings are pieces of text. When you give a variable a string value, you need to wrap it in single or double quote marks;

let dolphinGoodbye = 'So long and thanks for all the fish';

Booleans

Booleans are true/false values — they can have two values, true or false

let iAmAlive = true;

other uses

let test = 6 < 3;

Arrays

An array is a single object that contains multiple values enclosed in square brackets and separated by commas.

let myNameArray = ['Chris', 'Bob', 'Jim'];

let myNumberArray = [10, 15, 40];

Once these arrays are defined, you can access each value by their location within the array

myNameArray[0]; // should return 'Chris'

myNumberArray[2]; // should return 40

Objects

In programming, an object is a structure of code that models a real-life object. You can have a simple object that represents a box and contains information about its width, length, and height, or you could have an object that represents a person, and contains data about their name, height, weight, what language they speak

let dog = { name : 'Spot', breed : 'Dalmatian' };

To retrieve the information stored in the object, you can use the following syntax:

dog.name

Undefined

let var test;

typeof test; // undefined

typeof instance === "undefined"

Null

Let var test = null;

Typeof test; // Object

## Dynamic typing

JavaScript is a "dynamically typed language", which means that, unlike some other languages, you don't need to specify what data type a variable will contain (numbers, strings, arrays, etc).

## Constants in JavaScript

Many programming languages have the concept of a *constant* — a value that once declared can't be changed

const daysInWeek = 7;

const hoursInDay = 24;

## Basic maths in JS

Types of numbers

* **Integers** are whole numbers, e.g. 10, 400, or -5.
* **Floating point numbers** (floats) have decimal points and decimal places, for example 12.5, and 56.7786543.
* **Doubles** are a specific type of floating point number that have greater precision than standard floating point numbers (meaning that they are accurate to a greater number of decimal places).

We even have different types of number systems! Decimal is base 10 (meaning it uses 0–9 in each column), but we also have things like:

* **Binary** — The lowest level language of computers; 0s and 1s.
* **Octal** — Base 8, uses 0–7 in each column.
* **Hexadecimal** — Base 16, uses 0–9 and then a–f in each column. You may have encountered these numbers before when setting [colors in CSS](https://developer.mozilla.org/en-US/Learn/CSS/Introduction_to_CSS/Values_and_units#Hexadecimal_values).

Useful Number methods

The [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) object, an instance of which represents all standard numbers you'll use in your JavaScript, has a number of useful methods available on it for you to manipulate numbers

For example, to round your number to a fixed number of decimal places, use the [toFixed()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) method.

let lotsOfDecimal = 1.766584958675746364;

lotsOfDecimal;

let twoDecimalPlaces = lotsOfDecimal.toFixed(2);

twoDecimalPlaces;

Converting to number data types

Sometimes you might end up with a number that is stored as a string type, which makes it difficult to perform calculations with it. This most commonly happens when data is entered into a [form](https://developer.mozilla.org/en-US/docs/Learn/Forms) input, and the [input type is text](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/text). There is a way to solve this problem — passing the string value into the [Number()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/Number) constructor to return a number version of the same value.

let myNumber = '74';

myNumber + 3;

You end up with the result 743, not 77, because myNumber is actually defined as a string. We can test it as below.

typeof myNumber;

To fix the calculation, you can do this:

Number(myNumber) + 3;

All Maths rule works as expected

Operator precedence

Let's look at the last example from above, assuming that num2 holds the value 50 and num1 holds the value 10

