JavaScript First Steps

In our first JavaScript module, we first answer some fundamental questions such as "what is JavaScript?", "what does it look like?", and "what can it do?", before moving on to taking you through your first practical experience of writing JavaScript. After that, we discuss some key building blocks in detail, such as variables, strings, numbers and arrays.

Prerequisites

Before starting this module, you don't need any previous JavaScript knowledge, but you should have some familiarity with HTML and CSS. You are advised to work through the following modules before starting on JavaScript:

- Getting started with the Web (which includes a really basic JavaScript introduction).
- Introduction to HTML.
- Introduction to CSS.

> **Note**: If you are working on a computer/tablet/other device where you don't have the ability to create your own files, you could try out (most of) the code examples in an online coding program such as JSBin or Glitch.

Guides

What is JavaScript?

Welcome to the MDN beginner's JavaScript course! In this first article we will look at JavaScript from a high level, answering questions such as "what is it?", and "what is it doing?", and making sure you are comfortable with JavaScript's purpose.

A first splash into JavaScript

Now you've learned something about the theory of JavaScript, and what you can do with it, we are going to give you a crash course in the basic features of JavaScript via a completely practical tutorial. Here you'll build up a simple "Guess the number" game, step by step.

What went wrong? Troubleshooting JavaScript

When you built up the "Guess the number" game in the previous article, you may have found that it didn't work. Never fear — this article aims to save you from tearing your hair out over such problems by providing you with some simple tips on how to find and fix errors in JavaScript programs.

Storing the information you need — Variables

After reading the last couple of articles you should now know what JavaScript is, what it can do for you, how you use it alongside other web technologies, and what its main features look like from a high level. In this article we will get down to the real basics, looking at how to work with the most basic building blocks of JavaScript — Variables.

Basic math in JavaScript — numbers and operators

At this point in the course we discuss maths in JavaScript — how we can combine operators and other features to successfully manipulate numbers to do our bidding.

Handling text — strings in JavaScript

Next we'll turn our attention to strings — this is what pieces of text are called in programming. In this article we'll look at all the common things that you really ought to know about strings when learning JavaScript, such as creating strings, escaping quotes in string, and joining them together.

Useful string methods

Now we've looked at the very basics of strings, let's move up a gear and start thinking about what useful operations we can do on strings with built-in methods, such as finding the length of a text string, joining and splitting strings, substituting one character in a string for another, and more.

Arrays

In the final article of this module, we'll look at arrays — a neat way of storing a list of data items under a single variable name. Here we look at why this is useful, then explore how to create an array, retrieve, add, and remove items stored in an array, and more besides.

Assessments

The following assessment will test your understanding of the JavaScript basics covered in the guides above.

Silly story generator

In this assessment you'll be tasked with taking some of the knowledge you've picked up in this module's articles and applying it to creating a fun app that generates random silly stories. Have fun!

See also

Learn JavaScript

An excellent resource for aspiring web developers — Learn JavaScript in an interactive environment, with short lessons and interactive tests, guided by automated assessment. The first 40 lessons are free, and the complete course is available for a small one-time payment.

Last modified: Mar 30, 2020, by MDN contributors

Related Topics

Complete beginners start here!

- Getting started with the Web

HTML — Structuring the Web

What is JavaScript?

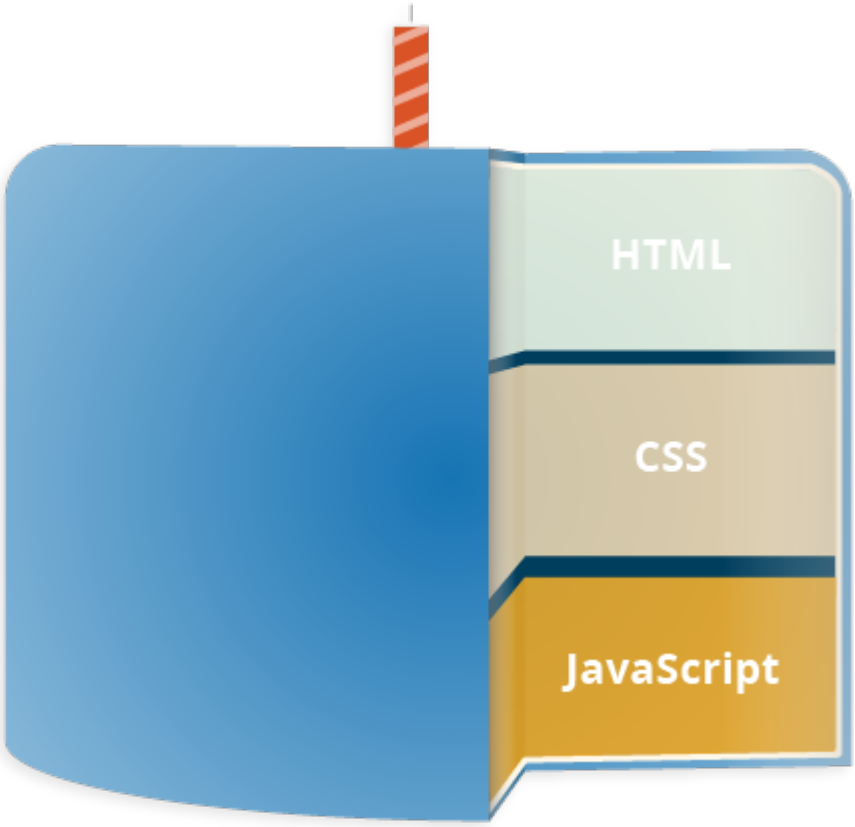
Overview: First steps Next

Welcome to the MDN beginner's JavaScript course! In this article we will look at JavaScript from a high level, answering questions such as "What is it?" and "What can you do with it?", and making sure you are comfortable with JavaScript's purpose.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS.
Objective:	To gain familiarity with what JavaScript is, what it can do, and how it fits into a web site.

A high-level definition

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. — you can bet that JavaScript is probably involved. It is the third layer of the layer cake of standard web technologies, two of which (HTML and CSS) we have covered in much more detail in other parts of the Learning Area.



- HTML is the markup language that we use to structure and give meaning to our web content, for example defining paragraphs, headings, and data tables, or embedding images and videos in the page.
- CSS is a language of style rules that we use to apply styling to our HTML content, for example setting background colors and fonts, and laying out our content in multiple columns.
- JavaScript is a scripting language that enables you to create dynamically updating content, control multimedia, animate images, and pretty much everything else. (Okay, not everything, but it is amazing what you can achieve with a few lines of JavaScript code.)

The three layers build on top of one another nicely. Let's take a simple text label as an example. We can mark it up using HTML to give it structure and purpose:

```
1 <p>Player 1: Chris</p>
```

Player 1: Chris

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

```css
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:

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

para.addEventListener('click', updateName);

function updateName() {
  let name = prompt('Enter a new name');
  para.textContent = 'Player 1: ' + name;
}
```

Try clicking on this last version of the text label to see what happens (note also that you can find this demo on GitHub — see the source code, or run it live)!

JavaScript can do a lot more than that — let's explore what in more detail.

So what can it really do?

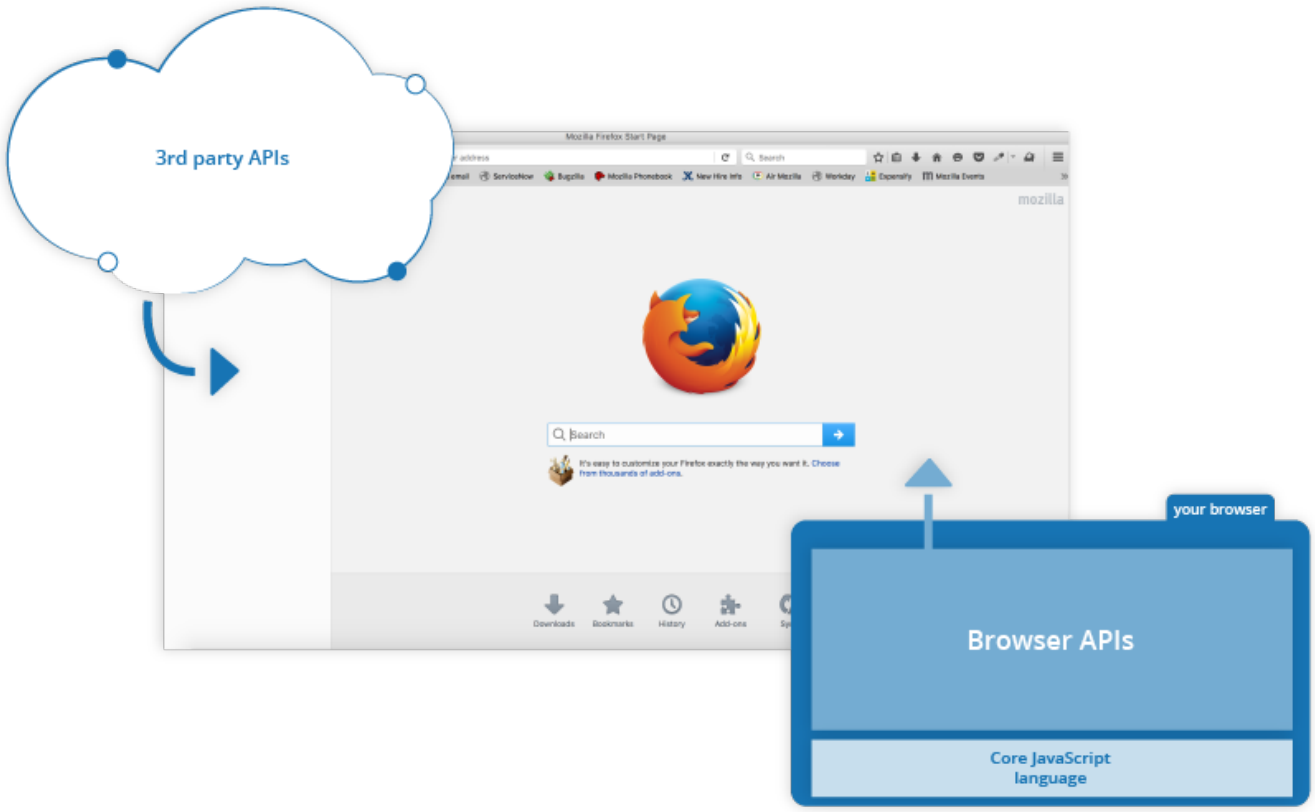
The core client-side JavaScript language consists of some common programming features that allow you to do things like:

- 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: Chris".
- Running code in response to certain events occurring on a web page. We used a `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 do the same thing for programming that ready-made furniture kits do for home building — it is much easier to take ready-cut panels and screw them together to make a bookshelf than it is to work out the design yourself, go and find the correct wood, cut all the panels to the right size and shape, find the correct-sized screws, and *then* put them together to make a bookshelf.

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` 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` retrieves geographical information. This is how Google Maps is able to find your location and plot it on a map.
- The `Canvas` and `WebGL` APIs allow you to create animated 2D and 3D graphics. People are doing some amazing things using these web technologies —see Chrome Experiments and webglsamples.
- Audio and Video APIs like `HTMLMediaElement` and `WebRTC` 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 (try our simple Snapshot demo to get the idea).

> **Note**: Many of the above demos won't work in an older browser — when experimenting, it's a good idea to use a modern browser like Firefox, Chrome, Edge or Opera to run your code

> in. You will need to consider cross browser testing in more detail when you get closer to delivering production code (i.e. real code that real customers will use).

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 allows you to do things like displaying your latest tweets on your website.
- The Google Maps API and OpenStreetMap API allows you to embed custom maps into your website, and other such functionality.

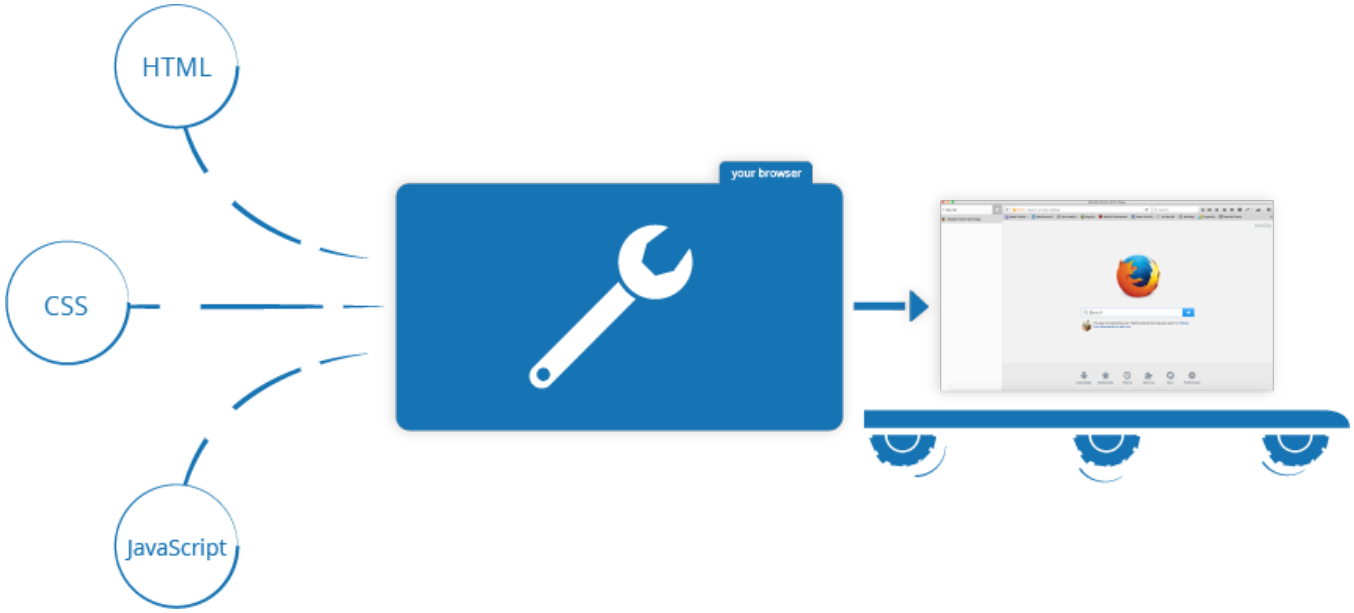
> **Note**: These APIs are advanced, and we'll not be covering any of these in this module. You can find out much more about these in our Client-side web APIs module.

There's a lot more available, too! However, don't get over excited just yet. You won't be able to build the next Facebook, Google Maps, or Instagram after studying JavaScript for 24 hours — there are a lot of basics to cover first. And that's why you're here — let's move on!

What is JavaScript doing on your page?

Here we'll actually start looking at some code, and while doing so, explore what actually happens when you run some JavaScript in your page.

Let's briefly recap the story of what happens when you load a web page in a browser (first talked about in our How CSS works article). 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. You will learn ways around this later in the article, in the Script loading strategies section.

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 pirates could start writing code to steal information from other websites, and other such bad things.

> **Note**: There are ways to send code and data between different websites/tabs in a safe manner, but these are advanced techniques that we won't cover in this course.

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:

```
1  const para = document.querySelector('p');
2
3  para.addEventListener('click', updateName);
4
5  function updateName() {
6    let name = prompt('Enter a new name');
7    para.textContent = 'Player 1: ' + name;
8  }
```

Here we are selecting a text paragraph (line 1), then attaching an event listener to it (line 3) so that when the paragraph is clicked, the `updateName()` code block (lines 5–8) is run. The `updateName()` code block (these types of reusable code blocks are called "functions") asks the user for a new name, and then inserts that name into the paragraph to update the display.

