

JavaScript Tutorial MDN Web Docs

[About MDN](#) by [Mozilla Contributors](#) is licensed under [CC-BY-SA 2.5](#).

This is offline text version of Mozilla JavaScript Tutorial copied from MDN Web Docs, just use it for educational purpose only.

Version History

- Offline First Edition on 28/02/2018

Note:

Some text or code snippet may have missed while copying, because embedded Iframe doesn't copy.

Encourage contribution to fix the missing part in the doc so this can help others.

JavaScript (JS) is a lightweight interpreted or JIT-compiled programming language with [first-class functions](#). While it is most well-known as the scripting language for Web pages, [many non-browser environments](#) also use it, such as [Node.js](#), [Apache CouchDB](#) and [Adobe Acrobat](#). JavaScript is a [prototype-based](#), multi-paradigm, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles.

What is JavaScript?

JavaScript® (often shortened to **JS**) is a lightweight, interpreted, object-oriented language with [first-class functions](#), and is best known as the scripting language for Web pages, but it's [used in many non-browser environments](#) as well. It is a [prototype-based](#), multi-paradigm scripting language that is dynamic, and supports object-oriented, imperative, and functional programming styles.

JavaScript runs on the client side of the web, which can be used to design / program how the web pages behave on the occurrence of an event. JavaScript is an easy to learn and also powerful scripting language, widely used for controlling web page behaviour.

Contrary to popular misconception, **JavaScript is not "Interpreted Java"**. In a nutshell, JavaScript is a dynamic scripting language supporting [prototype based](#) object construction. The basic syntax is intentionally similar to both Java and C++ to reduce the number of new concepts required to learn the language. Language constructs, such as `if` statements, `for` and `while` loops, and `switch` and `try ... catch` blocks function the same as in these languages (or nearly so).

JavaScript can function as both a [procedural](#) and an [object oriented language](#). Objects are created programmatically in JavaScript, by attaching methods and properties to otherwise empty objects **at run time**, as opposed to the syntactic class definitions common in compiled languages like C++ and Java. Once an object has been constructed it can be used as a blueprint (or prototype) for creating similar objects.

JavaScript's dynamic capabilities include runtime object construction, variable parameter lists, function variables, dynamic script creation (via [`eval`](#)), object introspection (via `for ... in`), and source code recovery (JavaScript programs can decompile function bodies back into their source text).

For a more in depth discussion of JavaScript programming follow the [JavaScript resources](#) links below.

What JavaScript implementations are available?

The Mozilla project provides two JavaScript implementations. The first **ever** JavaScript was created by Brendan Eich at Netscape, and has since been updated to conform to ECMA-262

Edition 5 and later versions. This engine, code named [SpiderMonkey](#), is implemented in C/C++. The [Rhino](#) engine, created primarily by Norris Boyd (also at Netscape) is a JavaScript implementation written in Java. Like SpiderMonkey, Rhino is ECMA-262 Edition 5 compliant.

Several major runtime optimizations such as TraceMonkey (Firefox 3.5), JägerMonkey (Firefox 4) and IonMonkey were added to the SpiderMonkey JavaScript engine over time. Work is always ongoing to improve JavaScript execution performance.

Besides the above implementations, there are other popular JavaScript engines such as:-

- Google's [V8](#), which is used in the Google Chrome browser and recent versions of Opera browser. This is also the engine used by [Node.js](#).
- The [JavaScriptCore](#) (SquirrelFish/Nitro) used in some WebKit browsers such as Apple Safari.
- [Carakan](#) in old versions of Opera.
- The [Chakra](#) engine used in Internet Explorer (although the language it implements is formally called "JScript" in order to avoid trademark issues).

Each of Mozilla's JavaScript engines expose a public API which application developers can use to integrate JavaScript into their software. By far, the most common host environment for JavaScript is web browsers. Web browsers typically use the public API to create **host objects** responsible for reflecting the [DOM](#) into JavaScript.

Another common application for JavaScript is as a (Web) server side scripting language. A JavaScript web server would expose host objects representing a HTTP request and response objects, which could then be manipulated by a JavaScript program to dynamically generate web pages. [Node.js](#) is a popular example of this.

[JavaScript](#) is a programming language that allows you to implement complex things 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, or interactive maps, or animated 2D/3D graphics, or scrolling video jukeboxes, and so on — you can bet that JavaScript is probably involved.

Learning pathway

JavaScript is arguably more difficult to learn than related technologies such as [HTML](#) and [CSS](#). Before attempting to learn JavaScript, you are strongly advised to get familiar with at least these two technologies first, and perhaps others as well. Start by working through the following modules:

- [Getting started with the Web](#)
- [Introduction to HTML](#)
- [Introduction to CSS](#)

Having previous experience with other programming languages might also help.

For complete beginners

What is JavaScript?

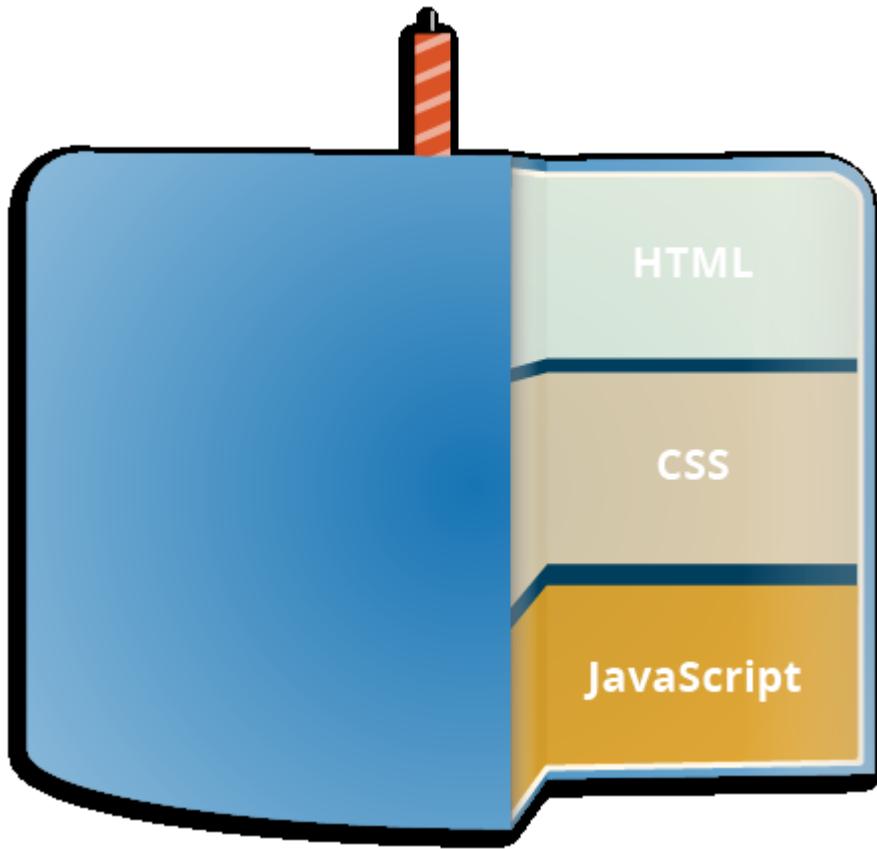
Welcome to the MDN beginner's JavaScript course! In this article we will look at JavaScript from a high level, answering questions such as "What is it?" and "What can you do with it?", and making sure you are comfortable with JavaScript's purpose.

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 things 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:

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

Player 1: Chris

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

```
p {  
  font-family: 'helvetica neue', helvetica, sans-serif;  
  letter-spacing: 1px;  
  text-transform: uppercase;  
  text-align: center;
```

```
border: 2px solid rgba(0,0,200,0.6);  
background: rgba(0,0,200,0.3);  
color: rgba(0,0,200,0.6);  
box-shadow: 1px 1px 2px rgba(0,0,200,0.4);  
border-radius: 10px;  
padding: 3px 10px;  
display: inline-block;  
cursor:pointer;  
}
```

PLAYER 1: CHRIS

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

```
var para = document.querySelector('p');  
  
para.addEventListener('click', updateName);  
  
function updateName() {  
  var 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?

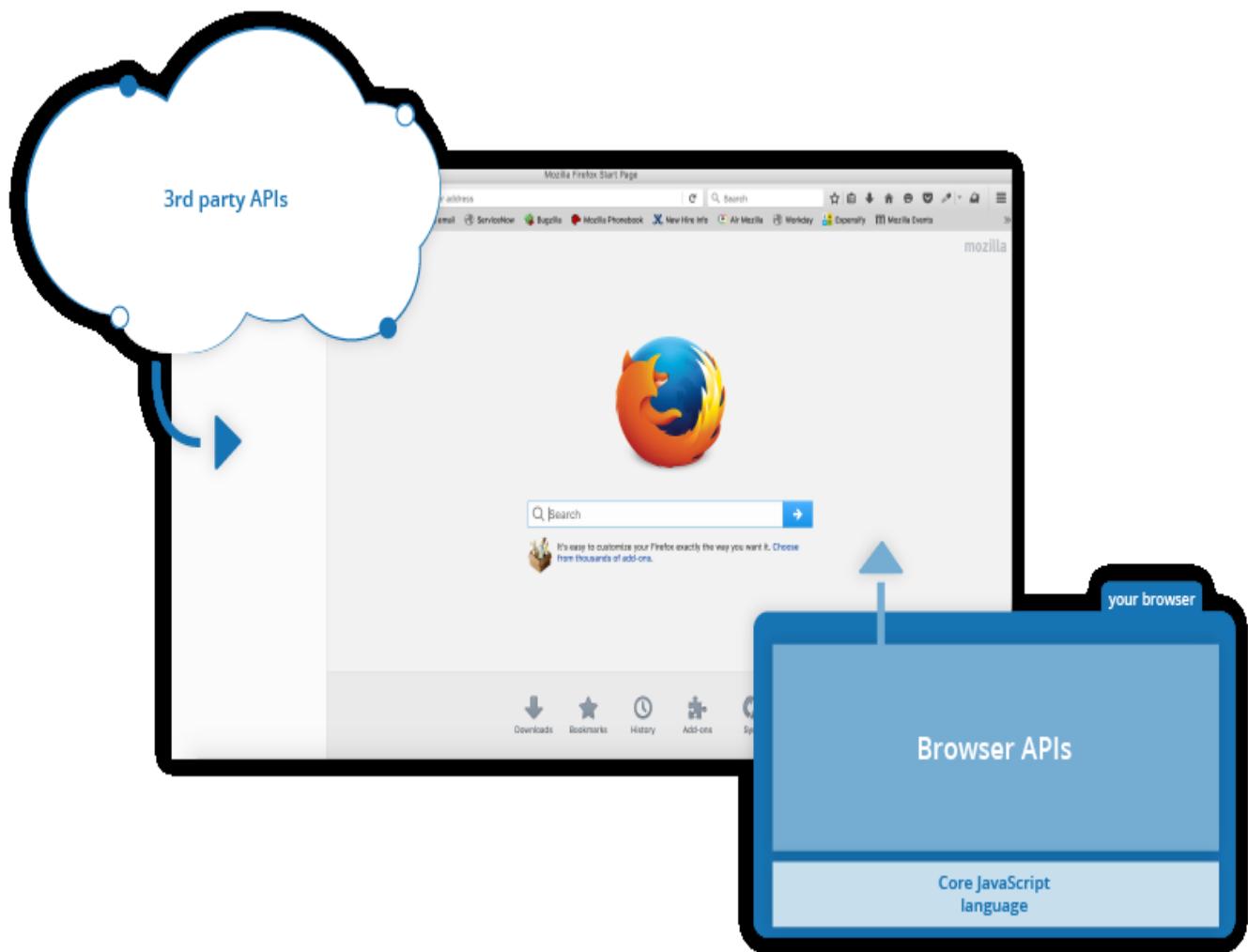
The core JavaScript language consists of some common programming features that allow you to do things like:

- 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 core 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. Everytime 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 API](#) allow you to do really interesting things with multimedia, such as play audio and video right in a web page, or grab video from your web camera and display it on someone else's computer (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](#) allows you to embed custom maps into your website, and other such functionality.

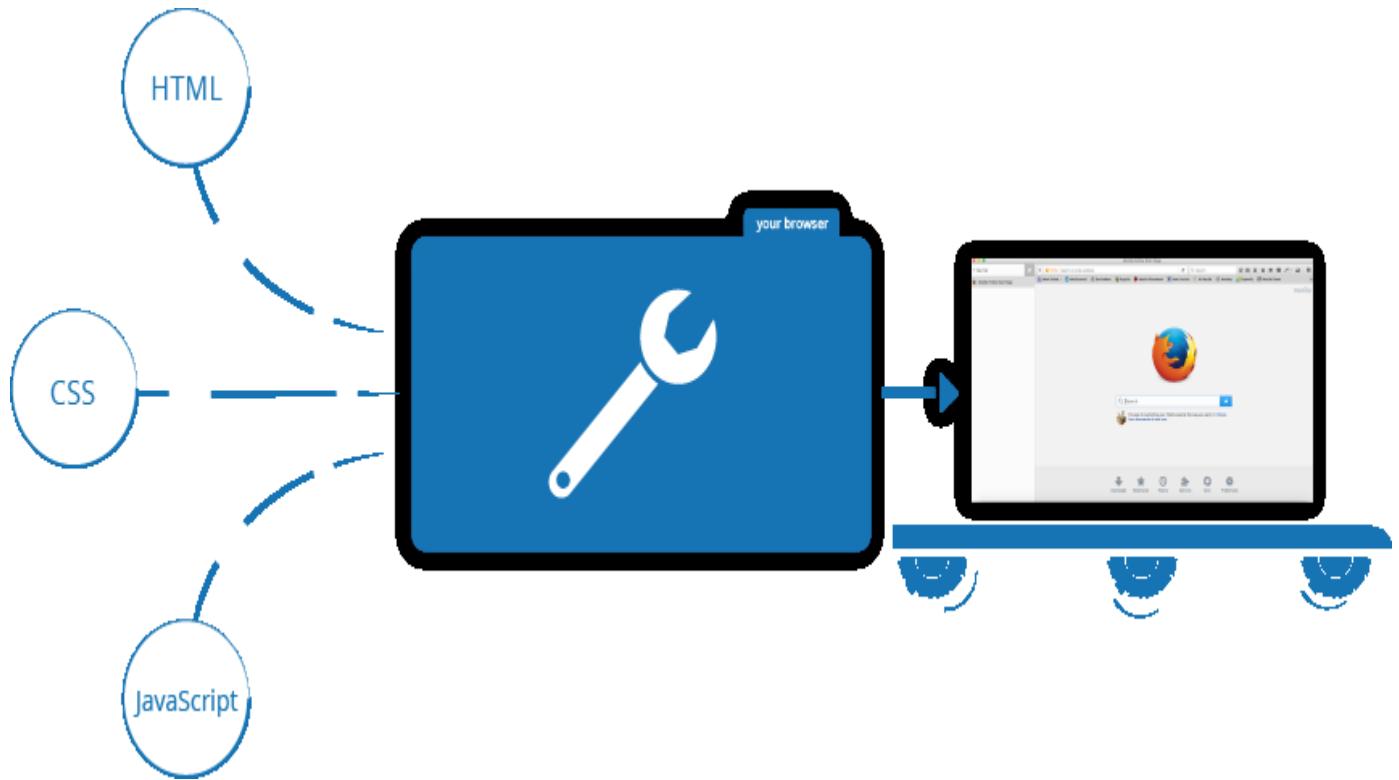
Note: These APIs are advanced, and we'll not be covering any of these in this module. You can find out much more about these in our [Client-side web APIs module](#).

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 start actually 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).



The JavaScript is executed by the browser's JavaScript engine, after the HTML and CSS have been assembled and put together into a web page. This ensures that the structure and style of the page are already in place by the time the JavaScript starts to run.

This is a good thing, as 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). If the JavaScript loaded and tried to run before the HTML and CSS were there to affect, then errors would occur.

Browser security

Each browser tab is 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:

```
var para = document.querySelector('p');

para.addEventListener('click', updateName);

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

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 block 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. JavaScript is an interpreted language — 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.

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.

Both approaches have different advantages, which we won't discuss at this point.

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 JavaScript 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, and 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 just before your closing `</body>` tag:

```
4.  <script>
5.
6.    // JavaScript goes here
7.
</script>
```

- • 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:

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

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

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

4.

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 `</body>` 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. Next, copy all of the script out of your current `<script>` element and paste it into the `.js` file. Save that file.
3. Now replace your current `<script>` element with the following:

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

- 3.
4. Save and refresh your browser, and you should see the same thing! It works just the same, but now we've got the 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() {
  var 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.

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 wanted 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:

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

for (var 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?

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 append it to the bottom of the HTML
body.

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

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

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

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

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

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.

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.

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 language — 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:

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

var guesses = document.querySelector('.guesses');
var lastResult = document.querySelector('.lastResult');
```

```

var lowOrHi = document.querySelector('.lowOrHi');

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

var guessCount = 1;
var resetButton;

```

This section of the code sets up the variables 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 `var` followed by a name for your variable. You can then assign a value to your variable 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 next three variables 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:
 - `<p class="guesses"></p>`
 - `<p class="lastResult"></p>`
 - `<p class="lowOrHi"></p>`
- • The next two variables 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"
id="guessField" class="guessField">
<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 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 maths, 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. If you don't have easy access to the browser developer tools, you can always use the simple built in console seen below:

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:

```
var name = 'Bingo';
name;
var hello = ' says hello!';
hello;
var 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 |
| != | Non-equality (is it not the same?) | 'Chris' != 'Ch' + 'ris' |
| < | Less than | 10 < 6 |
| > | Greater than | 10 > 20 |

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() {
  var 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()` method, 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):

```
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 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.appendChild(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;

    var resetParas = document.querySelectorAll('.resultParas p');
    for (var 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.
- Clears all the information paragraphs.
- 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 (var 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:

```
var resetParas = document.querySelectorAll('.resultParas p');
for (var 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 `var 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` variable that stores a reference to the text input form field in our HTML — the following line can be found amongst our variable declarations near the top:

```
var 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 will now have 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` variable contains a reference to a `<p>` element, and `guessCount` 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 in `guessField` and the console will show 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!
- Now try typing in `guesses` and pressing return. The console will show you that the variable contains a `<p>` element.
- Now try entering the following line:

```
guesses.value
```

- The browser will return `undefined`, because paragraphs don't have the `value` property.
- To change the text inside a paragraph, you need the `textContent` property instead. Try this:

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

- 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';
```

8. 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](#).

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.

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 simple 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 a resulting 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 favourite 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!

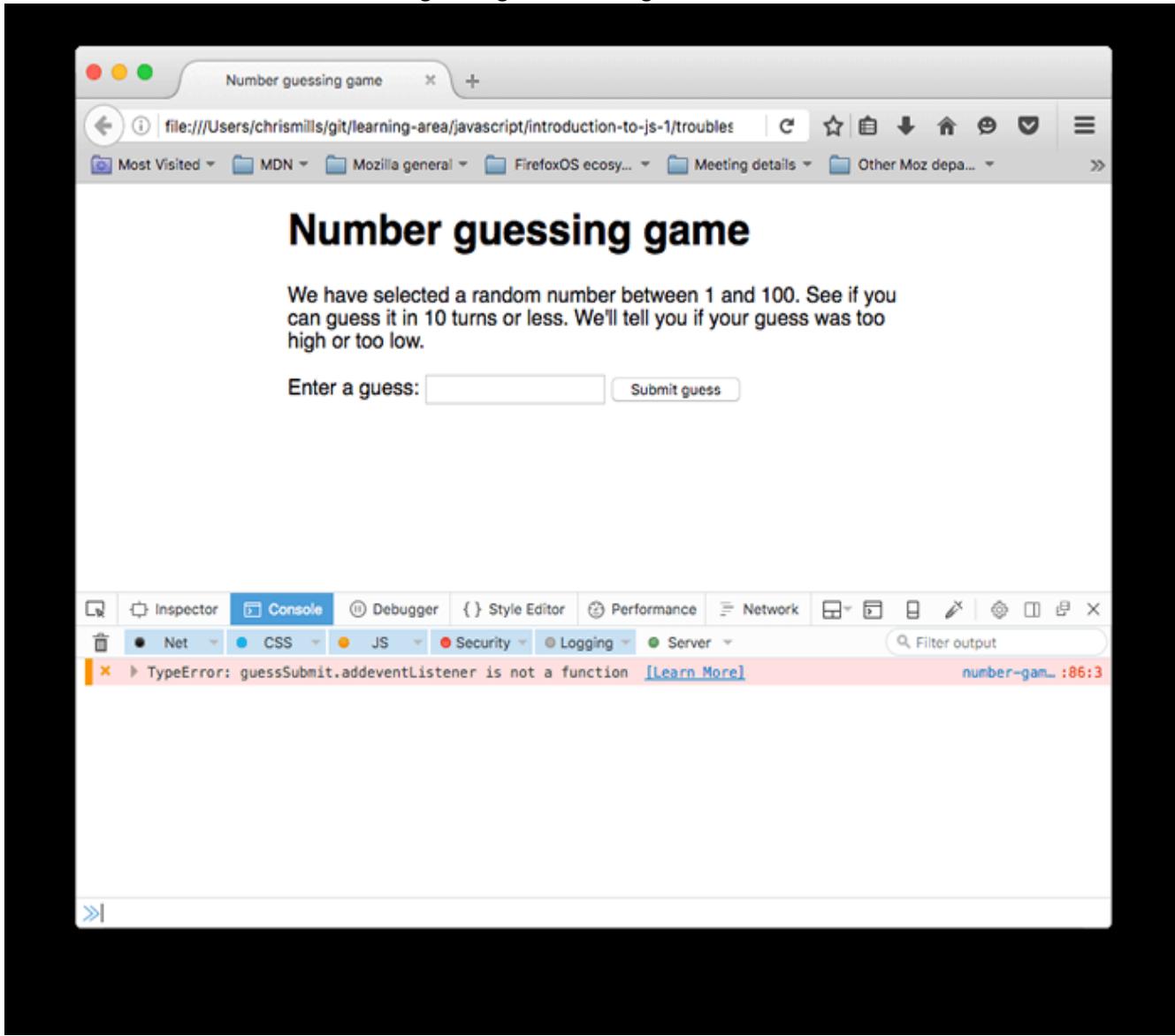
Note: You might well have your own version of the game example that doesn't work, which you might want to fix! We'd still like you to work through the article with our version, so that you can learn the techniques we are teaching here. Then you can go back and try to fix your example.

At this point, let's consult the developer console to see if we can see 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 huge amounts of detail.
 - The name of the JavaScript file, which links through to the Debugger tab of the devtools. 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);
```

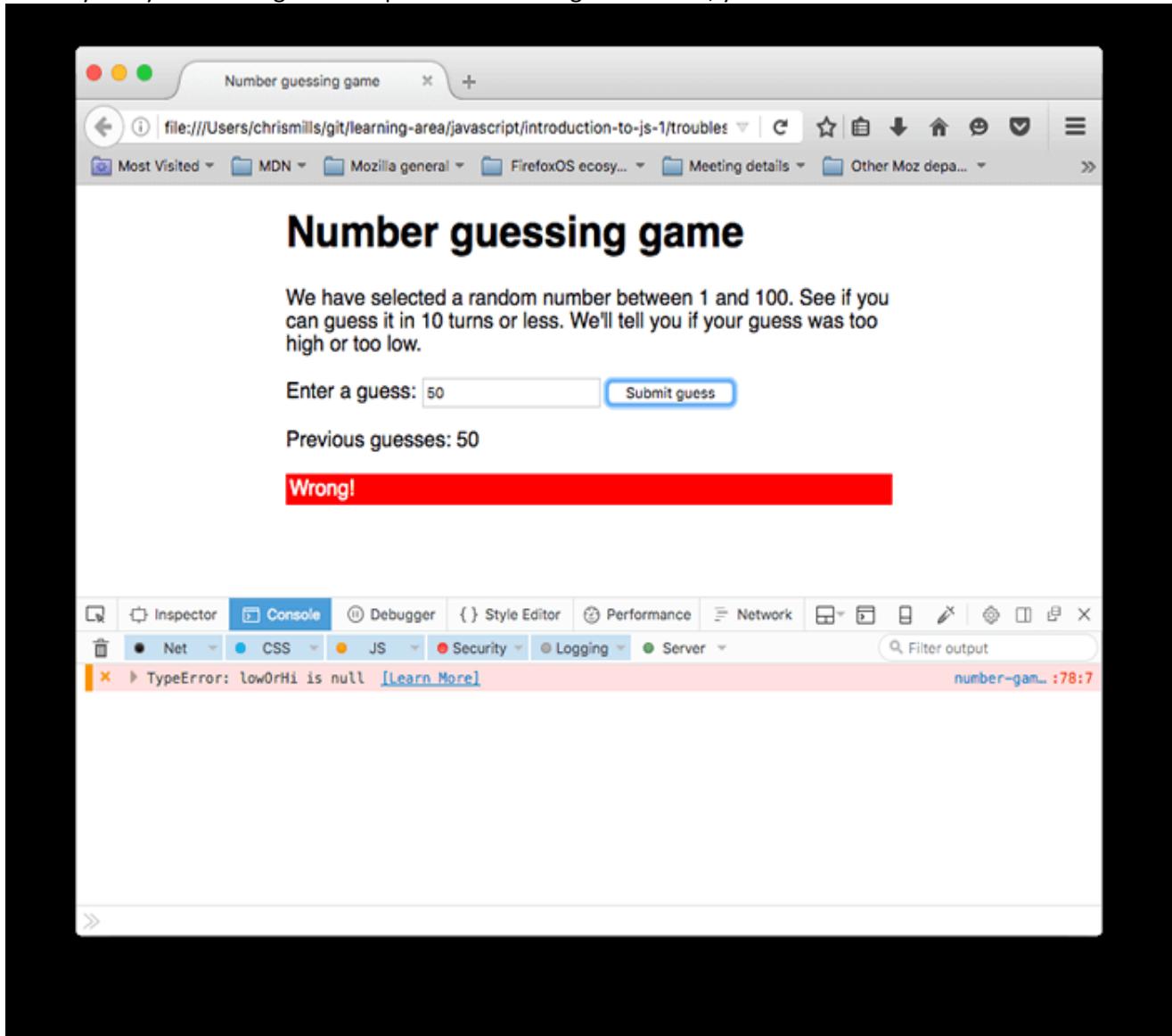
- 3.
4. The error message says "guessSubmit.addeventListener is not a function", so we've probably spelled something wrong. If you are not sure of the correct spelling of a piece of syntax, it is often good to look up the feature on MDN. The best way to do this currently is to search for "mdn *name-of-feature*" on your favourite 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.

- Now if you try to enter a guess and press the Submit guess button, you'll see ... another error!



- 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.

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

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

- • This line is trying to set the `textContent` property of the `lowOrHi` variable 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:

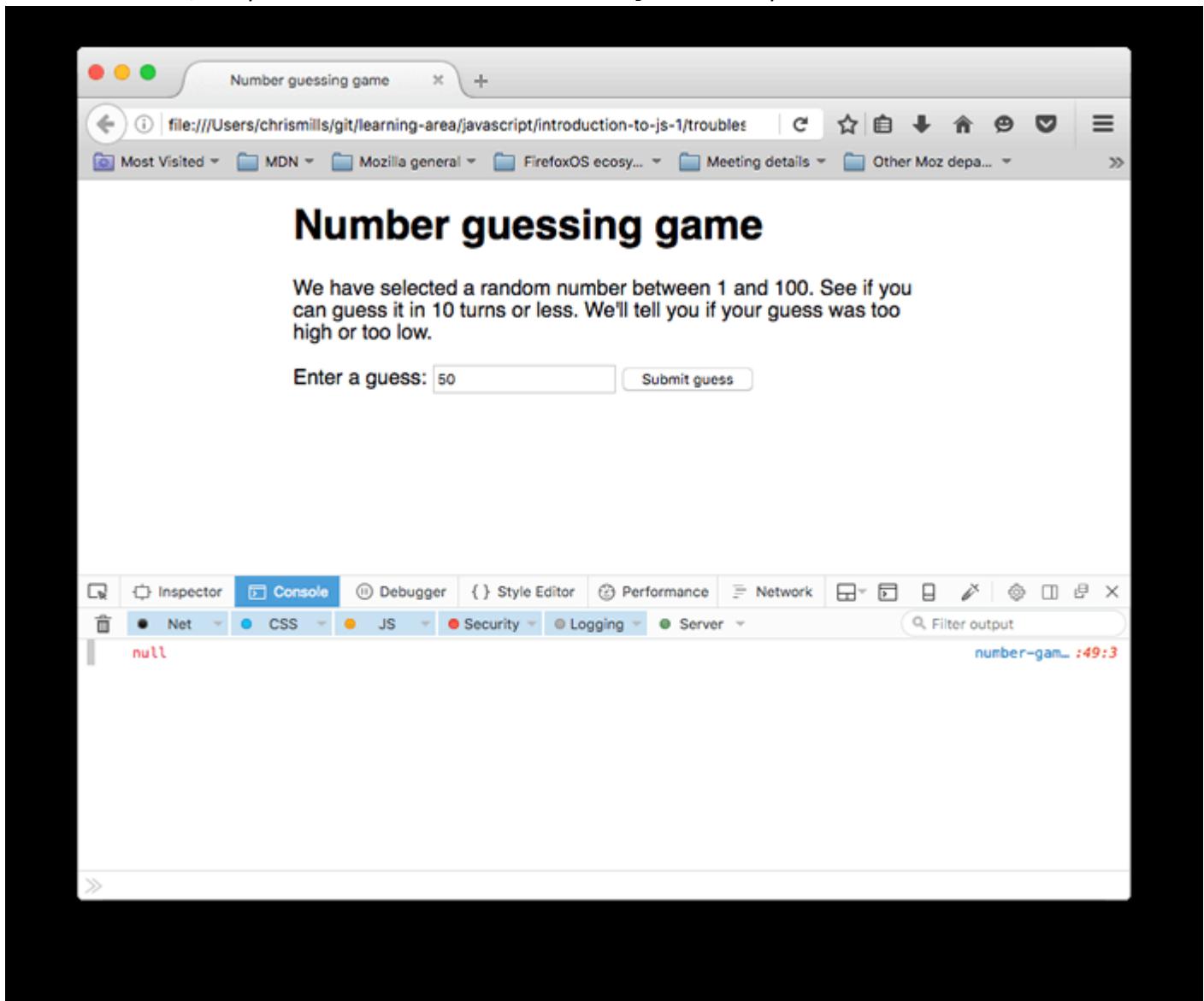
```
var lowOrHi = document.querySelector('lowOrHi');
```

- • 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:

```
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.