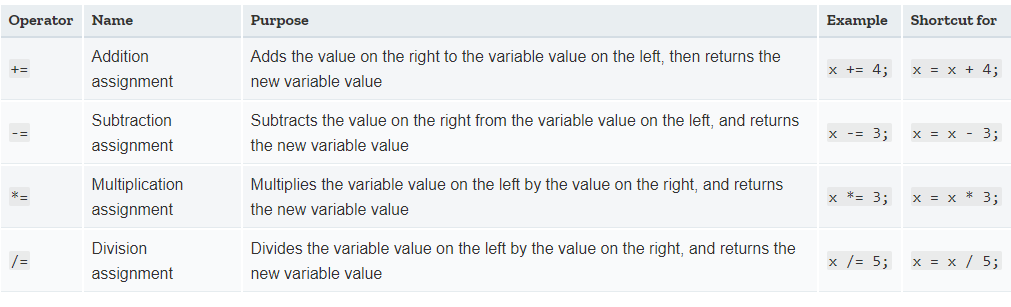
num2 + num1 / 8 + 2;

As a human being, you may read this as *"50 plus 10 equals 60"*, then *"8 plus 2 equals 10"*, and finally *"60 divided by 10 equals 6"*.

But the browser does *"10 divided by 8 equals 1.25"*, then *"50 plus 1.25 plus 2 equals 53.25"*.

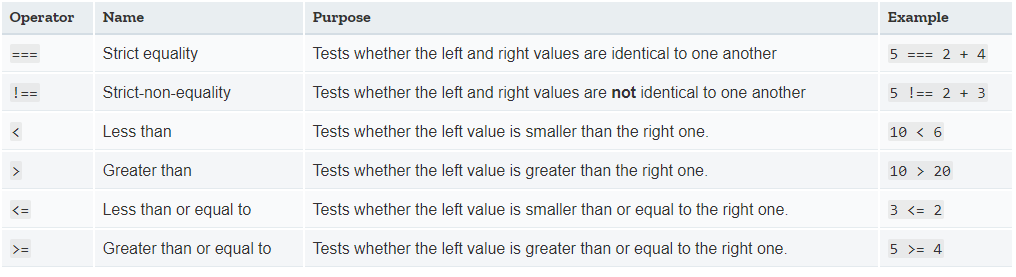
Increment and decrement operators

Sometimes you'll want to repeatedly add or subtract one to or from a numeric variable value. This can be conveniently done using the increment (++) and decrement(--) operators.



Comparison operators

Sometimes we will want to run true/false tests, then act accordingly depending on the result of that test — to do this we use **comparison operators**.



## Strings in Javascript

Creating a string

To start with, enter the following lines:

let string = 'The revolution will not be televised.';

string;

Just like we did with numbers, we are declaring a variable, initializing it with a string value, and then returning the value. The only difference here is that when writing a string, you need to surround the value with quotes.

Escaping characters in a string

let bigmouth = 'I\'ve got no right to take my place...';

bigmouth;

Concatenating strings

let one = 'Hello, ';

let two = 'how are you?';

let joined = one + two;

We can also use a mix of variables and actual strings. Try this:

let response = one + 'I am fine — ' + two;

response;

Numbers vs. strings

So what happens when we try to add (or concatenate) a string and a number? Let's try it in our console:

'Front ' + 242;

You can even do this with two numbers — you can force a number to become a string by wrapping it in quote marks. Try the following (we are using the typeof operator to check whether the variable is a number or a string):

let myDate = '19' + '67';

typeof myDate;

If you have a numeric variable that you want to convert to a string but not change otherwise, or a string variable that you want to convert to a number but not change otherwise, you can use the following two constructs:

* The [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) object converts anything passed to it into a number, if it can. Try the following:

let myString = '123';

let myNum = Number(myString);

typeof myNum;

* Conversely, every number has a method called [toString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toString) that converts it to the equivalent string. Try this:

let myNum = 123;

let myString = myNum.toString();

typeof myString;

## Template literals

Another type of string syntax that you may come across is **template literals** (sometimes referred to as template strings). This is a newer syntax that provides more flexible, easier to read strings.