If you swapped the order of the first two lines of code, it would no longer work — instead, you'd get an error returned in the browser developer console — `TypeError: para is undefined`. This means that the `para` object does not exist yet, so we can't add an event listener to it.

> **Note**: This is a very common error — you need to be careful that the objects referenced in your code exist before you try to do stuff to them.

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 assembly language 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.

There are advantages to both types of language, but we won't discuss them right now.

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 — you can find out more about server-side JavaScript in our Dynamic Websites – Server-side programming topic.

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. The meaning is slightly different in the two contexts, but related, and both approaches (server-side and client-side) usually work together.

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>` elements to apply external stylesheets and `<style>` elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML — the `<script>` element. Let's learn how this works.

Internal JavaScript

1. First of all, make a local copy of our example file apply-javascript.html. Save it in a directory somewhere sensible.
2. Open the file in your web browser and in your text editor. You'll see that the HTML creates a simple web page containing a clickable button.
3. Next, go to your text editor and add the following in your head — just before your closing `</head>` tag:

```
<script>

  // JavaScript goes here

</script>
```

4. Now we'll add some JavaScript inside our `<script>` element to make the page do something more interesting — add the following code just below the "// JavaScript goes here" line:

```
document.addEventListener("DOMContentLoaded", function() {
  function createParagraph() {
    let para = document.createElement('p');
    para.textContent = 'You clicked the button!';
    document.body.appendChild(para);
  }

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

  for(let i = 0; i < buttons.length ; i++) {
    buttons[i].addEventListener('click', createParagraph);
```

```
  }
});
```

5. Save your file and refresh the browser — now you should see that when you click the button, a new paragraph is generated and placed below.

> **Note**: If your example doesn't seem to work, go through the steps again and check that you did everything right. Did you save your local copy of the starting code as a `.html` file? Did you add your `<script>` element just before the `</head>` tag? Did you enter the JavaScript exactly as shown? **JavaScript is case sensitive, and very fussy, so you need to enter the syntax exactly as shown, otherwise it may not work.**

> **Note**: You can see this version on GitHub as apply-javascript-internal.html (see it live too).

External JavaScript

This works great, but what if we wanted to put our JavaScript in an external file? Let's explore this now.

1. First, create a new file in the same directory as your sample HTML file. Call it `script.js` — make sure it has that .js filename extension, as that's how it is recognized as JavaScript.
2. Replace your current `<script>` element with the following:

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

3. Inside `script.js`, add the following script:

```
function createParagraph() {
  let para = document.createElement('p');
  para.textContent = 'You clicked the button!';
  document.body.appendChild(para);
}

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

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

```

```
    buttons[i].addEventListener('click', createParagraph);
}
```

4. Save and refresh your browser, and you should see the same thing! It works just the same, but now we've got our JavaScript in an external file. This is generally a good thing in terms of organizing your code and making it reusable across multiple HTML files. Plus, the HTML is easier to read without huge chunks of script dumped in it.

> **Note**: You can see this version on GitHub as apply-javascript-external.html and script.js (see it live too).

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>
```

You can try this version of our demo below.

Click me!

This demo has exactly the same functionality as in the previous two sections, except that the `<button>` element includes an inline `onclick` handler to make the function run when the button is pressed.

Please don't do this, however. It is bad practice to pollute your HTML with JavaScript, and it is inefficient — you'd have to include the `onclick="createParagraph()"` attribute on every button you want the JavaScript to apply to.

Using a pure JavaScript construct allows you to select all the buttons using one instruction. The code we used above to serve this purpose looks like this:

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

for(let i = 0; i < buttons.length ; i++) {
  buttons[i].addEventListener('click', createParagraph);
}
```

This might be a bit longer than the `onclick` attribute, but it will work for all buttons — no matter how many are on the page, nor how many are added or removed. The JavaScript does not need to be changed.

> **Note**: Try editing your version of `apply-javascript.html` and add a few more buttons into the file. When you reload, you should find that all of the buttons when clicked will create a paragraph. Neat, huh?

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 above code examples, in the internal and external examples the JavaScript is loaded and run in the head of the document, before the HTML body is parsed. This could cause an error, so we've used some constructs to get around it.

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, therefore the error is avoided (you'll learn about events later in the course).

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

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

> **Note**: In the external case, we did not need to use the `DOMContentLoaded` event because the `defer` attribute solved the problem for us. We didn't use the `defer` solution for the internal JavaScript example because `defer` only works for external scripts.

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>
```

You can't rely on the order the scripts will load in. `jquery.js` may load before or after `script2.js` and `script3.js` and if this is the case, any functions in those scripts depending on `jquery` will produce an error because `jquery` will not be defined at the time the script runs.

`async` should be used when you have a bunch of background scripts to load in, and you just want to get them in place asap. For example, maybe you have some game data files to load, which will be needed when the game actually begins, but for now you just want to get on with showing the game intro, titles, and lobby, without them being blocked by script loading.

Scripts loaded using the `defer` attribute (see below) will run in the order they appear in the page and execute them as soon as the script and content are downloaded:

```
<script defer src="js/vendor/jquery.js"></script>

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

<script defer 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

As with HTML and CSS, it is possible to write comments into your JavaScript code that will be ignored by the browser, and exist simply to provide instructions to your fellow developers on how the code works (and you, if you come back to your code after six months and can't remember what you did). Comments are very useful, and you should use them often, particularly for larger applications. There are two types:

- 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
*/
```

So for example, we could annotate our last demo's JavaScript with comments like so:

```
// Function: creates a new paragraph and appends it to the bottom of the

function createParagraph() {
  let para = document.createElement('p');
  para.textContent = 'You clicked the button!';
```

```
  document.body.appendChild(para);
}

/*
  1. Get references to all the buttons on the page in an array format.
  2. Loop through all the buttons and add a click event listener to each

  When any button is pressed, the createParagraph() function will be run.
*/

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

for (let i = 0; i < buttons.length ; i++) {
  buttons[i].addEventListener('click', createParagraph);
}
```

Note: In general more comments are usually better than less, but you should be careful if you find yourself adding lots of comments to explain what variables are (your variable names perhaps should be more intuitive), or to explain very simple operations (maybe your code is overcomplicated).

Summary

So there you go, your first step into the world of JavaScript. We've begun with just theory, to start getting you used to why you'd use JavaScript and what kind of things you can do with it. Along the way, you saw a few code examples and learned how JavaScript fits in with the rest of the code on your website, amongst other things.

JavaScript may seem a bit daunting right now, but don't worry — in this course, we will take you through it in simple steps that will make sense going forward. In the next article, we will plunge straight into the practical, getting you to jump straight in and build your own JavaScript examples.

Overview: First steps Next

A first splash into JavaScript

Previous Overview: First steps Next

Now you've learned something about the theory of JavaScript, and what you can do with it, we are going to give you a crash course in the basic features of JavaScript via a completely practical tutorial. Here you'll build up a simple "Guess the number" game, step by step.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To have a first bit of experience at writing some JavaScript, and gain at least a basic understanding of what writing a JavaScript program involves.

You won't be expected to understand all of the code in detail immediately — we just want to introduce you to the high-level concepts for now, and give you an idea of how JavaScript (and other programming languages) work. In subsequent articles, you'll revisit all these features in a lot more detail!

> Note: Many of the code features you'll see in JavaScript are the same as in other programming languages — functions, loops, etc. The code syntax looks different, but the concepts are still largely the same.

Thinking like a programmer

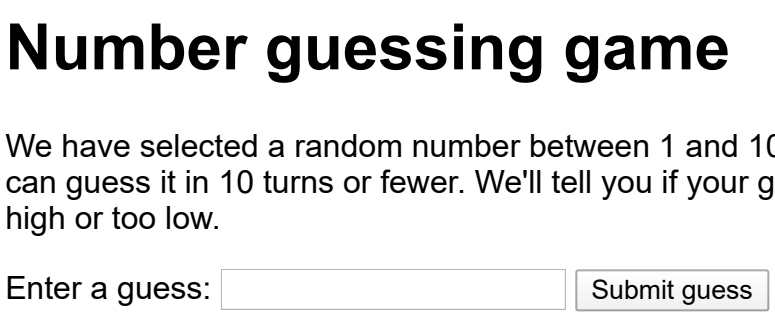
One of the hardest things to learn in programming is not the syntax you need to learn, but how to apply it to solve real world problems. You need to start thinking like a programmer — this generally involves looking at descriptions of what your program needs to do, working out what code features are needed to achieve those things, and how to make them work together.

This requires a mixture of hard work, experience with the programming syntax, and practice — plus a bit of creativity. The more you code, the better you'll get at it. We can't promise that you'll develop "programmer brain" in five minutes, but we will give you plenty of opportunity to practice thinking like a programmer throughout the course.

With that in mind, let's look at the example we'll be building up in this article, and review the general process of dissecting it into tangible tasks.

Example — Guess the number game

In this article we'll show you how to build up the simple game you can see below:



Have a go at playing it — familiarize yourself with the game before you move on.

Let's imagine your boss has given you the following brief for creating this game:

I want you to create a simple guess the number type game. It should choose a random number between 1 and 100, then challenge the player to guess the number in 10 turns. After each turn the player should be told if they are right or wrong, and if they are wrong, whether the guess was too low or too high. It should also tell the player what numbers they previously guessed. The game will end once the player guesses correctly, or once they run out of turns. When the game ends, the player should be given an option to start playing again.

Upon looking at this brief, the first thing we can do is to start breaking it down into simple actionable tasks, in as much of a programmer mindset as possible:

1. Generate a random number between 1 and 100.
2. Record the turn number the player is on. Start it on 1.
3. Provide the player with a way to guess what the number is.
4. Once a guess has been submitted first record it somewhere so the user can see their previous guesses.
5. Next, check whether it is the correct number.
6. If it is correct:
    1. Display congratulations message.
    2. Stop the player from being able to enter more guesses (this would mess the game up).
    3. Display control allowing the player to restart the game.
7. If it is wrong and the player has turns left:
    1. Tell the player they are wrong.
    2. Allow them to enter another guess.
    3. Increment the turn number by 1.
8. If it is wrong and the player has no turns left:
    1. Tell the player it is game over.
    2. Stop the player from being able to enter more guesses (this would mess the game up).

3. Display control allowing the player to restart the game.

9. Once the game restarts, make sure the game logic and UI are completely reset, then go back to step 1.

Let's now move forward, looking at how we can turn these steps into code, building up the example, and exploring JavaScript features as we go.

Initial setup

To begin this tutorial, we'd like you to make a local copy of the number-guessing-game-start.html file (see it live here). Open it in both your text editor and your web browser. At the moment you'll see a simple heading, paragraph of instructions and form for entering a guess, but the form won't currently do anything.

The place where we'll be adding all our code is inside the `<script>` element at the bottom of the HTML:

```
<script>

  // Your JavaScript goes here

</script>
```

Adding variables to store our data

Let's get started. First of all, add the following lines inside your `<script>` element:

```
let randomNumber = Math.floor(Math.random() * 100) + 1;

const guesses = document.querySelector('.guesses');
const lastResult = document.querySelector('.lastResult');
const lowOrHi = document.querySelector('.lowOrHi');

const guessSubmit = document.querySelector('.guessSubmit');
const guessField = document.querySelector('.guessField');

```

```
let guessCount = 1;
let resetButton;
```

This section of the code sets up the variables and constants we need to store the data our program will use. Variables are basically containers for values (such as numbers, or strings of text). You create a variable with the keyword `let` (or `var`) followed by a name for your variable (you'll read more about the difference between the keywords in a future article). Constants are used to store values that are immutable or can't be changed and are created with the keyword `const`. In this case, we are using constants to store references to parts of our user interface; the text inside some of them might change, but the HTML elements referenced stay the same.

You can assign a value to your variable or constant with an equals sign (`=`) followed by the value you want to give it.

In our example:

- The first variable — `randomNumber` — is assigned a random number between 1 and 100, calculated using a mathematical algorithm.
- The first three constants are each made to store a reference to the results paragraphs in our HTML, and are used to insert values into the paragraphs later on in the code (note how they are inside a `<div>` element, which is itself used to select all three later on for resetting, when we restart the game):

```
<div class="resultParas">
  <p class="guesses"></p>
  <p class="lastResult"></p>
  <p class="lowOrHi"></p>
</div>
```

- The next two constants store references to the form text input and submit button and are used to control submitting the guess later on.

```
<label for="guessField">Enter a guess: </label><input type="text"
<input type="submit" value="Submit guess" class="guessSubmit">
```

- Our final two variables store a guess count of 1 (used to keep track of how many guesses the player has had), and a reference to a reset button that doesn't exist yet (but will later).

> **Note**: You'll learn a lot more about variables/constants later on in the course, starting with the next article.

Functions

Next, add the following below your previous JavaScript:

```
function checkGuess() {
  alert('I am a placeholder');
}
```

Functions are reusable blocks of code that you can write once and run again and again, saving the need to keep repeating code all the time. This is really useful. There are a number of ways to define functions, but for now we'll concentrate on one simple type. Here we have defined a function by using the keyword `function`, followed by a name, with parentheses put after it. After that we put two curly braces (`{ }`). Inside the curly braces goes all the code that we want to run whenever we call the function.

When we want to run the code, we type the name of the function followed by the parentheses.

Let's try that now. Save your code and refresh the page in your browser. Then go into the developer tools JavaScript console, and enter the following line:

```
checkGuess();
```

After pressing `Return`/`Enter`, you should see an alert come up that says "`I am a placeholder`"; we have defined a function in our code that creates an alert whenever we call it.

> **Note**: You'll learn a lot more about functions later in the course.

Operators

JavaScript operators allow us to perform tests, do math, join strings together, and other such things.

If you haven't already done so, save your code, refresh the page in your browser, and open the developer tools JavaScript console. Then we can try typing in the examples shown below — type in each one from the "Example" columns exactly as shown, pressing `Return`/`Enter` after each one, and see what results they return.

First let's look at arithmetic operators, for example:

Operator	Name	Example
+	Addition	`6 + 9`
-	Subtraction	`20 - 15`
*	Multiplication	`3 * 7`
/	Division	`10 / 5`

You can also use the + operator to join text strings together (in programming, this is called *concatenation*). Try entering the following lines, one at a time:

```
let name = 'Bingo';
name;
let hello = ' says hello!';
hello;
let greeting = name + hello;
greeting;
```

There are also some shortcut operators available, called augmented assignment operators. For example, if you want to simply add a new text string to an existing one and return the result, you could do this:

```
name += ' says hello!';
```

This is equivalent to

```
name = name + ' says hello!';
```

When we are running true/false tests (for example inside conditionals — see below) we use comparison operators. For example:

Operator	Name	Example

===	Strict equality (is it exactly the same?)	`5 === 2 + 4 // false`
		`'Chris' === 'Bob' // false`
		`5 === 2 + 3 // true`
		`2 === '2' // false; number versus string`

!==	Non-equality (is it not the same?)	`5 !== 2 + 4 // true`
		`'Chris' !== 'Bob' // true`
		`5 !== 2 + 3 // false`
		`2 !== '2' // true; number versus string`

<	Less than	`6 < 10 // true` `20 < 10 // false`
>	Greater than	`6 > 10 // false` `20 > 10  // true`

Conditionals

Returning to our `checkGuess()` function, I think it's safe to say that we don't want it to just spit out a placeholder message. We want it to check whether a player's guess is correct or not, and respond appropriately.

At this point, replace your current `checkGuess()` function with this version instead:

```
function checkGuess() {
  let userGuess = Number(guessField.value);
  if (guessCount === 1) {
```