- Save and refresh, 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.

- 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:

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

8.

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 lives.
2. At the 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 0 or 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:

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

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

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

• • 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:

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

- 2.
3. 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.

```
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:

```
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 semi colon 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()`:

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

to

```
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 off 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

```
function checkGuess() {
```

to

```
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 missed 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 earlier on in your learning journey.

See also

- There are many other types of error 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 on the [Learning Area Discourse thread](#), or in the [#mdn](#) IRC channel on [Mozilla IRC](#). Tell us what your error is, and we'll try to help you. A listing of your code would be useful as well.

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 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).

However, we have also provided a simple JavaScript console embedded in the page below for you to enter this code into, in case you are not using a browser with a JavaScript console easily available, or find an in-page console more comfortable.

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>
var button = document.querySelector('button');

button.onclick = function() {
  var name = prompt('What is your name?');
  alert('Hello ' + name + ', nice to see you!');
}
```

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:

```
var 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!');
}

// ... and so on ...
```

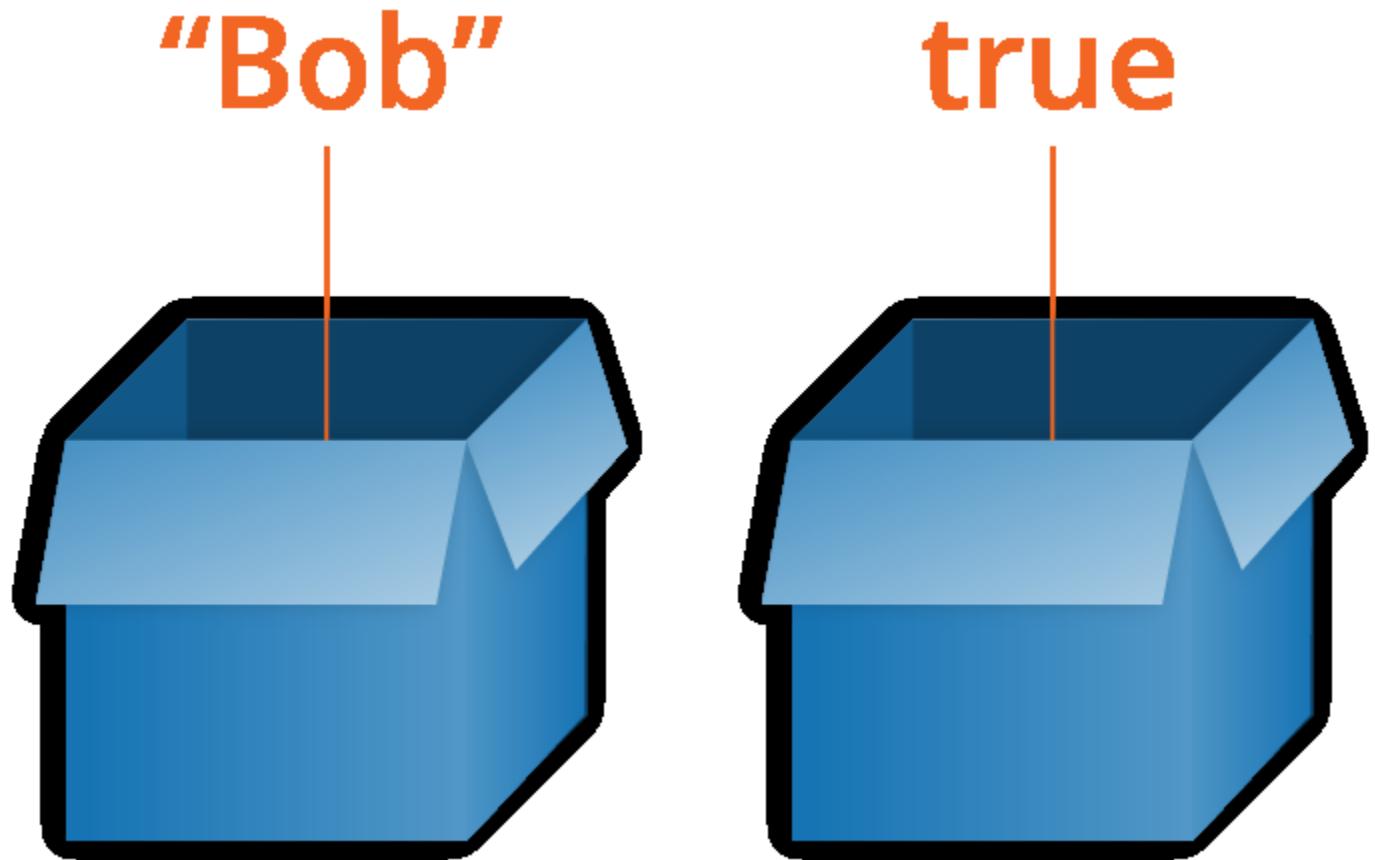
You may not fully understand the syntax we are using (yet!), but you should be able to get the idea — if we didn't have variables available, we'd have to implement a giant code block that checked what the entered name was, and then display the appropriate message for that name.

This is obviously really inefficient (the code is a lot bigger, even for only four 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 that 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` followed by the name you want to call your variable:

```
var myName;  
var myAge;
```

Here we're creating two variables called `myName` and `myAge`. Try typing these lines in now in your web browser's console, or in the below console (You can [open this console](#) in a separate tab or window if you'd prefer that). 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 either 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 value defined with a variable that doesn't exist at all — they are very different things.

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:

```
var myName = 'Chris';
```

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

Note: If you write a multiline JavaScript program that declares and initializes a variable, you can actually declare it after you initialize it and it will still work. This is because variable declarations are generally done first before the rest of the code is executed. This is called **hoisting** — read [var hoisting](#) for more detail on the subject.

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 will cause 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 to `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 will 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
```

Error-prone name examples:

```
var
Document
```

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:

```
var 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 will try to interpret it as another variable name.

```
var 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:

```
var iAmAlive = true;
```

Whereas in reality it would be used more like this:

```
var test = 6 < 3;
```

This is using the "less than" operator (`<`) to test whether 6 is less than 3. As you might expect, it will return `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:

```
var myNameArray = ['Chris', 'Bob', 'Jim'];
var 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 car park and contains information about its width and length, 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:

```
var 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.

Loose typing

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

For example, if you declare a variable and give it a value encapsulated in quotes, the browser will treat the variable as a string:

```
var myString = 'Hello';
```

It will still be a string, even if it contains numbers, so be careful:

```
var 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 pass into 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.

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.

Basic math in JavaScript — numbers and operators

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, etc. 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, 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.

It's all numbers to me

Let's have a quick play with some numbers to reacquaint ourselves with the basic syntax we need. Enter the commands listed below into your [developer tools JavaScript console](#), or use the simple built in console seen below if you'd prefer.

[Open in new window](#)

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:

```
2. var myInt = 5;
3. var myFloat = 6.667;
4. myInt;
myFloat;
```

- Number values are typed in without quote marks — try declaring and initializing a couple more variables containing numbers before you move on.

- 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!

Arithmetic operators

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

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

Note: You'll sometimes see numbers involved in sums referred to as [operands](#).

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](#), or use the simple built in console seen earlier if you'd prefer, to familiarize yourself with the syntax.

- First try entering some simple examples of your own, such as
2. 10 + 7
3. 9 * 8
- 60 % 3

- 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:

```
var num1 = 10;  
var num2 = 50;  
9 * num1;  
num2 / num1;
```

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

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

3.

Some of this last set of sums might not give you quite the result you were expecting; the below section 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 will be applied before others when calculating the result of a sum (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 sum 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/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: They 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:

```
var 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

```
var 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:

```

var x = 3; // x contains the value 3
var 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 |
|-----------------|---------------------------|--|---|-------------------------------------|
| <code>+=</code> | Addition assignment | Adds the value on the right to the variable value on the left, then returns the new variable value | <code>x = 3; x = 3; x += x = x + 4; 4;</code> | |
| <code>-=</code> | Subtraction assignment | Subtracts the value on the right from the variable value on the left, and returns the new variable value | <code>x = 6; x = 6; x -= x = x - 3; 3;</code> | |
| <code>*=</code> | Multiplication assignment | Multiples the variable value on the left by the value on the right, and returns the new variable value | <code>x = 2; x = 2; x *= x = x * 3; 3;</code> | |
| <code>/=</code> | Division assignment | Divides the variable value on the left by the value on the right, and returns the new variable value | <code>x = 10; x /= x / 5;</code> | <code>x = 10; x = x / 5;</code> |

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:

```

var x = 3; // x contains the value 3
var 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.

[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 |
|--------------------|--------------------------|---|--------------------------|
| <code>==</code> | Strict equality | Tests whether the left and right values are identical to one another | <code>5 === 2 + 4</code> |
| <code>!=</code> | Strict-non-equality | Tests whether the left and right values not identical to one another | <code>5 != 2 + 3</code> |
| <code><</code> | Less than | Tests whether the left value is smaller than the right one. | <code>10 < 6</code> |
| <code>></code> | Greater than | Tests whether the left value is greater than the right one. | <code>10 > 20</code> |
| <code><=</code> | Less than or equal to | Tests whether the left value is smaller than or equal to the right one. | <code>3 <= 2</code> |
| <code>>=</code> | Greater than or equal to | Tests whether the left value is greater than or equal to the right one. | <code>5 >= 4</code> |

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>
var btn = document.querySelector('button');
var 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.';
  }
}
```

[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.

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.

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 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, 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 a console to familiarize ourselves. We've provided one below (you can also [open this console](#) in a separate tab or window, or use the [browser developer console](#) if you'd prefer).

Creating a string

1. To start with, enter the following lines:

```
2. var 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.

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

```
var badString = This is a test;  
var badString = 'This is a test;  
var 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, 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.

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

```
var badString = string;  
badString;
```

3. `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:

```
2. var sgl = 'Single quotes.';  
3. var dbl = "Double quotes";  
4. sgl;  
   dbl;
```

- 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 the different quotes on the same string! The following will return an error:

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

- 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:

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

- 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:

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

4. 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:

```
var 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.

```
2. var one = 'Hello, ';
3. var two = 'how are you?';
4. var joined = one + two;
joined;
```

- The result of this is a variable called `joined`, which contains the value "Hello, how are you?".
- In the last instance, we just joined two strings together, but you can do as many as you like, as long as you include a + between each one. Try this:

```
var multiple = one + one + one + one + two;
multiple;
```

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

```
var response = one + 'I am fine - ' + two;
response;
```

3.

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>
var button = document.querySelector('button');
```

```
button.onclick = function() {
  var name = prompt('What is your name?');
  alert('Hello ' + name + ', nice to see you!');
}
```

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 an [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 throw 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 together.
- 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):

```
var myDate = '19' + '67';
typeof myDate;
```

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

- The [Number](#) object will convert anything passed to it into a number, if it can. Try the following:
 - ```
var myString = '123';
```
  - ```
var myNum = Number(myString);
```
 - ```
typeof myNum;
```
- • On the other hand, every number has a method called [toString\(\)](#) that will convert it to the equivalent string. Try this:

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

3.

○

These constructs can be really useful in some situations. For example, if a user enters a number into a form text field, it will be 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 61](#).

## 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.

### Useful string methods

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

We've said it before, and we'll say it again — *everything* is an object in JavaScript. When you create a string, for example by using

```
var 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 a fresh console. We've provided one below (you can also [open this console](#) in a separate tab or window, or use the [browser developer console](#) if you'd prefer).

### Finding the length of a string

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

```
var 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];
```

Computers count from 0, not 1! 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, hence us needing `length-1`. You could use this to, for example, find the first letter of a series of strings and order them alphabetically.

## Finding a substring inside a string and extracting it

1. 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. Try this:

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

1. 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
}
```

- 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. You could also say that the second parameter is equal to the length of the string being returned.

- 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);
```

4. 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:

```
var 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');
```

Note that to actually get the updated value reflected in 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.

### 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.

## 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:

```
MAN675847583748sjt567654;Manchester Piccadilly
```

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

```
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`.

## 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.

## 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.

**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 be bad enough, 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 a JavaScript console. We've provided one below (you can also [open this console](#) in a separate tab or window, or use the [browser developer console](#) if you prefer).

### Creating an array

Arrays are constructed of square brackets, which contain a list of items separated by commas.

1. Let's say we wanted to store a shopping list in an array — we'd do something like the following.  
Enter the following lines into your console:
  2. var shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];  
shopping;
  - In this case, each item in the array is a string, but bear in mind that you can store any item in an array — string, number, object, another variable, even another array. You can also mix and match item types — they don't all have to be numbers, strings, etc. Try these:

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

- 2.
3. Try creating a couple of arrays of your own, before you move on.

### 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:
  2. 

```
shopping[0];
// returns "bread"
```
- • 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", "noodles"
]
```

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

- 3.
4. Try further playing with, and making some more modifications to, your array examples before moving on.

## 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:

```
sequence.length;
// should return 7
```

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:

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

You'll learn about loops properly in a future 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 will work for an array of any length, but in this case it will stop looping at item number 7 (this is good, as the last item — which we want the loop to cover — is 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:

```
var myData = 'Manchester,London,Liverpool,Birmingham,Leeds,Carlisle';
```

- Now let's split it at each comma:

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

- 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
```

- You can also go the opposite way using the [join\(\)](#) method. Try the following:

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

- 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 (try running Step 4 with a different character than a comma).

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

5.

## 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:

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

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:

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

- • 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:

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

- • Removing the last item from the array is as simple as running [pop\(\)](#) on it. Try this:

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

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

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

4.

[unshift\(\)](#) and [shift\(\)](#) work in exactly the same way as [push\(\)](#) and [pop\(\)](#), except that they work on the beginning of the array, not the end.

1. First [unshift\(\)](#) — try the following commands:

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

- • Now [shift\(\);](#) try these!

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

2.

## 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 stop immediately, 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.

## 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.

## 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.

## JavaScript building blocks

In this module, we continue our coverage of all JavaScript's key fundamental features, turning our attention to commonly-encountered types of code block such as conditional statements, loops, functions, and events. You've seen this stuff already in the course, but only in passing — here we'll discuss it all explicitly.

In this module, we continue our coverage of all JavaScript's key fundamental features, turning our attention to commonly-encountered types of code block such as conditional statements, loops, functions, and events. You've seen this stuff already in the course, but only in passing — here we'll discuss it all explicitly.

## Prerequisites

Before starting this module, you should have some familiarity with the basics of [HTML](#) and [CSS](#), and you should have also worked through our previous module, [JavaScript first steps](#).

**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 [Thimble](#).

## Making decisions in your code — conditionals

In any programming language, code needs to make decisions and carry out actions accordingly depending on different inputs. For example, in a game, if the player's number of lives is 0, then it's game over. In a weather app, if it is being looked at in the morning, show a sunrise graphic; show stars and a moon if it is nighttime. In this article we'll explore how so-called conditional statements work in JavaScript.

**Prerequisites:** Basic computer literacy, a basic understanding of HTML and CSS, [JavaScript first steps](#).

**Objective:** To understand how to use conditional structures in JavaScript.

### You can have it on one condition..!

Human beings (and other animals) make decisions all the time that affect their lives, from small ("should I eat one cookie or two?") to large ("should I stay in my home country and work on my father's farm, or should I move to America and study astrophysics?")

Conditional statements allow us to represent such decision making in JavaScript, from the choice that must be made (e.g. "one cookie or two"), to the resulting outcome of those choices (perhaps the outcome of "ate one cookie" might be "still felt hungry", and the outcome of "ate two cookies" might be "felt full up, but mom scolded me for eating all the cookies".)



**if ... else statements**

Let's look at by far the most common type of conditional statement you'll use in JavaScript — the humble [if ... else statement](#).

## Basic if ... else syntax

Basic if...else syntax looks like so in [pseudocode](#):

```
if (condition) {
 code to run if condition is true
} else {
 run some other code instead
}
```

Here we've got:

1. The keyword `if` followed by some parentheses.
2. A condition to test, placed inside the parentheses (typically "is this value bigger than this other value", or "does this value exist"). This condition will make use of the [comparison operators](#) we discussed in the last module, and will return `true` or `false`.
3. A set of curly braces, inside which we have some code — this can be any code we like, and will only be run if the condition returns `true`.
4. The keyword `else`.
5. Another set of curly braces, inside which we have some more code — this can be any code we like, and will only be run if the condition is not `true`.

This code is pretty human-readable — it is saying "**if** the **condition** returns `true`, run code A, **else** run code B"

You should note that you don't have to include the `else` and the second curly brace block — the following is also perfectly legal code:

```
if (condition) {
 code to run if condition is true
}

run some other code
```

However, you need to be careful here — in this case, the second block of code is not controlled by the conditional statement, so it will **always** run, regardless of whether the condition returns `true` or `false`. This is not necessarily a bad thing, but it might not be what you want — often you will want to run one block of code *or* the other, not both.

As a final point, you may sometimes see if...else statements written without the curly braces, in the following shorthand style:

```
if (condition) code to run if condition is true
else run some other code instead
```

This is perfectly valid code, but using it is not recommended — it is much easier to read the code and work out what is going on if you use the curly braces to delimit the blocks of code, and use multiple lines and indentation.

## A real example

To understand this syntax better, let's consider a real example. Imagine a child being asked for help with a chore by their mother or father. The parent might say "Hey sweetheart, if you help me by going and doing the shopping, I'll give you some extra allowance so you can afford that toy you wanted." In JavaScript, we could represent this like so:

```
var shoppingDone = false;

if (shoppingDone === true) {
 var childAllowance = 10;
} else {
 var childAllowance = 5;
}
```

This code as shown will always result in the `shoppingDone` variable returning `false`, meaning disappointment for our poor child. It'd be up to us to provide a mechanism for the parent to set the `shoppingDone` variable to `true` if the child did the shopping.

**Note:** You can see a more [complete version of this example on GitHub](#) (also see it [running live](#).)

## else if

The last example provided us with two choices, or outcomes — but what if we want more than two?

There is a way to chain on extra choices/outcomes to your `if...else` — using `else if`. Each extra choice requires an additional block to put in between `if() { ... }` and `else { ... }` — check out the following more involved example, which could be part of a simple weather forecast application:

```
<label for="weather">Select the weather type today: </label>
<select id="weather">
 <option value="">--Make a choice--</option>
 <option value="sunny">Sunny</option>
 <option value="rainy">Rainy</option>
 <option value="snowing">Snowing</option>
 <option value="overcast">Overcast</option>
</select>

<p></p>
var select = document.querySelector('select');
var para = document.querySelector('p');

select.addEventListener('change', setWeather);
```

```

function setWeather() {
 var choice = select.value;

 if (choice === 'sunny') {
 para.textContent = 'It is nice and sunny outside today. Wear shorts! Go to the beach, or the park, and get an ice cream.';
 } else if (choice === 'rainy') {
 para.textContent = 'Rain is falling outside; take a rain coat and a brolly, and don\'t stay out for too long.';
 } else if (choice === 'snowing') {
 para.textContent = 'The snow is coming down – it is freezing! Best to stay in with a cup of hot chocolate, or go build a snowman.';
 } else if (choice === 'overcast') {
 para.textContent = 'It isn\'t raining, but the sky is grey and gloomy; it could turn any minute, so take a rain coat just in case.';
 } else {
 para.textContent = '';
 }
}

```

1. Here we've got an HTML `<select>` element allowing us to make different weather choices, and a simple paragraph.
2. In the JavaScript, we are storing a reference to both the `<select>` and `<p>` elements, and adding an event listener to the `<select>` element so that when its value is changed, the `setWeather()` function is run.
3. When this function is run, we first set a variable called `choice` to the current value selected in the `<select>` element. We then use a conditional statement to show different text inside the paragraph depending on what the value of `choice` is. Notice how all the conditions are tested in `else if() { ... }` blocks, except for the first one, which is tested in an `if() { ... }` block.
4. The very last choice, inside the `else { ... }` block, is basically a "last resort" option — the code inside it will be run if none of the conditions are `true`. In this case, it serves to empty the text out of the paragraph if nothing is selected, for example if a user decides to re-select the "--Make a choice--" placeholder option shown at the beginning.

**Note:** You can also [find this example on GitHub](#) ([see it running live](#) on there also.)

### A note on comparison operators

Comparison operators are used to test the conditions inside our conditional statements. We first looked at comparison operators back in our [Basic math in JavaScript — numbers and operators](#) article. Our choices are:

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

**Note:** Review the material at the previous link if you want to refresh your memories on these.

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

```
var cheese = 'Cheddar';

if (cheese) {
 console.log('Yay! Cheese available for making cheese on toast.');
} else {
 console.log('No cheese on toast for you today.');
}
```

And, returning to our previous example about the child doing a chore for their parent, you could write it like this:

```
var shoppingDone = false;

if (shoppingDone) { // don't need to explicitly specify '===' true'
 var childsAllowance = 10;
} else {
 var childsAllowance = 5;
}
```

### Nesting if ... else

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

```
if (choice === 'sunny') {
 if (temperature < 86) {
 para.textContent = 'It is ' + temperature + ' degrees outside – nice and
 sunny. Let\'s go out to the beach, or the park, and get an ice cream.';
 } else if (temperature >= 86) {
 para.textContent = 'It is ' + temperature + ' degrees outside – REALLY
 HOT! If you want to go outside, make sure to put some suncream on.';
 }
}
```

Even though the code all works together, each `if...else` statement works completely independently of the other one.

### Logical operators: AND, OR and NOT

If you want to test multiple conditions without writing nested `if...else` statements, [logical operators](#) can help you. When used in conditions, the first two do the following:

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

To give you an AND example, the previous example snippet can be rewritten to this:

```
if (choice === 'sunny' && temperature < 86) {
 para.textContent = 'It is ' + temperature + ' degrees outside - nice and
 sunny. Let\'s go out to the beach, or the park, and get an ice cream.';
} else if (choice === 'sunny' && temperature >= 86) {
 para.textContent = 'It is ' + temperature + ' degrees outside - REALLY HOT!
 If you want to go outside, make sure to put some suncream on.';
}
```

So for example, the first code block will only be run if `choice === 'sunny'` *and* `temperature < 86` return `true`.

Let's look at a quick OR example:

```
if (iceCreamVanOutside || houseStatus === 'on fire') {
 console.log('You should leave the house quickly.');
} else {
 console.log('Probably should just stay in then.');
}
```

The last type of logical operator, NOT, expressed by the `!` operator, can be used to negate an expression. Let's combine it with OR in the above example:

```
if (!(iceCreamVanOutside || houseStatus === 'on fire')) {
 console.log('Probably should just stay in then.');
} else {
 console.log('You should leave the house quickly.');
}
```

In this snippet, if the OR statement returns `true`, the NOT operator will negate it so that the overall expression returns `false`.

You can combine as many logical statements together as you want, in whatever structure. The following example executes the code inside only if both OR statements return true, meaning that the overall AND statement will return true:

```
if ((x === 5 || y > 3 || z <= 10) && (loggedIn || userName === 'Steve')) {
 // run the code
}
```

A common mistake when using the logical OR operator in conditional statements is to try to state the variable whose value you are checking once, and then give a list of values it could be to return true, separated by `||` (OR) operators. For example:

```
if (x === 5 || 7 || 10 || 20) {
 // run my code
}
```

In this case the condition inside `if(...)` will always evaluate to true since 7 (or any other non-zero value) always evaluates to true. This condition is actually saying "if x equals 5, or 7 is true — which it always is". This is logically not what we want! To make this work you've got to specify a complete test either side of each OR operator:

```
if (x === 5 || x === 7 || x === 10 || x === 20) {
 // run my code
}
```

## switch statements

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

[switch statements](#) are your friend here — they take a single expression/value as an input, and then look through a number of choices until they find one that matches that value, executing the corresponding code that goes along with it. Here's some more pseudocode, to give you an idea:

```
switch (expression) {
 case choice1:
 run this code
 break;

 case choice2:
 run this code instead
 break;

 // include as many cases as you like

 default:
 actually, just run this code
}
```

Here we've got:

1. The keyword `switch`, followed by a set of parentheses.
2. An expression or value inside the parentheses.
3. The keyword `case`, followed by a choice that the expression/value could be, followed by a colon.
4. Some code to run if the choice matches the expression.

5. A `break` statement, followed by a semi-colon. If the previous choice matches the expression/value, the browser stops executing the code block here, and moves on to any code that appears below the switch statement.
6. As many other cases (bullets 3–5) as you like.
7. The keyword `default`, followed by exactly the same code pattern as one of the cases (bullets 3–5), except that `default` does not have a choice after it, and you don't need to `break` statement as there is nothing to run after this in the block anyway. This is the default option that runs if none of the choices match.

**Note:** You don't have to include the `default` section — you can safely omit it if there is no chance that the expression could end up equaling an unknown value. If there is a chance of this however, you need to include it to handle unknown cases.

## A switch example

Let's have a look at a real example — we'll rewrite our weather forecast application to use a `switch` statement instead:

```
<label for="weather">Select the weather type today: </label>
<select id="weather">
 <option value="">--Make a choice--</option>
 <option value="sunny">Sunny</option>
 <option value="rainy">Rainy</option>
 <option value="snowing">Snowing</option>
 <option value="overcast">Overcast</option>
</select>

<p></p>
var select = document.querySelector('select');
var para = document.querySelector('p');

select.addEventListener('change', setWeather);

function setWeather() {
 var choice = select.value;

 switch (choice) {
 case 'sunny':
 para.textContent = 'It is nice and sunny outside today. Wear shorts! Go to the beach, or the park, and get an ice cream.';
 break;
 case 'rainy':
 para.textContent = 'Rain is falling outside; take a rain coat and a brolly, and don\'t stay out for too long.';
 break;
 case 'snowing':
 para.textContent = 'The snow is coming down – it is freezing! Best to stay in with a cup of hot chocolate, or go build a snowman.';
 break;
 case 'overcast':
 para.textContent = 'It isn\'t raining, but the sky is grey and gloomy; it could turn any minute, so take a rain coat just in case.';
 }
}
```

```

 break;
 default:
 para.textContent = '';
 }
}

```

**Note:** You can also [find this example on GitHub](#) (see it [running live](#) on there also.)

## Ternary operator

There is one final bit of syntax we want to introduce you to, before we get you to play with some examples. The [ternary or conditional operator](#) is a small bit of syntax that tests a condition and returns one value/expression if it is `true`, and another if it is `false` — this can be useful in some situations, and can take up a lot less code than an `if...else` block if you simply have two choices that are chosen between via a `true/false` condition. The pseudocode looks like this:

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

So let's look at a simple example:

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

Here we have a variable called `isBirthday` — if this is `true`, we give our guest a happy birthday message; if not, we give her the standard daily greeting.

### Ternary operator example

You don't just have to set variable values with the ternary operator; you can also run functions, or lines of code — anything you like. The following live example shows a simple theme chooser where the styling for the site is applied using a ternary operator.

```
<label for="theme">Select theme: </label>
<select id="theme">
 <option value="white">White</option>
 <option value="black">Black</option>
</select>

<h1>This is my website</h1>
var select = document.querySelector('select');
var html = document.querySelector('html');
document.body.style.padding = '10px';

function update(bgColor, textColor) {
 html.style.backgroundColor = bgColor;
 html.style.color = textColor;
}

select.onchange = function() {
```

```
(select.value === 'black') ? update('black','white') :
update('white','black');
}
```

Here we've got a `<select>` element to choose a theme (black or white), plus a simple `<h1>` to display a website title. We also have a function called `update()`, which takes two colors as parameters (inputs). The website's background color is set to the first provided color, and its text color is set to the second provided color.

Finally, we've also got an `onchange` event listener that serves to run a function containing a ternary operator. It starts with a test condition — `select.value === 'black'`. If this returns `true`, we run the `update()` function with parameters of black and white, meaning that we end up with background color of black and text color of white. If it returns `false`, we run the `update()` function with parameters of white and black, meaning that the site color are inverted.

**Note:** You can also [find this example on GitHub](#) (see it [running live](#) on there also.)

## Active learning: A simple calendar

In this example you are going to help us finish a simple calendar application. In the code you've got:

- A `<select>` element to allow the user to choose between different months.
- An `onchange` event handler to detect when the value selected in the `<select>` menu is changed.
- A function called `createCalendar()` that draws the calendar and displays the correct month in the `<h1>` element.

We need you to write a conditional statement inside the `onchange` handler function, just below the `// ADD CONDITIONAL HERE` comment. It should:

1. Look at the selected month (stored in the `choice` variable. This will be the `<select>` element value after the value changes, so "January" for example.)
2. Set a variable called `days` to be equal to the number of days in the selected month. To do this you'll have to look up the number of days in each month of the year. You can ignore leap years for the purposes of this example.

Hints:

- You are advised to use logical OR to group multiple months together into a single condition; many of them share the same number of days.
- Think about which number of days is the most common, and use that as a default value.

If you make a mistake, you can always reset the example with the "Reset" button. If you get really stuck, press "Show solution" to see a solution.