To turn a standard string literal into a template literal, you have to replace the quote marks (' ', or " ") with backtick characters (` `). So, taking a simple example:

let song = 'Fight the Youth';

Would be turned into a template literal like so:

song = `Fight the Youth`;

If we want to concatenate strings, or include expression results inside them, traditional strings can be fiddly to write:

let score = 9;

let highestScore = 10;

let output = 'I like the song "' + song + '". I gave it a score of ' + (score/highestScore \* 100) + '%.';

Template literals simplify this enormously:

output = `I like the song "${ song }". I gave it a score of ${ score/highestScore \* 100 }%.`;

There is no more need to open and close multiple string pieces — the whole lot can just be wrapped in a single pair of backticks. When you want to include a variable or expression inside the string, you include it inside a ${ } construct, which is called a *placeholder*.

You can include complex expressions inside template literals, for example:

let examScore = 45;

let examHighestScore = 70;

examReport = `You scored ${ examScore }/${ examHighestScore } (${ Math.round((examScore/examHighestScore\*100)) }%). ${ examScore >= 49 ? 'Well done, you passed!' : 'Bad luck, you didn\'t pass this time.' }`;

* The first two placeholders here are pretty simple, only including a simple value in the string.
* The third one calculates a percentage result and rounds it to the nearest integer.
* The fourth one includes uses a ternary operator to check whether the score is above a certain mark and print a pass or fail message depending on the result.

Another point to note is that if you want to split a traditional string over multiple lines, you need to include a newline character, \n:

output = 'I like the song "' + song + '".\nI gave it a score of ' + (score/highestScore \* 100) + '%.';

Template literals respect the line breaks in the source code, so newline characters are no longer needed. This would achieve the same result:

output = `I like the song "${ song }".

I gave it a score of ${ score/highestScore \* 100 }%.`;

## Strings as objects

Most things are objects in JavaScript. When you create a string, for example by using

let string = 'This is my string';

your variable becomes a string object instance, and as a result has a large number of properties and methods available to it.

Finding the length of a string

This is easy — you simply use the [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/length) property. Try entering the following lines:

let browserType = 'mozilla';

browserType.length;

Retrieving a specific string character

On a related note, you can return any character inside a string by using **square bracket notation** —

browserType[0];

browserType[browserType.length-1];

Finding a substring inside a string and extracting it

browserType.indexOf('zilla');

When you know where a substring starts inside a string, and you know at which character you want it to end, [slice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/slice) can be used to extract it

browserType.slice(0,3); // moz

if you know that you want to extract all of the remaining characters in a string after a certain character, you don't have to include the second parameter! Instead, you only need to include the character position from where you want to extract the remaining characters in a string. Try the following:

browserType.slice(2)

Changing case

The string methods [toLowerCase()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase) and [toUpperCase()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) take a string and convert all the characters to lower- or uppercase, respectively.

let radData = 'My NaMe Is MuD';

radData.toLowerCase();

radData.toUpperCase();

Updating parts of a string

You can replace one substring inside a string with another substring using the [replace()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace) method.

browserType.replace('moz','van');

## ARRAYS

Arrays are generally described as "list-like objects"; they are basically single objects that contain multiple values stored in a list. Array objects can be stored in variables and dealt with in much the same way as any other type of value, the difference being that we can access each value inside the list individually, and do super useful and efficient things with the list, like loop through it and do the same thing to every value.

Creating arrays

Arrays consist of square brackets and elements that are separated by commas.

let shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];

shopping;

Accessing and modifying array items

You can then access individual items in the array using bracket notation, in the same way that you accessed the letters in a string.

shopping[0];

You can also modify an item in an array by simply giving a single array item a new value. Try this:

shopping[0] = 'tahini';

shopping;

Note that an array inside an array is called a multidimensional array. You can access an item inside an array that is itself inside another array by chaining two sets of square brackets together.

let random = [[1,2,3],[3,4,5]];

random[2][2];

Finding the length of an array

shopping.length;

This has other uses, but it is most commonly used to tell a loop to keep going until it has looped through all the items in an array.

let sequence = [1, 1, 2, 3, 5, 8, 13];

for (let i = 0; i < sequence.length; i++) {

console.log(sequence[i]);