```
    guesses.textContent = 'Previous guesses: ';
  }
  guesses.textContent += userGuess + ' ';

  if (userGuess === randomNumber) {
    lastResult.textContent = 'Congratulations! You got it right!';
    lastResult.style.backgroundColor = 'green';
    lowOrHi.textContent = '';
    setGameOver();
  } else if (guessCount === 10) {
    lastResult.textContent = '!!!GAME OVER!!!';
    setGameOver();
  } else {
    lastResult.textContent = 'Wrong!';
    lastResult.style.backgroundColor = 'red';
    if(userGuess < randomNumber) {
      lowOrHi.textContent = 'Last guess was too low!';
    } else if(userGuess > randomNumber) {
      lowOrHi.textContent = 'Last guess was too high!';
    }
  }

  guessCount++;
  guessField.value = '';
  guessField.focus();
}

```

This is a lot of code — phew! Let's go through each section and explain what it does.

- The first line (line 2 above) declares a variable called `userGuess` and sets its value to the current value entered inside the text field. We also run this value through the built-in `Number()` constructor, just to make sure the value is definitely a number.
- Next, we encounter our first conditional code block (lines 3–5 above). A conditional code block allows you to run code selectively, depending on whether a certain condition is true or not. It looks a bit like a function, but it isn't. The simplest form of conditional block starts with the keyword `if`, then some parentheses, then some curly braces. Inside the parentheses we include a test. If the test returns `true`, we run the code inside the curly braces. If not, we don't, and move on to the next bit of code. In this case the test is testing

whether the `guessCount` variable is equal to `1` (i.e. whether this is the player's first go or not):

```
1 | guessCount === 1
```

If it is, we make the guesses paragraph's text content equal to "`Previous guesses: `". If not, we don't.

- Line 6 appends the current `userGuess` value onto the end of the `guesses` paragraph, plus a blank space so there will be a space between each guess shown.
- The next block (lines 8–24 above) does a few checks:
  - The first `if(){ }` checks whether the user's guess is equal to the `randomNumber` set at the top of our JavaScript. If it is, the player has guessed correctly and the game is won, so we show the player a congratulations message with a nice green color, clear the contents of the Low/High guess information box, and run a function called `setGameOver()`, which we'll discuss later.
  - Now we've chained another test onto the end of the last one using an `else if(){ }` structure. This one checks whether this turn is the user's last turn. If it is, the program does the same thing as in the previous block, except with a game over message instead of a congratulations message.
  - The final block chained onto the end of this code (the `else { }`) contains code that is only run if neither of the other two tests returns true (i.e. the player didn't guess right, but they have more guesses left). In this case we tell them they are wrong, then we perform another conditional test to check whether the guess was higher or lower than the answer, displaying a further message as appropriate to tell them higher or lower.
- The last three lines in the function (lines 26–28 above) get us ready for the next guess to be submitted. We add 1 to the `guessCount` variable so the player uses up their turn (`++` is an incrementation operation — increment by 1), and empty the value out of the form text field and focus it again, ready for the next guess to be entered.

Events

At this point we have a nicely implemented `checkGuess()` function, but it won't do anything because we haven't called it yet. Ideally we want to call it when the "Submit guess" button is pressed, and to do this we need to use an **event**. Events are things that happen in the browser — a button being clicked, a page loading, a video playing, etc. — in response to which we can run blocks of code. The constructs that listen out for the event happening are called **event**

listeners, and the blocks of code that run in response to the event firing are called **event handlers**.

Add the following line below your `checkGuess()` function:

```
guessSubmit.addEventListener('click', checkGuess);
```

Here we are adding an event listener to the `guessSubmit` button. This is a method that takes two input values (called *arguments*) — the type of event we are listening out for (in this case `click`) as a string, and the code we want to run when the event occurs (in this case the `checkGuess()` function). Note that we don't need to specify the parentheses when writing it inside `addEventListener()`.

Try saving and refreshing your code now, and your example should work — to a point. The only problem now is that if you guess the correct answer or run out of guesses, the game will break because we've not yet defined the `setGameOver()` function that is supposed to be run once the game is over. Let's add our missing code now and complete the example functionality.

Finishing the game functionality

Let's add that `setGameOver()` function to the bottom of our code and then walk through it. Add this now, below the rest of your JavaScript:

```
function setGameOver() {
  guessField.disabled = true;
  guessSubmit.disabled = true;
  resetButton = document.createElement('button');
  resetButton.textContent = 'Start new game';
  document.body.append(resetButton);
  resetButton.addEventListener('click', resetGame);
}
```

- The first two lines disable the form text input and button by setting their disabled properties to `true`. This is necessary, because if we didn't, the user could submit more guesses after the game is over, which would mess things up.

- The next three lines generate a new `<button>` element, set its text label to "Start new game", and add it to the bottom of our existing HTML.
- The final line sets an event listener on our new button so that when it is clicked, a function called `resetGame()` is run.

Now we need to define this function too! Add the following code, again to the bottom of your JavaScript:

```
function resetGame() {
  guessCount = 1;

  const resetParas = document.querySelectorAll('.resultParas p');
  for (let i = 0 ; i < resetParas.length ; i++) {
    resetParas[i].textContent = '';
  }

  resetButton.parentNode.removeChild(resetButton);

  guessField.disabled = false;
  guessSubmit.disabled = false;
  guessField.value = '';
  guessField.focus();

  lastResult.style.backgroundColor = 'white';

  randomNumber = Math.floor(Math.random() * 100) + 1;
}
```

This rather long block of code completely resets everything to how it was at the start of the game, so the player can have another go. It:

- Puts the `guessCount` back down to 1.
- Empties all the text out of the information paragraphs. We select all paragraphs inside `<div class="resultParas"></div>`, then loop through each one, setting their `textContent` to `''` (an empty string).
- Removes the reset button from our code.
- Enables the form elements, and empties and focuses the text field, ready for a new guess to be entered.

- Removes the background color from the `lastResult` paragraph.
- Generates a new random number so that you are not just guessing the same number again!

At this point you should have a fully working (simple) game — congratulations!

All we have left to do now in this article is talk about a few other important code features that you've already seen, although you may have not realized it.

Loops

One part of the above code that we need to take a more detailed look at is the for loop. Loops are a very important concept in programming, which allow you to keep running a piece of code over and over again, until a certain condition is met.

To start with, go to your browser developer tools JavaScript console again, and enter the following:

```
for (let i = 1 ; i < 21 ; i++) { console.log(i) }
```

What happened? The numbers `1` to `20` were printed out in your console. This is because of the loop. A `for` loop takes three input values (arguments):

1. **A starting value**: In this case we are starting a count at 1, but this could be any number you like. You could replace the letter `i` with any name you like too, but `i` is used as a convention because it's short and easy to remember.
2. **An exit condition**: Here we have specified `i < 21` — the loop will keep going until `i` is no longer less than 21. When `i` reaches 21, the loop will no longer run.
3. **An incrementor**: We have specified `i++`, which means "add 1 to i". The loop will run once for every value of `i`, until `i` reaches a value of 21 (as discussed above). In this case, we are simply printing the value of `i` out to the console on every iteration using `console.log()`.

Now let's look at the loop in our number guessing game — the following can be found inside the `resetGame()` function:

```
const resetParas = document.querySelectorAll('.resultParas p');
for (let i = 0 ; i < resetParas.length ; i++) {
  resetParas[i].textContent = '';
}
```

This code creates a variable containing a list of all the paragraphs inside `<div class="resultParas">` using the `querySelectorAll()` method, then it loops through each one, removing the text content of each.

A small discussion on objects

Let's add one more final improvement before we get to this discussion. Add the following line just below the `let resetButton;` line near the top of your JavaScript, then save your file:

```
guessField.focus();
```

This line uses the `focus()` method to automatically put the text cursor into the `<input>` text field as soon as the page loads, meaning that the user can start typing their first guess right away, without having to click the form field first. It's only a small addition, but it improves usability — giving the user a good visual clue as to what they've got to do to play the game.

Let's analyze what's going on here in a bit more detail. In JavaScript, everything is an object. An object is a collection of related functionality stored in a single grouping. You can create your own objects, but that is quite advanced and we won't be covering it until much later in the course. For now, we'll just briefly discuss the built-in objects that your browser contains, which allow you to do lots of useful things.

In this particular case, we first created a `guessField` constant that stores a reference to the text input form field in our HTML — the following line can be found amongst our declarations near the top of the code:

```
const guessField = document.querySelector('.guessField');
```

To get this reference, we used the `querySelector()` method of the `document` object. `querySelector()` takes one piece of information — a CSS selector that selects the element

you want a reference to.

Because `guessField` now contains a reference to an `<input>` element, it now has access to a number of properties (basically variables stored inside objects, some of which can't have their values changed) and methods (basically functions stored inside objects). One method available to input elements is `focus()`, so we can now use this line to focus the text input:

```
guessField.focus();
```

Variables that don't contain references to form elements won't have `focus()` available to them. For example, the `guesses` constant contains a reference to a `<p>` element, and the `guessCount` variable contains a number.

Playing with browser objects

Let's play with some browser objects a bit.

1. First of all, open up your program in a browser.
2. Next, open your browser developer tools, and make sure the JavaScript console tab is open.
3. Type `guessField` into the console and the console shows you that the variable contains an `<input>` element. You'll also notice that the console autocompletes the names of objects that exist inside the execution environment, including your variables!
4. Now type in the following:

   ```
   guessField.value = 'Hello';
   ```

   The `value` property represents the current value entered into the text field. You'll see that by entering this command, we've changed the text in the text field!
5. Now try typing `guesses` into the console and pressing return. The console shows you that the variable contains a `<p>` element.
6. Now try entering the following line:

   ```
   guesses.value
   ```

   The browser returns `undefined`, because paragraphs don't have the `value` property.

7. To change the text inside a paragraph, you need the `textContent` property instead. Try this:

```
guesses.textContent = 'Where is my paragraph?';
```

8. Now for some fun stuff. Try entering the below lines, one by one:

```
guesses.style.backgroundColor = 'yellow';
guesses.style.fontSize = '200%';
guesses.style.padding = '10px';
guesses.style.boxShadow = '3px 3px 6px black';
```

   Every element on a page has a `style` property, which itself contains an object whose properties contain all the inline CSS styles applied to that element. This allows us to dynamically set new CSS styles on elements using JavaScript.

Finished for now...

So that's it for building the example. You got to the end — well done! Try your final code out, or play with our finished version here. If you can't get the example to work, check it against the source code.

Previous Overview: First steps Next

In this module

- What is JavaScript?
- A first splash into JavaScript
- What went wrong? Troubleshooting JavaScript
- Storing the information you need — Variables
- Basic math in JavaScript — numbers and operators

What went wrong? Troubleshooting JavaScript

Previous Overview: First steps Next

When you built up the "Guess the number" game in the previous article, you may have found that it didn't work. Never fear — this article aims to save you from tearing your hair out over such problems by providing you with some tips on how to find and fix errors in JavaScript programs.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To gain the ability and confidence to start fixing problems in your own code.

Types of error

Generally speaking, when you do something wrong in code, there are two main types of error that you'll come across:

- **Syntax errors**: These are spelling errors in your code that actually cause the program not to run at all, or stop working part way through — you will usually be provided with some error messages too. These are usually okay to fix, as long as you are familiar with the right tools and know what the error messages mean!

- **Logic errors**: These are errors where the syntax is actually correct but the code is not what you intended it to be, meaning that program runs successfully but gives incorrect results. These are often harder to fix than syntax errors, as there usually isn't an error message to direct you to the source of the error.

Okay, so it's not quite *that* simple — there are some other differentiators as you drill down deeper. But the above classifications will do at this early stage in your career. We'll look at both of these types going forward.

An erroneous example

To get started, let's return to our number guessing game — except this time we'll be exploring a version that has some deliberate errors introduced. Go to Github and make yourself a local copy of number-game-errors.html (see it running live here).

1. To get started, open the local copy inside your favorite text editor, and your browser.
2. Try playing the game — you'll notice that when you press the "Submit guess" button, it doesn't work!

> **Note**: You might well have your own version of the game example that doesn't work, which you might want to fix! We'd still like you to work through the article with our version, so that you can learn the techniques we are teaching here. Then you can go back and try to fix your example.

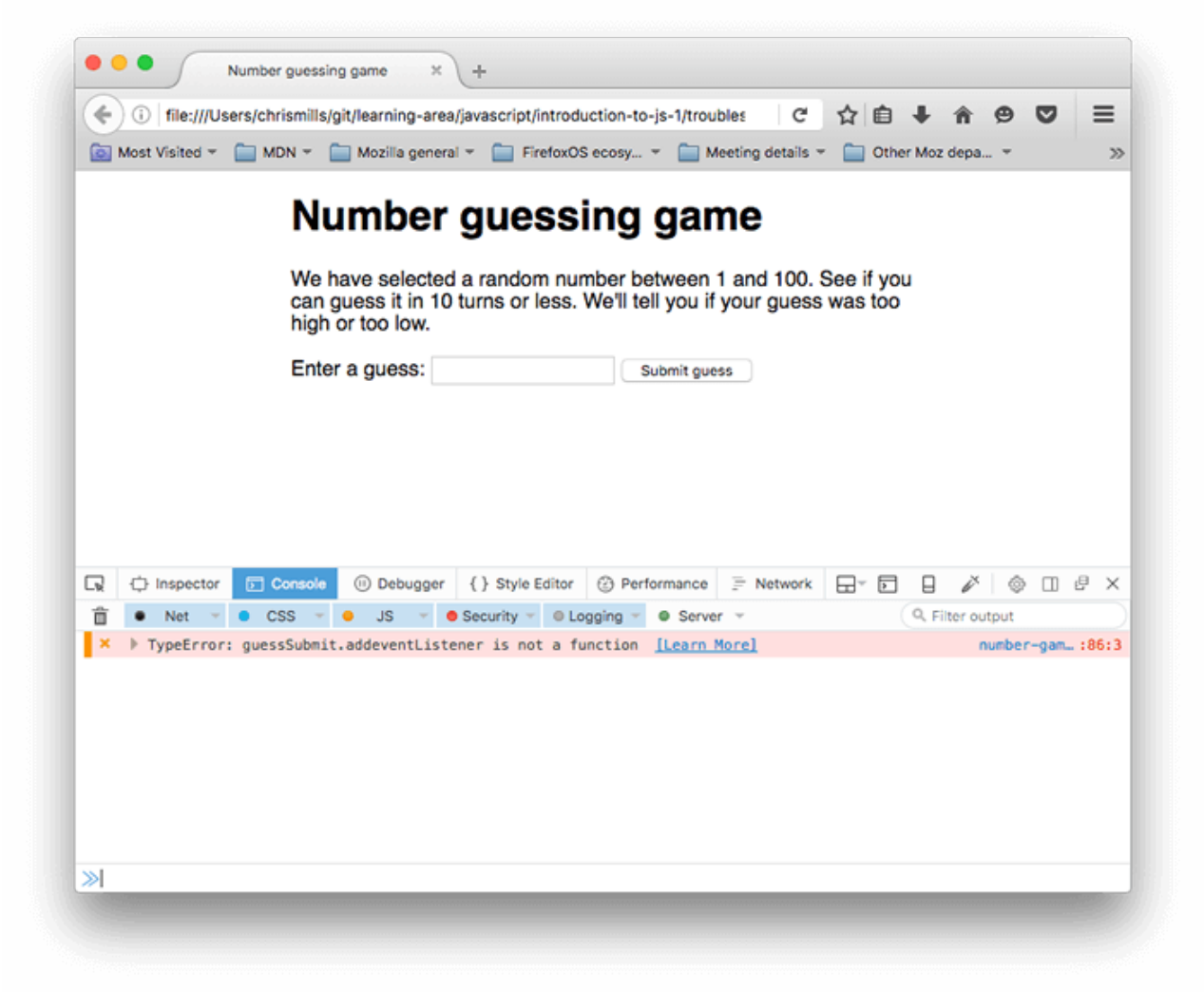
At this point, let's consult the developer console to see if it reports any syntax errors, then try to fix them. You'll learn how below.

Fixing syntax errors

Earlier on in the course we got you to type some simple JavaScript commands into the developer tools JavaScript console (if you can't remember how to open this in your browser, follow the previous link to find out how). What's even more useful is that the console gives you

error messages whenever a syntax error exists inside the JavaScript being fed into the browser's JavaScript engine. Now let's go hunting.

1. Go to the tab that you've got `number-game-errors.html` open in, and open your JavaScript console. You should see an error message along the following lines:



2. This is a pretty easy error to track down, and the browser gives you several useful bits of information to help you out (the screenshot above is from Firefox, but other browsers provide similar information). From left to right, we've got:
   - A red "x" to indicate that this is an error.
   - An error message to indicate what's gone wrong: "TypeError: guessSubmit.addeventListener is not a function"
   - A "Learn More" link that links through to an MDN page that explains what this error means in greater detail.
   - The name of the JavaScript file, which links through to the Debugger tab of the developer tools. If you follow this link, you'll see the exact line where the error is highlighted.

- The line number where the error is, and the character number in that line where the error is first seen. In this case, we've got line 86, character number 3.

3. If we look at line 86 in our code editor, we'll find this line:

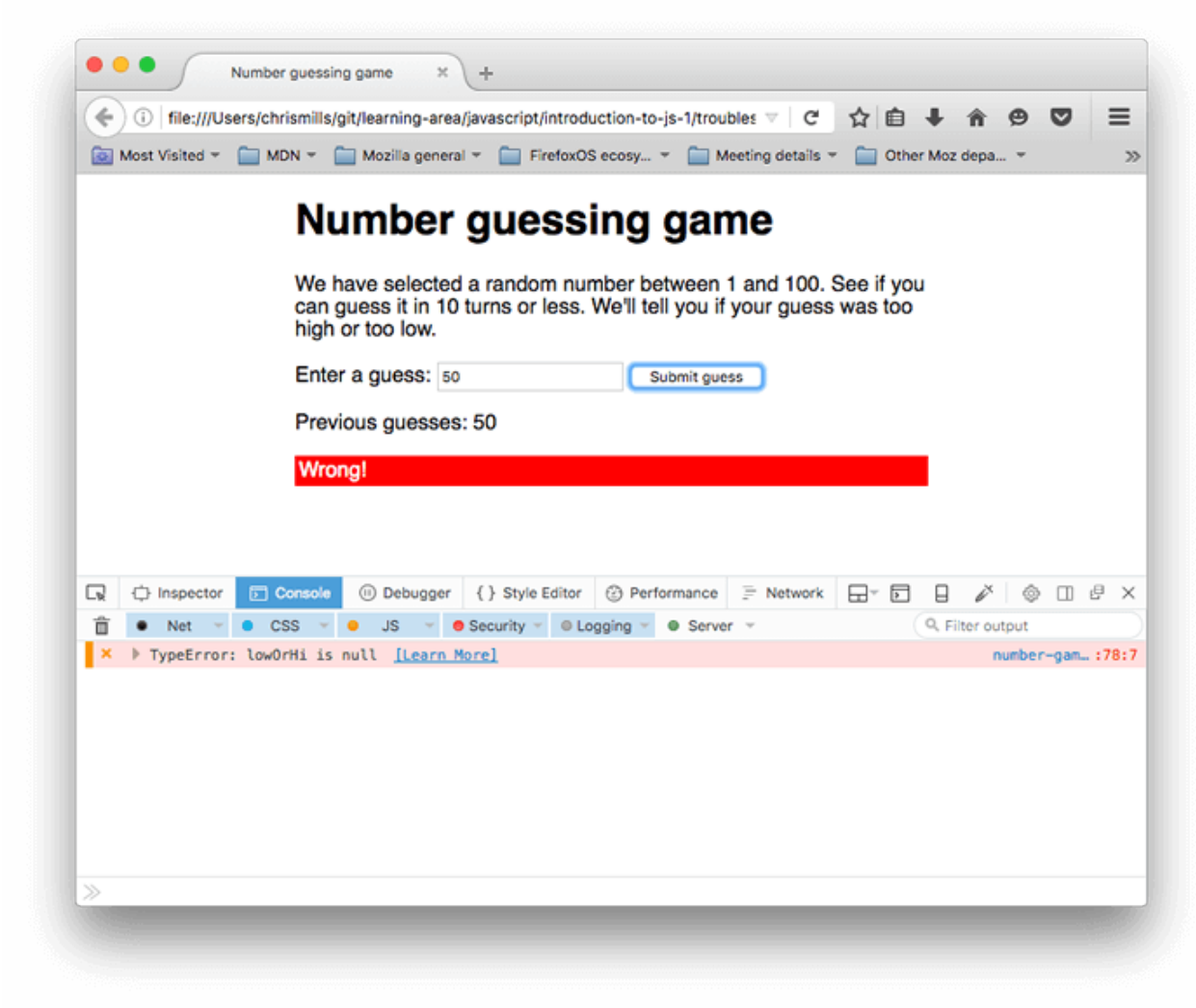
```
guessSubmit.addeventListener('click', checkGuess);
```

4. The error message says "guessSubmit.addeventListener is not a function", which means that the function we're calling is not recognized by the JavaScript interpreter. Often, this error message actually means that we've spelled something wrong. If you are not sure of the correct spelling of a piece of syntax, it is often good to look up the feature on MDN. The best way to do this currently is to search for "mdn *name-of-feature*" with your favorite search engine. Here's a shortcut to save you some time in this instance: `addEventListener()`.
5. So, looking at this page, the error appears to be that we've spelled the function name wrong! Remember that JavaScript is case sensitive, so any slight difference in spelling or casing will cause an error. Changing `addeventListener` to `addEventListener` should fix this. Do this now.

> **Note**: See our TypeError: "x" is not a function reference page for more details about this error.

Syntax errors round two

1. Save your page and refresh, and you should see the error has gone.
2. Now if you try to enter a guess and press the Submit guess button, you'll see ... another error!



3. This time the error being reported is "TypeError: lowOrHi is null", on line 78.

> **Note**: `Null` is a special value that means "nothing", or "no value". So `lowOrHi` has been declared and initialised, but not with any meaningful value — it has no type or value.

> **Note**: This error didn't come up as soon as the page was loaded because this error occurred inside a function (inside the `checkGuess() { ... }` block). As you'll learn in more detail in our later functions article, code inside functions runs in a separate scope than code outside functions. In this case, the code was not run and the error was not thrown until the `checkGuess()` function was run by line 86.

4. Have a look at line 78, and you'll see the following code:

```
1 lowOrHi.textContent = 'Last guess was too high!';
```

5. This line is trying to set the `textContent` property of the `lowOrHi` constant to a text string, but it's not working because `lowOrHi` does not contain what it's supposed to. Let's

see why this is — try searching for other instances of `lowOrHi` in the code. The earliest instance you'll find in the JavaScript is on line 48:

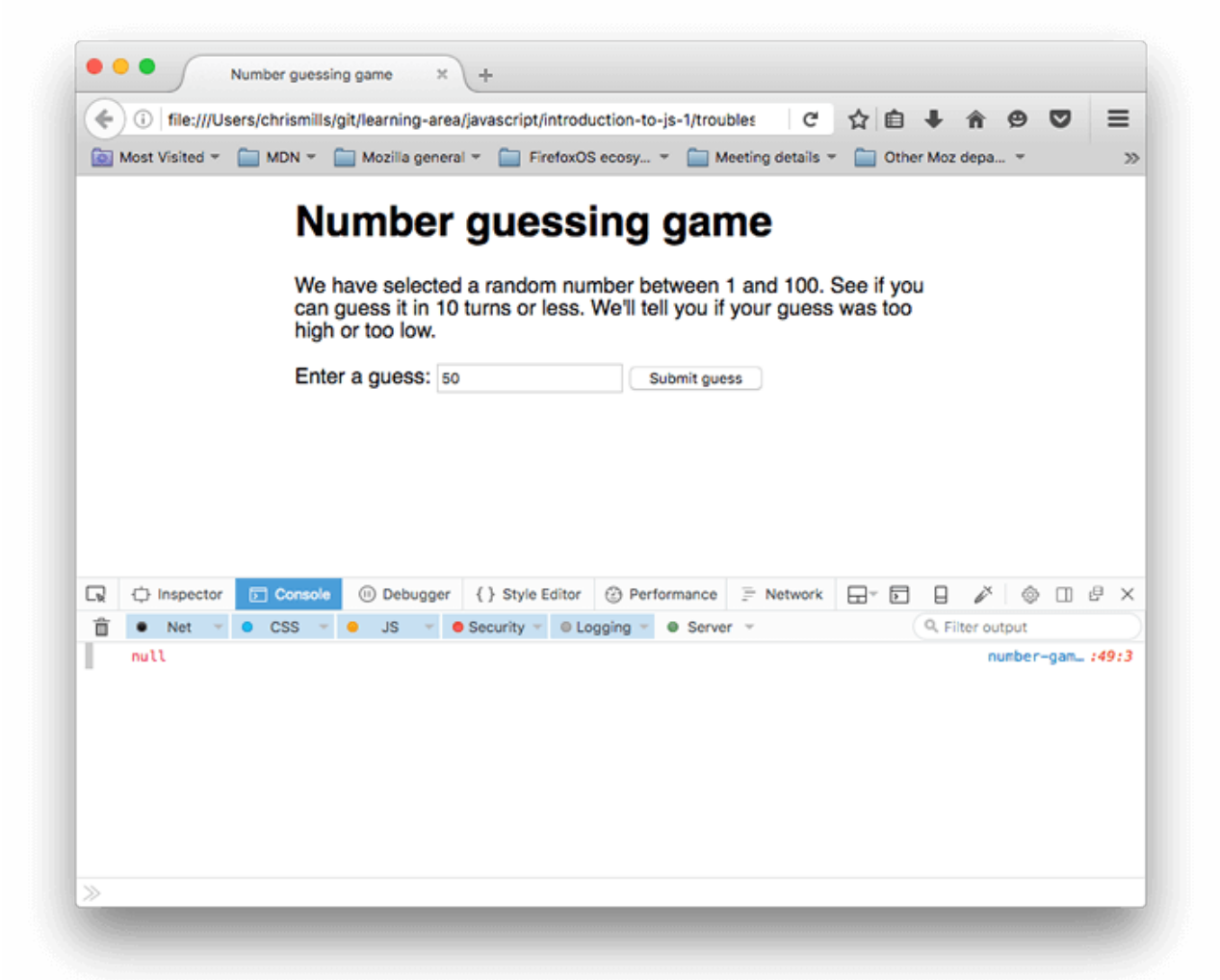
```
1 const lowOrHi = document.querySelector('lowOrHi');
```

6. At this point we are trying to make the variable contain a reference to an element in the document's HTML. Let's check whether the value is `null` after this line has been run. Add the following code on line 49:

```
1 console.log(lowOrHi);
```

> **Note**: `console.log()` is a really useful debugging function that prints a value to the console. So it will print the value of `lowOrHi` to the console as soon as we have tried to set it in line 48.

7. Save and refesh, and you should now see the `console.log()` result in your console.



Sure enough, `lowOrHi`'s value is `null` at this point, so there is definitely a problem with line 48.

8. Let's think about what the problem could be. Line 48 is using a `document.querySelector()` method to get a reference to an element by selecting it with a CSS selector. Looking further up our file, we can find the paragraph in question:

```
1 <p class="lowOrHi"></p>
```

9. So we need a class selector here, which begins with a dot (`.`), but the selector being passed into the `querySelector()` method in line 48 has no dot. This could be the problem! Try changing `lowOrHi` to `.lowOrHi` in line 48.
10. Try saving and refreshing again, and your `console.log()` statement should return the `<p>` element we want. Phew! Another error fixed! You can delete your `console.log()` line now, or keep it to reference later on — your choice.

> **Note**: See our TypeError: "x" is (not) "y" reference page for more details about this error.

Syntax errors round three

1. Now if you try playing the game through again, you should get more success — the game should play through absolutely fine, until you end the game, either by guessing the right number, or by running out of guesses.
2. At that point, the game fails again, and the same error is spat out that we got at the beginning — "TypeError: resetButton.addeventListener is not a function"! However, this time it's listed as coming from line 94.
3. Looking at line number 94, it is easy to see that we've made the same mistake here. We again just need to change `addeventListener` to `.addEventListener`. Do this now.

A logic error

At this point, the game should play through fine, however after playing through a few times you'll undoubtedly notice that the "random" number you've got to guess is always 1. Definitely not quite how we want the game to play out!

There's definitely a problem in the game logic somewhere — the game is not returning an error; it just isn't playing right.

1. Search for the `randomNumber` variable, and the lines where the random number is first set. The instance that stores the random number that we want to guess at the start of the game should be around line number 44:

   ```
   1 | let randomNumber = Math.floor(Math.random()) + 1;
   ```

   And the one that generates the random number before each subsequent game is around line 113:

2. ```
   1 | randomNumber = Math.floor(Math.random()) + 1;
   ```

3. To check whether these lines are indeed the problem, let's turn to our friend `console.log()` again — insert the following line directly below each of the above two lines:

   ```
   1 | console.log(randomNumber);
   ```

4. Save and refresh, then play a few games — you'll see that `randomNumber` is equal to 1 at each point where it is logged to the console.