## Active learning: More color choices!

In this example you are going to take the ternary operator example we saw earlier and convert the ternary operator into a switch statement that will allow us to apply more choices to the simple website. Look at the [`<select>`](#) — this time you'll see that it has not two theme options, but five. You need to add a switch statement just underneath the `// ADD SWITCH STATEMENT` comment:

- It should accept the `choice` variable as its input expression.
- For each case, the choice should equal one of the possible values that can be selected, i.e. white, black, purple, yellow, or psychedelic.
- For each case, the `update()` function should be run, and be passed two color values, the first one for the background color, and the second one for the text color. Remember that color values are strings, so need to be wrapped in quotes.

If you make a mistake, you can always reset the example with the "Reset" button. If you get really stuck, press "Show solution" to see a solution.

## Conclusion

And that's all you really need to know about conditional structures in JavaScript right now! I'm sure you'll have understood these concepts and worked through the examples with ease; if there is anything you didn't understand, feel free to read through the article again, or [contact us](#) to ask for help.

## Looping code

Programming languages are very useful for rapidly completing repetitive tasks, from multiple basic calculations to just about any other situation where you've got a lot of similar items of work to complete. Here we'll look at the loop structures available in JavaScript that handle such needs.

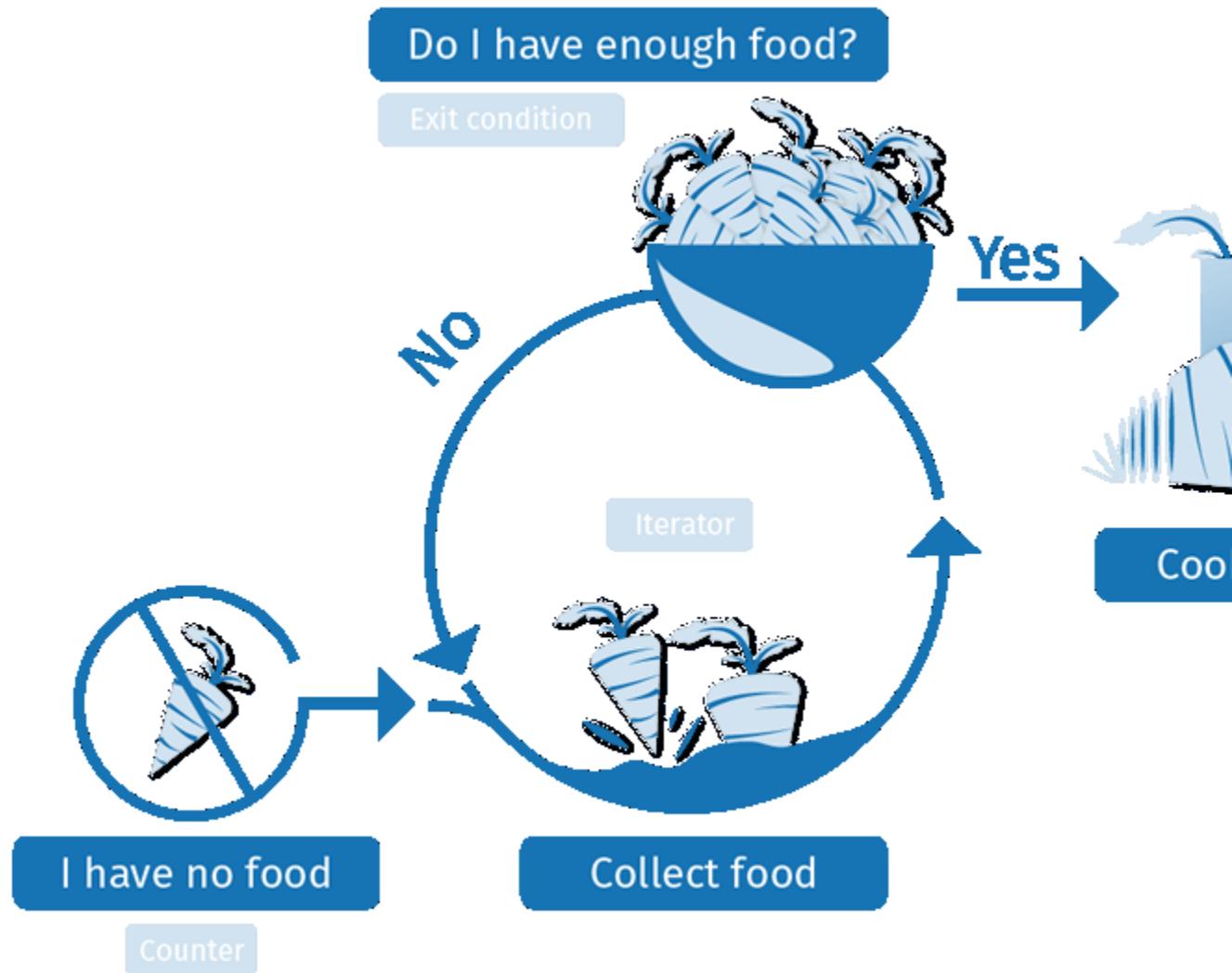
**Prerequisites:** Basic computer literacy, a basic understanding of HTML and CSS, [JavaScript first steps](#).

**Objective:** To understand how to use loops in JavaScript.

## Keep me in the loop

Loops, loops, loops. As well as being associated with [popular breakfast cereals](#), [roller coasters](#) and [musical production](#), they are also a critical concept in programming. Programming loops are all to do with doing the same thing over and over again — which is termed **iteration** in programming speak.

Let's consider the case of a farmer that is making sure he has enough food to feed his family for the week. He might use the following loop to achieve this:



A loop usually has one or more of the following features:

- A **counter**, which is initialized with a certain value — this is the starting point of the loop ("Start: I have no food", above).
- An **exit condition**, which is the criteria under which the loop stops — usually the counter reaching a certain value. This is illustrated by "Have I got enough food?", above. Let's say he needs 10 portions of food to feed his family.
- An **iterator**, which generally increments the counter by a small amount on each successive loop, until it reaches the exit condition. We haven't explicitly illustrated this above, but we could think about the farmer being able to collect say 2 portions of food per hour. After each hour, the amount of food he has collected is incremented by two, and he checks whether he has enough food. If he has reached 10 portions (the exit condition), he can stop collecting and go home.

In [pseudocode](#), this would look something like the following:

```

loop(food = 0; foodNeeded = 10) {
 if (food == foodNeeded) {
 exit loop;
 // We have enough food; let's go home
 } else {
 food += 2; // Spend an hour collecting 2 more food
 // loop will then run again
 }
}

```

So the amount of food needed is set at 10, and the amount the farmer currently has is set at 0. In each iteration of the loop we check whether the amount of food the farmer has is equal to the amount he needs. If so, we can exit the loop. If not, the farmer spends an hour collecting two portions of food, and the loop runs again.

## Why bother?

At this point you probably understand the high level concepts behind loops, but you are probably thinking "OK, great, but how does this help me write better JavaScript code?" As we said earlier, **loops are all to do with doing the same thing over and over again**, which is great for **rapidly completing repetitive tasks**.

Often, the code will be slightly different on each successive iteration of the loop, which means that you can complete a whole load of tasks that are similar but slightly different — if you've got a lot of different calculations to do, you want to do each different one, not the same one over and over again!

Let's look at an example to perfectly illustrate why loops are such a good thing. Let's say we wanted to draw 100 random circles on a `<canvas>` element (press the *Update* button to run the example again and again to see a different random sets):

You don't have to understand all the code for now, but let's look at the part of the code that actually draws the 100 circles:

```

for (var i = 0; i < 100; i++) {
 ctx.beginPath();
 ctx.fillStyle = 'rgba(255,0,0,0.5)';
 ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
 ctx.fill();
}

```

You should get the basic idea — we are using a loop to run 100 iterations of this code, each one of which draws a circle in a random position on the page. The amount of code needed would be the same whether we were drawing 100 circles, 1000, or 10,000. Only one number has to change.

If we weren't using a loop here, we'd have to repeat the following code for every circle we wanted to draw:

```
ctx.beginPath();
ctx.fillStyle = 'rgba(255,0,0,0.5)';
ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
ctx.fill();
```

This would get very boring and difficult to maintain very quickly. Loops really are the best.

## The standard `for` loop

Let's start exploring some specific loop constructs. The first, which you'll use most of the time, is the [for](#) loop — this has the following syntax:

```
for (initializer; exit-condition; final-expression) {
 // code to run
}
```

Here we have:

1. The keyword `for`, followed by some parentheses.
2. Inside the parentheses we have three items, separated by semi-colons:
  1. An **initializer** — this is usually a variable set to a number, which is incremented to count the number of times the loop has run. It is also sometimes referred to as a **counter variable**.
  2. An **exit-condition** — as mentioned before, this defines when the loop should stop looping. This is generally an expression featuring a comparison operator, a test to see if the exit condition has been met.
  3. A **final-expression** — this is always evaluated (or run) each time the loop has gone through a full iteration. It usually serves to increment (or in some cases decrement) the counter variable, to bring it closer to the exit condition value.
3. Some curly braces that contain a block of code — this code will be run each time the loop iterates.

Let's look at a real example so we can visualize what these do more clearly.

```
var cats = ['Bill', 'Jeff', 'Pete', 'Biggles', 'Jasmin'];
var info = 'My cats are called ';
var para = document.querySelector('p');

for (var i = 0; i < cats.length; i++) {
 info += cats[i] + ', ';
}

para.textContent = info;
```

This gives us the following output:

**Note:** You can find this [example code on GitHub](#) too (also [see it running live](#)).

This shows a loop being used to iterate over the items in an array and do something with each of them — a very common pattern in JavaScript. Here:

1. The iterator, `i`, starts at 0 (`var i = 0`).
2. It has been told to run until it is no longer smaller than the length of the `cats` array. This is important — the exit condition shows the condition under which the loop will still run. So in this case, `while i < cats.length` is still true, the loop will still run.
3. Inside the loop, we concatenate the current loop item (`cats[i]` is `cats[whatever i is at the time]`) along with a comma and a space, onto the end of the `info` variable. So:
  1. During the first run, `i = 0`, so `cats[0] + ', '` will be concatenated onto `info` ("Bill, ").
  2. During the second run, `i = 1`, so `cats[1] + ', '` will be concatenated onto `info` ("Jeff, ")
  3. And so on. After each time the loop has run, 1 will be added to `i` (`i++`), then the process will start again.
4. When `i` becomes equal to `cats.length`, the loop will stop, and the browser will move on to the next bit of code below the loop.

**Note:** We have made the exit condition `i < cats.length`, not `i <= cats.length`, because computers count from 0, not 1 — we are starting `i` at 0, and going up to `i = 4` (the index of the last array item). `cats.length` returns 5, as there are 5 items in the array, but we don't want to get up to `i = 5`, as that would return `undefined` for the last item (there is no array item with an index of 5). So therefore we want to go up to 1 less than `cats.length` (`i <`), not the same as `cats.length` (`i <=`).

**Note:** A common mistake with exit conditions is making them use "equal to" (`==`) rather than say "less than or equal to" (`<=`). If we wanted to run our loop up to `i = 5`, the exit condition would need to be `i <= cats.length`. If we set it to `i == cats.length`, the loop would not run at all because `i` is not equal to 5 on the first loop iteration, so it would stop immediately.

One small problem we are left with is that the final output sentence isn't very well-formed:

My cats are called Bill, Jeff, Pete, Biggles, Jasmin,

Ideally we want to change the concatenation on the final loop iteration so that we haven't got a comma on the end of the sentence. Well, no problem — we can quite happily insert a conditional inside our for loop to handle this special case:

```
for (var i = 0; i < cats.length; i++) {
 if (i === cats.length - 1) {
 info += 'and ' + cats[i] + '.';
 } else {
 info += cats[i] + ', '
 }
}
```

**Note:** You can find this [example code on GitHub](#) too (also [see it running live](#)).

**Important:** With for — as with all loops — you must make sure that the initializer is iterated so that it eventually reaches the exit condition. If not, the loop will go on forever, and either the browser will force it to stop, or it will crash. This is called an **infinite loop**.

## Exiting loops with break

If you want to exit a loop before all the iterations have been completed, you can use the `break` statement. We already met this in the previous article when we looked at [switch statements](#) — when a case is met in a switch statement that matches the input expression, the break statement immediately exits the switch statement and moves onto the code after it.

It's the same with loops — a `break` statement will immediately exit the loop and make the browser move on to any code that follows it.

Say we wanted to search through an array of contacts and telephone numbers and return just the number we wanted to find? First, some simple HTML — a text `<input>` allowing us to enter a name to search for, a `<button>` element to submit a search, and a `<p>` element to display the results in:

```
<label for="search">Search by contact name: </label>
<input id="search" type="text">
<button>Search</button>

<p></p>
```

Now on to the JavaScript:

```
var contacts = ['Chris:2232322', 'Sarah:3453456', 'Bill:7654322',
'Mary:9998769', 'Dianne:9384975'];
var para = document.querySelector('p');
var input = document.querySelector('input');
var btn = document.querySelector('button');

btn.addEventListener('click', function() {
 var searchName = input.value;
 input.value = '';
 input.focus();
 for (var i = 0; i < contacts.length; i++) {
 var splitContact = contacts[i].split(':');
 if (splitContact[0] === searchName) {
 para.textContent = splitContact[0] + '\'s number is ' + splitContact[1]
+ '.';
 break;
 } else {
 para.textContent = 'Contact not found.';
 }
 }
});
```

1. First of all we have some variable definitions — we have an array of contact information, with each item being a string containing a name and phone number separated by a colon.
2. Next, we attach an event listener to the button (`btn`), so that when it is pressed, some code is run to perform the search and return the results.
3. We store the value entered into the text input in a variable called `searchName`, before then emptying the text input and focusing it again, ready for the next search.
4. Now onto the interesting part, the for loop:
  1. We start the counter at 0, run the loop until the counter is no longer less than `contacts.length`, and increment `i` by 1 after each iteration of the loop.
  2. Inside the loop we first split the current contact (`contacts[i]`) at the colon character, and store the resulting two values in an array called `splitContact`.
  3. We then use a conditional statement to test whether `splitContact[0]` (the contact's name) is equal to the inputted `searchName`. If it is, we enter a string into the paragraph to report what the contact's number is, and use `break` to end the loop.
  5. If the contact name does not match the entered search, the paragraph text is set to "Contact not found.", and the loop continues iterating.

Note: You can view the [full source code on GitHub](#) too (also [see it running live](#)).

## Skipping iterations with continue

The [continue](#) statement works in a similar manner to `break`, but instead of breaking out of the loop entirely, it skips to the next iteration of the loop. Let's look at another example that takes a number as an input, and returns only the numbers that are squares of integers (whole numbers).

The HTML is basically the same as the last example — a simple text input, and a paragraph for output. The JavaScript is mostly the same too, although the loop itself is a bit different:

```
var num = input.value;

for (var i = 1; i <= num; i++) {
 var sqRoot = Math.sqrt(i);
 if (Math.floor(sqRoot) !== sqRoot) {
 continue;
 }

 para.textContent += i + ' ';
}
```

Here's the output:

1. In this case, the input should be a number (`num`). The `for` loop is given a counter starting at 1 (as we are not interested in 0 in this case), an exit condition that says the loop will stop when the counter becomes bigger than the input `num`, and an iterator that adds 1 to the counter each time.
2. Inside the loop, we find the square root of each number using [Math.sqrt\(i\)](#), then check whether the square root is an integer by testing whether it is the same as itself when it has been rounded down to the nearest integer (this is what [Math.floor\(\)](#) does to the number it is passed).

3. If the square root and the rounded down square root do not equal one another (`!==`), it means that the square root is not an integer, so we are not interested in it. In such a case, we use the `continue` statement to skip on to the next loop iteration without recording the number anywhere.
4. If the square root IS an integer, we skip past the if block entirely so the `continue` statement is not executed; instead, we concatenate the current `i` value plus a space on to the end of the paragraph content.

**Note:** You can view the [full source code on GitHub](#) too (also [see it running live](#)).

## while and do ... while

`for` is not the only type of loop available in JavaScript. There are actually many others and, while you don't need to understand all of these now, it is worth having a look at the structure of a couple of others so that you can recognize the same features at work in a slightly different way.

First, let's have a look at the [while](#) loop. This loop's syntax looks like so:

```
initializer
while (exit-condition) {
 // code to run

 final-expression
}
```

This works in a very similar way to the `for` loop, except that the initializer variable is set before the loop, and the final-expression is included inside the loop after the code to run — rather than these two items being included inside the parentheses. The exit-condition is included inside the parentheses, which are preceded by the `while` keyword rather than `for`.

The same three items are still present, and they are still defined in the same order as they are in the `for` loop — this makes sense, as you still have to have an initializer defined before you can check whether it has reached the exit-condition; the final-condition is then run after the code inside the loop has run (an iteration has been completed), which will only happen if the exit-condition has still not been reached.

Let's have a look again at our cats list example, but rewritten to use a `while` loop:

```
var i = 0;

while (i < cats.length) {
 if (i === cats.length - 1) {
 info += 'and ' + cats[i] + '.';
 } else {
 info += cats[i] + ', ';
 }

 i++;
}
```

**Note:** This still works just the same as expected — have a look at it [running live on GitHub](#) (also view the [full source code](#)).

The `do...while` loop is very similar, but provides a variation on the while structure:

```
initializer
do {
 // code to run

 final-expression
} while (exit-condition)
```

In this case, the initializer again comes first, before the loop starts. The `do` keyword directly precedes the curly braces containing the code to run and the final-expression.

The differentiator here is that the exit-condition comes after everything else, wrapped in parentheses and preceded by a `while` keyword. In a `do...while` loop, the code inside the curly braces is always run once before the check is made to see if it should be executed again (in `while` and `for`, the check comes first, so the code might never be executed).

Let's rewrite our cat listing example again to use a `do...while` loop:

```
var i = 0;

do {
 if (i === cats.length - 1) {
 info += 'and ' + cats[i] + '.';
 } else {
 info += cats[i] + ', ';
 }

 i++;
} while (i < cats.length);
```

**Note:** Again, this works just the same as expected — have a look at it [running live on GitHub](#) (also view the [full source code](#)).

**Important:** With `while` and `do...while` — as with all loops — you must make sure that the initializer is iterated so that it eventually reaches the exit condition. If not, the loop will go on forever, and either the browser will force it to stop, or it will crash. This is called an **infinite loop**.

## Active learning: Launch countdown!

In this exercise, we want you to print out a simple launch countdown to the output box, from 10 down to Blast off. Specifically, we want you to:

- Loop from 10 down to 0. We've provided you with an initializer — `var i = 10;`.

- For each iteration, create a new paragraph and append it to the output `<div>`, which we've selected using `var output = document.querySelector('.output');`. In comments, we've provided you with three code lines that need to be used somewhere inside the loop:
  - `var para = document.createElement('p');` — creates a new paragraph.
  - `output.appendChild(para);` — appends the paragraph to the output `<div>`.
  - `para.textContent =` — makes the text inside the paragraph equal to whatever you put on the right hand side, after the equals sign.
- Different iteration numbers require different text to be put in the paragraph for that iteration (you'll need a conditional statement and multiple `para.textContent = lines`):
  - If the number is 10, print "Countdown 10" to the paragraph.
  - If the number is 0, print "Blast off!" to the paragraph.
  - For any other number, print just the number to the paragraph.
- Remember to include an iterator! However, in this example we are counting down after each iteration, not up, so you **don't** want `i++` — how do you iterate downwards?

If you make a mistake, you can always reset the example with the "Reset" button. If you get really stuck, press "Show solution" to see a solution.

## Active learning: Filling in a guest list

In this exercise, we want you to take a list of names stored in an array, and put them into a guest list. But it's not quite that easy — we don't want to let Phil and Lola in because they are greedy and rude, and always eat all the food! We have two lists, one for guests to admit, and one for guests to refuse.

Specifically, we want you to:

- Write a loop that will iterate from 0 to the length of the `people` array. You'll need to start with an initializer of `var i = 0;`, but what exit condition do you need?
- During each loop iteration, check if the current array item is equal to "Phil" or "Lola" using a conditional statement:
  - If it is, concatenate the array item to the end of the `refused` paragraph's `textContent`, followed by a comma and a space.
  - If it isn't, concatenate the array item to the end of the `admitted` paragraph's `textContent`, followed by a comma and a space.

We've already provided you with:

- `var i = 0;` — Your initializer.
- `refused.textContent +=` — the beginnings of a line that will concatenate something on to the end of `refused.textContent`.
- `admitted.textContent +=` — the beginnings of a line that will concatenate something on to the end of `admitted.textContent`.

Extra bonus question — after completing the above tasks successfully, you will be left with two lists of names, separated by commas, but they will be untidy — there will be a comma at the end

of each one. Can you work out how to write lines that slice the last comma off in each case, and add a full stop to the end? Have a look at the [Useful string methods](#) article for help.

If you make a mistake, you can always reset the example with the "Reset" button. If you get really stuck, press "Show solution" to see a solution.

## Which loop type should you use?

For basic uses, `for`, `while`, and `do...while` loops are largely interchangeable. They can all be used to solve the same problems, and which one you use will largely depend on your personal preference — which one you find easiest to remember or most intuitive. Let's have a look at them again.

First `for`:

```
for (initializer; exit-condition; final-expression) {
 // code to run
}
```

`while`:

```
initializer
while (exit-condition) {
 // code to run

 final-expression
}
```

and finally `do...while`:

```
initializer
do {
 // code to run

 final-expression
} while (exit-condition)
```

We would recommend `for`, at least to begin with, as it is probably the easiest for remembering everything — the initializer, exit-condition, and final-expression all have to go neatly into the parentheses, so it is easy to see where they are and check that you aren't missing them.

**Note:** There are other loop types/features too, which are useful in advanced/specialized situations and beyond the scope of this article. If you want to go further with your loop learning, read our advanced [Loops and iteration guide](#).

## Conclusion

This article has revealed to you the basic concepts behind, and different options available when, looping code in JavaScript. You should now be clear on why loops are a good mechanism for dealing with repetitive code, and be raring to use them in your own examples!

If there is anything you didn't understand, feel free to read through the article again, or [contact us](#) to ask for help.

## Functions — reusable blocks of code

Another essential concept in coding is **functions**, which allow you to store a piece of code that does a single task inside a defined block, and then call that code whenever you need it using a single short command — rather than having to type out the same code multiple times. In this article we'll explore fundamental concepts behind functions such as basic syntax, how to invoke and define them, scope, and parameters.

**Prerequisites:** Basic computer literacy, a basic understanding of HTML and CSS, [JavaScript first steps](#).

**Objective:** To understand the fundamental concepts behind JavaScript functions.

## Where do I find functions?

In JavaScript, you'll find functions everywhere. In fact, we've been using functions all the way through the course so far; we've just not been talking about them very much. Now is the time, however, for us to start talking about functions explicitly, and really exploring their syntax.

Pretty much anytime you make use of a JavaScript structure that features a pair of parentheses — `()` — and you're **not** using a common built-in language structure like a [for loop](#), [while](#) or [do...while loop](#), or [if...else statement](#), you are making use of a function.

## Built-in browser functions

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

```
var myText = 'I am a string';
var newString = myText.replace('string', 'sausage');
console.log(newString);
// the replace() string function takes a string,
// replaces one substring with another, and returns
// a new string with the replacement made
```

Or every time we manipulated an array:

```
var myArray = ['I', 'love', 'chocolate', 'frogs'];
var madeAString = myArray.join(' ');
console.log(madeAString);
```

```
// the join() function takes an array, joins
// all the array items together into a single
// string, and returns this new string
```

Or every time we generated a random number:

```
var myNumber = Math.random();
// the random() function generates a random
// number between 0 and 1, and returns that
// number
```

...we were using a function!

**Note:** Feel free to enter these lines into your browser's JavaScript console to re-familiarize yourself with their functionality, if needed.

The JavaScript language has many built-in functions to allow you to do useful things without having to write all that code yourself. In fact, some of the code you are calling when you **invoke** (a fancy word for run, or execute) a built in browser function couldn't be written in JavaScript — many of these functions are calling parts of the background browser code, which is written largely in low-level system languages like C++, not web languages like JavaScript.

Bear in mind that some built-in browser functions are not part of the core JavaScript language — some are defined as part of browser APIs, which build on top of the default language to provide even more functionality (refer to [this early section of our course](#) for more descriptions). We'll look at using browser APIs in more detail in a later module.

## Functions versus methods

One thing we need to clear up before we move on — technically speaking, built in browser functions are not functions — they are **methods**. This sounds a bit scary and confusing, but don't worry — the words function and method are largely interchangeable, at least for our purposes, at this stage in your learning.

The distinction is that methods are functions defined inside objects. Built-in browser functions (methods) and variables (which are called **properties**) are stored inside structured objects, to make the code more efficient and easier to handle.

You don't need to learn about the inner workings of structured JavaScript objects yet — you can wait until our later module that will teach you all about the inner workings of objects, and how to create your own. For now, we just wanted to clear up any possible confusion of method versus function — you are likely to meet both terms as you look at the available related resources across the Web.

## Custom functions

You've also seen a lot of **custom functions** in the course so far — functions defined in your code, not inside the browser. Anytime you saw a custom name with parentheses straight after it, you were using a custom function. In our [random-canvas-circles.html](#) example (see also the full [source code](#)) from our [loops article](#), we included a custom `draw()` function that looked like this:

```
function draw() {
 ctx.clearRect(0,0,WIDTH,HEIGHT);
 for (var i = 0; i < 100; i++) {
 ctx.beginPath();
 ctx.fillStyle = 'rgba(255,0,0,0.5)';
 ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
 ctx.fill();
 }
}
```

This function draws 100 random circles inside an [`<canvas>`](#) element. Every time we want to do that, we can just invoke the function with this

```
draw();
```

rather than having to write all that code out again every time we want to repeat it. And functions can contain whatever code you like — you can even call other functions from inside functions. The above function for example calls the `random()` function three times, which is defined by the following code:

```
function random(number) {
 return Math.floor(Math.random()*number);
}
```

We needed this function because the browser's built-in [Math.random\(\)](#) function only generates a random decimal number between 0 and 1. We wanted a random whole number between 0 and a specified number.

## Invoking functions

You are probably clear on this by now, but just in case ... to actually use a function after it has been defined, you've got to run — or invoke — it. This is done by including the name of the function in the code somewhere, followed by parentheses.

```
function myFunction() {
 alert('hello');
}

myFunction()
// calls the function once
```

## Anonymous functions

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

```
function myFunction() {
 alert('hello');
}
```

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

```
function() {
 alert('hello');
}
```

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

```
var myButton = document.querySelector('button');

myButton.onclick = function() {
 alert('hello');
}
```

The above example would require there to be a [`<button>`](#) element available on the page to select and click. You've already seen this structure a few times throughout the course, and you'll learn more about and see it in use in the next article.

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

```
var myGreeting = function() {
 alert('hello');
}
```

This function could now be invoked using:

```
myGreeting();
```

This effectively gives the variable a name; you can also assign the function to be the value of multiple variables, for example:

```
var anotherGreeting = function() {
 alert('hello');
}
```

This function could now be invoked using either of

```
myGreeting();
anotherGreeting();
```

But this would just be confusing, so don't do it! When creating functions, it is better to just stick to this form:

```
function myGreeting() {
 alert('hello');
}
```

You will mainly use anonymous functions to just run a load of code in response to an event firing — like a button being clicked — using an event handler. Again, this looks something like this:

```
myButton.onclick = function() {
 alert('hello');
 // I can put as much code
 // inside here as I want
}
```

## Function parameters

Some functions require **parameters** to be specified when you are invoking them — these are values that need to be included inside the function parentheses, which it needs to do its job properly.

**Note:** Parameters are sometimes called arguments, properties, or even attributes.

As an example, the browser's built-in [Math.random\(\)](#) function doesn't require any parameters. When called, it always returns a random number between 0 and 1:

```
var myNumber = Math.random();
```

The browser's built-in string [replace\(\)](#) function however needs two parameters — the substring to find in the main string, and the substring to replace that string with:

```
var myText = 'I am a string';
var newString = myText.replace('string', 'sausage');
```

**Note:** When you need to specify multiple parameters, they are separated by commas.

It should also be noted that sometimes parameters are optional — you don't have to specify them. If you don't, the function will generally adopt some kind of default behavior. As an example, the array [join\(\)](#) function's parameter is optional:

```
var myArray = ['I', 'love', 'chocolate', 'frogs'];
var madeAString = myArray.join(' ');
// returns 'I love chocolate frogs'
var madeAString = myArray.join();
// returns 'I,love,chocolate,frogs'
```

If no parameter is included to specify a joining/delimiting character, a comma is used by default.

# Function scope and conflicts

Let's talk a bit about [scope](#) — a very important concept when dealing with functions. When you create a function, the variables and other things defined inside the function are inside their own separate **scope**, meaning that they are locked away in their own separate compartments, unreachable from inside other functions or from code outside the functions.

The top level outside all your functions is called the **global scope**. Values defined in the global scope are accessible from everywhere in the code.

JavaScript is set up like this for various reasons — but mainly because of security and organization. Sometimes you don't want variables to be accessible from everywhere in the code — external scripts that you call in from elsewhere could start to mess with your code and cause problems because they happen to be using the same variable names as other parts of the code, causing conflicts. This might be done maliciously, or just by accident.

For example, say you have an HTML file that is calling in two external JavaScript files, and both of them have a variable and a function defined that use the same name:

```
<!-- Excerpt from my HTML -->
<script src="first.js"></script>
<script src="second.js"></script>
<script>
 greeting();
</script>
// first.js
var name = 'Chris';
function greeting() {
 alert('Hello ' + name + ': welcome to our company.');
}
// second.js
var name = 'Zaptec';
function greeting() {
 alert('Our company is called ' + name + '.');
}
```

Both functions you want to call are called `greeting()`, but you can only ever access the `second.js` file's `greeting()` function — it is applied to the HTML later on in the source code, so its variable and function overwrite the ones in `first.js`.

**Note:** You can see this example [running live on GitHub](#) (see also the [source code](#)).

Keeping parts of your code locked away in functions avoids such problems, and is considered best practice.