}

Converting between strings and arrays

Often you'll be presented with some raw data contained in a big long string, and you might want to separate the useful items out into a more useful form and then do things to them, like display them in a data table. To do this, we can use the [split()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split) method.

let myData = 'Manchester,London,Liverpool,Birmingham,Leeds,Carlisle';

let myArray = myData.split(',');

myArray;

You can also go the opposite way using the [join()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join) method. Try the following

let myNewString = myArray.join(',');

myNewString;

Another way of converting an array to a string is to use the [toString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/toString) method. toString() is arguably simpler than join() as it doesn't take a parameter, but more limiting. With join() you can specify different separators, whereas toString() always uses a comma.

let dogNames = ['Rocket','Flash','Bella','Slugger'];

dogNames.toString(); // Rocket,Flash,Bella,Slugger

Adding and removing array items

let myArray = ['Manchester', 'London', 'Liverpool', 'Birmingham', 'Leeds', 'Carlisle'];

Let's use push() first — note that you need to include one or more items that you want to add to the end of your array.

myArray.push('Cardiff');

myArray;

myArray.push('Bradford', 'Brighton');

myArray;

Removing the last item from the array is as simple as running pop() on it. Try this:

myArray.pop();

let removedItem = myArray.pop();

myArray;

removedItem;

[unshift()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift) and [shift()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift) work in exactly the same way as push() and pop(), respectively, except that they work on the beginning of the array, not the end.

First unshift() — try the following commands:

myArray.unshift('Edinburgh');

myArray

Now shift(); try these!

let removedItem = myArray.shift();

myArray;

removedItem;

## Making Decisions in your code - conditions..!

In any programming language, the code needs to make decisions and carry out actions accordingly depending on different inputs

Basic if ... else syntax

Basic if...else syntax looks like the following in [pseudocode](https://developer.mozilla.org/en-US/docs/Glossary/pseudocode):

if (condition) {

code to run if condition is true

} else {

run some other code instead

}

A note on comparison operators

Comparison operators are used to test the conditions inside our conditional statements. We first looked at comparison operators back in our [Basic math in JavaScript — numbers and operators](https://developer.mozilla.org/en-US/Learn/JavaScript/First_steps/Math#Comparison_operators) article. Our choices are:

* === and !== — test if one value is identical to, or not identical to, another.
* < and > — test if one value is less than or greater than another.
* <= and >= — test if one value is less than or equal to, or greater than or equal to, another.

We wanted to make a special mention of testing boolean (true/false) values, and a common pattern you'll come across again and again. Any value that is not false, undefined, null, 0, NaN, or an empty string ('') actually returns true when tested as a conditional statement, therefore you can simply use a variable name on its own to test whether it is true, or even that it exists (that is, it is not undefined.) So for example:

let cheese = 'Cheddar';

if (cheese) {

console.log('Yay! Cheese available for making cheese on toast.');

} else {

console.log('No cheese on toast for you today.');

}

let shoppingDone = false;

if (shoppingDone) { // don't need to explicitly specify '=== true'

let childsAllowance = 10;

} else {

let childsAllowance = 5;

}

Nesting if ... else

It is perfectly OK to put one if...else statement inside another one — to nest them. For example, we could create weather forecast application to show a further set of choices depending on what the temperature is

if (choice === 'sunny') {

if (temperature < 86) {

para.textContent = 'It is ' + temperature + ' degrees outside — nice and sunny. Let\'s go out to the beach, or the park, and get an ice cream.';

} else if (temperature >= 86) {

para.textContent = 'It is ' + temperature + ' degrees outside — REALLY HOT! If you want to go outside, make sure to put some sunscreen on.';

}

}

Logical operators: AND, OR and NOT

* && — AND; allows you to chain together two or more expressions so that all of them have to individually evaluate to true for the whole expression to return true.
* || — OR; allows you to chain together two or more expressions so that one or more of them have to individually evaluate to true for the whole expression to return true.