Working through the logic

To fix this, let's consider how this line is working. First, we invoke `Math.random()`, which generates a random decimal number between 0 and 1, e.g. 0.5675493843.

```
1 | Math.random()
```

Next, we pass the result of invoking `Math.random()` through `Math.floor()`, which rounds the number passed to it down to the nearest whole number. We then add 1 to that result:

```
1 | Math.floor(Math.random()) + 1
```

Rounding a random decimal number between 0 and 1 down will always return 0, so adding 1 to it will always return 1. We need to multiply the random number by 100 before we round it down. The following would give us a random number between 0 and 99:
```

```
Math.floor(Math.random()*100);
```

Hence us wanting to add 1, to give us a random number between 1 and 100:

```
Math.floor(Math.random()*100) + 1;
```

Try updating both lines like this, then save and refresh — the game should now play like we are intending it to!

Other common errors

There are other common errors you'll come across in your code. This section highlights most of them.

SyntaxError: missing ; before statement

This error generally means that you have missed a semicolon at the end of one of your lines of code, but it can sometimes be more cryptic. For example, if we change this line inside the `checkGuess()` function:

```
var userGuess = Number(guessField.value);
```

to

```
var userGuess === Number(guessField.value);
```

It throws this error because it thinks you are trying to do something different. You should make sure that you don't mix up the assignment operator (`=`) — which sets a variable to be equal to a value — with the strict equality operator (`===`), which tests whether one value is equal to another, and returns a `true`/`false` result.

> **Note**: See our SyntaxError: missing ; before statement reference page for more details about this error.

The program always says you've won, regardless of the guess you enter

This could be another symptom of mixing up the assignment and strict equality operators. For example, if we were to change this line inside `checkGuess()`:

```
1 | if (userGuess === randomNumber) {
```

to

```
1 | if (userGuess = randomNumber) {
```

the test would always return `true`, causing the program to report that the game has been won. Be careful!

SyntaxError: missing) after argument list

This one is pretty simple — it generally means that you've missed the closing parenthesis at the end of a function/method call.

> **Note**: See our SyntaxError: missing) after argument list reference page for more details about this error.

SyntaxError: missing : after property id

This error usually relates to an incorrectly formed JavaScript object, but in this case we managed to get it by changing

```
1 | function checkGuess() {
```

to

```
1 function checkGuess( {
```

This has caused the browser to think that we are trying to pass the contents of the function into the function as an argument. Be careful with those parentheses!

SyntaxError: missing } after function body

This is easy — it generally means that you've missed one of your curly braces from a function or conditional structure. We got this error by deleting one of the closing curly braces near the bottom of the `checkGuess()` function.

SyntaxError: expected expression, got '*string*' or SyntaxError: unterminated string literal

These errors generally mean that you've left off a string value's opening or closing quote mark. In the first error above, *string* would be replaced with the unexpected character(s) that the browser found instead of a quote mark at the start of a string. The second error means that the string has not been ended with a quote mark.

For all of these errors, think about how we tackled the examples we looked at in the walkthrough. When an error arises, look at the line number you are given, go to that line and see if you can spot what's wrong. Bear in mind that the error is not necessarily going to be on that line, and also that the error might not be caused by the exact same problem we cited above!

> **Note**: See our SyntaxError: Unexpected token and SyntaxError: unterminated string literal reference pages for more details about these errors.

Summary

So there we have it, the basics of figuring out errors in simple JavaScript programs. It won't always be that simple to work out what's wrong in your code, but at least this will save you a few hours of sleep and allow you to progress a bit faster when things don't turn out right, especially in the earlier stages of your learning journey.

See also

- There are many other types of errors that aren't listed here; we are compiling a reference that explains what they mean in detail — see the JavaScript error reference.
- If you come across any errors in your code that you aren't sure how to fix after reading this article, you can get help! Ask for help on the MDN Discourse forum Learning category, or in the MDN Web Docs room on Matrix. Tell us what your error is, and we'll try to help you. A listing of your code would be useful as well.

Previous Overview: First steps Next

In this module

- What is JavaScript?
- A first splash into JavaScript
- What went wrong? Troubleshooting JavaScript
- Storing the information you need — Variables
- Basic math in JavaScript — numbers and operators
- Handling text — strings in JavaScript
- Useful string methods
- Arrays
- Assessment: Silly story generator

Storing the information you need — Variables

Previous Overview: First steps Next

After reading the last couple of articles you should now know what JavaScript is, what it can do for you, how you use it alongside other web technologies, and what its main features look like from a high level. In this article, we will get down to the real basics, looking at how to work with the most basic building blocks of JavaScript — Variables.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To gain familiarity with the basics of JavaScript variables.

Tools you need

Throughout this article, you'll be asked to type in lines of code to test your understanding of the content. If you are using a desktop browser, the best place to type your sample code is your browser's JavaScript console (see What are browser developer tools for more information on how to access this tool).

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!');
}
```

Press me

In this example pressing the button runs a couple of lines of code. The first line pops a box up on the screen that asks the reader to enter their name, and then stores the value in a variable. The second line displays a welcome message that includes their name, taken from the variable value.

To understand why this is so useful, let's think about how we'd write this example without using a variable. It would end up looking something like this:

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

if (name === 'Adam') {
  alert('Hello Adam, nice to see you!');
} else if (name === 'Alan') {
  alert('Hello Alan, nice to see you!');
} else if (name === 'Bella') {
  alert('Hello Bella, nice to see you!');
} else if (name === 'Bianca') {
  alert('Hello Bianca, nice to see you!');
} else if (name === 'Chris') {
  alert('Hello Chris, nice to see you!');
}
```

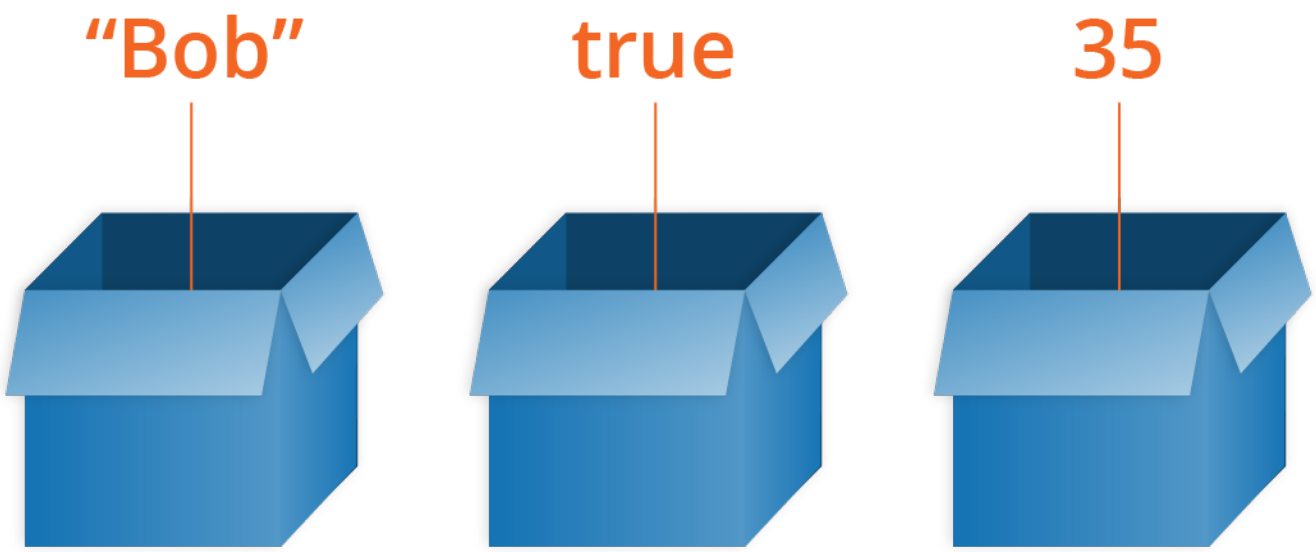
```
14
15 // ... and so on ...
```

You may not fully understand the syntax we are using (yet!), but you should be able to get the idea — if we didn't have variables available, we'd have to implement a giant code block that checked what the entered name was, and then display the appropriate message for any name. This is obviously really inefficient (the code is a lot bigger, even for only five choices), and it just wouldn't work — you couldn't possibly store all possible choices.

Variables just make sense, and as you learn more about JavaScript they will start to become second nature.

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. You'll learn more about this as you go along.

> **Note**: We say variables contain values. This is an important distinction to make. Variables aren't the values themselves; they are containers for values. You can think of them being like little cardboard boxes that you can store things in.



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. After that, try creating a variable (or two) with your own name choices.

> **Note**: In JavaScript, all code instructions should end with a semi-colon (`;`) — your code may work correctly for single lines, but probably won't when you are writing multiple lines of code together. Try to get into the habit of including it.

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

```
myName;
myAge;
```

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 — try typing in

```
scoobyDoo;
```

> **Note**: Don't confuse a variable that exists but has no defined value with a variable that doesn't exist at all — they are very different things. In the box analogy you saw above, not existing would mean there's no box (variable) for a value to go in. No value defined would mean that there IS a box, but it has no value inside it.

Initializing a variable

Once you've declared a variable, you can initialize it with a value. You do this by typing the variable name, followed by an equals sign (=), followed by the value you want to give it. For example:

```
myName = 'Chris';
myAge = 37;
```

Try going back to the console now and typing in these lines. You should see the value you've assigned to the variable returned in the console to confirm it, in each case. Again, you can return your variable values by simply typing their name into the console — try these again:

```
myName;
myAge;
```

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

```
let myDog = 'Rover';
```

This is probably what you'll do most of the time, as it is quicker than doing the two actions on two separate lines.

The difference between var and let

At this point you may be thinking "why do we need two keywords for defining variables?? Why have `var` *and* `let`?".

The reasons are somewhat historical. Back when JavaScript was first created, there was only `var`. This works basically fine in most cases, but it has some issues in the way it works — its design can sometimes be confusing or downright annoying. So, `let` was created in modern versions of JavaScript, a new keyword for creating variables that works somewhat differently to `var`, fixing its issues in the process.

A couple of simple differences are explained below. We won't go into all the differences now, but you'll start to discover them as you learn more about JavaScript (if you really want to read about them now, feel free to check out our let reference page).

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. For example:

```
myName = 'Chris';

function logName() {
  console.log(myName);
}

logName();

var myName;
```

> **Note**: This won't work when typing individual lines into a JavaScript console, just when running multiple lines of JavaScript in a web document.

This works because of **hoisting** — read var hoisting for more detail on the subject.

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.

For these reasons and more, we recommend that you use `let` as much as possible in your code, rather than `var`. There is no reason to use `var`, unless you need to support old versions of Internet Explorer with your code (it doesn't support `let` until version 11; the modern Windows Edge browser supports `let` just fine).

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. Try entering the following lines into your console:

```
myName = 'Bob';
myAge = 40;
```

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", 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.

> **Note**: You can find a fairly complete list of reserved keywords to avoid at Lexical grammar — keywords.

Good name examples:

```
age
myAge
init
initialColor
finalOutputValue
audio1
audio2
```

Bad name examples:

```
1
a
_12
myage
MYAGE
var
Document
skjfndskjfnbdskjfb
thisisareallylongstupidvariablenameman
```

Try creating a few more variables now, with the above guidance in mind.

Variable types

There are a few different types of data we can store in variables. In this section we'll describe these in brief, then in future articles, you'll learn about them in more detail.

So far we've looked at the first two, but there are others.

Numbers

You can store numbers in variables, either whole numbers like 30 (also called integers) or decimal numbers like 2.456 (also called floats or floating point numbers). You don't need to declare variable types in JavaScript, unlike some other programming languages. When you give a variable a number value, you don't include quotes:

```
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; otherwise, JavaScript tries to interpret it as another variable name.

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

Booleans

Booleans are true/false values — they can have two values, `true` or `false`. These are generally used to test a condition, after which code is run as appropriate. So for example, a simple case would be:

```
let iAmAlive = true;
```

Whereas in reality it would be used more like this:

```
let test = 6 < 3;
```

This is using the "less than" operator (<) to test whether 6 is less than 3. As you might expect, it returns `false`, because 6 is not less than 3! You will learn a lot more about such operators later on in the course.

Arrays

An array is a single object that contains multiple values enclosed in square brackets and separated by commas. Try entering the following lines into your console:

```
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. Try these lines:

```
myNameArray[0]; // should return 'Chris'
myNumberArray[2]; // should return 40
```

The square brackets specify an index value corresponding to the position of the value you want returned. You might have noticed that arrays in JavaScript are zero-indexed: the first element is at index 0.

You'll learn a lot more about arrays in a future article.

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, how to say hello to them, and more.

Try entering the following line into your console:

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

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

```
dog.name
```

We won't be looking at objects any more for now — you can learn more about those in a future module.

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).

For example, if you declare a variable and give it a value enclosed in quotes, the browser treats the variable as a string:

```
let myString = 'Hello';
```

Even if the value contains numbers, it is still a string, so be careful:

```
let myNumber = '500'; // oops, this is still a string
typeof myNumber;
myNumber = 500; // much better — now this is a number
typeof myNumber;
```

Try entering the four lines above into your console one by one, and see what the results are. You'll notice that we are using a special operator called `typeof` — this returns the data type of the variable you type after it. The first time it is called, it should return `string`, as at that point the `myNumber` variable contains a string, `'500'`. Have a look and see what it returns the second time you call it.

Constants in JavaScript

Many programming languages have the concept of a *constant* — a value that once declared can't be changed. There are many reasons why you'd want to do this, from security (if a third party script changed such values it could cause problems) to debugging and code comprehension (it is harder to accidentally change values that shouldn't be changed and mess things up).

In the early days of JavaScript, constants didn't exist. In modern JavaScript, we have the keyword `const`, which lets us store values that can never be changed:

```
const daysInWeek = 7;
const hoursInDay = 24;
```

`const` works in exactly the same way as `let`, except that you can't give a `const` a new value. In the following example, the second line would throw an error:

```
const daysInWeek = 7;
daysInWeek = 8;
```

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: variables.

Summary

By now you should know a reasonable amount about JavaScript variables and how to create them. In the next article, we'll focus on numbers in more detail, looking at how to do basic math in JavaScript.

Previous Overview: First steps Next

In this module

- What is JavaScript?
- The first splash into JavaScript
- What went wrong? Troubleshooting JavaScript
- Storing the information you need — Variables
- Basic math in JavaScript — numbers and operators
- Handling text — strings in JavaScript
- Useful string methods
- Arrays
- Assessment: Silly story generator

Last modified: Feb 6, 2020, by MDN contributors

Related Topics

Complete beginners start here!

- Getting started with the Web

HTML — Structuring the Web

Basic math in JavaScript — numbers and operators

Previous Overview: First steps Next

At this point in the course we discuss math in JavaScript — how we can use operators and other features to successfully manipulate numbers to do our bidding.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To gain familiarity with the basics of math in JavaScript.

Everybody loves math

Okay, maybe not. Some of us like math, some of us have hated math ever since we had to learn multiplication tables and long division in school, and some of us sit somewhere in between the two. But none of us can deny that math is a fundamental part of life that we can't get very far without. This is especially true when we are learning to program JavaScript (or any other language for that matter) — so much of what we do relies on processing numerical data, calculating new values, and so on, that you won't be surprised to learn that JavaScript has a full-featured set of math functions available.

This article discusses only the basic parts that you need to know now.

Types of numbers

In programming, even the humble decimal number system that we all know so well is more complicated than you might think. We use different terms to describe different types of decimal numbers, for example:

- **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.

Before you start to get worried about your brain melting, stop right there! For a start, we are just going to stick to decimal numbers throughout this course; you'll rarely come across a need to start thinking about other types, if ever.

The second bit of good news is that unlike some other programming languages, JavaScript only has one data type for numbers, both integers and decimals — you guessed it, `Number`. This means that whatever type of numbers you are dealing with in JavaScript, you handle them in exactly the same way.

> **Note**: Actually, JavaScript has a second number type, BigInt, used for very, very large integers. But for the purposes of this course, we'll just worry about `Number` values.

It's all numbers to me

Let's quickly play with some numbers to reacquaint ourselves with the basic syntax we need. Enter the commands listed below into your developer tools JavaScript console.

1. First of all, let's declare a couple of variables and initialize them with an integer and a float, respectively, then type the variable names back in to check that everything is in order:

```
let myInt = 5;
let myFloat = 6.667;
myInt;
myFloat;
```

2. Number values are typed in without quote marks — try declaring and initializing a couple more variables containing numbers before you move on.
3. Now let's check that both our original variables are of the same datatype. There is an operator called `typeof` in JavaScript that does this. Enter the below two lines as shown:

```
typeof myInt;
typeof myFloat;
```

   You should get `"number"` returned in both cases — this makes things a lot easier for us than if different numbers had different data types, and we had to deal with them in different ways. Phew!

Useful Number methods

The `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. We don't cover these in detail in this article because we wanted to keep it as a simple introduction and only cover the real basic essentials for now; however, once you've read through this module a couple of times it is worth going to the object reference pages and learning more about what's available.

For example, to round your number to a fixed number of decimal places, use the `toFixed()` method. Type the following lines into your browser's console:

```
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 input, and the input type is text. There is a way to solve this problem — passing the string value into the `Number()` constructor to return a number version of the same value.