It is a bit like a zoo. The lions, zebras, tigers, and penguins are kept in their own enclosures, and only have access to the things inside their enclosures — in the same manner as the function scopes. If they were able to get into other enclosures, problems would occur. At best, different animals would feel really uncomfortable inside unfamiliar habitats — a lion or tiger would feel

terrible inside the penguins' watery, icy domain. At worst, the lions and tigers might try to eat the penguins!



The zoo keeper is like the global scope — he or she has the keys to access every enclosure, to restock food, tend to sick animals, etc.

### Active learning: Playing with scope

Let's look at a real example to demonstrate scoping.

1. First, make a local copy of our [function-scope.html](#) example. This contains two functions called `a()` and `b()`, and three variables — `x`, `y`, and `z` — two of which are defined inside the functions, and one in the global scope. It also contains a third function called `output()`, which takes a single parameter and outputs it in a paragraph on the page.
2. Open the example up in a browser and in your text editor.
3. Open the JavaScript console in your browser developer tools. In the JavaScript console, enter the following command:

```
output(x);
```

- You should see the value of variable `x` output to the screen.
- Now try entering the following in your console

```
output(y);
```

```
output(z);
```

- Both of these should return an error along the lines of "[ReferenceError: y is not defined](#)". Why is that?

Because of function scope — y and z are locked inside the a() and b() functions, so output() can't access them when called from the global scope.

- However, what about when it's called from inside another function? Try editing a() and b() so they look like this:

```
function a() {
 var y = 2;
 output(y);
}
```

```
function b() {
 var z = 3;
 output(z);
}
```

Save the code and reload it in your browser, then try calling the a() and b() functions from the JavaScript console:

```
a();
b();
```

- You should see the y and z values output in the page. This works fine, as the output() function is being called inside the other functions — in the same scope as the variables it is printing are defined in, in each case. output() itself is available from anywhere, as it is defined in the global scope.

- Now try updating your code like this:

```
function a() {
 var y = 2;
 output(x);
}
```

```
function b() {
 var z = 3;
 output(x);
}
```

Save and reload again, and try this again in your JavaScript console:

```
a();
b();
```

- Both the a() and b() call should output the value of x — 1. These work fine because even though the output() calls are not in the same scope as x is defined in, x is a global variable so is available inside all code, everywhere.

- Finally, try updating your code like this:

```
function a() {
 var y = 2;
 output(z);
}
```

```
function b() {
 var z = 3;
```

```
 output(y);
}
Save and reload again, and try this again in your JavaScript console:
```

```
a();
b();
```

7. This time the `a()` and `b()` calls will both return that annoying "[ReferenceError: z is not defined](#)" error — this is because the `output()` calls and the variables they are trying to print are not defined inside the same function scopes — the variables are effectively invisible to those function calls.

**Note:** The same scoping rules do not apply to loop (e.g. `for() { ... }`) and conditional blocks (e.g. `if() { ... }`) — they look very similar, but they are not the same thing! Take care not to get these confused.

**Note:** The [ReferenceError: "x" is not defined](#) error is one of the most common you'll encounter. If you get this error and you are sure that you have defined the variable in question, check what scope it is in.

## Functions inside functions

Keep in mind that you can call a function from anywhere, even inside another function. This is often used as a way to keep code tidy — if you have a big complex function, it is easier to understand if you break it down into several sub-functions:

```
function myBigFunction() {
 var myValue;

 subFunction1();
 subFunction2();
 subFunction3();
}

function subFunction1() {
 console.log(myValue);
}

function subFunction2() {
 console.log(myValue);
}

function subFunction3() {
 console.log(myValue);
}
```

Just make sure that the values being used inside the function are properly in scope. The example above would throw an error `ReferenceError: myValue is not defined`, because although the `myValue` variable is defined in the same scope as the function calls, it is not defined inside the function definitions — the actual code that is run when the functions are called. To make this work, you'd have to pass the value into the function as a parameter, like this:

```
function myBigFunction() {
 var myValue = 1;

 subFunction1(myValue);
 subFunction2(myValue);
 subFunction3(myValue);
}

function subFunction1(value) {
 console.log(value);
}

function subFunction2(value) {
 console.log(value);
}

function subFunction3(value) {
 console.log(value);
}
```

## Conclusion

This article has explored the fundamental concepts behind functions, paving the way for the next one in which we get practical and take you through the steps to building up your own custom function.

### Build your own function

With most of the essential theory dealt with in the previous article, this article provides practical experience. Here you will get some practice building your own, custom function. Along the way, we'll also explain some useful details of dealing with functions.

**Prerequisites:** Basic computer literacy, a basic understanding of HTML and CSS, [JavaScript first steps](#), [Functions — reusable blocks of code](#).

**Objective:** To provide some practice in building a custom function, and explain a few more useful associated details.

## Active learning: Let's build a function

The custom function we are going to build will be called `displayMessage()`. It will display a custom message box on a web page and will act as a customized replacement for a browser's built-in `alert()` function. We've seen this before, but let's just refresh our memories. Type the following in your browser's JavaScript console, on any page you like:

```
alert('This is a message');
```

The `alert` function takes a single argument — the string that is displayed in the alert box. Try varying the string to change the message.

The `alert` function is limited: you can alter the message, but you can't easily vary anything else, such as the color, icon, or anything else. We'll build one that will prove to be more fun.

**Note:** This example should work in all modern browsers fine, but the styling might look a bit funny in slightly older browsers. We'd recommend you doing this exercise in a modern browser like Firefox, Opera, or Chrome.

## The basic function

To begin with, let's put together a basic function.

**Note:** For function naming conventions, you should follow the same rules as [variable naming conventions](#). This is fine, as you can tell them apart — function names appear with parentheses after them, and variables don't.

1. Start by accessing the [function-start.html](#) file and making a local copy. You'll see that the HTML is simple — the body contains just a single button. We've also provided some basic CSS to style the custom message box, and an empty `<script>` element to put our JavaScript in.
  2. Next, add the following inside the `<script>` element:
    3.   `function displayMessage() {`
    4.   `}`
- We start off with the keyword `function`, which means we are defining a function. This is followed by the name we want to give to our function, a set of parentheses, and a set of curly braces. Any parameters we want to give to our function go inside the parentheses, and the code that runs when we call the function goes inside the curly braces.
  - Finally, add the following code inside the curly braces:

```
var html = document.querySelector('html');

var panel = document.createElement('div');
panel.setAttribute('class', 'msgBox');
html.appendChild(panel);

var msg = document.createElement('p');
msg.textContent = 'This is a message box';
panel.appendChild(msg);

var closeBtn = document.createElement('button');
closeBtn.textContent = 'x';
panel.appendChild(closeBtn);

closeBtn.onclick = function() {
 panel.parentNode.removeChild(panel);
}
```

3.

This is quite a lot of code to go through, so we'll walk you through it bit by bit.

The first line uses a DOM API function called [document.querySelector\(\)](#) to select the [`<html>`](#) element and store a reference to it in a variable called `html`, so we can do things to it later on:

```
var html = document.querySelector('html');
```

The next section uses another DOM API function called [Document.createElement\(\)](#) to create a [`<div>`](#) element and store a reference to it in a variable called `panel`. This element will be the outer container of our message box.

We then use yet another DOM API function called [Element.setAttribute\(\)](#) to set a `class` attribute on our panel with a value of `msgBox`. This is to make it easier to style the element — if you look at the CSS on the page, you'll see that we are using a `.msgBox` class selector to style the message box and its contents.

Finally, we call a DOM function called [Node.appendChild\(\)](#) on the `html` variable we stored earlier, which nests one element inside the other as a child of it. We specify the panel [`<div>`](#) as the child we want to append inside the [`<html>`](#) element. We need to do this as the element we created won't just appear on the page on its own — we need to specify where to put it.

```
var panel = document.createElement('div');
panel.setAttribute('class', 'msgBox');
html.appendChild(panel);
```

The next two sections make use of the same `createElement()` and `appendChild()` functions we've already seen to create two new elements — a [`<p>`](#) and a [`<button>`](#) — and insert them in the page as children of the panel [`<div>`](#). We use their [Node.textContent](#) property — which represents the text content of an element — to insert a message inside the paragraph, and an 'x' inside the button. This button will be what needs to be clicked/activated when the user wants to close the message box.

```
var msg = document.createElement('p');
msg.textContent = 'This is a message box';
panel.appendChild(msg);

var closeBtn = document.createElement('button');
closeBtn.textContent = 'x';
panel.appendChild(closeBtn);
```

Finally, we use an [GlobalEventHandlers.onclick](#) event handler to make it so that when the button is clicked, some code is run to delete the whole panel from the page — to close the message box.

Briefly, the `onclick` handler is a property available on the button (or in fact, any element on the page) that can be set to a function to specify what code to run when the button is clicked. You'll learn a lot more about these in our later events article. We are making the `onclick` handler equal to an anonymous function, which contains the code to run when the button is clicked. The line inside the function uses the [Node.removeChild\(\)](#) DOM API function to specify that we want to remove a specific child element of the HTML element — in this case the panel `<div>`.

```
closeBtn.onclick = function() {
 panel.parentNode.removeChild(panel);
}
```

Basically, this whole block of code is generating a block of HTML that looks like so, and inserting it into the page:

```
<div class="msgBox">
 <p>This is a message box</p>
 <button>x</button>
</div>
```

That was a lot of code to work through — don't worry too much if you don't remember exactly how every bit of it works right now! The main part we want to focus on here is the function's structure and usage, but we wanted to show something interesting for this example.

## Calling the function

You've now got your function definition written into your `<script>` element just fine, but it will do nothing as it stands.

1. Try including the following line below your function to call it:

```
displayMessage();
```

- This line invokes the function, making it run immediately. When you save your code and reload it in the browser, you'll see the little message box appear immediately, only once. We are only calling it once, after all.
- Now open your browser developer tools on the example page, go to the JavaScript console and type the line again there, you'll see it appear again! So this is fun — we now have a reusable function that we can call any time we like.

But we probably want it to appear in response to user and system actions. In a real application, such a message box would probably be called in response to new data being available, or an error having occurred, or the user trying to delete their profile ("are you sure about this?"), or the user adding a new contact and the operation completing successfully ... etc.

In this demo, we'll get the message box to appear when the user clicks the button.

- Delete the previous line you added.

- Next, we'll select the button and store a reference to it in a variable. Add the following line to your code, above the function definition:

```
var btn = document.querySelector('button');
```

- Finally, add the following line below the previous one:

```
btn.onclick = displayMessage;
```

5. In a similar way to our `closeBtn.onclick` line inside the function, here we are calling some code in response to a button being clicked. But in this case, instead of calling an anonymous function containing some code, we are calling our function name directly.
6. Try saving and refreshing the page — now you should see the message box appear when you click the button.

You might be wondering why we haven't included the parentheses after the function name. This is because we don't want to call the function immediately — only after the button has been clicked. If you try changing the line to

```
btn.onclick = displayMessage();
```

and saving and reloading, you'll see that the message box appears without the button being clicked! The parentheses in this context are sometimes called the "function invocation operator". You only use them when you want to run the function immediately in the current scope. In the same respect, the code inside the anonymous function is not run immediately, as it is inside the function scope.

If you tried the last experiment, make sure to undo the last change before carrying on.

## Improving the function with parameters

As it stands, the function is still not very useful — we don't want to just show the same default message every time. Let's improve our function by adding some parameters, allowing us to call it with some different options.

1. First of all, update the first line of the function:

```
function displayMessage() {
```

to this:

```
function displayMessage(msgText, msgType) {
```

- Now when we call the function, we can provide two variable values inside the parentheses to specify the message to display in the message box, and the type of message it is.

- To make use of the first parameter, update the following line inside your function:

```
msg.textContent = 'This is a message box';
to
```

```
msg.textContent = msgText;
```

- Last but not least, you now need to update your function call to include some updated message text. Change the following line:

```
btn.onclick = displayMessage;
to this block:
```

```
btn.onclick = function() {
 displayMessage('Woo, this is a different message!');
};
```

- If we want to specify parameters inside parentheses for the function we are calling, then we can't call it directly — we need to put it inside an anonymous function so that it isn't in the immediate scope and therefore isn't called immediately. Now it will not be called until the button is clicked.
- Reload and try the code again and you'll see that it still works just fine, except that now you can also vary the message inside the parameter to get different messages displayed in the box!

## A more complex parameter

On to the next parameter. This one is going to involve slightly more work — we are going to set it so that depending on what the `msgType` parameter is set to, the function will display a different icon and a different background color.

- First of all, download the icons needed for this exercise ([warning](#) and [chat](#)) from GitHub. Save them in a new folder called `icons` in the same location as your HTML file.

**Note:** [warning](#) and [chat](#) icons found on iconfinder.com, and designed by [Nazarrudin Ansyari](#). Thanks!

- Next, find the CSS inside your HTML file. We'll make a few changes to make way for the icons. First, update the `.msgBox width` from:

```
width: 200px;
to
```

```
width: 242px;
```

- Next, add the following lines inside the `.msgBox p { ... }` rule:

```
padding-left: 82px;
background-position: 25px center;
background-repeat: no-repeat;
```

- Now we need to add code to our `displayMessage()` function to handle displaying the icons. Add the following block just above the closing curly brace `()` of your function:

```
if (msgType === 'warning') {
 msg.style.backgroundImage = 'url(Icons/warning.png)';
 panel.style.backgroundColor = 'red';
} else if (msgType === 'chat') {
 msg.style.backgroundImage = 'url(Icons/chat.png)';
 panel.style.backgroundColor = 'aqua';
} else {
```

```
 msg.style.paddingLeft = '20px';
}
```

- Here, if the `msgType` parameter is set as `'warning'`, the warning icon is displayed and the panel's background color is set to red. If it is set to `'chat'`, the chat icon is displayed and the panel's background color is set to aqua blue. If the `msgType` parameter is not set at all (or to something different), then the `else { ... }` part of the code comes into play, and the paragraph is simply given default padding and no icon, with no background panel color set either. This provides a default state if no `msgType` parameter is provided, meaning that it is an optional parameter!

- Let's test out our updated function, try updating the `displayMessage()` call from this:

```
displayMessage('Woo, this is a different message!');
to one of these:
```

```
displayMessage('Your inbox is almost full - delete some mails', 'warning');
displayMessage('Brian: Hi there, how are you today?', 'chat');
```

5. You can see how useful our (now not so) little function is becoming.

**Note:** If you have trouble getting the example to work, feel free to check your code against the [finished version on GitHub](#) ([see it running live](#) also), or ask us for help.

## Conclusion

Congratulations on reaching the end! This article took you through the entire process of building up a practical custom function, which with a bit more work could be transplanted into a real project. In the next article we'll wrap up functions by explaining another essential related concept — return values.

## Function return values

There's one last essential concept for us to discuss in this course, to close our look at functions — return values. Some functions don't return a significant value after completion, but others do, and it's important to understand what their values are, how to make use of them in your code, and how to make your own custom functions return useful values. We'll cover all of these below.

**Prerequisites:** Basic computer literacy, a basic understanding of HTML and CSS, [JavaScript first steps](#), [Functions — reusable blocks of code](#).

**Objective:** To understand function return values, and how to make use of them.

## What are return values?

**Return values** are just what they sound like — values returned by the function when it completes. You've already met return values a number of times, although you may not have thought about them explicitly. Let's return to some familiar code:

```
var myText = 'I am a string';
var newString = myText.replace('string', 'sausage');
console.log(newString);
// the replace() string function takes a string,
// replaces one substring with another, and returns
// a new string with the replacement made
```

We saw exactly this block of code in our first function article. We are invoking the [replace\(\)](#) function on the `myText` string, and passing it two parameters — the substring to find, and the substring to replace it with. When this function completes (finishes running), it returns a value, which is a new string with the replacement made. In the code above, we are saving this return value as the value of the `newString` variable.

If you look at the `replace` function MDN reference page, you'll see a section called [Return value](#). It is very useful to know and understand what values are returned by functions, so we try to include this information wherever possible.

Some functions don't return a return value as such (in our reference pages, the return value is listed as `void` or `undefined` in such cases). For example, in the [displayMessage\(\) function](#) we built in the previous article, no specific value is returned as a result of the function being invoked. It just makes a box appear somewhere on the screen — that's it!

Generally, a return value is used where the function is an intermediate step in a calculation of some kind. You want to get to a final result, which involves some values. Those values need to be calculated by a function, which then returns the results so they can be used in the next stage of the calculation.

## Using return values in your own functions

To return a value from a custom function, you need to use ... wait for it ... the [return](#) keyword. We saw this in action recently in our [random-canvas-circles.html](#) example. Our `draw()` function draws 100 random circles somewhere on an HTML [`<canvas>`](#):

```
function draw() {
 ctx.clearRect(0,0,WIDTH,HEIGHT);
 for (var i = 0; i < 100; i++) {
 ctx.beginPath();
 ctx.fillStyle = 'rgba(255,0,0,0.5)';
 ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
 ctx.fill();
 }
}
```

Inside each loop iteration, three calls are made to the `random()` function, to generate a random value for the current circle's x coordinate, y coordinate, and radius, respectively. The `random()` function takes one parameter — a whole number — and it returns a whole random number between 0 and that number. It looks like this:

```
function random(number) {
```

```
 return Math.floor(Math.random()*number);
}
```

This could be written as follows:

```
function random(number) {
 var result = Math.floor(Math.random()*number);
 return result;
}
```

But the first version is quicker to write, and more compact.

We are returning the result of the calculation `Math.floor(Math.random()*number)` each time the function is called. This return value appears at the point the function was called, and the code continues. So for example, if we ran the following line:

```
ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
```

and the three `random()` calls returned the values 500, 200, and 35, respectively, the line would actually be run as if it were this:

```
ctx.arc(500, 200, 35, 0, 2 * Math.PI);
```

The function calls on the line are run first and their return values substituted for the function calls, before the line itself is then executed.

## Active learning: our own return value function

Let's have a go at writing our own functions featuring return values.

1. First of all, make a local copy of the [function-library.html](#) file from GitHub. This is a simple HTML page containing a text `<input>` field and a paragraph. There's also a `<script>` element in which we have stored a reference to both HTML elements in two variables. This little page will allow you to enter a number into the text box, and display different numbers related to it in the paragraph below.
2. Let's add some useful functions to this `<script>` element. Below the existing two lines of JavaScript, add the following function definitions:

```
3. function squared(num) {
4. return num * num;
5.
6.
7. function cubed(num) {
8. return num * num * num;
9. }
10.
11. function factorial(num) {
12. var x = num;
13. while (x > 1) {
14. num *= x-1;
15. x--;
16. }
17. return num;
18. }
```

```
16. }
17. return num;
}
```

- The `squared()` and `cubed()` functions are fairly obvious — they return the square or cube of the number given as a parameter. The `factorial()` function returns the [factorial](#) of the given number.
- Next we're going to include a way to print out information about the number entered into the text input. Enter the following event handler below the existing functions:

```
input.onchange = function() {
 var num = input.value;
 if (isNaN(num)) {
 para.textContent = 'You need to enter a number!';
 } else {
 para.textContent = num + ' squared is ' + squared(num) + '. ' +
 num + ' cubed is ' + cubed(num) + '. ' +
 num + ' factorial is ' + factorial(num) + '.';
 }
}
```

3. Here we are creating an `onchange` event handler that runs whenever the `change` event fires on the text input — that is, when a new value is entered into the text input, and submitted (enter a value then press tab for example). When this anonymous function runs, the existing value entered into the input is stored in the `num` variable.

Next, we do a conditional test — if the entered value is not a number, we print an error message into the paragraph. The test looks at whether the expression `isNaN(num)` returns true. We use the [`isNaN\(\)`](#) function to test whether the `num` value is not a number — if so, it returns `true`, and if not, `false`.

If the test returns `false`, the `num` value is a number, so we print out a sentence inside the paragraph element stating what the square, cube, and factorial of the number are. The sentence calls the `squared()`, `cubed()`, and `factorial()` functions to get the required values.

4. Save your code, load it in a browser, and try it out.

**Note:** If you have trouble getting the example to work, feel free to check your code against the [finished version on GitHub](#) ([see it running live](#) also), or ask us for help.

At this point, we'd like you to have a go at writing out a couple of functions of your own and adding them to the library. How about the square or cube root of the number, or the circumference of a circle with a radius of length `num`?

This exercise has brought up a couple of important points besides being a study on how to use the `return` statement. In addition, we have:

- Looked at another example of writing error handling into our functions. It is generally a good idea to check that any necessary parameters have been provided, and in the right datatype, and

if they are optional, that some kind of default value is provided to allow for that. This way, your program will be less likely to throw errors.

- Thought about the idea of creating a function library. As you go further into your programming career, you'll start to do the same kinds of things over and over again. It is a good idea to start keeping your own library of utility functions that you use very often — you can then copy them over to your new code, or even just apply it to any HTML pages where you need it.

## Conclusion

So there we have it — functions are fun, very useful and, although there's a lot to talk about in regards to their syntax and functionality, fairly understandable given the right articles to study.

If there is anything you didn't understand, feel free to read through the article again, or [contact us](#) to ask for help.

### Introduction to events

Events are actions or occurrences that happen in the system you are programming, which the system tells you about so you can respond to them in some way if desired. For example if the user clicks a button on a webpage, you might want to respond to that action by displaying an information box. In this article we will discuss some important concepts surrounding events, and look at how they work in browsers. This won't be an exhaustive study; just what you need to know at this stage.

**Prerequisites:** Basic computer literacy, a basic understanding of HTML and CSS, [JavaScript first steps](#).

**Objective:** To understand the fundamental theory of events, how they work in browsers, and how events may differ in different programming environments.

## A series of fortunate events

As mentioned above, **events** are actions or occurrences that happen in the system you are programming — the system will fire a signal of some kind when an event occurs, and also provide a mechanism by which some kind of action can be automatically taken (e.g. some code running) when the event occurs. For example in an airport when the runway is clear for a plane to take off, a signal is communicated to the pilot, and as a result they commence piloting the plane.



In the case of the Web, events are fired inside the browser window, and tend to be attached to a specific item that resides in it — this might be a single element, set of elements, the HTML document loaded in the current tab, or the entire browser window. There are a lot of different types of event that can occur, for example:

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

You will gather from this (and from glancing at the MDN [Event reference](#)) that there are **a lot** of events that can be responded to.

Each available event has an **event handler**, which is block of code (usually a user-defined JavaScript function) that will be run when the event fires. When such a block of code is defined to be run in response to an event firing, we say we are **registering an event handler**. Note that event handlers are sometimes called **event listeners** — they are pretty much interchangeable for

our purposes, although strictly speaking they work together. The listener listens out for the event happening, and the handler is the code that is run in response to it happening.

**Note:** It is important to note that web events are not part of the core JavaScript language — they are defined as part of the JavaScript APIs built into the browser.

## A simple example

Let's look at a simple example to explain what we mean here. You've already seen events and event handlers used in many of the examples in this course already, but let's recap just to cement our knowledge. In the following example, we have a single `<button>`, which when pressed, will make the background change to a random color:

```
<button>Change color</button>
```

The JavaScript looks like so:

```
var btn = document.querySelector('button');

function random(number) {
 return Math.floor(Math.random() * (number+1));
}

btn.onclick = function() {
 var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) +
')';
 document.body.style.backgroundColor = rndCol;
}
```

In this code, we store a reference to the button inside a variable called `btn`, using the [Document.querySelector\(\)](#) function. We also define a function that returns a random number. The third part of the code is the event handler. The `btn` variable points to a `<button>` element, and this type of object has a number of events that can fire on it, and therefore, event handlers available. We are listening for the click event firing, by setting the `onclick` event handler property to equal an anonymous function containing code that generated a random RGB color and sets the `<body>` background-color equal to it.

This code will now be run whenever the click event fires on the `<button>` element, i.e., whenever a user clicks on it.

The example output is as follows:

## It's not just web pages

Another thing worth mentioning at this point is that events are not particular to JavaScript — most programming languages have some kind of event model, and the way it works will often differ from JavaScript's way. In fact, the event model in JavaScript for web pages differs from the event model for JavaScript as it is used in other environments.

For example, [Node.js](#) is a very popular JavaScript runtime that enables developers to use JavaScript to build network and server-side applications. The [Node.js event model](#) relies on listeners to listen for events and emitters to emit events periodically — it doesn't sound that different, but the code is quite different, making use of functions like `on()` to register an event listener, and `once()` to register an event listener that unregisters after it has run once. The [HTTP connect event docs](#) provide a good example of use.

As another example, you can now also use JavaScript to build cross-browser add-ons — browser functionality enhancements — using a technology called [WebExtensions](#). The event model is similar to the web events model, but a bit different — event listeners properties are camel-cased (e.g. `onMessage` rather than `onmessage`), and need to be combined with the `addListener` function. See the [runtime.onMessage page](#) for an example.

You don't need to understand anything about other such environments at this stage in your learning; we just wanted to make it clear that events can differ in different programming environments.

## Ways of using web events

There are a number of different ways in which you can add event listener code to web pages so that it will be run when the associated event fires. In this section we will review the different mechanisms and discuss which ones you should use.

### Event handler properties

These are the properties that exist to contain event handler code that we have seen most frequently during the course. Returning to the above example:

```
var btn = document.querySelector('button');

btn.onclick = function() {
 var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) +
')';
 document.body.style.backgroundColor = rndCol;
}
```

The `onclick` property is the event handler property being used in this situation. It is essentially a property like any other available on the button (e.g. `btn.textContent`, or `btn.style`), but it is a special type — when you set it to be equal to some code, that code will be run when the event fires on the button.

You could also set the handler property to be equal to a named function name (like we saw in [Build your own function](#)). The following would work just the same:

```
var btn = document.querySelector('button');

function bgChange() {
```

```

var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) +
')';
document.body.style.backgroundColor = rndCol;
}

btn.onclick = bgChange;

```

There are many different event handler properties available. Let's do an experiment.

First of all, make a local copy of [random-color-eventhandlerproperty.html](#), and open it in your browser. It's just a copy of the simple random color example we've been playing with already in this article. Now try changing `btn.onclick` to the following different values in turn, and observing the results in the example:

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

Some events are very general and available nearly anywhere (for example an `onclick` handler can be registered on nearly any element), whereas some are more specific and only useful in certain situations (for example it makes sense to use `onplay` only on specific elements, such as `<video>`).

## Inline event handlers — don't use these

You might also see a pattern like this in your code:

```

<button onclick="bgChange () ">Press me</button>
function bgChange() {
 var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) +
')';
 document.body.style.backgroundColor = rndCol;
}

```

**Note:** You can find the [full source code](#) for this example on GitHub (also [see it running live](#)).

The earliest method of registering event handlers found on the Web involved **event handler HTML attributes** (aka **inline event handlers**) like the one shown above — the attribute value is

literally the JavaScript code you want to run when the event occurs. The above example invokes a function defined inside a `<script>` element on the same page, but you could also insert JavaScript directly inside the attribute, for example:

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

You'll find HTML attribute equivalents for many of the event handler properties; however, you shouldn't use these — they are considered bad practice. It might seem easy to use an event handler attribute if you are just doing something really quick, but they very quickly become unmanageable and inefficient.