if (choice === 'sunny' && temperature < 86) {

para.textContent = 'It is ' + temperature + ' degrees outside — nice and sunny. Let\'s go out to the beach, or the park, and get an ice cream.';

} else if (choice === 'sunny' && temperature >= 86) {

para.textContent = 'It is ' + temperature + ' degrees outside — REALLY HOT! If you want to go outside, make sure to put some sunscreen on.';

}

switch statements

if...else statements do the job of enabling conditional code well, but they are not without their downsides. They are mainly good for cases where you've got a couple of choices, and each one requires a reasonable amount of code to be run, and/or the conditions are complex (for example, multiple logical operators). For cases where you just want to set a variable to a certain choice of value or print out a particular statement depending on a condition, the syntax can be a bit cumbersome, especially if you've got a large number of choices.

switch (expression) {

case choice1:

run this code

break;

case choice2:

run this code instead

break;

// include as many cases as you like

default:

actually, just run this code

}

Ternary operator

The [ternary or conditional operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) is a small bit of syntax that tests a condition and returns one value/expression if it is true, and another if it is false

( condition ) ? run this code : run this code instead

let greeting = ( isBirthday ) ? 'Happy birthday Mrs. Smith — we hope you have a great day!' : 'Good morning Mrs. Smith.';

## loopin Code

Programming languages are very useful for rapidly completing repetitive tasks, from multiple basic calculations to just about any other situation where you've got a lot of similar items of work to complete

for (initializer; condition; final-expression) {

// code to run

}

Break; and continue;

While and Do while

initializer

while (condition) {

// code to run

final-expression

}

initializer

do {

// code to run

final-expression

} while (condition)

## Functions- Reusable block of code

Another essential concept in coding is **functions**, which allow you to store a piece of code that does a single task inside a defined block, and then call that code whenever you need it using a single short command

Built in Browser functions

We've made use of functions built in to the browser a lot in this course. Every time we manipulated a text string, for example:

let myText = 'I am a string';

let newString = myText.replace('string', 'sausage');

console.log(newString);

// the replace() string function takes a source string,

// and a target string and replaces the source string,

// with the target string, and returns the newly formed string

Or every time we manipulated an array:

let myArray = ['I', 'love', 'chocolate', 'frogs'];

let madeAString = myArray.join(' ');

console.log(madeAString);

// the join() function takes an array, joins

// all the array items together into a single

// string, and returns this new string

Custom Function

function myFunction() {

alert('hello');

}

myFunction();

// calls the function once

Anonymous Functions

You may see functions defined and invoked in slightly different ways. So far we have just created a function like so:

function myFunction() {

alert('hello');

}

But you can also create a function that doesn't have a name:

function() {

alert('hello');

}

This is called an **anonymous function** — it has no name! It also won't do anything on its own. You generally use an anonymous function along with an event handler, for example the following would run the code inside the function whenever the associated button is clicked:

const myButton = document.querySelector('button');

myButton.onclick = function() {

alert('hello');

}

You can also assign an anonymous function to be the value of a variable, for example:

let myGreeting = function() {

alert('hello');

}

## Introduction to events

Events are actions or occurrences that happen in the system you are programming, which the system tells you about so you can respond to them in some way if desired.

For example, if the user clicks a button on a webpage, you might want to respond to that action by displaying an information box.

There are a lot of different types of events that can occur, for example:

* The user clicking the mouse over a certain element or hovering the cursor over a certain element.
* The user pressing a key on the keyboard.
* The user resizing or closing the browser window.
* A web page finishing loading.
* A form being submitted.
* A video being played, or paused, or finishing play.
* An error occurring.

Each available event has an **event handler**, which is a block of code (usually a JavaScript function that you as a programmer create) that will be run when the event fires. When such a block of code is defined to be run in response to an event firing, we say we are **registering an event handler**. Note that event handlers are sometimes called **event listeners**.