For example, try typing these lines into your console:

```
let myNumber = '74';
myNumber + 3;
```

You end up with the result 743, not 77, because `myNumber` is actually defined as a string. You can test this by typing in the following:

```
typeof myNumber;
```

To fix the calculation, you can do this:

```
Number(myNumber) + 3;
```

Arithmetic operators

Arithmetic operators are the basic operators that we use to do sums in JavaScript:

Operator	Name	Purpose	Example
+	Addition	Adds two numbers together.	6 + 9

Operator	Name	Purpose	Example
`-`	Subtraction	Subtracts the right number from the left.	`20 - 15`
`*`	Multiplication	Multiplies two numbers together.	`3 * 7`
`/`	Division	Divides the left number by the right.	`10 / 5`
`%`	Remainder (sometimes called modulo)	Returns the remainder left over after you've divided the left number into a number of integer portions equal to the right number.	`8 % 3` (returns 2, as three goes into 8 twice, leaving 2 left over).
`**`	Exponent	Raises a `base` number to the `exponent` power, that is, the `base` number multiplied by itself, `exponent` times. It was first Introduced in EcmaScript 2016.	`5 ** 2` (returns `25`, which is the same as `5 * 5`).

> **Note**: You'll sometimes see numbers involved in arithmetic referred to as operands.

> **Note**: You may sometimes see exponents expressed using the older `Math.pow()` method, which works in a very similar way. For example, in `Math.pow(7, 3)`, `7` is the base and `3` is the exponent, so the result of the expression is `343`. `Math.pow(7, 3)` is equivalent to `7**3`.

We probably don't need to teach you how to do basic math, but we would like to test your understanding of the syntax involved. Try entering the examples below into your developer tools JavaScript console to familiarize yourself with the syntax.

1. First try entering some simple examples of your own, such as

```
10 + 7
9 * 8
60 % 3
```

2. You can also try declaring and initializing some numbers inside variables, and try using those in the sums — the variables will behave exactly like the values they hold for the purposes of the sum. For example:

```
let num1 = 10;
let num2 = 50;
9 * num1;
num1 ** 3;
num2 / num1;
```

3. Last for this section, try entering some more complex expressions, such as:

```
5 + 10 * 3;
num2 % 9 * num1;
num2 + num1 / 8 + 2;
```

Some of this last set of calculations might not give you quite the result you were expecting; the section below might well give the answer as to why.

Operator precedence

Let's look at the last example from above, assuming that `num2` holds the value 50 and `num1` holds the value 10 (as originally stated above):

```
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"*.

This is because of **operator precedence** — some operators are applied before others when calculating the result of a calculation (referred to as an *expression*, in programming). Operator precedence in JavaScript is the same as is taught in math classes in school — Multiply and divide are always done first, then add and subtract (the calculation is always evaluated from left to right).

If you want to override operator precedence, you can put parentheses round the parts that you want to be explicitly dealt with first. So to get a result of 6, we could do this:

```
(num2 + num1) / (8 + 2);
```

Try it and see.

> **Note**: A full list of all JavaScript operators and their precedence can be found in Expressions and operators.

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. We used `++` in our "Guess the number" game back in our first splash into JavaScript article, when we added 1 to our `guessCount` variable to keep track of how many guesses the user has left after each turn.

```
guessCount++;
```

> **Note**: These operators are most commonly used in loops, which you'll learn about later on in the course. For example, say you wanted to loop through a list of prices, and add sales tax to each one. You'd use a loop to go through each value in turn and do the necessary calculation for adding the sales tax in each case. The incrementor is used to move to the next value when needed. We've actually provided a simple example showing how this is done — check it out live, and look at the source code to see if you can spot the incrementors! We'll look at loops in detail later on in the course.

Let's try playing with these in your console. For a start, note that you can't apply these directly to a number, which might seem strange, but we are assigning a variable a new updated value, not operating on the value itself. The following will return an error:

```
3++;
```

So, you can only increment an existing variable. Try this:

```
let num1 = 4;
num1++;
```

Okay, strangeness number 2! When you do this, you'll see a value of 4 returned — this is because the browser returns the current value, *then* increments the variable. You can see that it's been incremented if you return the variable value again:

```
num1;
```

The same is true of `--` : try the following

```
let num2 = 6;
num2--;
num2;
```

Note: You can make the browser do it the other way round — increment/decrement the variable *then* return the value — by putting the operator at the start of the variable instead of the end. Try the above examples again, but this time use `++num1` and `--num2`.

Assignment operators

Assignment operators are operators that assign a value to a variable. We have already used the most basic one, `=`, loads of times — it simply assigns the variable on the left the value stated on the right:

```
let x = 3; // x contains the value 3
let y = 4; // y contains the value 4
x = y; // x now contains the same value y contains, 4
```

But there are some more complex types, which provide useful shortcuts to keep your code neater and more efficient. The most common are listed below:

Operator	Name	Purpose	Example	Shortcut for
`+=`	Addition assignment	Adds the value on the right to the variable value on the left, then returns the new variable value	`x = 3; x += 4;`	`x = 3; x = x + 4;`
`-=`	Subtraction assignment	Subtracts the value on the right from the variable value on the left, and returns the new variable value	`x = 6; x -= 3;`	`x = 6; x = x - 3;`
`*=`	Multiplication assignment	Multiplies the variable value on the left by the value on the right, and returns the new variable value	`x = 2; x *= 3;`	`x = 2; x = x * 3;`
`/=`	Division assignment	Divides the variable value on the left by the value on the right, and returns the new variable value	`x = 10; x /= 5;`	`x = 10; x = x / 5;`

Try typing some of the above examples into your console, to get an idea of how they work. In each case, see if you can guess what the value is before you type in the second line.

Note that you can quite happily use other variables on the right hand side of each expression, for example:

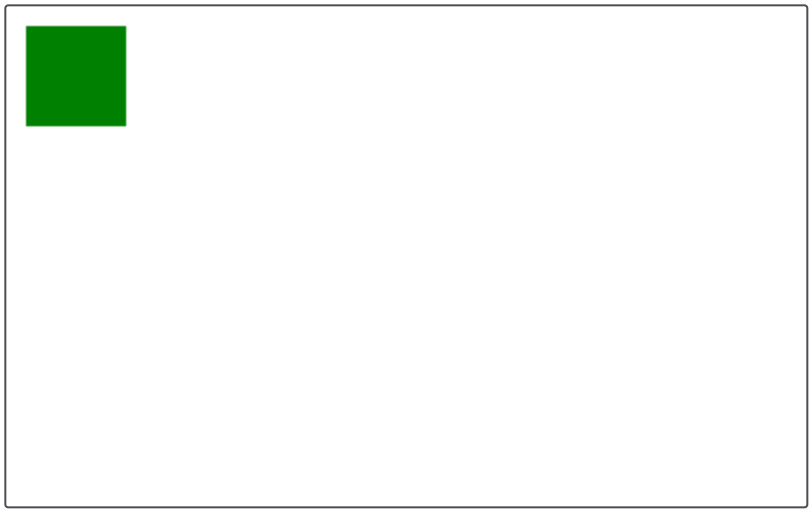
```
let x = 3; // x contains the value 3
let y = 4; // y contains the value 4
x *= y; // x now contains the value 12
```

Note: There are lots of other assignment operators available, but these are the basic ones you should learn now.

Active learning: sizing a canvas box

In this exercise, you will manipulate some numbers and operators to change the size of a box. The box is drawn using a browser API called the Canvas API. There is no need to worry about how this works — just concentrate on the math for now. The width and height of the box (in pixels) are defined by the variables `x` and `y`, which are initially both given a value of 50.

Live output



The rectangle is 50px wide and 50px high.

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
let x = 50; let y = 50;

// Edit the two lines below here ONLY
x = 50;
y = 50;

ctx.fillStyle = 'green';
ctx.fillRect(10, 10, x, y);
```

Reset

Open in new window

In the editable code box above, there are two lines marked with a comment that we'd like you to update to make the box grow/shrink to certain sizes, using certain operators and/or values in each case. Let's try the following:

- Change the line that calculates x so the box is still 50px wide, but the 50 is calculated using the numbers 43 and 7 and an arithmetic operator.

- Change the line that calculates y so the box is 75px high, but the 75 is calculated using the numbers 25 and 3 and an arithmetic operator.
- Change the line that calculates x so the box is 250px wide, but the 250 is calculated using two numbers and the remainder (modulo) operator.
- Change the line that calculates y so the box is 150px high, but the 150 is calculated using three numbers and the subtraction and division operators.
- Change the line that calculates x so the box is 200px wide, but the 200 is calculated using the number 4 and an assignment operator.
- Change the line that calculates y so the box is 200px high, but the 200 is calculated using the numbers 50 and 3, the multiplication operator, and the addition assignment operator.

Don't worry if you totally mess the code up. You can always press the Reset button to get things working again. After you've answered all the above questions correctly, feel free to play with the code some more or create your own challenges.

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**.

Operator	Name	Purpose	Example
`===`	Strict equality	Tests whether the left and right values are identical to one another	`5 === 2 + 4`
`!==`	Strict-non-equality	Tests whether the left and right values are **not** identical to one another	`5 !== 2 + 3`
`<`	Less than	Tests whether the left value is smaller than the right one.	`10 < 6`
`>`	Greater than	Tests whether the left value is greater than the right one.	`10 > 20`
`<=`	Less than or equal to	Tests whether the left value is smaller than or equal to the right one.	`3 <= 2`

`>=`	Greater than or equal to	Tests whether the left value is greater than or equal to the right one.	`5 >= 4`

> **Note**: You may see some people using `==` and `!=` in their tests for equality and non-equality. These are valid operators in JavaScript, but they differ from `===` / `!==`. The former versions test whether the values are the same but not whether the values' datatypes are the same. The latter, strict versions test the equality of both the values and their datatypes. The strict versions tend to result in fewer errors, so we recommend you use them.

If you try entering some of these values in a console, you'll see that they all return `true`/`false` values — those booleans we mentioned in the last article. These are very useful, as they allow us to make decisions in our code, and they are used every time we want to make a choice of some kind. For example, booleans can be used to:

- Display the correct text label on a button depending on whether a feature is turned on or off
- Display a game over message if a game is over or a victory message if the game has been won
- Display the correct seasonal greeting depending what holiday season it is
- Zoom a map in or out depending on what zoom level is selected

We'll look at how to code such logic when we look at conditional statements in a future article. For now, let's look at a quick example:

```
<button>Start machine</button>
<p>The machine is stopped.</p>
```

```
const btn = document.querySelector('button');
const txt = document.querySelector('p');

btn.addEventListener('click', updateBtn);

function updateBtn() {
  if (btn.textContent === 'Start machine') {
    btn.textContent = 'Stop machine';
    txt.textContent = 'The machine has started!';
```

```
  } else {
    btn.textContent = 'Start machine';
    txt.textContent = 'The machine is stopped.';
  }
}
```

Start machine

The machine is stopped.

Open in new window

You can see the equality operator being used just inside the `updateBtn()` function. In this case, we are not testing if two mathematical expressions have the same value — we are testing whether the text content of a button contains a certain string — but it is still the same principle at work. If the button is currently saying "Start machine" when it is pressed, we change its label to "Stop machine", and update the label as appropriate. If the button is currently saying "Stop machine" when it is pressed, we swap the display back again.

> **Note**: Such a control that swaps between two states is generally referred to as a **toggle**. It toggles between one state and another — light on, light off, etc.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Math.

Summary

In this article we have covered the fundamental information you need to know about numbers in JavaScript, for now. You'll see numbers used again and again, all the way through your JavaScript learning, so it's a good idea to get this out of the way now. If you are one of those people that doesn't enjoy math, you can take comfort in the fact that this chapter was pretty short.

In the next article, we'll explore text and how JavaScript allows us to manipulate it.

> **Note**: If you do enjoy math and want to read more about how it is implemented in JavaScript, you can find a lot more detail in MDN's main JavaScript section. Great places to start are our Numbers and dates and Expressions and operators articles.

Previous Overview: First steps Next

In this module

- What is JavaScript?
- A first splash into JavaScript
- What went wrong? Troubleshooting JavaScript
- Storing the information you need — Variables
- Basic math in JavaScript — numbers and operators
- Handling text — strings in JavaScript
- Useful string methods
- Arrays
- Assessment: Silly story generator

Last modified: Apr 22, 2020, by MDN contributors

Handling text — strings in JavaScript

Previous Overview: First steps Next

Next, we'll turn our attention to strings — this is what pieces of text are called in programming. In this article, we'll look at all the common things that you really ought to know about strings when learning JavaScript, such as creating strings, escaping quotes in strings, and joining strings together.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To gain familiarity with the basics of strings in JavaScript.

The power of words

Words are very important to humans — they are a large part of how we communicate. Since the Web is a largely text-based medium designed to allow humans to communicate and share information, it is useful for us to have control over the words that appear on it. HTML provides structure and meaning to our text, CSS allows us to precisely style it, and JavaScript contains a number of features for manipulating strings, creating custom welcome messages and prompts, showing the right text labels when needed, sorting terms into the desired order, and much more.

Pretty much all of the programs we've shown you so far in the course have involved some string manipulation.

Strings — the basics

Strings are dealt with similarly to numbers at first glance, but when you dig deeper you'll start to see some notable differences. Let's start by entering some basic lines into the browser developer console to familiarize ourselves.

Creating a string

1. 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.

2. If you don't do this, or miss one of the quotes, you'll get an error. Try entering the following lines:

```
let badString = This is a test;
let badString = 'This is a test;
let badString = This is a test';
```

   These lines don't work because any text without quotes around it is assumed to be a variable name, property name, a reserved word, or similar. If the browser can't find it, then an error is raised (e.g. "missing; before statement"). If the browser can see where a string starts, but can't find the end of the string, as indicated by the 2nd quote, it complains with an error (with "unterminated string literal"). If your program is raising such errors, then go back and check all your strings to make sure you have no missing quote marks.

3. The following will work if you previously defined the variable `string` — try it now:

```
let badString = string;
badString;
```

   `badString` is now set to have the same value as `string`.

Single quotes vs. double quotes

1. In JavaScript, you can choose single quotes or double quotes to wrap your strings in. Both of the following will work okay:

```
let sgl = 'Single quotes.';
let dbl = "Double quotes";
sgl;
dbl;
```

2. There is very little difference between the two, and which you use is down to personal preference. You should choose one and stick to it, however; differently quoted code can be confusing, especially if you use two different quotes on the same string! The following will return an error:

```
let badQuotes = 'What on earth?";
```

3. The browser will think the string has not been closed because the other type of quote you are not using to contain your strings can appear in the string. For example, both of these are okay:

```
let sglDbl = 'Would you eat a "fish supper"?';
let dblSgl = "I'm feeling blue.";
sglDbl;
dblSgl;
```

4. However, you can't include the same quote mark inside the string if it's being used to contain them. The following will error, as it confuses the browser as to where the string ends:

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

   This leads us very nicely into our next subject.

Escaping characters in a string

To fix our previous problem code line, we need to escape the problem quote mark. Escaping characters means that we do something to them to make sure they are recognized as text, not

part of the code. In JavaScript, we do this by putting a backslash just before the character. Try this:

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

This works fine. You can escape other characters in the same way, e.g. `\"`, and there are some special codes besides. See Escape notation for more details.

Concatenating strings

1. Concatenate is a fancy programming word that means "join together". Joining together strings in JavaScript uses the plus (+) operator, the same one we use to add numbers together, but in this context it does something different. Let's try an example in our console.

   ```
   let one = 'Hello, ';
   let two = 'how are you?';
   let joined = one + two;
   joined;
   ```

   The result of this is a variable called `joined`, which contains the value "Hello, how are you?".

2. In the last instance, we joined only two strings, but you can join as many as you like, as long as you include a `+` between each pair. Try this:

   ```
   let multiple = one + one + one + one + two;
   multiple;
   ```

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

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

> **Note**: When you enter an actual string in your code, enclosed in single or double quotes, it is called a **string literal**.