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

Even in a single file, inline event handlers are not a good idea. One button is OK, but what if you had 100 buttons? You'd have to add 100 attributes to the file; it would very quickly turn into a maintenance nightmare. With JavaScript, you could easily add an event handler function to all the buttons on the page no matter how many there were, using something like this:

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

for (var i = 0; i < buttons.length; i++) {
 buttons[i].onclick = bgChange;
}
```

**Note:** Separating your programming logic from your content also makes your site more friendly to search engines.

### `addEventListener()` and `removeEventListener()`

The newest type of event mechanism is defined in the [Document Object Model \(DOM\) Level 2 Events](#) Specification, which provides browsers with a new function — [`addEventListener\(\)`](#). This functions in a similar way to the event handler properties, but the syntax is obviously different. We could rewrite our random color example to look like this:

```
var btn = document.querySelector('button');

function bgChange() {
 var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) +
')';
 document.body.style.backgroundColor = rndCol;
}

btn.addEventListener('click', bgChange);
```

**Note:** You can find the [full source code](#) for this example on GitHub (also [see it running live](#)).

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

```
btn.addEventListener('click', function() {
 var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) +
')';
 document.body.style.backgroundColor = rndCol;
});
```

This mechanism has some advantages over the older mechanisms discussed earlier. For a start, there is a counterpart function, [removeEventListener\(\)](#), which removes a previously added listener. For example, this would remove the listener set in the first code block in this section:

```
btn.removeEventListener('click', bgChange);
```

This isn't significant for simple, small programs, but for larger, more complex programs it can improve efficiency to clean up old unused event handlers. Plus, for example, this allows you to have the same button performing different actions in different circumstances — all you've got to do is add/remove event handlers as appropriate.

Second, you can also register multiple handlers for the same listener. The following two handlers would not be applied:

```
myElement.onclick = functionA;
myElement.onclick = functionB;
```

As the second line would overwrite the first value of `onclick` set. This would work, however:

```
myElement.addEventListener('click', functionA);
myElement.addEventListener('click', functionB);
```

Both functions would now run when the element is clicked.

In addition, there are a number of other powerful features and options available with this event mechanism. These are a little out of scope for this article, but if you want to read up on them, have a look at the [addEventListener\(\)](#) and [removeEventListener\(\)](#) reference pages.

### What mechanism should I use?

Of the three mechanisms, you definitely shouldn't use the HTML event handler attributes — these are outdated, and bad practice, as mentioned above.

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

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

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

```
element.onclick = function1;
element.onclick = function2;
etc.
```

**Note:** If you are called upon to support browsers older than Internet Explorer 8 in your work, you may run into difficulties, as such ancient browsers use different event models from newer browsers. But never fear, most JavaScript libraries (for example `jQuery`) have built in functions that abstract away cross browser differences. Don't worry about this too much at this stage in your learning journey.

## Other event concepts

In this section we will briefly cover some advanced concepts that are relevant to events. It is not important to understand these fully at this point, but it might serve to explain some code patterns you'll likely come across from time to time.

### Event objects

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

```
function bgChange(e) {
 var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) +
')';
 e.target.style.backgroundColor = rndCol;
 console.log(e);
}

btn.addEventListener('click', bgChange);
```

**Note:** You can find the [full source code](#) for this example on GitHub (also [see it running live](#)).

Here you can see that we are including an event object, `e`, in the function, and in the function setting a background color style on `e.target` — which is the button itself. The `target` property of the event object is always a reference to the element that the event has just occurred upon. So in this example we are setting a random background color on the button, not the page.

**Note:** You can use any name you like for the event object — you just need to choose a name that you can then use to reference it inside the event handler function. `e/evt/event` are most commonly used by developers because they are short and easy to remember. It's always good to stick to a standard.

`e.target` is incredibly useful when you want to set the same event handler on multiple elements, and do something to all of them when an event occurs on them. You might for example have a set of 16 tiles that disappear when they are clicked on. It is useful to always be able to just set the thing to disappear as `e.target`, rather than having to select it in some more difficult way. In the following example (see [useful-eventtarget.html](#) for the full source code; also see it [running live here](#)), we create 16 `<div>` elements using JavaScript. We then select all of them using `document.querySelectorAll()`, then loop through each one, adding an `onclick` handler to each that makes it so that a random color is applied to each one when clicked:

```
var divs = document.querySelectorAll('div');

for (var i = 0; i < divs.length; i++) {
 divs[i].onclick = function(e) {
 e.target.style.backgroundColor = bgChange();
 }
}
```

The output is as follows (try clicking around on it — have fun):

Most event handlers you'll encounter just have a standard set of properties and functions (methods) available on the event object (see the [Event](#) object reference for a full list). Some more advanced handlers however add specialist properties containing extra data that they need to function. The [Media Recorder API](#) for example has a `dataavailable` event, which fires when some audio or video has been recorded and is available for doing something with (for example saving it, or playing it back). The corresponding `ondataavailable` handler's event object has a `data` property available containing the recorded audio or video data to allow you to access it and do something with it.

## Preventing default behavior

Sometimes, you'll come across a situation where you want to stop an event doing what it does by default. The most common example is that of a web form, for example, a custom registration form. When you fill in the details and press the submit button, the natural behaviour is for the data to be submitted to a specified page on the server for processing, and the browser to be redirected to a "success message" page of some kind (or the same page, if another is not specified.)

The trouble comes when the user has not submitted the data correctly — as a developer, you'll want to stop the submission to the server and give them an error message telling them what's wrong and what needs to be done to put things right. Some browsers support automatic form data validation features, but since many don't, you are advised to not rely on those, and implement your own validation checks. Let's look at a simple example.

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

```
<form>
 <div>
 <label for="fname">First name: </label>
 <input id="fname" type="text">
 </div>
 <div>
 <label for="lname">Last name: </label>
 <input id="lname" type="text">
 </div>
 <div>
 <input id="submit" type="submit">
 </div>
</form>
<p></p>
```

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

```
var form = document.querySelector('form');
var fname = document.getElementById('fname');
var lname = document.getElementById('lname');
var submit = document.getElementById('submit');
var para = document.querySelector('p');

form.onsubmit = function(e) {
 if (fname.value === '' || lname.value === '') {
 e.preventDefault();
 para.textContent = 'You need to fill in both names!';
 }
}
```

Obviously, this is pretty weak form validation — it wouldn't stop the user validating the form with spaces or numbers entered into the fields, for example — but it is ok for example purposes. The output is as follows:

**Note:** for the full source code, see [preventdefault-validation.html](#) (also see it [running live](#) here.)

## Event bubbling and capture

The final subject to cover here is something that you'll not come across often, but it can be a real pain if you don't understand it. Event bubbling and capture are two mechanisms that describe what happens when two handlers of the same event type are activated on one element. Let's look at an example to make this easier — open up the [show-video-box.html](#) example in a new tab (and the [source code](#) in another tab.) It is also available live below:

This is a pretty simple example that shows and hides a `<div>` with a `<video>` element inside it:

```
<button>Display video</button>

<div class="hidden">
 <video>
 <source src="rabbit320.mp4" type="video/mp4">
 <source src="rabbit320.webm" type="video/webm">
 <p>Your browser doesn't support HTML5 video. Here is a link to the video instead.</p>
 </video>
</div>
```

When the `<button>` is clicked, the video is displayed, by changing the class attribute on the `<div>` from `hidden` to `showing` (the example's CSS contains these two classes, which position the box off the screen and on the screen, respectively):

```
btn.onclick = function() {
 videoBox.setAttribute('class', 'showing');
}
```

We then add a couple more `onclick` event handlers — the first one to the `<div>` and the second one to the `<video>`. The idea is that when the area of the `<div>` outside the video is clicked, the box should be hidden again; when the video itself is clicked, the video should start to play.

```
videoBox.onclick = function() {
 videoBox.setAttribute('class', 'hidden');
};

video.onclick = function() {
 video.play();
};
```

But there's a problem — currently when you click the video it starts to play, but it causes the `<div>` to also be hidden at the same time. This is because the video is inside the `<div>` — it is part of it — so clicking on the video actually runs *both* the above event handlers.

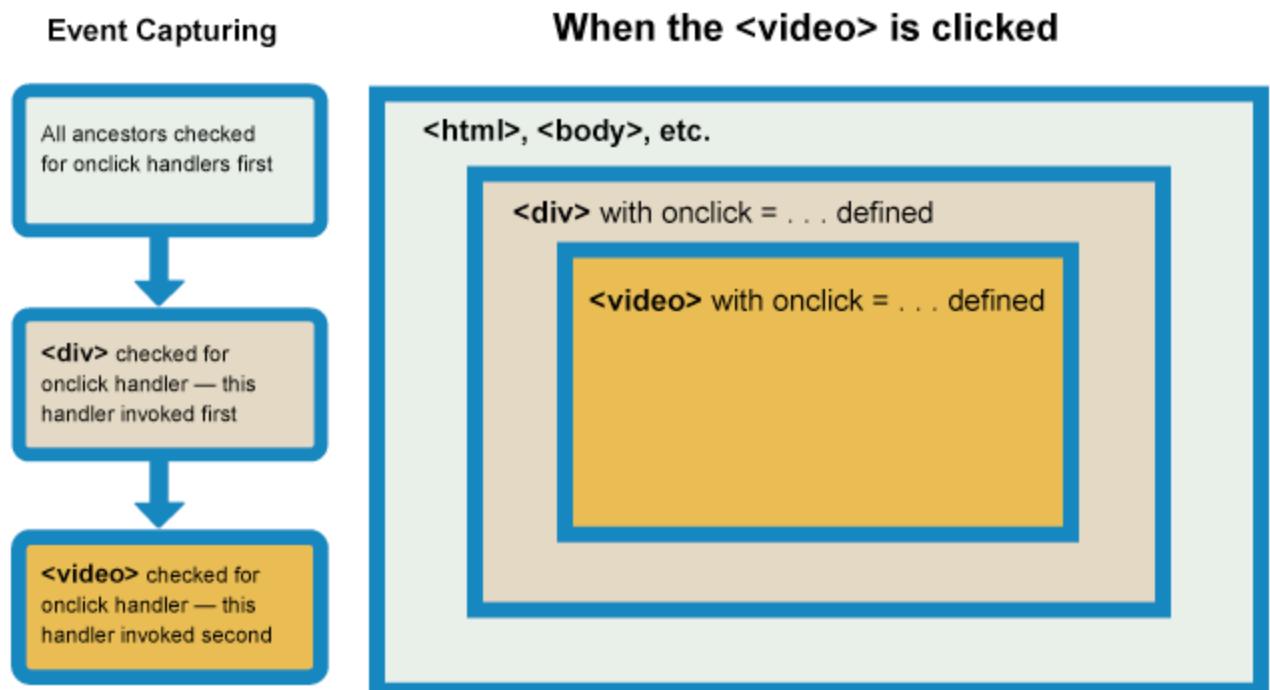
## Bubbling and capturing explained

When an event is fired on an element that has parent elements (e.g. the `<video>` in our case), modern browsers run two different phases — the capturing phase and the bubbling phase. In the capturing phase:

- The browser checks to see if the element's outer-most ancestor (`<html>`) has an `onclick` event handler registered on it in the capturing phase, and runs it if so.
- Then it moves on to the next element inside `<html>` and does the same thing, then the next one, and so on until it reaches the element that was actually clicked on.

In the bubbling phase, the exact opposite occurs:

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



(Click on image for bigger diagram)

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

- It finds the `video.onclick...` handler and runs it, so the video first starts playing.
- It then finds the `videoBox.onclick...` handler and runs it, so the video is hidden as well.

## Fixing the problem with `stopPropagation()`

This is annoying behavior, but there is a way to fix it! The standard event object has a function available on it called [`stopPropagation\(\)`](#), which when invoked on a handler's event object makes it so that handler is run, but the event doesn't bubble any further up the chain, so no more handlers will be run.

We can therefore fix our current problem by changing the second handler function in the previous code block to this:

```
video.onclick = function(e) {
 e.stopPropagation();
 video.play();
};
```

You can try making a local copy of the [show-video-box.html source code](#) and having a go at fixing it yourself, or looking at the fixed result in [show-video-box-fixed.html](#) (also see the [source code](#) here).

**Note:** Why bother with both capturing and bubbling? Well, in the bad old days when browsers were much less cross-compatible than they are now, Netscape used only event capturing, and Internet Explorer used only event bubbling. When the W3C decided to try to standardize the behavior and reach a consensus, they ended up with this system that included both, which is the one modern browsers implemented.

**Note:** As mentioned above, by default all event handlers are registered in the bubbling phase, and this makes more sense most of the time. If you really want to register an event in the capturing phase instead, you can do so by registering your handler using [`addEventListener\(\)`](#), and setting the optional third property to `true`.

## Event delegation

Bubbling also allows us to take advantage of **event delegation** — this concept relies on the fact that if you want some code to run when you click on any one of a large number of child elements, you can set the event listener on their parent and have the effect of the event listener bubble to each child, rather than having to set the event listener on every child individually.

A good example is a series of list items — if you want each one of them to pop up a message when clicked, you can set the `click` event listener on the parent `<ul>`, and it will bubble to the list items.

This concept is explained further on David Walsh's blog, with multiple examples — see [How JavaScript Event Delegation Works](#).

## Conclusion

You should now know all you need to know about web events at this early stage. As mentioned above, events are not really part of the core JavaScript — they are defined in browser Web APIs.

Also, it is important to understand that the different contexts that JavaScript is used in tend to have different event models — from Web APIs to other areas such as browser WebExtensions and Node.js (server-side JavaScript). We are not expecting you to understand all these areas now, but it certainly helps to understand the basics of events as you forge ahead with learning web development.

If there is anything you didn't understand, feel free to read through the article again, or [contact us](#) to ask for help.

## Intermediate

### Introducing JavaScript objects

In JavaScript, most things are objects, from core JavaScript features like strings and arrays to the browser APIs built on top of JavaScript. You can even create your own objects to encapsulate related functions and variables into efficient packages, and act as handy data containers. The object-based nature of JavaScript is important to understand if you want to go further with your knowledge of the language, therefore we've provided this module to help you. Here we teach object theory and syntax in detail, then look at how to create your own objects.

### JavaScript object basics

In this article, we'll look at fundamental JavaScript object syntax, and revisit some JavaScript features that we've already seen earlier in the course, reiterating the fact that many of the features you've already dealt with are objects.

**Prerequisites:** Basic computer literacy, a basic understanding of HTML and CSS, familiarity with JavaScript basics (see [First steps](#) and [Building blocks](#)).

**Objective:** To understand the basic theory behind object-oriented programming, how this relates to JavaScript ("most things are objects"), and how to start working with JavaScript objects.

## Object basics

An object is a collection of related data and/or functionality (which usually consists of several variables and functions — which are called properties and methods when they are inside objects.) Let's work through an example to understand what they look like.

To begin with, make a local copy of our [oojs.html](#) file. This contains very little — a `<script>` element for us to write our source code into. We'll use this as a basis for exploring basic object syntax. While working with this example you should have your [developer tools JavaScript console](#) open and ready to type in some commands.

As with many things in JavaScript, creating an object often begins with defining and initializing a variable. Try entering the following below the JavaScript code that's already in your file, then saving and refreshing:

```
var person = {};
```

If you enter `person` into your JS console and press the button, you should get the following result:

```
[object Object]
```

Congratulations, you've just created your first object. Job done! But this is an empty object, so we can't really do much with it. Let's update our object to look like this:

```
var person = {
 name: ['Bob', 'Smith'],
 age: 32,
 gender: 'male',
 interests: ['music', 'skiing'],
 bio: function() {
 alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years
old. He likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');
 },
 greeting: function() {
 alert('Hi! I\'m ' + this.name[0] + '.');
 }
};
```

After saving and refreshing, try entering some of the following into your JS console:

```
person.name[0]
person.age
person.interests[1]
person.bio()
```

```
person.greeting()
```

You have now got some data and functionality inside your object, and are now able to access them with some nice simple syntax!

**Note:** If you are having trouble getting this to work, try comparing your code against our version — see [ojs-finished.html](#) (also [see it running live](#)). One common mistake when you are starting out with objects is to put a comma on the end of the last member — this will cause an error.

So what is going on here? Well, an object is made up of multiple members, each of which has a name (e.g. `name` and `age` above), and a value (e.g. `['Bob', 'Smith']` and `32`). Each name/value pair must be separated by a comma, and the name and value in each case are separated by a colon. The syntax always follows this pattern:

```
var objectName = {
 member1Name: member1Value,
 member2Name: member2Value,
 member3Name: member3Value
}
```

The value of an object member can be pretty much anything — in our `person` object we've got a string, a number, two arrays, and two functions. The first four items are data items, and are referred to as the object's **properties**. The last two items are functions that allow the object to do something with that data, and are referred to as the object's **methods**.

An object like this is referred to as an **object literal** — we've literally written out the object contents as we've come to create it. This is in contrast to objects instantiated from classes, which we'll look at later on.

It is very common to create an object using an object literal when you want to transfer a series of structured, related data items in some manner, for example sending a request to the server to be put into a database. Sending a single object is much more efficient than sending several items individually, and it is easier to work with than an array, when you want to identify individual items by name.

## Dot notation

Above, you accessed the object's properties and methods using **dot notation**. The object name (`person`) acts as the **namespace** — it must be entered first to access anything **encapsulated** inside the object. Next you write a dot, then the item you want to access — this can be the name of a simple property, an item of an array property, or a call to one of the object's methods, for example:

```
person.age
person.interests[1]
person.bio()
```

## Sub-namespaces

It is even possible to make the value of an object member another object. For example, try changing the name member from

```
name: ['Bob', 'Smith'],
```

to

```
name : {
 first: 'Bob',
 last: 'Smith'
},
```

Here we are effectively creating a **sub-namespace**. This sounds complex, but really it's not — to access these items you just need to chain the extra step onto the end with another dot. Try these in the JS console:

```
person.name.first
person.name.last
```

**Important:** At this point you'll also need to go through your method code and change any instances of

```
name[0]
name[1]
```

to

```
name.first
name.last
```

Otherwise your methods will no longer work.

## Bracket notation

There is another way to access object properties — using bracket notation. Instead of using these:

```
person.age
person.name.first
```

You can use

```
person['age']
person['name']['first']
```

This looks very similar to how you access the items in an array, and it is basically the same thing — instead of using an index number to select an item, you are using the name associated with each member's value. It is no wonder that objects are sometimes called **associative arrays** — they map strings to values in the same way that arrays map numbers to values.

## Setting object members

So far we've only looked at retrieving (or **getting**) object members — you can also **set** (update) the value of object members by simply declaring the member you want to set (using dot or bracket notation), like this:

```
person.age = 45;
person['name']['last'] = 'Cratchit';
```

Try entering the above lines, and then getting the members again to see how they've changed, like so:

```
person.age
person['name']['last']
```

Setting members doesn't just stop at updating the values of existing properties and methods; you can also create completely new members. Try these in the JS console:

```
person['eyes'] = 'hazel';
person.farewell = function() { alert("Bye everybody!"); }
```

You can now test out your new members:

```
person['eyes']
person.farewell()
```

One useful aspect of bracket notation is that it can be used to set not only member values dynamically, but member names too. Let's say we wanted users to be able to store custom value types in their people data, by typing the member name and value into two text inputs? We could get those values like this:

```
var myDataName = nameInput.value;
var myDataValue = nameValue.value;
```

we could then add this new member name and value to the `person` object like this:

```
person[myDataName] = myDataValue;
```

To test this, try adding the following lines into your code, just below the closing curly brace of the `person` object:

```
var myDataName = 'height';
var myDataValue = '1.75m';
```

```
person[myDataName] = myDataValue;
```

Now try saving and refreshing, and entering the following into your text input:

```
person.height
```

Adding a property to an object using the method above isn't possible with dot notation, which can only accept a literal member name, not a variable value pointing to a name.

## What is "this"?

You may have noticed something slightly strange in our methods. Look at this one for example:

```
greeting: function() {
 alert('Hi! I\'m ' + this.name.first + '.');
}
```

You are probably wondering what "this" is. The `this` keyword refers to the current object the code is being written inside — so in this case `this` is equivalent to `person`. So why not just write `person` instead? As you'll see in the [Object-oriented JavaScript for beginners](#) article when we start creating constructors, etc., `this` is very useful — it will always ensure that the correct values are used when a member's context changes (e.g. two different `person` object instances may have different names, but will want to use their own name when saying their greeting).

Let's illustrate what we mean with a simplified pair of person objects:

```
var person1 = {
 name: 'Chris',
 greeting: function() {
 alert('Hi! I\'m ' + this.name + '.');
 }
}

var person2 = {
 name: 'Brian',
 greeting: function() {
 alert('Hi! I\'m ' + this.name + '.');
 }
}
```

In this case, `person1.greeting()` will output "Hi! I'm Chris."; `person2.greeting()` on the other hand will output "Hi! I'm Brian.", even though the method's code is exactly the same in each case. As we said earlier, `this` is equal to the object the code is inside — this isn't hugely useful when you are writing out object literals by hand, but it really comes into its own when you are dynamically generating objects (for example using constructors). It will all become clearer later on.

## You've been using objects all along

As you've been going through these examples, you have probably been thinking that the dot notation you've been using is very familiar. That's because you've been using it throughout the course! Every time we've been working through an example that uses a built-in browser API or JavaScript object, we've been using objects, because such features are built using exactly the same kind of object structures that we've been looking at here, albeit more complex ones than our own basic custom examples.

So when you used string methods like:

```
myString.split(',') ;
```

You were using a method available on an instance of the [String](#) class. Every time you create a string in your code, that string is automatically created as an instance of `String`, and therefore has several common methods/properties available on it.

When you accessed the document object model using lines like this:

```
var myDiv = document.createElement('div');
var myVideo = document.querySelector('video');
```

You were using methods available on an instance of the [Document](#) class. For each webpage loaded, an instance of `Document` is created, called `document`, which represents the entire page's structure, content, and other features such as its URL. Again, this means that it has several common methods/properties available on it.

The same is true of pretty much any other built-in object/API you've been using — [Array](#), [Math](#), etc.

Note that built in Objects/APIs don't always create object instances automatically. As an example, the [Notifications API](#) — which allows modern browsers to fire system notifications — requires you to instantiate a new object instance using the constructor for each notification you want to fire. Try entering the following into your JavaScript console:

```
var myNotification = new Notification('Hello!');
```

Again, we'll look at constructors in a later article.

**Note:** It is useful to think about the way objects communicate as **message passing** — when an object needs another object to perform some kind of action often it will send a message to another object via one of its methods, and wait for a response, which we know as a return value.

## Summary

Congratulations, you've reached the end of our first JS objects article — you should now have a good idea of how to work with objects in JavaScript — including creating your own simple objects. You should also appreciate that objects are very useful as structures for storing related

data and functionality — if you tried to keep track of all the properties and methods in our `person` object as separate variables and functions, it would be inefficient and frustrating, and we'd run the risk of clashing with other variables and functions that have the same names. Objects let us keep the information safely locked away in their own package, out of harm's way.

In the next article we'll start to look at object-oriented programming (OOP) theory, and how such techniques can be used in JavaScript.

## Object-oriented JavaScript for beginners

With the basics out of the way, we'll now focus on object-oriented JavaScript (OOJS) — this article presents a basic view of object-oriented programming (OOP) theory, then explores how JavaScript emulates object classes via constructor functions, and how to create object instances.

Basic computer literacy, a basic understanding of HTML and CSS, familiarity with

**Prerequisites:** JavaScript basics (see [First steps](#) and [Building blocks](#)) and OOJS basics (see [Introduction to objects](#)).

To understand the basic theory behind object-oriented programming, how this relates to

**Objective:** JavaScript ("everything is an object"), and how to create constructors and object instances.

## Object-oriented programming — the basics

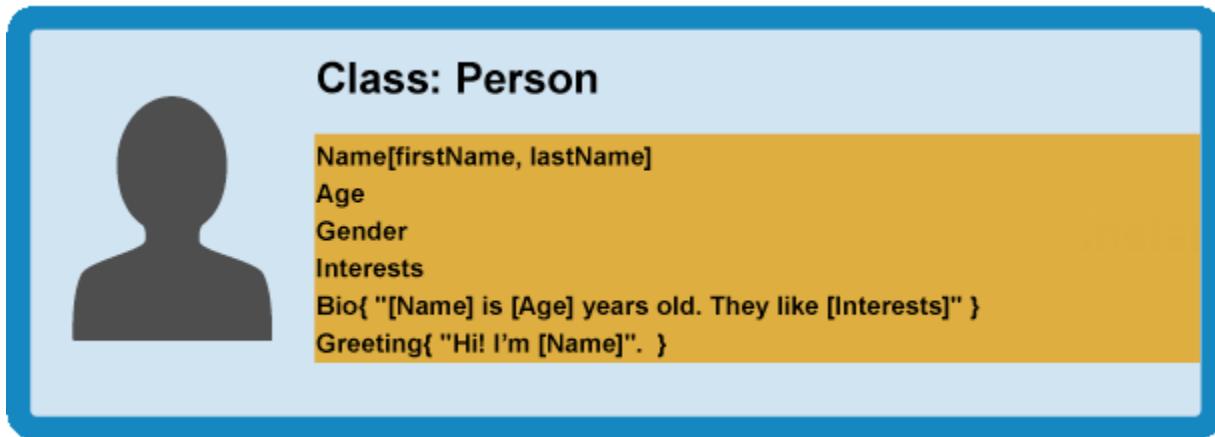
To start with, let's give you a simplistic, high-level view of what Object-oriented programming (OOP) is. We say simplistic, because OOP can quickly get very complicated, and giving it a full treatment now would probably confuse more than help. The basic idea of OOP is that we use objects to model real world things that we want to represent inside our programs, and/or provide a simple way to access functionality that would otherwise be hard or impossible to make use of.

Objects can contain related data and code, which represent information about the thing you are trying to model, and functionality or behavior that you want it to have. Object data (and often, functions too) can be stored neatly (the official word is **encapsulated**) inside an object package (which can be given a specific name to refer to, which is sometimes called a **namespace**), making it easy to structure and access; objects are also commonly used as data stores that can be easily sent across the network.

### Defining an object template

Let's consider a simple program that displays information about the students and teachers at a school. Here we'll look at OOP theory in general, not in the context of any specific programming language.

To start this off, we could return to our Person object type from our [first objects article](#), which defines the generic data and functionality of a person. There are lots of things you *could* know about a person (their address, height, shoe size, DNA profile, passport number, significant personality traits ...), but in this case we are only interested in showing their name, age, gender, and interests, and we also want to be able to write a short introduction about them based on this data, and get them to say hello. This is known as **abstraction** — creating a simple model of a more complex thing, which represents its most important aspects in a way that is easy to work with for our program's purposes.



In some OOP languages, this generic object type definition is called a **class** (JavaScript uses a different mechanism and terminology, as you'll see below) — it isn't actually an object, rather it is a template that defines what characteristics an object should have.

### Creating actual objects

From our class, we can create **object instances** — objects that contain the data and functionality defined in the class. From our Person class, we can now create some actual people:

## Class: Person



```
Name(firstName, lastName)
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]" }
Greeting{ "Hi! I'm [Name]" }
```

Instantiation

## Object: person1



```
Name[Bob, Smith]
Age: 32
Gender: Male
Interests: Music, Skiing
Bio{ "Bob Smith is 32 years old. He likes Music and Skiing." }
Greeting{ "Hi! I'm Bob." }
```

## Object: person2

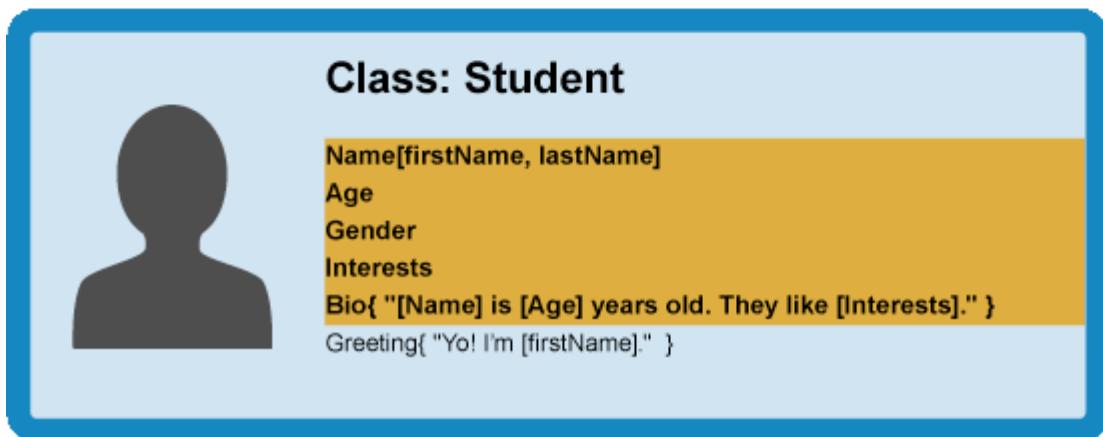
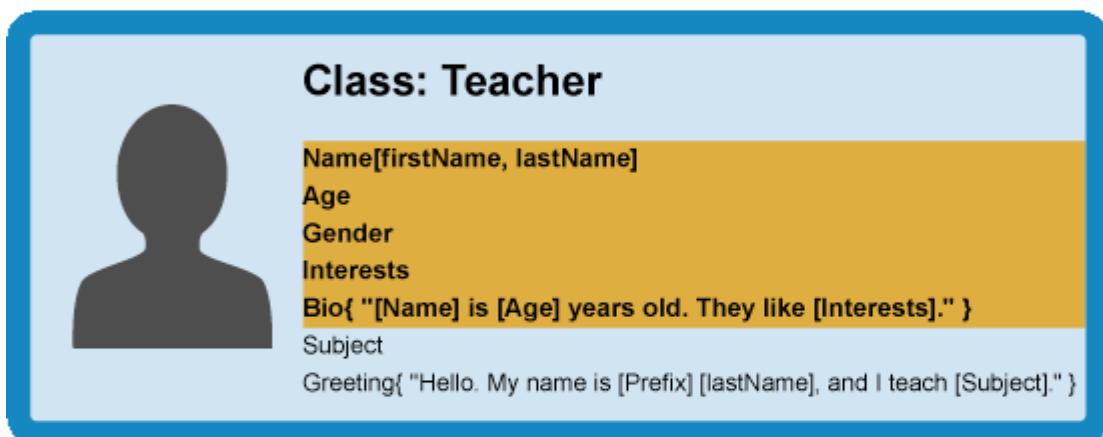
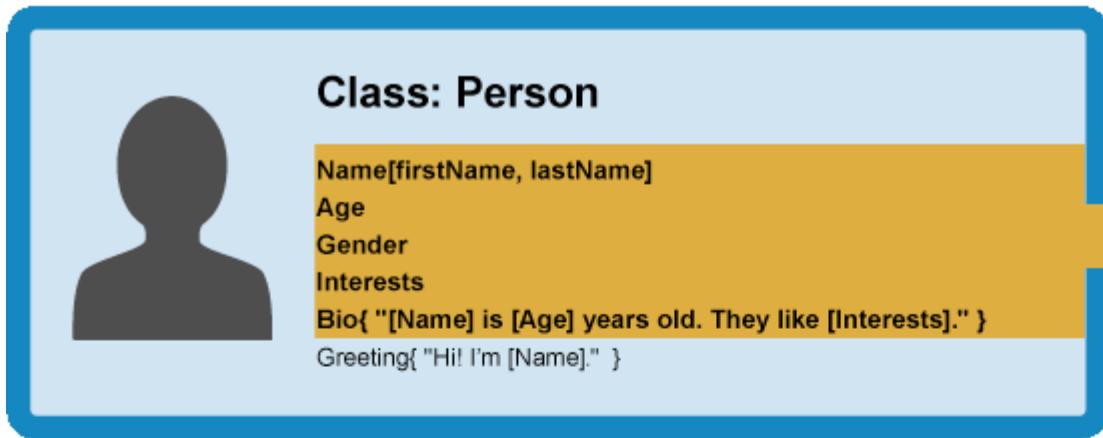