Web events are not part of the core JavaScript language — they are defined as part of the APIs built into the browser.

A simple example

 In the following example, we have a single [<button>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button), which when pressed, makes the background change to a random color:

<button>Change color</button>

The JavaScript looks like so:

const btn = document.querySelector('button');

function random(number) {

return Math.floor(Math.random() \* (number+1));

}

btn.onclick = function() {

const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';

document.body.style.backgroundColor = rndCol;

}

Now let us try changing btn.onclick to the following different values in turn, and observing the results in the example:

* [btn.onfocus](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onfocus) and [btn.onblur](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onblur) — The color changes when the button is focused and unfocused; try pressing tab to focus on the button and press tab again to focus away from the button. These are often used to display information about how to fill in form fields when they are focused, or display an error message if a form field has just been filled in with an incorrect value.
* [btn.ondblclick](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/ondblclick) — The color changes only when the button is double-clicked.
* [window.onkeypress](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onkeypress), [window.onkeydown](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onkeydown), [window.onkeyup](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onkeyup) — The color changes when a key is pressed on the keyboard. The keypress event refers to a general press (button down and then up), while keydown and keyup refer to just the key down and key up parts of the keystroke, respectively. Note that it doesn't work if you try to register this event handler on the button itself — we've had to register it on the [window](https://developer.mozilla.org/en-US/docs/Web/API/Window) object, which represents the entire browser window.
* [btn.onmouseover](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onmouseover) and [btn.onmouseout](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onmouseout) — The color changes when the mouse pointer is moved so it begins hovering over the button, or when pointer stops hovering over the button and moves off of it, respectively.

Inline event handlers

<button onclick="bgChange()">Press me</button>

function bgChange() {

const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';

document.body.style.backgroundColor = rndCol;

}

you could also insert JavaScript directly inside the attribute, for example:

<button onclick="alert('Hello, this is my old-fashioned event handler!');">Press me</button>

For a start, it is not a good idea to mix up your HTML and your JavaScript, as it becomes hard to parse — keeping your JavaScript all in one place is better; if it is in a separate file you can apply it to multiple HTML documents.

const buttons = document.querySelectorAll('button');

for (let i = 0; i < buttons.length; i++) {

buttons[i].onclick = bgChange;

}

addEventListener() and removeEventListener()

const btn = document.querySelector('button');

function bgChange() {

const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';

document.body.style.backgroundColor = rndCol;

}

btn.addEventListener('click', bgChange);

Inside the addEventListener() function, we specify two parameters — the name of the event we want to register this handler for, and the code that comprises the handler function we want to run in response to it. Note that it is perfectly appropriate to put all the code inside the addEventListener() function, in an anonymous function, like this:

btn.addEventListener('click', function() {

var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';

document.body.style.backgroundColor = rndCol;

});

removeEventListener :

btn.removeEventListener('click', bgChange);

What mechanism should I use?

The other two are relatively interchangeable, at least for simple uses:

* Event handler properties have less power and options, but better cross-browser compatibility (being supported as far back as Internet Explorer 8). You should probably start with these as you are learning.
* DOM Level 2 Events (addEventListener(), etc.) are more powerful, but can also become more complex and are less well supported (supported as far back as Internet Explorer 9). You should also experiment with these, and aim to use them where possible.

The main advantages of the third mechanism are that you can remove event handler code if needed, using removeEventListener(), and you can add multiple listeners of the same type to elements if required. For example, you can call addEventListener('click', function() { ... }) on an element multiple times, with different functions specified in the second argument. This is impossible with event handler properties because any subsequent attempts to set a property will overwrite earlier ones, e.g.:

element.onclick = function1;

element.onclick = function2;

## Other event concepts

In this section, we briefly cover some advanced concepts that are relevant to events