Concatenation in context

Let's have a look at concatenation being used in action — here's an example from earlier in the course:

```
<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!');
}
```

Press me

Here we're using a `window.prompt()` function in line 4, which asks the user to answer a question via a popup dialog box then stores the text they enter inside a given variable — in this case `name`. We then use a `window.alert()` function in line 5 to display another popup containing a string we've assembled from two string literals and the `name` variable, via concatenation.

Numbers vs. strings

1. 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 might expect this to return an error, but it works just fine. Trying to represent a string as a number doesn't really make sense, but representing a number as a string does, so

the browser rather cleverly converts the number to a string and concatenates the two strings.

2. 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;
```

3. 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` 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()` that converts it to the equivalent string. Try this:

```
let myNum = 123;
let myString = myNum.toString();
typeof myString;
```

   These constructs can be really useful in some situations. For example, if a user enters a number into a form's text field, it's a string. However, if you want to add this number to something, you'll need it to be a number, so you could pass it through `Number()` to handle this. We did exactly this in our Number Guessing Game, in line 54.

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.

> **Note**: Try entering the below examples into your browser's JavaScript console, to see what results you get.

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 ' + (s
```

Template literals simplify this enormously:

```
output = `I like the song "${ song }". I gave it a score of ${ score/high
```

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;
```

```
3 examReport = `You scored ${ examScore }/${ examHighestScore } (${ Math.rc
```

- 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`:

```
1 output = 'I like the song "' + song + '".\nI gave it a score of ' + (scor
```

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

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

We would recommend that you get used to using template literals as soon as possible. They are well-supported in modern browsers, and the only place you'll find a lack of support is Internet Explorer. Many of our examples still use standard string literals, but we will include more template literals going forward.

See our Template literals reference page for more examples and details of advanced features.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Strings. Note that this also requires knowledge from the next article, so you might want to read that first.

Conclusion

So that's the very basics of strings covered in JavaScript. In the next article, we'll build on this, looking at some of the built-in methods available to strings in JavaScript and how we can use them to manipulate our strings into just the form we want.

Previous Overview: First steps Next

In this module

- What is JavaScript?
- A first splash into JavaScript
- What went wrong? Troubleshooting JavaScript
- Storing the information you need — Variables
- Basic math in JavaScript — numbers and operators
- Handling text — strings in JavaScript
- Useful string methods
- Arrays
- Assessment: Silly story generator

Last modified: Mar 29, 2020, by MDN contributors

Related Topics

Complete beginners start here!

- Getting started with the Web

HTML — Structuring the Web

Useful string methods

Previous Overview: First steps Next

Now that we've looked at the very basics of strings, let's move up a gear and start thinking about what useful operations we can do on strings with built-in methods, such as finding the length of a text string, joining and splitting strings, substituting one character in a string for another, and more.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To understand that strings are objects, and learn how to use some of the basic methods available on those objects to manipulate strings.

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. You can see this if you go to the `String` object page and look down the list on the side of the page!

Now, before your brain starts melting, don't worry! You really don't need to know about most of these early on in your learning journey. But there are a few that you'll potentially use quite often that we'll look at here.

Let's enter some examples into the browser developer console.

Finding the length of a string

This is easy — you simply use the `length` property. Try entering the following lines:

```
let browserType = 'mozilla';
browserType.length;
```

This should return the number 7, because "mozilla" is 7 characters long. This is useful for many reasons; for example, you might want to find the lengths of a series of names so you can display them in order of length, or let a user know that a username they have entered into a form field is too long if it is over a certain length.

Retrieving a specific string character

On a related note, you can return any character inside a string by using **square bracket notation** — this means you include square brackets (`[]`) on the end of your variable name. Inside the square brackets you include the number of the character you want to return, so for example to retrieve the first letter you'd do this:

```
browserType[0];
```

Remember: computers count from 0, not 1! You could use this to, for example, find the first letter of a series of strings and order them alphabetically.

To retrieve the last character of *any* string, we could use the following line, combining this technique with the `length` property we looked at above:

```
browserType[browserType.length-1];
```

The length of "mozilla" is 7, but because the count starts at 0, the character position is 6; using `length-1` gets us the last character.

Finding a substring inside a string and extracting it

Sometimes you'll want to find if a smaller string is present inside a larger one (we generally say *if a substring is present inside a string*). This can be done using the `indexOf()` method, which takes a single parameter — the substring you want to search for.

If the substring *is* found inside the main string, it returns a number representing the index position of the stubstring — which character number of the main string the substring starts at. If the substring is *not found* inside the main string, it returns a value of `-1`.

1. Try this:

```
browserType.indexOf('zilla');
```

   This gives us a result of 2, because the substring "zilla" starts at position 2 (0, 1, 2 — so 3 characters in) inside "mozilla". Such code could be used to filter strings. For example, we may have a list of web addresses and only want to print out the ones that contain "mozilla".

2. This can be done in another way, which is possibly even more effective. Try the following:

```
browserType.indexOf('vanilla');
```

   This should give you a result of `-1` — this is returned when the substring, in this case 'vanilla', is not found in the main string.

   You could use this to find all instances of strings that **don't** contain the substring 'mozilla', or **do,** if you use the negation operator, as shown below. You could do something like this:

```
if(browserType.indexOf('mozilla') !== -1) {
  // do stuff with the string
}
```

3. When you know where a substring starts inside a string, and you know at which character you want it to end, `slice()` can be used to extract it. Try the following:

```
browserType.slice(0,3);
```

This returns "moz" — the first parameter is the character position to start extracting at, and the second parameter is the character position after the last one to be extracted. So the slice happens from the first position, up to, but not including, the last position. In this example, since the starting index is 0, the second parameter is equal to the length of the string being returned.

4. Also, 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);
```

This returns "zilla" — this is because the character position of 2 is the letter z, and because you didn't include a second parameter, the substring that was returned was all of the remaining characters in the string.

> **Note**: The second parameter of `slice()` is optional: if you don't include it, the slice ends at the end of the original string. There are other options too; study the `slice()` page to see what else you can find out.

Changing case

The string methods `toLowerCase()` and `toUpperCase()` take a string and convert all the characters to lower- or uppercase, respectively. This can be useful for example if you want to normalize all user-entered data before storing it in a database.

Let's try entering the following lines to see what happens:

```
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()` method. This works very simply at a basic level, although there are some advanced things you can do with it that we won't go into yet.

It takes two parameters — the string you want to replace, and the string you want to replace it with. Try this example:

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

This returns "vanilla" in the console. But if you check the value of `browserType`, it is still "mozilla'. To actually update the value of the `browserType` variable in a real program, you'd have to set the variable value to be the result of the operation; it doesn't just update the substring value automatically. So you'd have to actually write this: `browserType = browserType.replace('moz','van');`

Active learning examples

In this section we'll get you to try your hand at writing some string manipulation code. In each exercise below, we have an array of strings, and a loop that processes each value in the array and displays it in a bulleted list. You don't need to understand arrays or loops right now — these will be explained in future articles. All you need to do in each case is write the code that will output the strings in the format that we want them in.

Each example comes with a "Reset" button, which you can use to reset the code if you make a mistake and can't get it working again, and a "Show solution" button you can press to see a potential answer if you get really stuck.

Filtering greeting messages

In the first exercise we'll start you off simple — we have an array of greeting card messages, but we want to sort them to list just the Christmas messages. We want you to fill in a conditional

test inside the `if( ... )` structure, to test each string and only print it in the list if it is a Christmas message.

1. First think about how you could test whether the message in each case is a Christmas message. What string is present in all of those messages, and what method could you use to test whether it is present?
2. You'll then need to write a conditional test of the form *operand1 operator operand2*. Is the thing on the left equal to the thing on the right? Or in this case, does the method call on the left return the result on the right?
3. Hint: In this case it is probably more useful to test whether the method call *isn't* equal to a certain result.

Live output

- Happy Birthday!
- Merry Christmas my love
- A happy Christmas to all the family
- You're all I want for Christmas
- Get well soon

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
const list = document.querySelector('.output ul');
list.innerHTML = '';
let greetings = ['Happy Birthday!',
                 'Merry Christmas my love',
                 'A happy Christmas to all the family',
                 'You\'re all I want for Christmas',
                 'Get well soon'];

for (let i = 0; i < greetings.length; i++) {
  let input = greetings[i];
  // Your conditional test needs to go inside the parentheses
  // in the line below, replacing what's currently there
  if (greetings[i]) {
    let listItem = document.createElement('li');
    listItem.textContent = input;
    list.appendChild(listItem);
  }
}
```

Reset Show solution

Fixing capitalization

In this exercise we have the names of cities in the United Kingdom, but the capitalization is all messed up. We want you to change them so that they are all lower case, except for a capital first letter. A good way to do this is to:

1. Convert the whole of the string contained in the `input` variable to lower case and store it in a new variable.
2. Grab the first letter of the string in this new variable and store it in another variable.
3. Using this latest variable as a substring, replace the first letter of the lowercase string with the first letter of the lowercase string changed to upper case. Store the result of this replace procedure in another new variable.
4. Change the value of the `result` variable to equal to the final result, not the `input`.

> **Note**: A hint — the parameters of the string methods don't have to be string literals; they can also be variables, or even variables with a method being invoked on them.

Live output

- lonDon
- ManCHESTer
- BiRmiNGHAM
- liVERpoOL

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
const list = document.querySelector('.output ul');
list.innerHTML = '';
let cities = ['lonDon', 'ManCHESTer', 'BiRmiNGHAM',
'liVERpoOL'];

for (let i = 0; i < cities.length; i++) {
  let input = cities[i];
  // write your code just below here

  let result = input;
  let listItem = document.createElement('li');
  listItem.textContent = result;
  list.appendChild(listItem);
}
```

Reset Show solution

Making new strings from old parts

In this last exercise, the array contains a bunch of strings containing information about train stations in the North of England. The strings are data items that contain the three-letter station code, followed by some machine-readable data, followed by a semicolon, followed by the human-readable station name. For example:

```
1 MAN675847583748sjt567654;Manchester Piccadilly
```

We want to extract the station code and name, and put them together in a string with the following structure:

```
1 MAN: Manchester Piccadilly
```

We'd recommend doing it like this:

1. Extract the three-letter station code and store it in a new variable.
2. Find the character index number of the semicolon.
3. Extract the human-readable station name using the semicolon character index number as a reference point, and store it in a new variable.
4. Concatenate the two new variables and a string literal to make the final string.
5. Change the value of the `result` variable to equal to the final string, not the `input`.

Live output

- MAN675847583748sjt567654;Manchester Piccadilly
- GNF576746573fhdg4737dh4;Greenfield
- LIV5hg65hd737456236dch46dg4;Liverpool Lime Street
- SYB4f65hf75f736463;Stalybridge
- HUD5767ghtyfyr4536dh45dg45dg3;Huddersfield

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
const list = document.querySelector('.output ul');
list.innerHTML = '';
let stations = ['MAN675847583748sjt567654;Manchester
Piccadilly',
                'GNF576746573fhdg4737dh4;Greenfield',
                'LIV5hg65hd737456236dch46dg4;Liverpool Lime
Street',
                'SYB4f65hf75f736463;Stalybridge',

'HUD5767ghtyfyr4536dh45dg45dg3;Huddersfield'];

for (let i = 0; i < stations.length; i++) {
  let input = stations[i];
  // write your code just below here

  let result = input;
  let listItem = document.createElement('li');
  listItem.textContent = result;
  list.appendChild(listItem);
```

Reset Show solution

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Strings.

Conclusion

You can't escape the fact that being able to handle words and sentences in programming is very important — particularly in JavaScript, as websites are all about communicating with people. This article has given you the basics that you need to know about manipulating strings for now. This should serve you well as you go into more complex topics in the future. Next, we're going to look at the last major type of data we need to focus on in the short term — arrays.

Previous Overview: First steps Next

In this module

- What is JavaScript?
- A first splash into JavaScript
- What went wrong? Troubleshooting JavaScript
- Storing the information you need — Variables
- Basic math in JavaScript — numbers and operators
- Handling text — strings in JavaScript
- Useful string methods
- Arrays
- Assessment: Silly story generator

Last modified: Apr 24, 2020, by MDN contributors

Arrays

Previous Overview: First steps Next

In the final article of this module, we'll look at arrays — a neat way of storing a list of data items under a single variable name. Here we look at why this is useful, then explore how to create an array, retrieve, add, and remove items stored in an array, and more besides.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To understand what arrays are and how to manipulate them in JavaScript.

What is an array?

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. Maybe we've got a series of product items and their prices stored in an array, and we want to loop through them all and print them out on an invoice, while totaling all the prices together and printing out the total price at the bottom.

If we didn't have arrays, we'd have to store every item in a separate variable, then call the code that does the printing and adding separately for each item. This would be much longer to write out, less efficient, and more error-prone. If we had 10 items to add to the invoice it would already be annoying, but what about 100 items, or 1000? We'll return to this example later on in the article.

As in previous articles, let's learn about the real basics of arrays by entering some examples into browser developer console.

Creating arrays

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

1. Suppose we want to store a shopping list in an array. Paste the following code into the console:

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

2. In the above example, each element is a string, but in an array we can store various data types — strings, numbers, objects, and even other arrays. We can also mix data types in a single array — we do not have to limit ourselves to storing only numbers in one array, and in another only strings. For example:

```
let sequence = [1, 1, 2, 3, 5, 8, 13];
let random = ['tree', 795, [0, 1, 2]];
```

3. Before proceeding, create a few example arrays.

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.

1. Enter the following into your console:

```
shopping[0];
// returns "bread"
```

2. 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;
// shopping will now return [ "tahini", "milk", "cheese", "hummus
```

> **Note**: We've said it before, but just as a reminder — computers start counting from 0!

3. 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. For example, to access one of the items inside the array that is the third item inside the `random` array (see previous section), we could do something like this:

```
random[2][2];
```

4. Try making some more modifications to your array examples before moving on. Play around a bit, and see what works and what doesn't.

Finding the length of an array

You can find out the length of an array (how many items are in it) in exactly the same way as you find out the length (in characters) of a string — by using the `length` property. Try the following:

```
shopping.length;
// should return 5
```

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. So for example:

```
let sequence = [1, 1, 2, 3, 5, 8, 13];
for (let i = 0; i < sequence.length; i++) {
  console.log(sequence[i]);
}
```

You'll learn about loops properly later on (in our Looping code article), but briefly, this code is saying:

1. Start looping at item number 0 in the array.
2. Stop looping at the item number equal to the length of the array. This works for an array of any length, but in this case it stops looping at item number 7 (this is good, as the last item — which we want the loop to include — is item 6).
3. For each item, print it out to the browser console with `console.log()`.

Some useful array methods

In this section we'll look at some rather useful array-related methods that allow us to split strings into array items and vice versa, and add new items into arrays.

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()` method. In its simplest form, this takes a single parameter, the character you want to separate the string at, and returns the substrings between the separator as items in an array.

> **Note**: Okay, this is technically a string method, not an array method, but we've put it in with arrays as it goes well here.

1. Let's play with this, to see how it works. First, create a string in your console:

```
let myData = 'Manchester,London,Liverpool,Birmingham,Leeds,Carlis
```

2. Now let's split it at each comma:

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

3. Finally, try finding the length of your new array, and retrieving some items from it:

```
myArray.length;
myArray[0]; // the first item in the array
myArray[1]; // the second item in the array
myArray[myArray.length-1]; // the last item in the array
```

4. You can also go the opposite way using the `join()` method. Try the following:

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

5. Another way of converting an array to a string is to use the `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. (Try running Step 4 with a different character than a comma.)