```
Name[Diana, Cope]
Age: 28
Gender: Female
Interests: Kickboxing, Brewing
Bio{ "Diana Cope is 28 years old. She likes Kickboxing and Brewing." }
Greeting{ "Hi! I'm Diana." }
```

When an object instance is created from a class, the class's **constructor function** is run to create it. This process of creating an object instance from a class is called **instantiation** — the object instance is **instantiated** from the class.

## Specialist classes

In this case we don't want generic people — we want teachers and students, which are both more specific types of people. In OOP, we can create new classes based on other classes — these new **child classes** can be made to **inherit** the data and code features of their **parent class**, so you can reuse functionality common to all the object types rather than having to duplicate it. Where functionality differs between classes, you can define specialized features directly on them as needed.

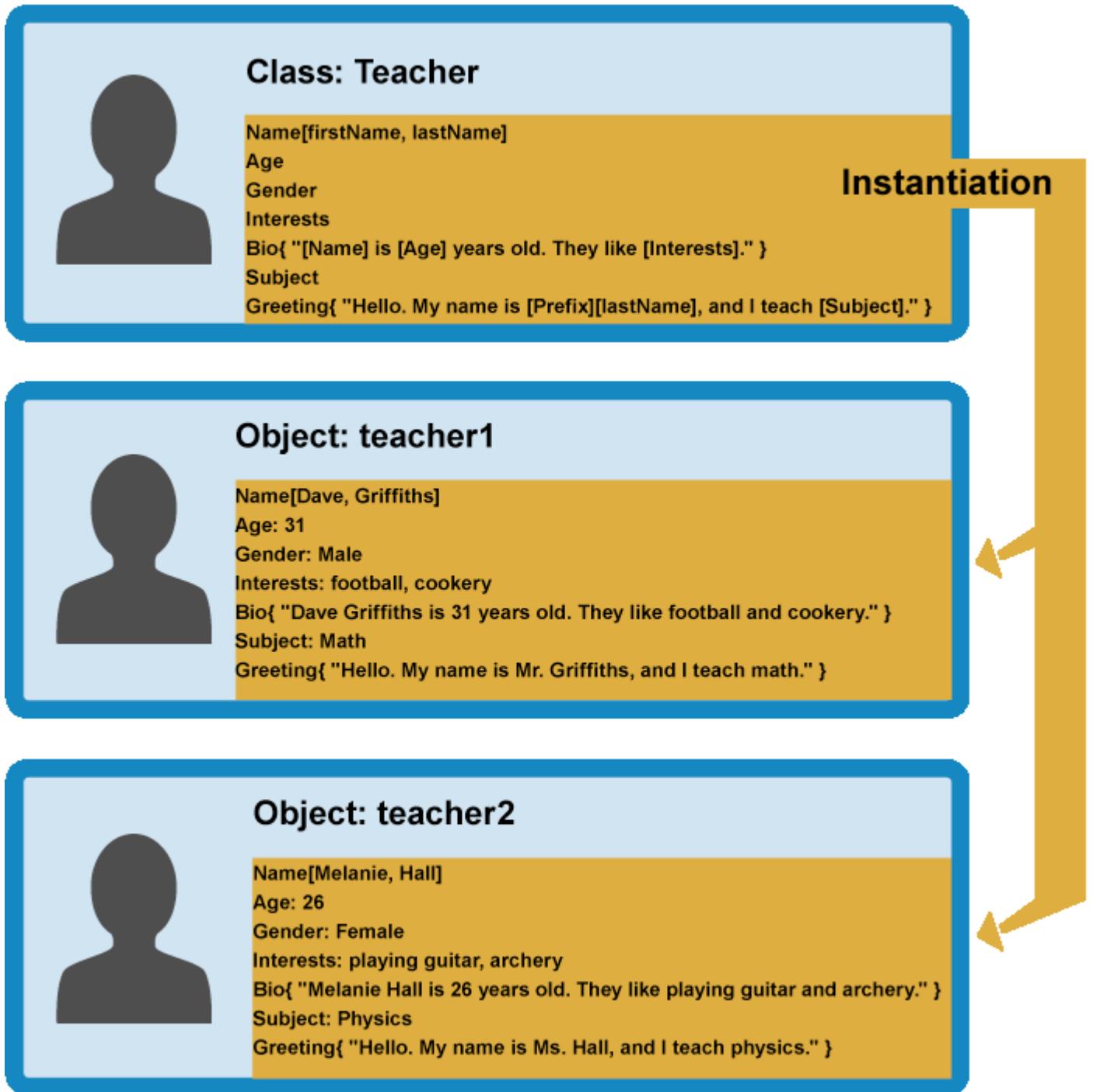


This is really useful — teachers and students share many common features such as name, gender, and age, so it is convenient to only have to define those features once. You can also define the same feature separately in different classes, as each definition of that feature will be in a different namespace. For example, a student's greeting might be of the form "Yo, I'm [firstName]" (e.g *Yo, I'm Sam*), whereas a teacher might use something more formal, such as "Hello, my name is

[Prefix] [lastName], and I teach [Subject]." (e.g *Hello, My name is Mr Griffiths, and I teach Chemistry*).

**Note:** The fancy word for the ability of multiple object types to implement the same functionality is **polymorphism**. Just in case you were wondering.

You can now create object instances from your child classes. For example:



In the rest of the article, we'll start to look at how OOP theory can be put into practice in JavaScript.

## Constructors and object instances

Some people argue that JavaScript is not a true object-oriented language — for example, its `class` statement is just syntactical sugar over existing prototypical inheritance and is not a `class` in a traditional sense. JavaScript, uses special functions called **constructor functions** to define objects and their features. They are useful because you'll often come across situations in which you don't know how many objects you will be creating; constructors provide the means to create as many objects as you need in an effective way, attaching data and functions to them as required.

When a new object instance is created from a constructor function, its core functionality (as defined by its prototype, which we'll explore in the article [Object prototypes](#)) is not all copied over to the new object like "classic" OO languages — instead the functionality is linked to via a reference chain called a **prototype chain**. So this is not true instantiation, strictly speaking — JavaScript uses a different mechanism to share functionality between objects.

**Note:** Not being "classic OOP" is not necessarily a bad thing; as mentioned above, OOP can get very complex very quickly, and JavaScript has some nice ways to take advantage of OO features without having to get too deep into it.

Let's explore creating classes via constructors and creating object instances from them in JavaScript. First of all, we'd like you to make a new local copy of the [ojs.html](#) file we saw in our first Objects article.

## A simple example

1. Let's start by looking at how you could define a person with a normal function. Add this function within the `script` element:

```
2. function createNewPerson(name) {
3. var obj = {};
4. obj.name = name;
5. obj.greeting = function() {
6. alert('Hi! I\'m ' + this.name + '.');
7. };
8. return obj;
 }
```

- You can now create a new person by calling this function — try the following lines in your browser's JavaScript console:

```
var salva = createNewPerson('Salva');
salva.name;
salva.greeting();
```

- This works well enough, but it is a bit long-winded; if we know we want to create an object, why do we need to explicitly create a new empty object and return it? Fortunately JavaScript provides us with a handy shortcut, in the form of constructor functions — let's make one now!

- Replace your previous function with the following:

```
function Person(name) {
 this.name = name;
 this.greeting = function() {
 alert('Hi! I\'m ' + this.name + '.');
```

```
};
}
```

3.

The constructor function is JavaScript's version of a class. You'll notice that it has all the features you'd expect in a function, although it doesn't return anything or explicitly create an object — it basically just defines properties and methods. You'll see the `this` keyword being used here as well — it is basically saying that whenever one of these object instances is created, the object's `name` property will be equal to the name value passed to the constructor call, and the `greeting()` method will use the name value passed to the constructor call too.

**Note:** A constructor function name usually starts with a capital letter — this convention is used to make constructor functions easier to recognize in code.

So how do we call a constructor to create some objects?

1. Add the following lines below your previous code addition:

```
2. var person1 = new Person('Bob');
 var person2 = new Person('Sarah');
```

• • Save your code and reload it in the browser, and try entering the following lines into your JS console:

```
person1.name
person1.greeting()
person2.name
person2.greeting()
```

2.

Cool! You'll now see that we have two new objects on the page, each of which is stored under a different namespace — when you access their properties and methods, you have to start calls with `person1` or `person2`; they are neatly packaged away so they won't clash with other functionality. They do, however, have the same `name` property and `greeting()` method available. Note that they are using their own `name` value that was assigned to them when they were created; this is one reason why it is very important to use `this`, so they will use their own values, and not some other value.

Let's look at the constructor calls again:

```
var person1 = new Person('Bob');
var person2 = new Person('Sarah');
```

In each case, the `new` keyword is used to tell the browser we want to create a new object instance, followed by the function name with its required parameters contained in parentheses, and the result is stored in a variable — very similar to how a standard function is called. Each instance is created according to this definition:

```

function Person(name) {
 this.name = name;
 this.greeting = function() {
 alert('Hi! I\'m ' + this.name + '.');
 };
}

```

After the new objects have been created, the `person1` and `person2` variables contain the following objects:

```

{
 name: 'Bob',
 greeting: function() {
 alert('Hi! I\'m ' + this.name + '.');
 }
}

{
 name: 'Sarah',
 greeting: function() {
 alert('Hi! I\'m ' + this.name + '.');
 }
}

```

Note that when we are calling our constructor function, we are defining `greeting()` every time, which isn't ideal. To avoid this, we can define functions on the prototype instead, which we will look at later.

## Creating our finished constructor

The example we looked at above was only a simple example to get us started. Let's now get on and create our final `Person()` constructor function.

1. Remove the code you inserted so far, and add in this replacement constructor — this is exactly the same as the simple example in principle, with just a bit more complexity:

```

2. function Person(first, last, age, gender, interests) {
3. this.name = {
4. 'first': first,
5. 'last' : last
6. };
7. this.age = age;
8. this.gender = gender;
9. this.interests = interests;
10. this.bio = function() {
11. alert(this.name.first + ' ' + this.name.last + ' is ' + this.age +
12. ' years old. He likes ' + this.interests[0] + ' and ' +
13. this.interests[1] + '.');
14. };
15. this.greeting = function() {
16. alert('Hi! I\'m ' + this.name.first + '.');
17. };
}

```

- Now add in the following line below it, to create an object instance from it:

```
var person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

2.

You'll now see that you can access the properties and methods just like we did previously — try these in your JS console:

```
person1['age']
person1.interests[1]
person1.bio()
// etc.
```

**Note:** If you are having trouble getting this to work, try comparing your code against our version — see [ojs-class-finished.html](#) (also [see it running live](#)).

### Further exercises

To start with, try adding a couple more object creation lines of your own, and try getting and setting the members of the resulting object instances.

In addition, there are a couple of problems with our `bio()` method — the output always includes the pronoun "He", even if your person is female, or some other preferred gender classification. And the bio will only include two interests, even if more are listed in the `interests` array. Can you work out how to fix this in the class definition (constructor)? You can put any code you like inside a constructor (you'll probably need a few conditionals and a loop). Think about how the sentences should be structured differently depending on gender, and depending on whether the number of listed interests is 1, 2, or more than 2.

**Note:** If you get stuck, we have provided an [answer inside our GitHub repo \(see it live\)](#) — try writing it yourself first though!

## Other ways to create object instances

So far we've seen two different ways to create an object instance — [declaring an object literal](#), and using a constructor function (see above).

These make sense, but there are other ways — we want to make you familiar with these in case you come across them in your travels around the Web.

### The `Object()` constructor

First of all, you can use the `Object()` constructor to create a new object. Yes, even generic objects have a constructor, which generates an empty object.

1. Try entering this into your browser's JavaScript console:

```
var person1 = new Object();
```

- • This stores an empty object in the `person1` variable. You can then add properties and methods to this object using dot or bracket notation as desired; try these examples in your console:

```
person1.name = 'Chris';
person1['age'] = 38;
person1.greeting = function() {
 alert('Hi! I\'m ' + this.name + '.');
};
```

- • You can also pass an object literal to the `Object()` constructor as a parameter, to prefill it with properties/methods. Try this in your JS console:

```
var person1 = new Object({
 name: 'Chris',
 age: 38,
 greeting: function() {
 alert('Hi! I\'m ' + this.name + '.');
}
});
```

3.

## Using the `create()` method

Constructors can help you give your code order—you can create constructors in one place, then create instances as needed, and it is clear where they came from.

However, some people prefer to create object instances without first creating constructors, especially if they are creating only a few instances of an object. JavaScript has a built-in method called [create\(\)](#) that allows you to do that. With it, you can create a new object based on any existing object.

1. With your finished exercise from the previous sections loaded in the browser, try this in your JavaScript console:

```
var person2 = Object.create(person1);
```

- • Now try these:

```
person2.name
person2.greeting()
```

2.

You'll see that `person2` has been created based on `person1`—it has the same properties and method available to it.

One limitation of `create()` is that IE8 does not support it. So constructors may be more effective if you want to support older browsers.

We'll explore the effects of `create()` in more detail later on.

## Summary

This article has provided a simplified view of object-oriented theory — this isn't the whole story, but it gives you an idea of what we are dealing with here. In addition, we have started to look at how JavaScript relates to and how it differs from "classic OOP", how to implement classes in JavaScript using constructor functions, and different ways of generating object instances.

In the next article, we'll explore JavaScript object prototypes.

## Object prototypes

Prototypes are the mechanism by which JavaScript objects inherit features from one another, and they work differently than inheritance mechanisms in classical object-oriented programming languages. In this article we explore that difference, explain how prototype chains work, and look at how the prototype property can be used to add methods to existing constructors.

Basic computer literacy, a basic understanding of HTML and CSS, familiarity

**Prerequisites:** with JavaScript basics (see [First steps](#) and [Building blocks](#)) and OOJS basics (see [Introduction to objects](#)).

**Objective:** To understand JavaScript object prototypes, how prototype chains work, and how to add new methods onto the prototype property.

## A prototype-based language?

JavaScript is often described as a **prototype-based language** — each object has a **prototype object**, which acts as a template object that it inherits methods and properties from. An object's prototype object may also have a prototype object, which it inherits methods and properties from, and so on. This is often referred to as a **prototype chain**, and explains why different objects have properties and methods defined on other objects available to them.

Well, to be exact, the properties and methods are defined on the `prototype` property on the Objects' constructor functions, not the object instances themselves.

In classic OOP, classes are defined, then when object instances are created all the properties and methods defined on the class are copied over to the instance. In JavaScript, they are not copied over — instead, a link is made between the object instance and its prototype (its `__proto__` property, which is derived from the `prototype` property on the constructor), and the properties and methods are found by walking up the chain of prototypes.

**Note:** It's important to understand that there is a distinction between an object's prototype (which is available via `Object.getPrototypeOf(obj)`, or via the deprecated `__proto__` property) and the `prototype` property on constructor functions. The former is the property on each instance, and the latter is the property on the constructor. That is, `Object.getPrototypeOf(new Foobar())` refers to the same object as `Foobar.prototype`.

Let's look at an example to make this a bit clearer.

## Understanding prototype objects

Let's go back to the example in which we finished writing our `Person()` constructor — load the example in your browser. If you don't still have it from working through the last article, use our [oojs-class-further-exercises.html](#) example (see also the [source code](#)).

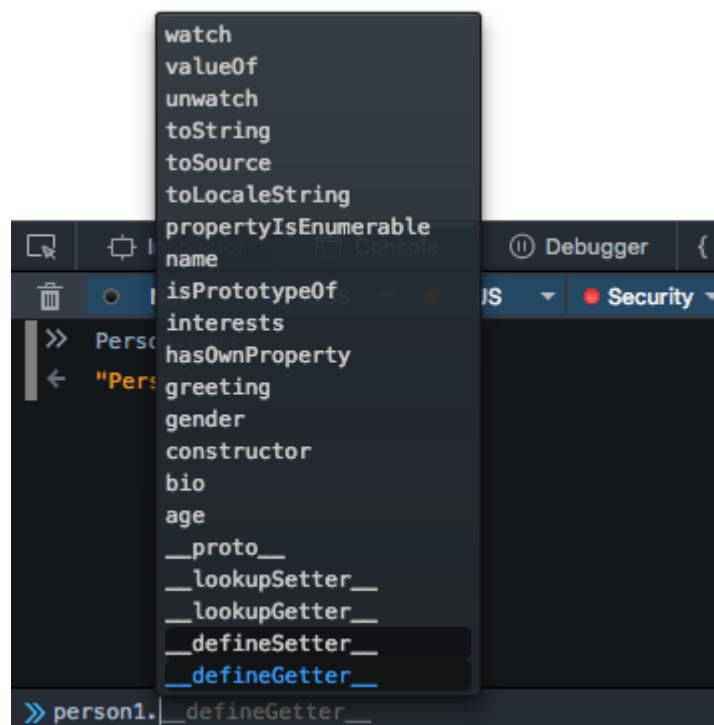
In this example, we have defined a constructor function, like so:

```
function Person(first, last, age, gender, interests) {
 // property and method definitions
}
```

We have then created an object instance like this:

```
var person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

If you type "person1." into your JavaScript console, you should see the browser try to autocomplete this with the member names available on this object:



In this list, you will see the members defined on `person1`'s prototype object, which is the `Person()` constructor — `name`, `age`, `gender`, `interests`, `bio`, and `greeting`. You will however

also see some other members — `watch`, `valueOf`, etc — these are defined on the `Person()` constructor's prototype object, which is [Object](#). This demonstrates the prototype chain working.



So what happens if you call a method on `person1`, which is actually defined on `Object`? For example:

```
person1.valueOf()
```

This method simply returns the value of the object it is called on — try it and see! In this case, what happens is:

- The browser initially checks to see if the `person1` object has a `valueOf()` method available on it.
- It doesn't, so the browser then checks to see if the `person1` object's prototype object (`Person()` constructor's prototype) has a `valueOf()` method available on it.
- It doesn't either, so the browser then checks to see if the `Person()` constructor's prototype object's prototype object (`Object()` constructor's prototype) has a `valueOf()` method available on it. It does, so it is called, and all is good!

**Note:** We want to reiterate that the methods and properties are **not** copied from one object to another in the prototype chain — they are accessed by walking up the chain as described above.

**Note:** There isn't officially a way to access an object's prototype object directly — the "links" between the items in the chain are defined in an internal property, referred to as `[ [prototype] ]` in the specification for the JavaScript language (see [ECMAScript](#)). Most modern browsers however do have a property available on them called `__proto__` (that's 2 underscores either side), which contains the object's prototype object. For example, try `person1.__proto__` and `person1.__proto__.__proto__` to see what the chain looks like in code!

Since ECMAScript 2015 you *can* access an objects prototype object indirectly via `Object.getPrototypeOf(obj)`.

## The prototype property: Where inherited members are defined

So, where are the inherited properties and methods defined? If you look at the [Object](#) reference page, you'll see listed in the left hand side a large number of properties and methods — many more than the number of inherited members we saw available on the `person1` object in the above screenshot. Some are inherited, and some aren't — why is this?

The answer is that the inherited ones are the ones defined on the `prototype` property (you could call it a sub-namespace) — that is, the ones that begin with `Object.prototype.`, and not the ones that begin with just `Object`. The `prototype` property's value is an object, which is basically a bucket for storing properties and methods that we want to be inherited by objects further down the prototype chain.

So `Object.prototype.watch()`, `Object.prototype.valueOf()`, etc., are available to any object types that inherit from `Object.prototype`, including new object instances created from the constructor.

`Object.is()`, `Object.keys()`, and other members not defined inside the `prototype` bucket are not inherited by object instances or object types that inherit from `Object.prototype`. They are methods/properties available just on the `Object()` constructor itself.

**Note:** This seems strange — how can you have a method defined on a constructor, which is itself a function? Well, a function is also a type of object — see the [Function\(\)](#) constructor reference if you don't believe us.

1. You can check out existing prototype properties for yourself — go back to our previous example and try entering the following into the JavaScript console:

```
Person.prototype
```

- • The output won't show you very much — after all, we haven't defined anything on our custom constructor's prototype! By default, a constructor's `prototype` always starts empty. Now try the following:

```
Object.prototype
```

- 2.

You'll see a large number of methods defined on `Object`'s `prototype` property, which are then available on objects that inherit from `Object`, as shown earlier.

You'll see other examples of prototype chain inheritance all over JavaScript — try looking for the methods and properties defined on the prototype of the [String](#), [Date](#), [Number](#), and [Array](#) global objects, for example. These all have a number of members defined on their prototype, which is why for example when you create a string, like this:

```
var myString = 'This is my string.';
```

`myString` immediately has a number of useful methods available on it, like [`split\(\)`](#), [`indexOf\(\)`](#), [`replace\(\)`](#), etc.

**Important:** The `prototype` property is one of the most confusingly-named parts of JavaScript — you might think that `this` points to the prototype object of the current object, but it doesn't (that's an internal object that can be accessed by `__proto__`, remember?). `prototype` instead is a property containing an object on which you define members that you want to be inherited.

## Revisiting `create()`

Earlier on we showed how the `Object.create()` method can be used to create a new object instance.

1. For example, try this in your previous example's JavaScript console:

```
var person2 = Object.create(person1);
```

- • What `create()` actually does is to create a new object from a specified prototype object. Here, `person2` is being created using `person1` as a prototype object. You can check this by entering the following in the console:

```
person2.__proto__
```

- 2.

This will return the `person1` object.

## The `constructor` property

Every constructor function has a `prototype` property whose value is an object containing a `constructor` property. This `constructor` property points to the original constructor function. As you will see in the next section that properties defined on the `Person.prototype` property (or in general on a constructor function's `prototype` property, which is an object, as mentioned in the above section) become available to all the instance objects created using the `Person()` constructor. Hence, the `constructor` property is also available to both `person1` and `person2` objects.

1. For example, try these commands in the console:

```
2. person1.constructor
person2.constructor
```

- These should both return the `Person()` constructor, as it contains the original definition of these instances.

A clever trick is that you can put parentheses onto the end of the `constructor` property (containing any required parameters) to create another object instance from that constructor. The `constructor` is a function after all, so can be invoked using parentheses; you just need to include the `new` keyword to specify that you want to use the function as a constructor.

- Try this in the console:

```
var person3 = new person1.constructor('Karen', 'Stephenson', 26, 'female',
['playing drums', 'mountain climbing']);
• • Now try accessing your new object's features, for example:
person3.name.first
person3.age
person3.bio()
```

3.

This works well. You won't need to use it often, but it can be really useful when you want to create a new instance and don't have a reference to the original constructor easily available for some reason.

The `constructor` property has other uses besides. For example, if you have an object instance and you want to return the name of the constructor it is an instance of, you can use the following:

```
instanceName.constructor.name
```

Try this, for example:

```
person1.constructor.name
```

**Note:** The value of `constructor.name` can change (due to prototypical inheritance, binding, preprocessors, transpilers, etc.), so for more complex examples you'll want to use the `instanceof` operator instead.

## Modifying prototypes

Let's have a look at an example of modifying the `prototype` property of a constructor function (methods added to the prototype are then available on all object instances created from the constructor).

1. Go back to our [oojs-class-further-exercises.html](#) example and make a local copy of the [source code](#). Below the existing JavaScript, add the following code, which adds a new method to the constructor's `prototype` property:

```
2. Person.prototype.farewell = function() {
3. alert(this.name.first + ' has left the building. Bye for now!');
};
```

- • Save the code and load the page in the browser, and try entering the following into the text input:

```
person1.farewell();
```

2.

You should get an alert message displayed, featuring the person's name as defined inside the constructor. This is really useful, but what is even more useful is that the whole inheritance chain

has updated dynamically, automatically making this new method available on all object instances derived from the constructor.

Think about this for a moment. In our code we define the constructor, then we create an instance object from the constructor, *then* we add a new method to the constructor's prototype:

```
function Person(first, last, age, gender, interests) {
 // property and method definitions
}

var person1 = new Person('Tammi', 'Smith', 32, 'neutral', ['music', 'skiing',
'kickboxing']);

Person.prototype.farewell = function() {
 alert(this.name.first + ' has left the building. Bye for now!');
};
```

But the `farewell()` method is *still* available on the `person1` object instance — its available functionality has been automatically updated. This proves what we said earlier about the prototype chain, and the browser looking upwards in the chain to find methods that aren't defined on the object instance itself rather than those methods being copied to the instance. This provides a very powerful, extensible system of functionality.

**Note:** If you are having trouble getting this example to work, have a look at our [oojs-class-prototype.html](#) example (see it [running live](#) also).

You will rarely see properties defined on the `prototype` property, because they are not very flexible when defined like this. For example you could add a property like so:

```
Person.prototype.fullName = 'Bob Smith';
```

But this isn't very flexible, as the person might not be called that. It'd be much better to do this, to build the `fullName` out of `name.first` and `name.last`:

```
Person.prototype.fullName = this.name.first + ' ' + this.name.last;
```

This however doesn't work, as `this` will be referencing the global scope in this case, not the function scope. Calling this property would return `undefined undefined`. This worked fine on the method we defined earlier in the prototype because it is sitting inside a function scope, which will be transferred successfully to the object instance scope. So you might define constant properties on the prototype (i.e. ones that never need to change), but generally it works better to define properties inside the constructor.

In fact, a fairly common pattern for more object definitions is to define the properties inside the constructor, and the methods on the prototype. This makes the code easier to read, as the constructor only contains the property definitions, and the methods are split off into separate blocks. For example:

```
// Constructor with property definitions

function Test(a, b, c, d) {
 // property definitions
}

// First method definition

Test.prototype.x = function() { ... };

// Second method definition

Test.prototype.y = function() { ... };

// etc.
```

This pattern can be seen in action in Piotr Zalewa's [school plan app](#) example.

## Summary

This article has covered JavaScript object prototypes, including how prototype object chains allow objects to inherit features from one another, the `prototype` property and how it can be used to add methods to constructors, and other related topics.

In the next article we'll look at how you can implement inheritance of functionality between two of your own custom objects.

## Inheritance in JavaScript

With most of the gory details of OOJS now explained, this article shows how to create "child" object classes (constructors) that inherit features from their "parent" classes. In addition, we present some advice on when and where you might use OOJS.

Basic computer literacy, a basic understanding of HTML and CSS, familiarity with

**Prerequisites:** JavaScript basics (see [First steps](#) and [Building blocks](#)) and OOJS basics (see [Introduction to objects](#)).

**Objective:** To understand how it is possible to implement inheritance in JavaScript.

## Prototypal inheritance

So far we have seen some inheritance in action — we have seen how prototype chains work, and how members are inherited going up a chain. But mostly this has involved built-in browser functions. How do we create an object in JavaScript that inherits from another object?

As mentioned earlier in the course, some people think JavaScript is not a true object-oriented language. In "classic OO" languages, you tend to define class objects of some kind, and you can

then simply define which classes inherit from which other classes (see [C++ inheritance](#) for some simple examples). JavaScript uses a different system — "inheriting" objects do not have functionality copied over to them, instead the functionality they inherit is linked to via the prototype chain (often referred to as **prototypal inheritance**).

Let's explore how to do this with a concrete example.

## Getting started

First of all, make yourself a local copy of our [oojs-class-inheritance-start.html](#) file (see it [running live](#) also). Inside here you'll find the same `Person()` constructor example that we've been using all the way through the module, with a slight difference — we've defined only the properties inside the constructor:

```
function Person(first, last, age, gender, interests) {
 this.name = {
 first,
 last
 };
 this.age = age;
 this.gender = gender;
 this.interests = interests;
};
```

The methods are *all* defined on the constructor's prototype. For example:

```
Person.prototype.greeting = function() {
 alert('Hi! I\'m ' + this.name.first + '.');
};
```

**Note:** In the source code, you'll also see `bio()` and `farewell()` methods defined. Later you'll see how these can be inherited by other constructors.

Say we wanted to create a `Teacher` class, like the one we described in our initial object-oriented definition, which inherits all the members from `Person`, but also includes:

1. A new property, `subject` — this will contain the subject the teacher teaches.
2. An updated `greeting()` method, which sounds a bit more formal than the standard `greeting()` method — more suitable for a teacher addressing some students at school.

## Defining a Teacher() constructor function

The first thing we need to do is create a `Teacher()` constructor — add the following below the existing code:

```
function Teacher(first, last, age, gender, interests, subject) {
 Person.call(this, first, last, age, gender, interests);
```

```
 this.subject = subject;
}
```

This looks similar to the `Person` constructor in many ways, but there is something strange here that we've not seen before — the `call()` function. This function basically allows you to call a function defined somewhere else, but in the current context. The first parameter specifies the value of `this` that you want to use when running the function, and the other parameters are those that should be passed to the function when it is invoked.

We want the `Teacher()` constructor to take the same parameters as the `Person()` constructor it is inheriting from, so we specify them all as parameters in the `call()` invocation.

The last line inside the constructor simply defines the new `subject` property that teachers are going to have, which generic people don't have.

As a note, we could have simply done this:

```
function Teacher(first, last, age, gender, interests, subject) {
 this.name = {
 first,
 last
 };
 this.age = age;
 this.gender = gender;
 this.interests = interests;
 this.subject = subject;
}
```

But this is just redefining the properties anew, not inheriting them from `Person()`, so it defeats the point of what we are trying to do. It also takes more lines of code.

## Inheriting from a constructor with no parameters

Note that if the constructor you are inheriting from doesn't take its property values from parameters, you don't need to specify them as additional arguments in `call()`. So, for example, if you had something really simple like this:

```
function Brick() {
 this.width = 10;
 this.height = 20;
}
```

You could inherit the `width` and `height` properties by doing this (as well as the other steps described below, of course):

```
function BlueGlassBrick() {
 Brick.call(this);

 this.opacity = 0.5;
 this.color = 'blue';
}
```

```
}
```

Note that we've only specified `this` inside `call()` — no other parameters are required as we are not inheriting any properties from the parent that are set via parameters.

## Setting Teacher()'s prototype and constructor reference

All is good so far, but we have a problem. We have defined a new constructor, and it has a `prototype` property, which by default just contains a reference to the constructor function itself. It does not contain the methods of the `Person` constructor's `prototype` property. To see this, enter `Object.getOwnPropertyNames(Teacher.prototype)` into either the text input field or your JavaScript console. Then enter it again, replacing `Teacher` with `Person`. Nor does the new constructor *inherit* those methods. To see this, compare the outputs of `Person.prototype.greeting` and `Teacher.prototype.greeting`. We need to get `Teacher()` to inherit the methods defined on `Person()`'s `prototype`. So how do we do that?

1. Add the following line below your previous addition:

```
Teacher.prototype = Object.create(Person.prototype);
```

- Here our friend [create\(\)](#) comes to the rescue again. In this case we are using it to create a new object and make it the value of `Teacher.prototype`. The new object has `Person.prototype` as its `prototype` and will therefore inherit, if and when needed, all the methods available on `Person.prototype`.
- We need to do one more thing before we move on. After adding the last line, `Teacher.prototype`'s `constructor` property is now equal to `Person()`, because we just set `Teacher.prototype` to reference an object that inherits its properties from `Person.prototype`! Try saving your code, loading the page in a browser, and entering `Teacher.prototype.constructor` into the console to verify.
- This can become a problem, so we need to set this right. You can do so by going back to your source code and adding the following line at the bottom:

```
Teacher.prototype.constructor = Teacher;
```

- 3.

4. Now if you save and refresh, entering `Teacher.prototype.constructor` should return `Teacher()`, as desired, plus we are now inheriting from `Person()`!

## Giving Teacher() a new greeting() function

To finish off our code, we need to define a new `greeting()` function on the `Teacher()` constructor.

The easiest way to do this is to define it on `Teacher()`'s `prototype` — add the following at the bottom of your code:

```

Teacher.prototype.greeting = function() {
 var prefix;

 if (this.gender === 'male' || this.gender === 'Male' || this.gender === 'm'
 || this.gender === 'M') {
 prefix = 'Mr.';
 } else if (this.gender === 'female' || this.gender === 'Female' ||
 this.gender === 'f' || this.gender === 'F') {
 prefix = 'Mrs.';
 } else {
 prefix = 'Mx.';
 }

 alert('Hello. My name is ' + prefix + ' ' + this.name.last + ', and I teach
 ' + this.subject + '.');
};

```

This alerts the teacher's greeting, which also uses an appropriate name prefix for their gender, worked out using a conditional statement.

## Trying the example out

Now that you've entered all the code, try creating an object instance from `Teacher()` by putting the following at the bottom of your JavaScript (or something similar of your choosing):

```
var teacher1 = new Teacher('Dave', 'Griffiths', 31, 'male', ['football',
'cookery'], 'mathematics');
```

Now save and refresh, and try accessing the properties and methods of your new `teacher1` object, for example:

```

teacher1.name.first;
teacher1.interests[0];
teacher1.bio();
teacher1.subject;
teacher1.greeting();
teacher1.farewell();

```

These should all work just fine. The queries on lines 1, 2, 3, and 6 access members inherited from the generic `Person()` constructor (class). The query on line 4 accesses a member that is available only on the more specialized `Teacher()` constructor (class). The query on line 5 would have accessed a member inherited from `Person()`, except for the fact that `Teacher()` has its own member with the same name, so the query accesses that member.

**Note:** If you have trouble getting this to work, compare your code to our [finished version](#) (see it [running live](#) also).

The technique we covered here is not the only way to create inheriting classes in JavaScript, but it works OK, and it gives you a good idea about how to implement inheritance in JavaScript.

You might also be interested in checking out some of the new [ECMAScript](#) features that allow us to do inheritance more cleanly in JavaScript (see [Classes](#)). We didn't cover those here, as they are not yet supported very widely across browsers. All the other code constructs we discussed in this set of articles are supported as far back as IE9 or earlier, and there are ways to achieve earlier support than that.

A common way is to use a JavaScript library — most of the popular options have an easy set of functionality available for doing inheritance more easily and quickly. [CoffeeScript](#) for example provides `class`, `extends`, etc.

## A further exercise

In our [OOP theory section](#), we also included a `Student` class as a concept, which inherits all the features of `Person`, and also has a different `greeting()` method from `Person` that is much more informal than the `Teacher`'s `greeting`. Have a look at what the student's `greeting` looks like in that section, and try implementing your own `Student()` constructor that inherits all the features of `Person()`, and implements the different `greeting()` function.

**Note:** If you have trouble getting this to work, have a look at our [finished version](#) (see it [running live](#) also).

## Object member summary

To summarize, you've basically got three types of property/method to worry about:

1. Those defined inside a constructor function that are given to object instances. These are fairly easy to spot — in your own custom code, they are the members defined inside a constructor using the `this.x = x` type lines; in built in browser code, they are the members only available to object instances (usually created by calling a constructor using the `new` keyword, e.g. `var myInstance = new myConstructor()`).
2. Those defined directly on the constructor themselves, that are available only on the constructor. These are commonly only available on built-in browser objects, and are recognized by being chained directly onto a constructor, *not* an instance. For example, [`Object.keys\(\)`](#).
3. Those defined on a constructor's prototype, which are inherited by all instances and inheriting object classes. These include any member defined on a Constructor's prototype property, e.g. `myConstructor.prototype.x()`.

If you are not sure which is which, don't worry about it just yet — you are still learning, and familiarity will come with practice.

## When would you use inheritance in JavaScript?

Particularly after this last article, you might be thinking "woo, this is complicated". Well, you are right. Prototypes and inheritance represent some of the most complex aspects of JavaScript, but a

lot of JavaScript's power and flexibility comes from its object structure and inheritance, and it is worth understanding how it works.

In a way, you use inheritance all the time. Whenever you use various features of a Web API , or methods/properties defined on a built-in browser object that you call on your strings, arrays, etc., you are implicitly using inheritance.

In terms of using inheritance in your own code, you probably won't use it often, especially to begin with, and in small projects. It is a waste of time to use objects and inheritance just for the sake of it when you don't need them. But as your code bases get larger, you are more likely to find a need for it. If you find yourself starting to create a number of objects that have similar features, then creating a generic object type to contain all the shared functionality and inheriting those features in more specialized object types can be convenient and useful.

**Note:** Because of the way JavaScript works, with the prototype chain, etc., the sharing of functionality between objects is often called **delegation**. Specialized objects are delegated functionality from a generic object type. This is probably more accurate than calling it *inheritance*, as the "inherited" functionality is not copied to the objects that are doing the "inheriting". Instead it still remains in the generic object.

When using inheritance, you are advised to not have too many levels of inheritance, and to keep careful track of where you define your methods and properties. It is possible to start writing code that temporarily modifies the prototypes of built-in browser objects, but you should not do this unless you have a really good reason. Too much inheritance can lead to endless confusion, and endless pain when you try to debug such code.

Ultimately, objects are just another form of code reuse, like functions or loops, with their own specific roles and advantages. If you find yourself creating a bunch of related variables and functions and want to track them all together and package them neatly, an object is a good idea. Objects are also very useful when you want to pass a collection of data from one place to another. Both of these things can be achieved without use of constructors or inheritance. If you only need a single instance of an object, then you are probably better off just using an object literal, and you certainly don't need inheritance.

## Summary

This article has covered the remainder of the core OOJS theory and syntax that we think you should know now. At this point you should understand JavaScript object and OOP basics, prototypes and prototypal inheritance, how to create classes (constructors) and object instances, add features to classes, and create subclasses that inherit from other classes.

In the next article we'll have a look at how to work with JavaScript Object Notation (JSON), a common data exchange format written using JavaScript objects.

# Working with JSON

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa). You'll come across it quite often, so in this article we give you all you need to work with JSON using JavaScript, including parsing JSON so you can access data within it, and creating JSON.

Basic computer literacy, a basic understanding of HTML and CSS, familiarity with

**Prerequisites:** JavaScript basics (see [First steps](#) and [Building blocks](#)) and OOJS basics (see [Introduction to objects](#)).

**Objective:** To understand how to work with data stored in JSON, and create your own JSON objects.

## No, really, what is JSON?

[JSON](#) is a text-based data format following JavaScript object syntax, which was popularized by [Douglas Crockford](#). Even though it closely resembles JavaScript object literal syntax, it can be used independently from JavaScript, and many programming environments feature the ability to read (parse) and generate JSON.

JSON exists as a string — useful when you want to transmit data across a network. It needs to be converted to a native JavaScript object when you want to access the data. This is not a big issue — JavaScript provides a global [JSON](#) object that has methods available for converting between the two.

**Note:** Converting a string to a native object is called *parsing*, while converting a native object to a string so it can be transmitted across the network is called *stringification*.

A JSON object can be stored in its own file, which is basically just a text file with an extension of `.json`, and a [MIME type](#) of `application/json`.

### JSON structure

As described above, a JSON is a string whose format very much resembles JavaScript object literal format. You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals. This allows you to construct a data hierarchy, like so:

```
{
 "squadName": "Super hero squad",
 "homeTown": "Metro City",
 "formed": 2016,
 "secretBase": "Super tower",
```

```

"active": true,
"members": [
 {
 "name": "Molecule Man",
 "age": 29,
 "secretIdentity": "Dan Jukes",
 "powers": [
 "Radiation resistance",
 "Turning tiny",
 "Radiation blast"
]
 },
 {
 "name": "Madame Uppercut",
 "age": 39,
 "secretIdentity": "Jane Wilson",
 "powers": [
 "Million tonne punch",
 "Damage resistance",
 "Superhuman reflexes"
]
 },
 {
 "name": "Eternal Flame",
 "age": 1000000,
 "secretIdentity": "Unknown",
 "powers": [
 "Immortality",
 "Heat Immunity",
 "Inferno",
 "Teleportation",
 "Interdimensional travel"
]
 }
]
}

```

If we loaded this object into a JavaScript program, parsed in a variable called `superHeroes` for example, we could then access the data inside it using the same dot/bracket notation we looked at in the [JavaScript object basics](#) article. For example:

```

superHeroes.homeTown
superHeroes['active']

```

To access data further down the hierarchy, you simply have to chain the required property names and array indexes together. For example, to access the third superpower of the second hero listed in the `members` list, you'd do this:

```

superHeroes['members'][1]['powers'][2]

```

1. First we have the variable name — `superHeroes`.
2. Inside that we want to access the `members` property, so we use `["members"]`.

3. `members` contains an array populated by objects. We want to access the second object inside the array, so we use `[1]`.
4. Inside this object, we want to access the `powers` property, so we use `["powers"]`.
5. Inside the `powers` property is an array containing the selected hero's superpowers. We want the third one, so we use `[2]`.

**Note:** We've made the JSON seen above available inside a variable in our [JSONTest.html](#) example (see the [source code](#)). Try loading this up and then accessing data inside the variable via your browser's JavaScript console.

## Arrays as JSON

Above we mentioned that JSON text basically looks like a JavaScript object, and this is mostly right. The reason we said "mostly right" is that an array is also valid JSON, for example:

```
[
 {
 "name": "Molecule Man",
 "age": 29,
 "secretIdentity": "Dan Jukes",
 "powers": [
 "Radiation resistance",
 "Turning tiny",
 "Radiation blast"
]
 },
 {
 "name": "Madame Uppercut",
 "age": 39,
 "secretIdentity": "Jane Wilson",
 "powers": [
 "Million tonne punch",
 "Damage resistance",
 "Superhuman reflexes"
]
 }
]
```

The above is perfectly valid JSON. You'd just have to access array items (in its parsed version) by starting with an array index, for example `[0]["powers"][0]`.

## Other notes

- JSON is purely a data format — it contains only properties, no methods.
- JSON requires double quotes to be used around strings and property names. Single quotes are not valid.
- Even a single misplaced comma or colon can cause a JSON file to go wrong, and not work. You should be careful to validate any data you are attempting to use (although computer-generated JSON is less likely to include errors, as long as the generator program is working correctly). You can validate JSON using an application like [JSONLint](#).

- JSON can actually take the form of any data type that is valid for inclusion inside JSON, not just arrays or objects. So for example, a single string or number would be a valid JSON object.
- Unlike in JavaScript code in which object properties may be unquoted, in JSON, only quoted strings may be used as properties.

## Active learning: Working through a JSON example

So, let's work through an example to show how we could make use of some JSON data on a website.

### Getting started

To begin with, make local copies of our [heroes.html](#) and [style.css](#) files. The latter contains some simple CSS to style our page, while the former contains some very simple body HTML:

```
<header>
</header>

<section>
</section>
```

Plus a [`<script>`](#) element to contain the JavaScript code we will be writing in this exercise. At the moment it only contains two lines, which grab references to the [`<header>`](#) and [`<section>`](#) elements and store them in variables:

```
var header = document.querySelector('header');
var section = document.querySelector('section');
```

We have made our JSON data available on our GitHub, at <https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json>.

We are going to load it into our page, and use some nifty DOM manipulation to display it, like this:

# **SUPER HERO SQUAD**

**Hometown: Metro City // Formed: 2016**

## **MOLECULE MAN**

Secret identity: Dan Jukes

Age: 29

Superpowers:

- Radiation resistance
- Turning tiny
- Radiation blast

## **MADAME UPPERCUT**

Secret identity: Jane Wilson

Age: 39

Superpowers:

- Million tonne punch
- Damage resistance
- Superhuman reflexes

## **ETERNAL FLAME**

Secret identity: Unknown

Age: 1000000

Superpowers:

- Immortality
- Heat Immunity
- Inferno
- Teleportation
- Interdimensional travel

## Obtaining the JSON

To obtain the JSON, we are going to use an API called [XMLHttpRequest](#) (often called **XHR**). This is a very useful JavaScript object that allows us to make network requests to retrieve resources from a server via JavaScript (e.g. images, text, JSON, even HTML snippets), meaning that we can update small sections of content without having to reload the entire page. This has led to more responsive web pages, and sounds exciting, but it is unfortunately beyond the scope of this article to teach it in much more detail.

1. To start with, we are going to store the URL of the JSON we want to retrieve in a variable. Add the following at the bottom of your JavaScript code:

```
var requestURL = 'https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json';
```

- To create a request, we need to create a new request object instance from the `XMLHttpRequest` constructor, using the `new` keyword. Add the following below your last line:

```
var request = new XMLHttpRequest();
```

- • Now we need to open a new request using the [open\(\)](#) method. Add the following line:

```
request.open('GET', requestURL);
```

- This takes at least two parameters — there are other optional parameters available. We only need the two mandatory ones for this simple example:

- The HTTP method to use when making the network request. In this case [GET](#) is fine, as we are just retrieving some simple data.
- The URL to make the request to — this is the URL of the JSON file that we stored earlier.

- Next, add the following two lines — here we are setting the [responseType](#) to JSON, so that XHR knows that the server will be returning JSON, and that this should be converted behind the scenes into a JavaScript object. Then we send the request with the [send\(\)](#) method:

```
request.responseType = 'json';
request.send();
```

- • The last bit of this section involves waiting for the response to return from the server, then dealing with it. Add the following code below your previous code:

```
request.onload = function() {
 var superHeroes = request.response;
 populateHeader(superHeroes);
 showHeroes(superHeroes);
}
```

5.

Here we are storing the response to our request (available in the [response](#) property) in a variable called `superHeroes`; this variable will now contain the JavaScript object based on the JSON! We are then passing that object to two function calls — the first one will fill the `<header>` with the correct data, while the second one will create an information card for each hero on the team, and insert it into the `<section>`.

We have wrapped the code in an event handler that runs when the load event fires on the request object (see [onload](#)) — this is because the load event fires when the response has successfully returned; doing it this way guarantees that `request.response` will definitely be available when we come to try to do something with it.

## Populating the header

Now we've retrieved the JSON data and converted it into a JavaScript object, let's make use of it by writing the two functions we referenced above. First of all, add the following function definition below the previous code:

```
function populateHeader(jsonObj) {
 var myH1 = document.createElement('h1');
 myH1.textContent = jsonObj['squadName'];
 header.appendChild(myH1);
```

```

var myPara = document.createElement('p');
myPara.textContent = 'Hometown: ' + jsonObj['homeTown'] + ' // Formed: ' +
jsonObj['formed'];
header.appendChild(myPara);
}

```

We have called the parameter `jsonObj`, to remind ourselves that this JavaScript object originated from JSON. Here we first create an `<h1>` element with `createElement()`, set its `textContent` to equal the `squadName` property of the object, then append it to the header using `appendChild()`. We then do a very similar operation with a paragraph: create it, set its text content and append it to the header. The only difference is that its text is set to a concatenated string containing both the `homeTown` and `formed` properties of the object.

## Creating the hero information cards

Next, add the following function at the bottom of the code, which creates and displays the superhero cards:

```

function showHeroes(jsonObj) {
 var heroes = jsonObj['members'];

 for (var i = 0; i < heroes.length; i++) {
 var myArticle = document.createElement('article');
 var myH2 = document.createElement('h2');
 var myPara1 = document.createElement('p');
 var myPara2 = document.createElement('p');
 var myPara3 = document.createElement('p');
 var myList = document.createElement('ul');

 myH2.textContent = heroes[i].name;
 myPara1.textContent = 'Secret identity: ' + heroes[i].secretIdentity;
 myPara2.textContent = 'Age: ' + heroes[i].age;
 myPara3.textContent = 'Superpowers:';

 var superPowers = heroes[i].powers;
 for (var j = 0; j < superPowers.length; j++) {
 var listItem = document.createElement('li');
 listItem.textContent = superPowers[j];
 myList.appendChild(listItem);
 }

 myArticle.appendChild(myH2);
 myArticle.appendChild(myPara1);
 myArticle.appendChild(myPara2);
 myArticle.appendChild(myPara3);
 myArticle.appendChild(myList);

 section.appendChild(myArticle);
 }
}

```

To start with, we store the `members` property of the JavaScript object in a new variable. This array contains multiple objects that contain the information for each hero.

Next, we use a [for loop](#) to loop through each object in the array. For each one, we:

1. Create several new elements: an `<article>`, an `<h2>`, three `<p>`s, and a `<ul>`.
2. Set the `<h2>` to contain the current hero's name.
3. Fill the three paragraphs with their `secretIdentity`, `age`, and a line saying "Superpowers:" to introduce the information in the list.
4. Store the `powers` property in another new variable called `superPowers` — this contains an array that lists the current hero's superpowers.
5. Use another `for` loop to loop through the current hero's superpowers — for each one we create a `<li>` element, put the superpower inside it, then put the `listItem` inside the `<ul>` element (`myList`) using `appendChild()`.
6. The very last thing we do is to append the `<h2>`, `<p>`s, and `<ul>` inside the `<article>` (`myArticle`), then append the `<article>` inside the `<section>`. The order in which things are appended is important, as this is the order they will be displayed inside the HTML.

**Note:** If you are having trouble getting the example to work, try referring to our [heroes-finished.html](#) source code (see it [running live](#) also.)

**Note:** If you are having trouble following the dot/bracket notation we are using to access the JavaScript object, it can help to have the [superheroes.json](#) file open in another tab or your text editor, and refer to it as you look at our JavaScript. You should also refer back to our [JavaScript object basics](#) article for more information on dot and bracket notation.

## Converting between objects and text

The above example was simple in terms of accessing the JavaScript object, because we set the XHR request to convert the JSON response directly into a JavaScript object using:

```
request.responseType = 'json';
```

But sometimes we aren't so lucky — sometimes we'll receive a raw JSON string, and we'll need to convert it to an object ourselves. And when we want to send a JavaScript object across the network, we'll need to convert it to JSON (a string) before sending. Luckily, these two problems are so common in web development that a built-in [JSON](#) object is available in browsers, which contains the following two methods:

- [`parse\(\)`](#): Accepts a JSON string as a parameter, and returns the corresponding JavaScript object.
- [`stringify\(\)`](#): Accepts an object as a parameter, and returns the equivalent JSON string form.

You can see the first one in action in our [heroes-finished-json-parse.html](#) example (see the [source code](#)) — this does exactly the same thing as the example we built up earlier, except that

we set the XHR to return the raw JSON text, then used `parse()` to convert it to an actual JavaScript object. The key snippet of code is here:

```
request.open('GET', requestURL);
request.responseType = 'text'; // now we're getting a string!
request.send();

request.onload = function() {
 var superHeroesText = request.response; // get the string from the response
 var superHeroes = JSON.parse(superHeroesText); // convert it to an object
 populateHeader(superHeroes);
 showHeroes(superHeroes);
}
```

As you might guess, `stringify()` works the opposite way. Try entering the following lines into your browser's JavaScript console one by one to see it in action:

```
var myJSON = { "name": "Chris", "age": "38" };
myJSON
var myString = JSON.stringify(myJSON);
myString
```

Here we're creating a JavaScript object, then checking what it contains, then converting it to a JSON string using `stringify()` — saving the return value in a new variable — then checking it again.

## Summary

In this article, we've given you a simple guide to using JSON in your programs, including how to create and parse JSON, and how to access data locked inside it. In the next article, we'll begin looking at object-oriented JavaScript.

## Object building practice

In previous articles we looked at all the essential JavaScript object theory and syntax details, giving you a solid base to start from. In this article we dive into a practical exercise, giving you some more practice in building custom JavaScript objects, with a fun and colorful result.

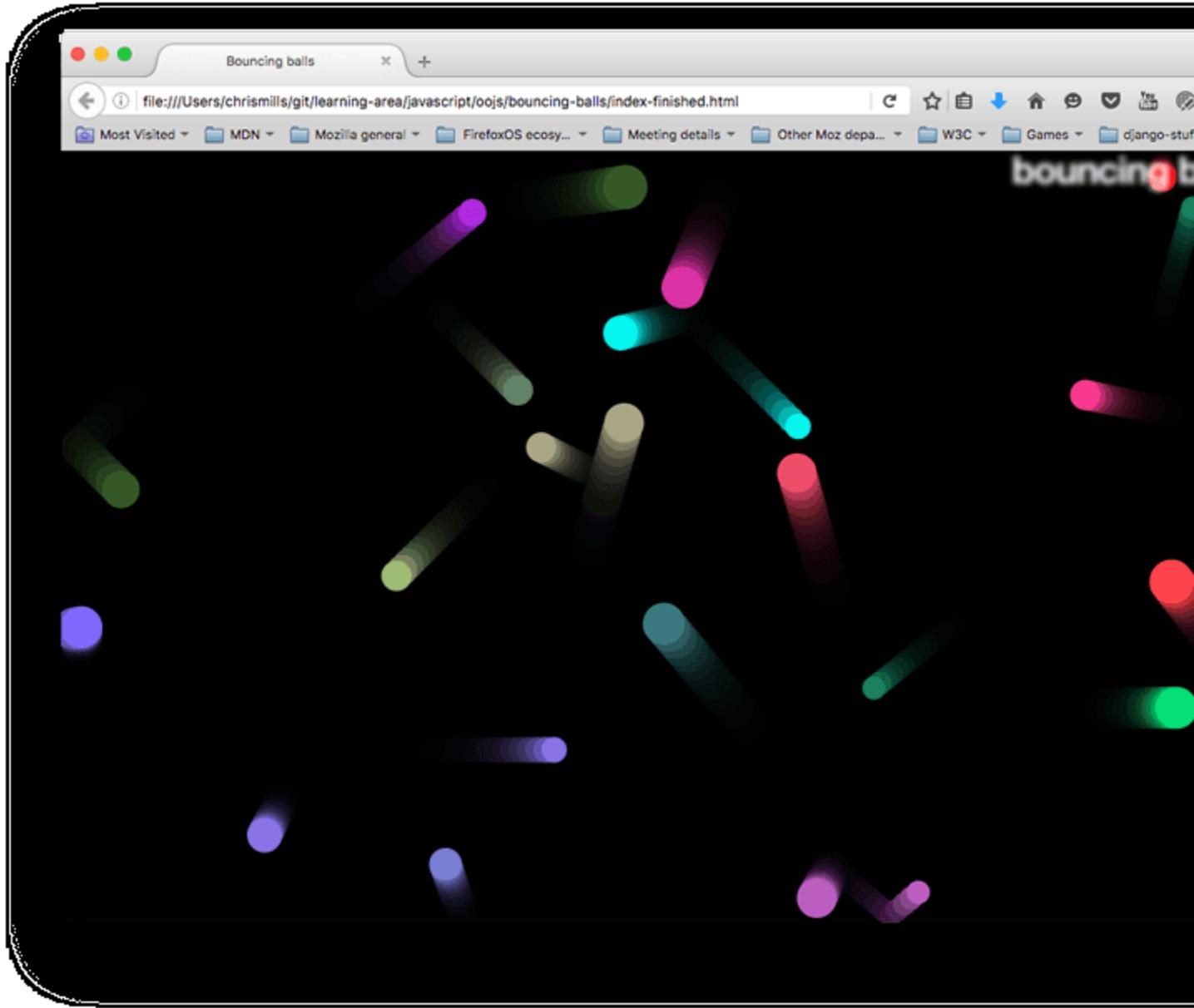
Basic computer literacy, a basic understanding of HTML and CSS, familiarity with

**Prerequisites:** JavaScript basics (see [First steps](#) and [Building blocks](#)) and OOJS basics (see [Introduction to objects](#)).

**Objective:** To get some practice with using objects and object-oriented techniques in a real world context.

## Let's bounce some balls

In this article we will write a classic "bouncing balls" demo, to show you how useful objects can be in JavaScript. Our little balls will bounce around on the screen, and change color when they touch each other. The finished example will look a little something like this:



This example will make use of the [Canvas API](#) for drawing the balls to the screen, and the [requestAnimationFrame](#) API for animating the whole display — you don't need to have any previous knowledge of these APIs, and we hope that by the time you've finished this article you'll be interested in exploring them more. Along the way we'll make use of some nifty objects, and show you a couple of nice techniques like bouncing balls off walls, and checking whether they have hit each other (otherwise known as **collision detection**).

## Getting started

To begin with, make local copies of our `index.html`, `style.css`, and `main.js` files. These contain the following, respectively:

1. A very simple HTML document featuring an `<h1>` element, a `<canvas>` element to draw our balls on, and elements to apply our CSS and JavaScript to our HTML.
2. Some very simple styles, which mainly serve to style and position the `<h1>`, and get rid of any scrollbars or margin round the edge of the page (so that it looks nice and neat).
3. Some JavaScript that serves to set up the `<canvas>` element and provide a general function that we're going to use.

The first part of the script looks like so:

```
var canvas = document.querySelector('canvas');

var ctx = canvas.getContext('2d');

var width = canvas.width = window.innerWidth;
var height = canvas.height = window.innerHeight;
```

This script gets a reference to the `<canvas>` element, then calls the `getContext()` method on it to give us a context on which we can start to draw. The resulting variable (`ctx`) is the object that directly represents the drawing area of the canvas and allows us to draw 2D shapes on it.

Next, we set variables called `width` and `height`, and the width and height of the canvas element (represented by the `canvas.width` and `canvas.height` properties) to equal the width and height of the browser viewport (the area that the webpage appears on — this can be got from the `Window.innerWidth` and `Window.innerHeight` properties).

You'll see here that we are chaining multiple assignments together, to get the variables all set quicker — this is perfectly OK.

The last bit of the initial script looks as follows:

```
function random(min, max) {
 var num = Math.floor(Math.random() * (max - min + 1)) + min;
 return num;
}
```

This function takes two numbers as arguments, and returns a random number in the range between the two.

## Modeling a ball in our program

Our program will feature lots of balls bouncing around the screen. Since these balls will all behave in the same kind of way, it makes sense to represent them with an object. Let's start by adding the following constructor to the bottom of our code.