Event objects

Sometimes inside an event handler function, you might see a parameter specified with a name such as event, evt, or simply e. This is called the **event object**, and it is automatically passed to event handlers to provide extra features and information. For example, let's write our random color example again slightly:

function bgChange(e) {

const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';

e.target.style.backgroundColor = rndCol;

console.log(e);

}

btn.addEventListener('click', bgChange);

e.target is incredibly useful when you want to set the same event handler on multiple elements and do something to all of them when an event occurs on them. You might, for example, have a set of 16 tiles that disappear when they are clicked on. It is useful to always be able to just set the thing to disappear as e.target,

const divs = document.querySelectorAll('div');

for (let i = 0; i < divs.length; i++) {

divs[i].onclick = function(e) {

e.target.style.backgroundColor = bgChange();

}

}

Preventing default behavior

Sometimes, you'll come across a situation where you want to prevent an event from doing what it does by default. The most common example is that of a web form, for example, a custom registration form. When you fill in the details and press the submit button, the natural behavior is for the data to be submitted to a specified page on the server for processing, and the browser to be redirected to a "success message" page of some kind

First, a simple HTML form that requires you to enter your first and last name:

<form>

<div>

<label for="fname">First name: </label>

<input id="fname" type="text">

</div>

<div>

<label for="lname">Last name: </label>

<input id="lname" type="text">

</div>

<div>

<input id="submit" type="submit">

</div>

</form>

<p></p>

Now some JavaScript — here we implement a very simple check inside an [onsubmit](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onsubmit) event handler (the submit event is fired on a form when it is submitted) that tests whether the text fields are empty. If they are, we call the [preventDefault()](https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault) function on the event object — which stops the form submission — and then display an error message in the paragraph below our form to tell the user what's wrong:

const form = document.querySelector('form');

const fname = document.getElementById('fname');

const lname = document.getElementById('lname');

const para = document.querySelector('p');

form.onsubmit = function(e) {

if (fname.value === '' || lname.value === '') {

e.preventDefault();

para.textContent = 'You need to fill in both names!';

}

}

Event bubbling and capture

Event bubbling and capture are two mechanisms that describe what happens when two handlers of the same event type are activated on one element

Bubbling and capturing explained

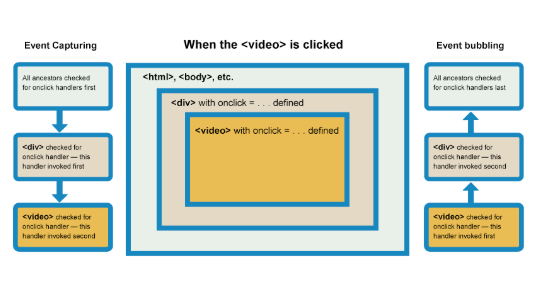
When an event is fired on an element that has parent elements (in this case, the [<video>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video) has the [<div>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div) as a parent), modern browsers run two different phases — the **capturing** phase and the **bubbling** phase.

In the **capturing** phase:

* The browser checks to see if the element's outer-most ancestor ([<html>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/html)) has an onclick event handler registered on it for the capturing phase, and runs it if so.
* Then it moves on to the next element inside <html> and does the same thing, then the next one, and so on until it reaches the element that was actually clicked on.

In the **bubbling** phase, the exact opposite occurs:

* The browser checks to see if the element that was actually clicked on has an onclick event handler registered on it for the bubbling phase, and runs it if so.
* Then it moves on to the next immediate ancestor element and does the same thing, then the next one, and so on until it reaches the <html> element.



n modern browsers, by default, all event handlers are registered for the bubbling phase. So in our current example, when you click the video, the click event bubbles from the <video> element outwards to the <html> element.

Fixing the problem with stopPropagation()

This is annoying behavior, but there is a way to fix it! The standard [Event](https://developer.mozilla.org/en-US/docs/Web/API/Event) object has a function available on it called [stopPropagation()](https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation) which, when invoked on a handler's event object, makes it so that first handler is run but the event doesn't bubble any further up the chain, so no more handlers will be run.