```
let dogNames = ['Rocket','Flash','Bella','Slugger'];
dogNames.toString(); // Rocket,Flash,Bella,Slugger
```

Adding and removing array items

We've not yet covered adding and removing array items — let us look at this now. We'll use the `myArray` array we ended up with in the last section. If you've not already followed that section, create the array first in your console:

```
let myArray = ['Manchester', 'London', 'Liverpool', 'Birmingham', 'Leeds'
```

First of all, to add or remove an item at the end of an array we can use `push()` and `pop()` respectively.

1. 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. Try this:

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

2. The new length of the array is returned when the method call completes. If you wanted to store the new array length in a variable, you could do something like this:

```
let newLength = myArray.push('Bristol');
myArray;
newLength;
```

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

```
myArray.pop();
```

4. The item that was removed is returned when the method call completes. To save that item in a new variable, you could do this:

```
let removedItem = myArray.pop();
myArray;
removedItem;
```

`unshift()` and `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.

1. First `unshift()` — try the following commands:

```
myArray.unshift('Edinburgh');
myArray;
```

2. Now `shift()`; try these!

```
let removedItem = myArray.shift();
myArray;
removedItem;
```

Active learning: Printing those products!

Let's return to the example we described earlier — printing out product names and prices on an invoice, then totaling the prices and printing them at the bottom. In the editable example below there are comments containing numbers — each of these marks a place where you have to add something to the code. They are as follows:

1. Below the `// number 1` comment are a number of strings, each one containing a product name and price separated by a colon. We'd like you to turn this into an array and store it in an array called `products`.
2. On the same line as the `// number 2` comment is the beginning of a for loop. In this line we currently have `i <= 0`, which is a conditional test that causes the for loop to only run once, because it is saying "stop when `i` is no longer less than or equal to 0", and `i` starts at 0. We'd like you to replace this with a conditional test that stops the loop when `i` is no longer less than the `products` array's length.
3. Just below the `// number 3` comment we want you to write a line of code that splits the current array item (`name:price`) into two separate items, one containing just the name and one containing just the price. If you are not sure how to do this, consult the Useful string methods article for some help, or even better, look at the Converting between strings and arrays section of this article.
4. As part of the above line of code, you'll also want to convert the price from a string to a number. If you can't remember how to do this, check out the first strings article.
5. There is a variable called `total` that is created and given a value of 0 at the top of the code. Inside the loop (below `// number 4`) we want you to add a line that adds the current item price to that total in each iteration of the loop, so that at the end of the code the correct total is printed onto the invoice. You might need an assignment operator to do this.
6. We want you to change the line just below `// number 5` so that the `itemText` variable is made equal to "current item name — $current item price", for example "Shoes — $23.99" in each case, so the correct information for each item is printed on the invoice. This is just simple string concatenation, which should be familiar to you.

Live output

- 0

Total: $0.00

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
const list = document.querySelector('.output ul');
const totalBox = document.querySelector('.output p');
let total = 0;
list.innerHTML = '';
totalBox.textContent = '';
// number 1
                'Underpants:6.99'
                'Socks:5.99'
                'T-shirt:14.99'
                'Trousers:31.99'
                'Shoes:23.99';

for (let i = 0; i <= 0; i++) { // number 2
  // number 3

  // number 4

  // number 5
  let itemText = 0;

  const listItem = document.createElement('li');
  listItem.textContent = itemText;
  list.appendChild(listItem);
}

totalBox.textContent = 'Total: $' + total.toFixed(2);
```

Reset Show solution

Active learning: Top 5 searches

A good use for array methods like `push()` and `pop()` is when you are maintaining a record of currently active items in a web app. In an animated scene for example, you might have an array of objects representing the background graphics currently displayed, and you might only want

50 displayed at once, for performance or clutter reasons. As new objects are created and added to the array, older ones can be deleted from the array to maintain the desired number.

In this example we're going to show a much simpler use — here we're giving you a fake search site, with a search box. The idea is that when terms are entered in the search box, the top 5 previous search terms are displayed in the list. When the number of terms goes over 5, the last term starts being deleted each time a new term is added to the top, so the 5 previous terms are always displayed.

> **Note**: In a real search app, you'd probably be able to click the previous search terms to return to previous searches, and it would display actual search results! We are just keeping it simple for now.

To complete the app, we need you to:

1. Add a line below the `// number 1` comment that adds the current value entered into the search input to the start of the array. This can be retrieved using `searchInput.value`.
2. Add a line below the `// number 2` comment that removes the value currently at the end of the array.

Live output

Search

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
const list = document.querySelector('.output ul');
const searchInput = document.querySelector('.output input');
const searchBtn = document.querySelector('.output button');

list.innerHTML = '';

let myHistory = [];

searchBtn.onclick = function() {
  // we will only allow a term to be entered if the search input isn't empty
  if (searchInput.value !== '') {
    // number 1

    // empty the list so that we don't display duplicate entries
    // the display is regenerated every time a search term is entered.
    list.innerHTML = '';

    // loop through the array, and display all the search terms in the list
    for (let i = 0; i < myHistory.length; i++) {
      itemText = myHistory[i];
      const listItem = document.createElement('li');
      listItem.textContent = itemText;
      list.appendChild(listItem);
    }
```

Reset Show solution

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Arrays.

Conclusion

After reading through this article, we are sure you will agree that arrays seem pretty darn useful; you'll see them crop up everywhere in JavaScript, often in association with loops in order to do the same thing to every item in an array. We'll be teaching you all the useful basics there are to know about loops in the next module, but for now you should give yourself a clap and take a well-deserved break; you've worked through all the articles in this module!

The only thing left to do is work through this module's assessment, which will test your understanding of the articles that came before it.

See also

- Indexed collections — an advanced level guide to arrays and their cousins, typed arrays.
- `Array` — the `Array` object reference page — for a detailed reference guide to the features discussed in this page, and many more.

Previous Overview: First steps Next

In this module

- What is JavaScript?
- A first splash into JavaScript
- What went wrong? Troubleshooting JavaScript
- Storing the information you need — Variables
- Basic math in JavaScript — numbers and operators
- Handling text — strings in JavaScript
- Useful string methods

Silly story generator

Previous Overview: First steps

In this assessment you'll be tasked with taking some of the knowledge you've picked up in this module's articles and applying it to creating a fun app that generates random silly stories. Have fun!

Prerequisites:	Before attempting this assessment you should have already worked through all the articles in this module.
Objective:	To test comprehension of JavaScript fundamentals, such as variables, numbers, operators, strings, and arrays.

Starting point

To get this assessment started, you should:

- Go and grab the HTML file for the example, save a local copy of it as `index.html` in a new directory somewhere on your computer, and do the assessment locally to begin with. This also has the CSS to style the example contained within it.
- Go to the page containing the raw text and keep this open in a separate browser tab somewhere. You'll need it later.

Alternatively, you could use a site like JSBin or Glitch to do your assessment. You could paste the HTML, CSS and JavaScript into one of these online editors. If the online editor you

are using doesn't have a separate JavaScript panel, feel free to put it inline in a `<script>` element inside the HTML page.

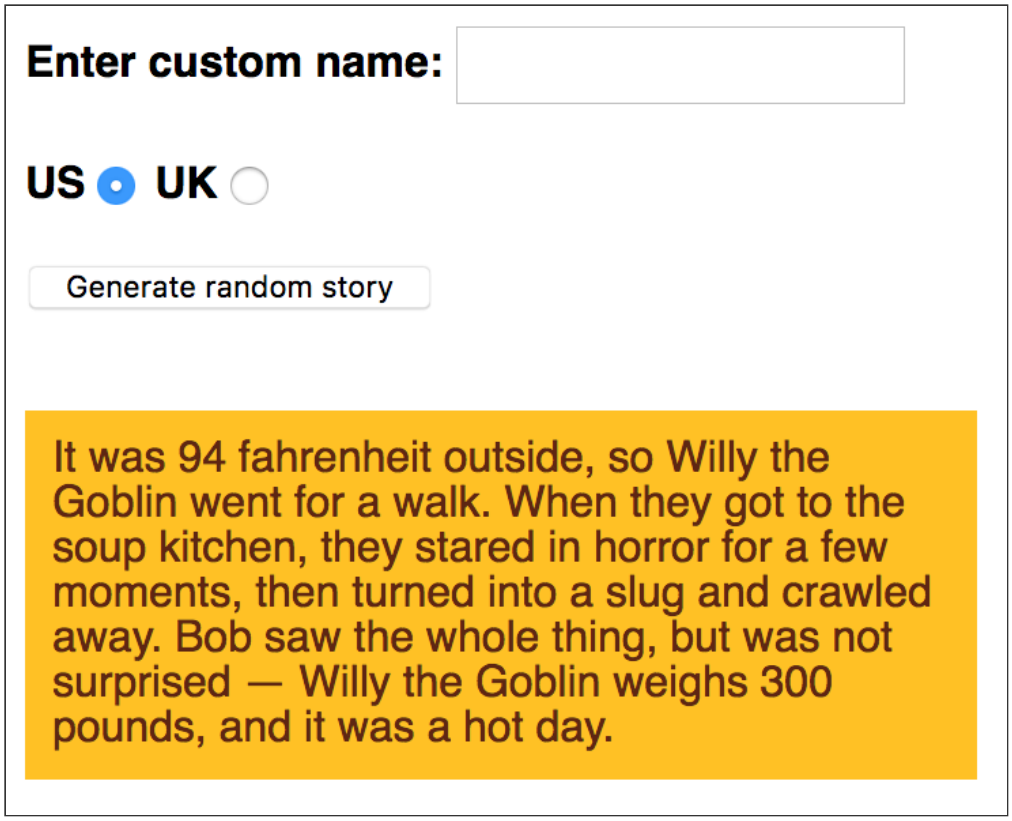
> **Note**: If you get stuck, then ask us for help — see the Assessment or further help section at the bottom of this page.

Project brief

You have been provided with some raw HTML/CSS and a few text strings and JavaScript functions; you need to write the necessary JavaScript to turn this into a working program, which does the following:

- Generates a silly story when the "Generate random story" button is pressed.
- Replaces the default name "Bob" in the story with a custom name, only if a custom name is entered into the "Enter custom name" text field before the generate button is pressed.
- Converts the default US weight and temperature quantities and units in the story into UK equivalents if the UK radio button is checked before the generate button is pressed.
- Will generate another random silly story if you press the button again (and again...)

The following screenshot shows an example of what the finished program should output:



To give you more of an idea, have a look at the finished example (no peeking at the source code!)

Steps to complete

The following sections describe what you need to do.

Basic setup:

1. Create a new file called `main.js`, in the same directory as your `index.html` file.
2. Apply the external JavaScript file to your HTML by inserting a `<script>` element into your HTML referencing `main.js`. Put it just before the closing `</body>` tag.

Initial variables and functions:

1. In the raw text file, copy all of the code underneath the heading "1. COMPLETE VARIABLE AND FUNCTION DEFINITIONS" and paste it into the top of the `main.js` file. This gives you three variables that store references to the "Enter custom name" text field

(`customName`), the "Generate random story" button (`randomize`), and the `<p>` element at the bottom of the HTML body that the story will be copied into (`story`), respectively. In addition you've got a function called `randomValueFromArray()` that takes an array, and returns one of the items stored inside the array at random.

2. Now look at the second section of the raw text file — "2. RAW TEXT STRINGS". This contains text strings that will act as input into our program. We'd like you to contain these inside variables inside `main.js`:
    1. Store the first, big long, string of text inside a variable called `storyText`.
    2. Store the first set of three strings inside an array called `insertX`.
    3. Store the second set of three strings inside an array called `insertY`.
    4. Store the third set of three strings inside an array called `insertZ`.

Placing the event handler and incomplete function:

1. Now return to the raw text file.
2. Copy the code found underneath the heading "3. EVENT LISTENER AND PARTIAL FUNCTION DEFINITION" and paste it into the bottom of your `main.js` file. This:
    - Adds a click event listener to the `randomize` variable so that when the button it represents is clicked, the `result()` function is run.
    - Adds a partially-completed `result()` function definiton to your code. For the remainder of the assessment, you'll be filling in lines inside this function to complete it and make it work properly.

Completing the `result()` function:

1. Create a new variable called `newStory`, and set it's value to equal `storyText`. This is needed so we can create a new random story each time the button is pressed and the function is run. If we made changes directly to `storyText`, we'd only be able to generate a new story once.
2. Create three new variables called `xItem`, `yItem`, and `zItem`, and make them equal to the result of calling `randomValueFromArray()` on your three arrays (the result in each case will be a random item out of each array it is called on). For example you can call the function and get it to return one random string out of `insertX` by writing `randomValueFromArray(insertX)`.
3. Next we want to replace the three placeholders in the `newStory` string — `:insertx:`, `:inserty:`, and `:insertz:` — with the strings stored in `xItem`, `yItem`, and `zItem`.

There is a particular string method that will help you here — in each case, make the call to the method equal to `newStory`, so each time it is called, `newStory` is made equal to itself, but with substitutions made. So each time the button is pressed, these placeholders are each replaced with a random silly string. As a further hint, the method in question only replaces the first instance of the substring it finds, so you might need to make one of the calls twice.

4. Inside the first `if` block, add another string replacement method call to replace the name 'Bob' found in the `newStory` string with the `name` variable. In this block we are saying "If a value has been entered into the `customName` text input, replace Bob in the story with that custom name."
5. Inside the second `if` block, we are checking to see if the `uk` radio button has been selected. If so, we want to convert the weight and temperature values in the story from pounds and Fahrenheit into stones and centigrade. What you need to do is as follows:
    1. Look up the formulae for converting pounds to stone, and Fahrenheit to centigrade.
    2. Inside the line that defines the `weight` variable, replace 300 with a calculation that converts 300 pounds into stones. Concatenate `' stone'` onto the end of the result of the overall `Math.round()` call.
    3. Inside the line that defines the `temperature` variable, replace 94 with a calculation that converts 94 Fahrenheit into centigrade. Concatenate `' centigrade'` onto the end of the result of the overall `Math.round()` call.
    4. Just under the two variable definitions, add two more string replacement lines that replace '94 fahrenheit' with the contents of the `temperature` variable, and '300 pounds' with the contents of the `weight` variable.
6. Finally, in the second-to-last line of the function, make the `textContent` property of the `story` variable (which references the paragraph) equal to `newStory`.

Hints and tips

- You don't need to edit the HTML in any way, except to apply the JavaScript to your HTML.
- If you are unsure whether the JavaScript is applied to your HTML properly, try removing everything else from the JavaScript file temporarily, adding in a simple bit of JavaScript that you know will create an obvious effect, then saving and refreshing. The following for example turns the background of the `<html>` element red — so the entire browser window should go red if the JavaScript is applied properly:

```
document.querySelector('html').style.backgroundColor = 'red';
```

- `Math.round()` is a built-in JavaScript method that simply rounds the result of a calculation to the nearest whole number.
- There are three instances of strings that need to be replaced. You may repeat the `replace()` method multiple times, or you can use regular expressions. For instance, `let text = 'I am the biggest lover, I love my love'; text.replace(/love/g,'like');` will replace all instances of 'love' to 'like'. Remember, Strings are immutable!

Assessment or further help

If you would like your work assessed, or are stuck and want to ask for help:

1. Put your work into an online shareable editor such as CodePen, jsFiddle, or Glitch.
2. Write a post asking for assessment and/or help at the MDN Discourse forum Learning category. Your post should include:
    - A descriptive title such as "Assessment wanted for Silly story generator".
    - Details of what you have already tried, and what you would like us to do, e.g. if you are stuck and need help, or want an assessment.
    - A link to the example you want assessed or need help with, in an online shareable editor (as mentioned in step 1 above). This is a good practice to get into — it's very hard to help someone with a coding problem if you can't see their code.
    - A link to the actual task or assessment page, so we can find the question you want help with.

Previous Overview: First steps

In this module