```
function Ball(x, y, velX, velY, color, size) {
```

```

this.x = x;
this.y = y;
this.velX = velX;
this.velY = velY;
this.color = color;
this.size = size;
}

```

Here we include some parameters that define the properties each ball needs to function to in our program:

- `x` and `y` coordinates — the horizontal and vertical coordinates where the ball will start on the screen. This can range between 0 (top left hand corner) to the width and height of the browser viewport (bottom right hand corner).
- horizontal and vertical velocity (`velX` and `velY`) — each ball is given a horizontal and vertical velocity; in real terms these values will be regularly added to the `x/y` coordinate values when we start to animate the balls, to move them by this much on each frame.
- `color` — each ball gets a color.
- `size` — each ball gets a size — this will be its radius, in pixels.

This sorts the properties out, but what about the methods? We want to actually get our balls to do something in our program.

## Drawing the ball

First add the following `draw()` method to the `Ball()`'s prototype:

```

Ball.prototype.draw = function() {
 ctx.beginPath();
 ctx.fillStyle = this.color;
 ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
 ctx.fill();
}

```

Using this function, we can tell our ball to draw itself onto the screen, by calling a series of members of the 2D canvas context we defined earlier (`ctx`). The context is like the paper, and now we want to command our pen to draw something on it:

- First, we use `beginPath()` to state that we want to draw a shape on the paper.
- Next, we use `fillStyle` to define what color we want the shape to be — we set it to our ball's `color` property.
- Next, we use the `arc()` method to trace an arc shape on the paper. Its parameters are:
  - The `x` and `y` position of the arc's center — we are specifying our ball's `x` and `y` properties.
  - The radius of our arc — we are specifying our ball's `size` property.
  - The last two parameters specify the start and end number of degrees round the circle that the arc is drawn between. Here we specify 0 degrees, and `2 * PI`, which is the equivalent of 360 degrees in radians (annoyingly, you have to specify this in radians).

That gives us a complete circle. If you had specified only `1 * PI`, you'd get a semi-circle (180 degrees).

- Last of all, we use the `fill()` method, which basically states "finish drawing the path we started with `beginPath()`, and fill the area it takes up with the color we specified earlier in `fillStyle`."

You can start testing your object out already.

- Save the code so far, and load the HTML file in a browser.
- Open the browser's JavaScript console, and then refresh the page so that the canvas size change to the smaller visible viewport left when the console opens.
- Type in the following to create a new ball instance:

```
var testBall = new Ball(50, 100, 4, 4, 'blue', 10);
```

- Try calling its members:

```
testBall.x
testBall.size
testBall.color
testBall.draw()
```

4.

- When you enter the last line, you should see the ball draw itself somewhere on your canvas.

## Updating the ball's data

We can draw the ball in position, but to actually start moving the ball, we need an update function of some kind. Add the following code at the bottom of your JavaScript file, to add an `update()` method to the `Ball()`'s prototype:

```
Ball.prototype.update = function() {
 if ((this.x + this.size) >= width) {
 this.velX = -(this.velX);
 }

 if ((this.x - this.size) <= 0) {
 this.velX = -(this.velX);
 }

 if ((this.y + this.size) >= height) {
 this.velY = -(this.velY);
 }

 if ((this.y - this.size) <= 0) {
 this.velY = -(this.velY);
 }

 this.x += this.velX;
 this.y += this.velY;
}
```

The first four parts of the function check whether the ball has reached the edge of the canvas. If it has, we reverse the polarity of the relevant velocity to make the ball travel in the opposite direction. So for example, if the ball was traveling upwards (positive `velY`), then the vertical velocity is changed so that it starts to travel downwards instead (negative `velY`).

In the four cases, we are:

- Checking to see whether the `x` coordinate is greater than the width of the canvas (the ball is going off the right hand edge).
- Checking to see whether the `x` coordinate is smaller than 0 (the ball is going off the left hand edge).
- Checking to see whether the `y` coordinate is greater than the height of the canvas (the ball is going off the bottom edge).
- Checking to see whether the `y` coordinate is smaller than 0 (the ball is going off the top edge).

In each case, we are including the `size` of the ball in the calculation because the `x/y` coordinates are in the center of the ball, but we want the edge of the ball to bounce off the perimeter — we don't want the ball to go halfway off the screen before it starts to bounce back.

The last two lines add the `velX` value to the `x` coordinate, and the `velY` value to the `y` coordinate — the ball is in effect moved each time this method is called.

This will do for now; let's get on with some animation!

## Animating the ball

Now let's make this fun. We are now going to start adding balls to the canvas, and animating them.

1. First, we need somewhere to store all our balls. The following array will do this job — add it to the bottom of your code now:

```
var balls = [];
```

- All programs that animate things generally involve an animation loop, which serves to update the information in the program and then render the resulting view on each frame of the animation; this is the basis for most games and other such programs.

- Add the following to the bottom of your code now:

```
function loop() {
 ctx.fillStyle = 'rgba(0, 0, 0, 0.25)';
 ctx.fillRect(0, 0, width, height);

 while (balls.length < 25) {
 var ball = new Ball(
 random(0, width),
 random(0, height),
```

```

 random(-7, 7),
 random(-7, 7),
 'rgb(' + random(0,255) + ',' + random(0,255) + ',' + random(0,255)
+')',
 random(10,20)
);
balls.push(ball);
}

for (var i = 0; i < balls.length; i++) {
 balls[i].draw();
 balls[i].update();
}

requestAnimationFrame(loop);
}

```

- Our `loop()` function does the following:

- Sets the canvas fill color to semi-transparent black, then draws a rectangle of the color across the whole width and height of the canvas, using `fillRect()` (the four parameters provide a start coordinate, and a width and height for the rectangle drawn). This serves to cover up the previous frame's drawing before the next one is drawn. If you don't do this, you'll just see long snakes worming their way around the canvas instead of balls moving! The color of the fill is set to semi-transparent, `rgba(0, 0, 0, 0.25)`, to allow the previous few frames to shine through slightly, producing the little trails behind the balls as they move. If you changed 0.25 to 1, you won't see them at all any more. Try varying this number to see the effect it has.
- Creates a new instance of our `Ball()` using random values generated with our `random()` function, then `push()` es it onto the end of our `balls` array, but only while the number of balls in the array is less than 25. So when we have 25 balls on screen, no more balls appear. You can try varying the number in `balls.length < 25` to get more or less balls on screen. Depending on how much processing power your computer/browser has, specifying several thousand balls might slow down the animation rather a lot!
- loops through all the balls in the `balls` array, and runs each ball's `draw()` and `update()` function to draw each one on the screen, then do the necessary updates to position and velocity in time for the next frame.
- Runs the function again using the `requestAnimationFrame()` method — when this method is constantly run and passed the same function name, it will run that function a set number of times per second to create a smooth animation. This is generally done recursively — which means that the function is calling itself every time it runs, so it will run over and over again.

- Last but not least, add the following line to the bottom of your code — we need to call the function once to get the animation started.

```
loop();
```

3.

That's it for the basics — try saving and refreshing to test your bouncing balls out!

# Adding collision detection

Now for a bit of fun, let's add some collision detection to our program, so our balls will know when they have hit another ball.

1. First of all, add the following method definition below where you defined the `update()` method (i.e. the `Ball.prototype.update` block).

```
1. Ball.prototype.collisionDetect = function() {
2. for (var j = 0; j < balls.length; j++) {
3. if (!(this === balls[j])) {
4. var dx = this.x - balls[j].x;
5. var dy = this.y - balls[j].y;
6. var distance = Math.sqrt(dx * dx + dy * dy);
7.
8. if (distance < this.size + balls[j].size) {
9. balls[j].color = this.color = 'rgb(' + random(0, 255) + ',' +
10. random(0, 255) + ',' + random(0, 255) + ')';
11. }
12. }
13. }
}
```

- This method is a little complex, so don't worry if you don't understand exactly how it works for now. An explanation follows:

- For each ball, we need to check every other ball to see if it has collided with the current ball. To do this, we open up another `for` loop to loop through all the balls in the `balls[]` array.
- Immediately inside our `for` loop, we use an `if` statement to check whether the current ball being looped through is the same ball as the one we are currently checking. We don't want to check whether a ball has collided with itself! To do this, we check whether the current ball (i.e., the ball whose `collisionDetect` method is being invoked) is the same as the loop ball (i.e., the ball that is being referred to by the current iteration of the `for` loop in the `collisionDetect` method). We then use `!` to negate the check, so that the code inside the `if` statement only runs if they are **not** the same.
- We then use a common algorithm to check the collision of two circles. We are basically checking whether any of the two circle's areas overlap. This is explained further in [2D collision detection](#).
- If a collision is detected, the code inside the inner `if` statement is run. In this case we are just setting the `color` property of both the circles to a new random color. We could have done something far more complex, like get the balls to bounce off each other realistically, but that would have been far more complex to implement. For such physics simulations, developers tend to use a games or physics library such as [PhysicsJS](#), [matter.js](#), [Phaser](#), etc.

- You also need to call this method in each frame of the animation. Add the following below the `balls[i].update();` line:  
`balls[i].collisionDetect();`

- 2.
3. Save and refresh the demo again, and you'll see your balls change color when they collide!

**Note:** If you have trouble getting this example to work, try comparing your JavaScript code against our [finished version](#) (also see it [running live](#)).

## Summary

We hope you had fun writing your own real world random bouncing balls example, using various object and object-oriented techniques from throughout the module! This should have given you some useful practice in using objects, and good real world context.

That's it for object articles — all that remains now is for you to test your skills in the object assessment.

## Client-side web APIs

When writing client-side JavaScript for web sites or applications, you won't go very far before you start to use APIs — interfaces for manipulating different aspects of the browser and operating system the site is running on, or even data from other web sites or services. In this module, we will explore what APIs are, and how to use some of the most common APIs you'll come across often in your development work.

When writing client-side JavaScript for web sites or applications, you won't go very far before you start to use APIs — interfaces for manipulating different aspects of the browser and operating system the site is running on, or even data from other web sites or services. In this module, we will explore what APIs are, and how to use some of the most common APIs you'll come across often in your development work.

## Introduction to web APIs

First up, we'll start by looking at APIs from a high level — what are they, how do they work, how do you use them in your code, and how are they structured? We'll also take a look at what the different main classes of APIs are, and what kind of uses they have.

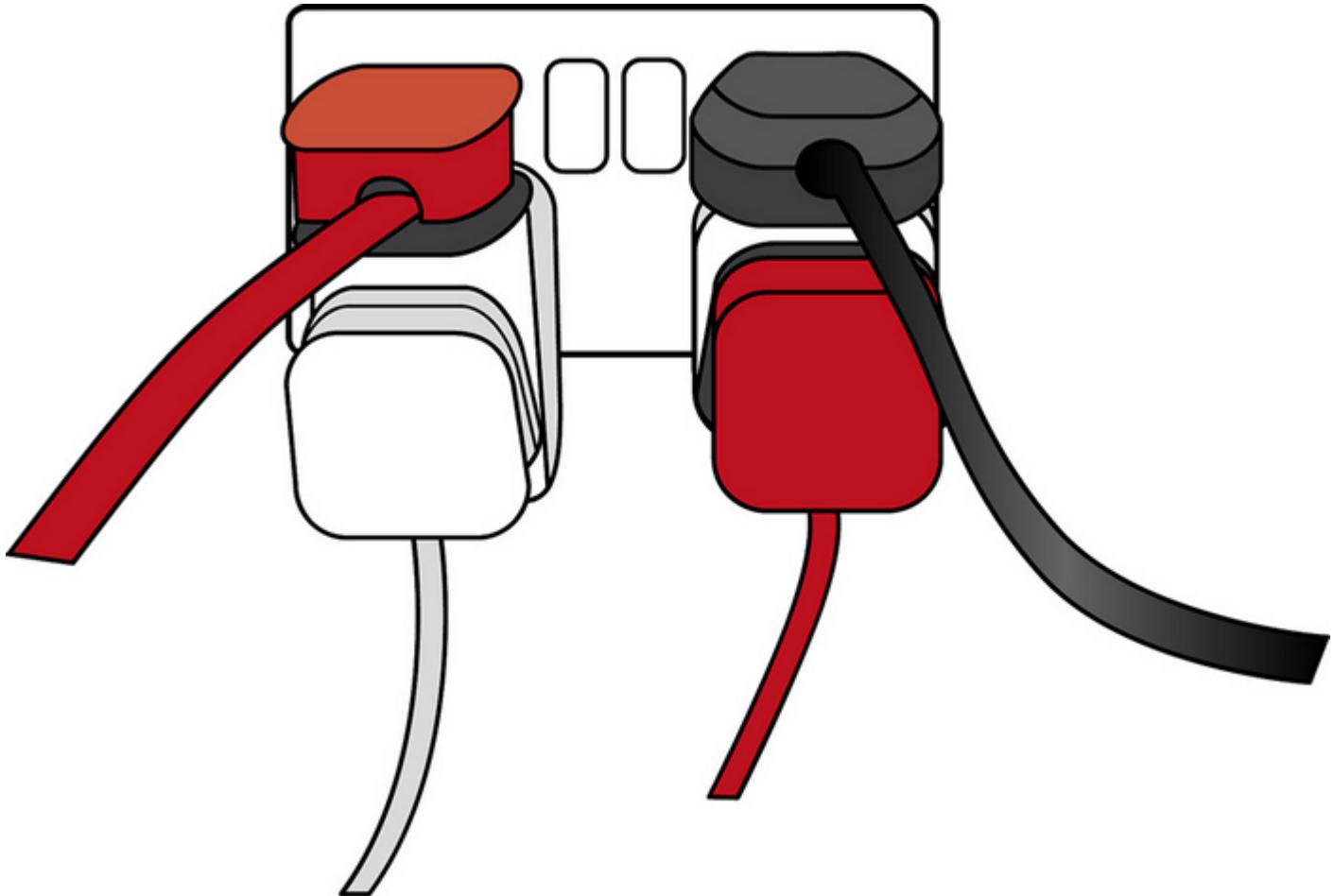
**Prerequisites:** Basic computer literacy, a basic understanding of [HTML](#) and [CSS](#), JavaScript basics (see [first steps](#), [building blocks](#), [JavaScript objects](#)).

**Objective:** To gain familiarity with APIs, what they can do, and how you can use them in your code.

## What are APIs?

Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily. They abstract more complex code away from you, providing some easier syntax to use in its place.

As a real-world example, think about the electricity supply in your house, apartment, or other dwellings. If you want to use an appliance in your house, you simply plug it into a plug socket and it works. You don't try to wire it directly into the power supply — to do so would be really inefficient and, if you are not an electrician, difficult and dangerous to attempt.



*Image source: [Overloaded plug socket](#) by [The Clear Communication People](#), on Flickr.*

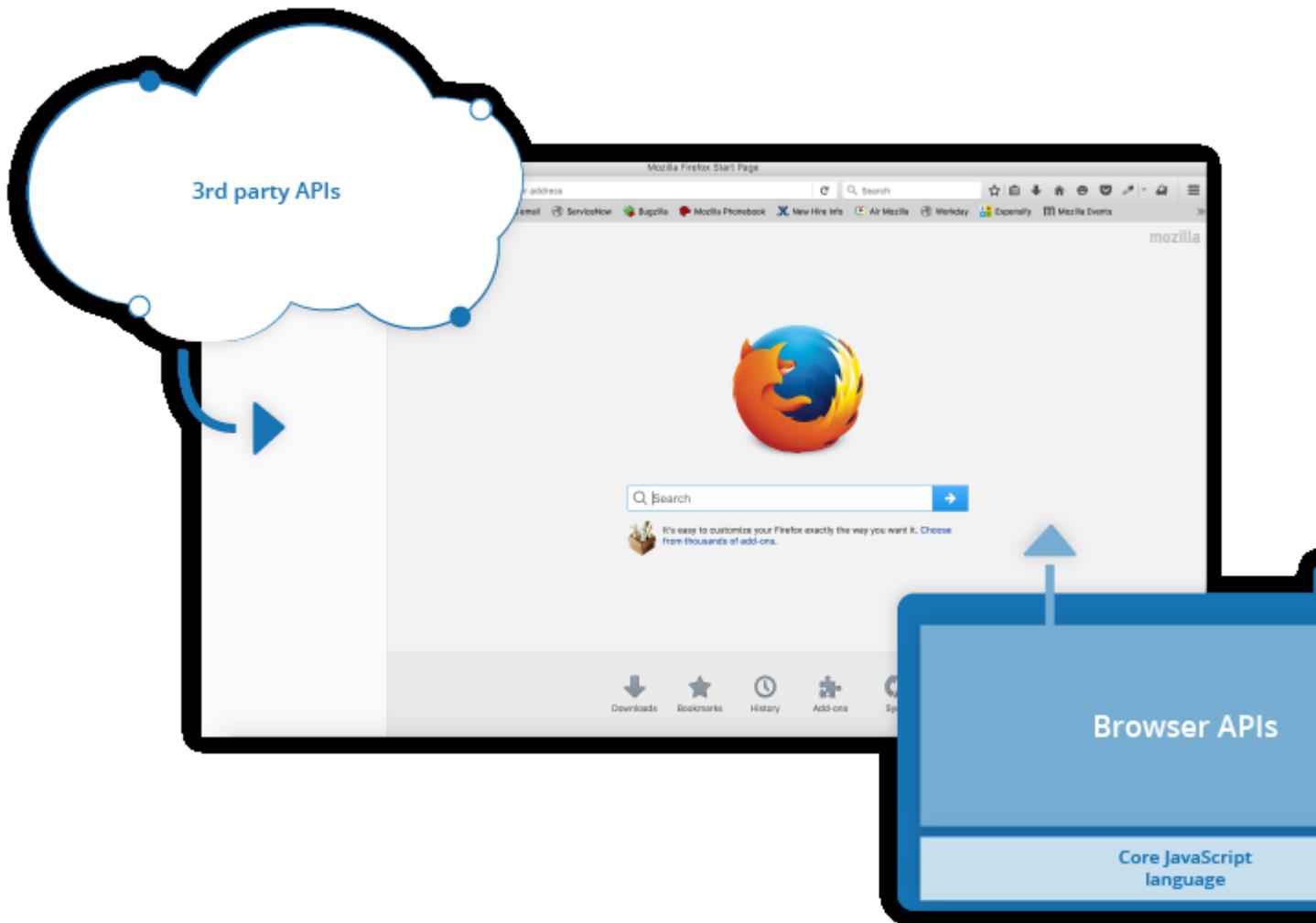
In the same way, if you want to say, program some 3D graphics, it is a lot easier to do it using an API written in a higher level language such as JavaScript or Python, rather than try to directly write low level code (say C or C++) that directly controls the computer's GPU or other graphics functions.

**Note:** See also the [API glossary entry](#) for further description.

## APIs in client-side JavaScript

Client-side JavaScript, in particular, has many APIs available to it — these are not part of the JavaScript language itself, rather they are built on top of the core JavaScript language, providing you with extra superpowers to use in your JavaScript code. They generally fall into two categories:

- **Browser APIs** are built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it. For example, the [Geolocation API](#) provides some simple JavaScript constructs for retrieving location data so you can say, plot your location on a Google Map. In the background, the browser is actually using some complex lower-level code (e.g. C++) to communicate with the device's GPS hardware (or whatever is available to determine position data), retrieve position data, and return it to the browser environment to use in your code. But again, this complexity is abstracted away from you by the API.
- **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. It provides a special set of constructs you can use to query the Twitter service and return specific information.



## Relationship between JavaScript, APIs, and other JavaScript tools

So above, we talked about what client-side JavaScript APIs are, and how they relate to the JavaScript language. Let's recap this to make it clearer, and also mention where other JavaScript tools fit in:

- **JavaScript** — A high-level scripting language built into browsers that allows you to implement functionality on web pages/apps. Note that JavaScript is also available in other programming environments, such as [Node](#). But don't worry about that for now.
- **Browser APIs** — constructs built into the browser that sit on top of the JavaScript language and allow you to implement functionality more easily.
- **Third party APIs** — constructs built into third-party platforms (e.g. Twitter, Facebook) that allow you to use some of those platform's functionality in your own web pages (for example, display your latest Tweets on your web page).

- JavaScript libraries — Usually one or more JavaScript files containing [custom functions](#) that you can attach to your web page to speed up or enable writing common functionality. Examples include jQuery, Mootools and React.
- JavaScript frameworks — The next step up from libraries, JavaScript frameworks (e.g. Angular and Ember) tend to be packages of HTML, CSS, JavaScript, and other technologies that you install and then use to write an entire web application from scratch. The key difference between a library and a framework is “Inversion of Control”. When calling a method from a library, the developer is in control. With a framework, the control is inverted: the framework calls the developer's code.

## What can APIs do?

There are a huge number of APIs available in modern browsers that allow you to do a wide variety of things in your code. You can see this by taking a look at the [MDN APIs index page](#).

### Common browser APIs

In particular, the most common categories of browser APIs you'll use (and which we'll cover in this module in greater detail) are:

- **APIs for manipulating documents** loaded into the browser. The most obvious example is the [DOM \(Document Object Model\) API](#), which 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, for example, that's the DOM in action. Find out more about these types of API in [Manipulating documents](#).
- **APIs that fetch data from the server** to update small sections of a webpage on their own are very commonly used. This seemingly small detail has had a huge impact on the performance and behaviour of sites — if you just need to update a stock listing or list of available new stories, doing it instantly without having to reload the whole entire page from the server can make the site or app feel much more responsive and "snappy". APIs that make this possible include [XMLHttpRequest](#) and the [Fetch API](#). You may also come across the term **Ajax**, which describes this technique. Find out more about such APIs in [Fetching data from the server](#).
- **APIs for drawing and manipulating graphics** are now widely supported in browsers — the most popular ones are [Canvas](#) and [WebGL](#), which allow you to programmatically update the pixel data contained in an HTML [`<canvas>`](#) element to create 2D and 3D scenes. For example, you might draw shapes such as rectangles or circles, import an image onto the canvas, and apply a filter to it such as sepia or grayscale using the Canvas API, or create a complex 3D scene with lighting and textures using WebGL. Such APIs are often combined with APIs for creating animation loops (such as [`window.requestAnimationFrame\(\)`](#)) and others to make constantly updating scenes like cartoons and games.
- **Audio and Video APIs** like [HTMLMediaElement](#), the [Web Audio API](#), and [WebRTC](#) allow you to do really interesting things with multimedia such as creating custom UI controls for playing audio and video, displaying text tracks like captions and subtitles along with your videos, grabbing video from your web camera to be manipulated via a canvas (see above) or displayed on someone else's computer in a web conference, or adding effects to audio tracks (such as gain, distortion, panning, etc).

- **Device APIs** are basically APIs for manipulating and retrieving data from modern device hardware in a way that is useful for web apps. We've already talked about the Geolocation API accessing the device's location data so you can plot your position on a map. Other examples include telling the user that a useful update is available on a web app via system notifications (see the [Notifications API](#)) or vibration hardware (see the [Vibration API](#)).
- **Client-side storage APIs** are becoming a lot more widespread in web browsers — the ability to store data on the client-side is very useful if you want to create an app that will save its state between page loads, and perhaps even work when the device is offline. There are a number of options available, e.g. simple name/value storage with the [Web Storage API](#), and more complex tabular data storage with the [IndexedDB API](#).

## Common third-party APIs

Third party APIs come in a large variety; some of the more popular ones that you are likely to make use of sooner or later are:

- The [Twitter API](#), which allows you to do things like displaying your latest tweets on your website.
- The [Google Maps API](#) allows you to do all sorts of things with maps on your web pages (funnily enough, it also powers Google Maps). This is now an entire suite of APIs, which handle a wide variety of tasks, as evidenced by the [Google Maps API Picker](#).
- The [Facebook suite of APIs](#) enables you to use various parts of the Facebook ecosystem to benefit your app, for example by providing app login using Facebook login, accepting in-app payments, rolling out targeted ad campaigns, etc.
- The [YouTube API](#), which allows you to embed YouTube videos on your site, search YouTube, build playlists, and more.
- The [Twilio API](#), which provides a framework for building voice and video call functionality into your app, sending SMS/MMS from your apps, and more.

**Note:** You can find information on a lot more 3rd party APIs at the [Programmable Web API directory](#).

## How do APIs work?

Different JavaScript APIs work in slightly different ways, but generally, they have common features and similar themes to how they work.

### They are based on objects

APIs are interacted with in your code using one or more [JavaScript objects](#), which serve as containers for the data the API uses (contained in object properties), and the functionality the API makes available (contained in object methods).

**Note:** If you are not already familiar with how objects work, you should go back and work through our [JavaScript objects](#) module before continuing.

Let's return to the example of the Geolocation API — this is a very simple API that consists of a few simple objects:

- [Geolocation](#), which contains three methods for controlling the retrieval of geodata.
- [Position](#), which represents the position of a device at a given time — this contains a [Coordinates](#) object that contains the actual position information, plus a timestamp representing the given time.
- [Coordinates](#), which contains a whole lot of useful data on the device position, including latitude and longitude, altitude, velocity and direction of movement, and more.

So how do these objects interact? If you look at our [maps-example.html](#) example ([see it live also](#)), you'll see the following code:

```
navigator.geolocation.getCurrentPosition(function(position) {
 var latlng = new
google.maps.LatLng(position.coords.latitude,position.coords.longitude);
 var myOptions = {
 zoom: 8,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.TERRAIN,
 disableDefaultUI: true
 }
 var map = new google.maps.Map(document.querySelector("#map_canvas"),
myOptions);
});
```

**Note:** When you first load up the above example, you should be given a dialog box asking if you are happy to share your location with this application (see the [They have additional security mechanisms where appropriate](#) section later in the article). You need to agree to this to be able to plot your location on the map. If you still can't see the map, you may need to set your permissions manually; you can do this in various ways depending on what browser you are using; for example in Firefox go to > *Tools* > *Page Info* > *Permissions*, then change the setting for *Share Location*; in Chrome go to *Settings* > *Privacy* > *Show advanced settings* > *Content settings* then change the settings for *Location*.

We first want to use the [Geolocation.getCurrentPosition\(\)](#) method to return the current location of our device. The browser's [Geolocation](#) object is accessed by calling the [Navigator.geolocation](#) property, so we start off by using

```
navigator.geolocation.getCurrentPosition(function(position) { ... });
```

This is equivalent to doing something like

```
var myGeo = navigator.geolocation;
myGeo.getCurrentPosition(function(position) { ... });
```

But we can use the dot syntax to chain our property/method access together, reducing the number of lines we have to write.

The `Geolocation.getCurrentPosition()` method only has a single mandatory parameter, which is an anonymous function that will run when the device's current position has been successfully retrieved. This function itself has a parameter, which contains a `Position` object representing the current position data.

**Note:** A function that is taken by another function as an argument is called a [callback function](#).

This pattern of invoking a function only when an operation has been completed is very common in JavaScript APIs — making sure one operation has completed before trying to use the data the operation returns in another operation. These are called **asynchronous operations**. Because getting the device's current position relies on an external component (the device's GPS or other geolocation hardware), we can't guarantee that it will be done in time to just immediately use the data it returns. Therefore, something like this wouldn't work:

```
var position = navigator.geolocation.getCurrentPosition();
var myLatitude = position.coords.latitude;
```

If the first line had not yet returned its result, the second line would throw an error, because the position data would not yet be available. For this reason, APIs involving asynchronous operations are designed to use [callback functions](#), or the more modern system of [Promises](#), which were made available in ECMAScript 6 and are widely used in newer APIs.

We are combining the Geolocation API with a third party API — the Google Maps API — which we are using to plot the location returned by `getCurrentPosition()` on a Google Map. We make this API available on our page by linking to it — you'll find this line in the HTML:

```
<script type="text/javascript"
src="https://maps.google.com/maps/api/js?key=AIzaSyDDuGt0E5IEGkcE6ZfrKfUtE9Ko
_de66pA"></script>
```

To use the API, we first create a `LatLng` object instance using the `google.maps.LatLng()` constructor, which takes our geolocated [Coordinates.latitude](#) and [Coordinates.longitude](#) values as parameters:

```
var latlng = new
google.maps.LatLng(position.coords.latitude, position.coords.longitude);
```

This object is itself set as the value of the `center` property of an options object that we've called `myOptions`. We then create an object instance to represent our map by calling the `google.maps.Map()` constructor, passing it two parameters — a reference to the `<div>` element we want to render the map on (with an ID of `map_canvas`), and the options object we defined just above it.

```
var myOptions = {
 zoom: 8,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.TERRAIN,
 disableDefaultUI: true
```

```
}

var map = new google.maps.Map(document.querySelector("#map_canvas"),
myOptions);
```

With this done, our map now renders.

This last block of code highlights two common patterns you'll see across many APIs. First of all, API objects commonly contain constructors, which are invoked to create instances of those objects that you'll use to write your program. Second, API objects often have several options available that can be tweaked to get the exact environment you want for your program. API constructors commonly accept options objects as parameters, which is where you'd set such options.

**Note:** Don't worry if you don't understand all the details of this example immediately. We'll cover using third party APIs in a lot more detail in a future article.

### They have recognizable entry points

When using an API, you should make sure you know where the entry point is for the API. In The Geolocation API, this is pretty simple — it is the [Navigator.geolocation](#) property, which returns the browser's [Geolocation](#) object that all the useful geolocation methods are available inside.

The Document Object Model (DOM) API has an even simpler entry point — its features tend to be found hanging off the [Document](#) object, or an instance of an HTML element that you want to affect in some way, for example:

```
var em = document.createElement('em'); // create a new em element
var para = document.querySelector('p'); // reference an existing p element
em.textContent = 'Hello there!'; // give em some text content
para.appendChild(em); // embed em inside para
```

Other APIs have slightly more complex entry points, often involving creating a specific context for the API code to be written in. For example, the Canvas API's context object is created by getting a reference to the [<canvas>](#) element you want to draw on, and then calling its [HTMLCanvasElement.getContext\(\)](#) method:

```
var canvas = document.querySelector('canvas');
var ctx = canvas.getContext('2d');
```

Anything that we want to do to the canvas is then achieved by calling properties and methods of the content object (which is an instance of [CanvasRenderingContext2D](#)), for example:

```
Ball.prototype.draw = function() {
 ctx.beginPath();
 ctx.fillStyle = this.color;
 ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
 ctx.fill();
```

```
};
```

**Note:** You can see this code in action in our [bouncing balls demo](#) (see it [running live](#) also).

## They use events to handle changes in state

We already discussed events earlier on in the course, in our [Introduction to events](#) article — this article looks in detail at what client-side web events are and how they are used in your code. If you are not already familiar with how client-side web API events work, you should go and read this article first before continuing.

Some web APIs contain no events, but some contain a number of events. The handler properties that allow us to run functions when events fire are generally listed in our reference material in separate "Event handlers" sections. As a simple example, instances of the [XMLHttpRequest](#) object (each one represents an HTTP request to the server to retrieve a new resource of some kind) have a number of events available on them, for example the `load` event is fired when a response has been successfully returned containing the requested resource, and it is now available.

The following code provides a simple example of how this would be used:

```
var requestURL = 'https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json';
var request = new XMLHttpRequest();
request.open('GET', requestURL);
request.responseType = 'json';
request.send();

request.onload = function() {
 var superHeroes = request.response;
 populateHeader(superHeroes);
 showHeroes(superHeroes);
}
```

**Note:** You can see this code in action in our [ajax.html](#) example ([see it live](#) also).

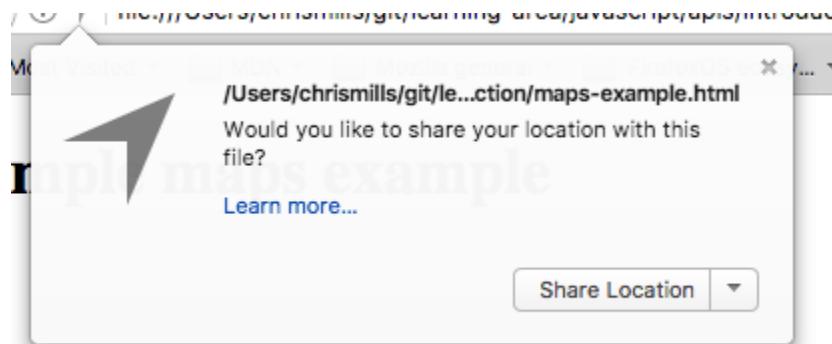
The first five lines specify the location of resource we want to fetch, create a new instance of a request object using the `XMLHttpRequest()` constructor, open an HTTP GET request to retrieve the specified resource, specify that the response should be sent in JSON format, then send the request.

The `onload` handler function then specifies what we do with the response. We know the response will be successfully returned and available after the `load` event has required (unless an error occurred), so we save the response containing the returned JSON in the `superHeroes` variable, then pass it to two different functions for further processing.

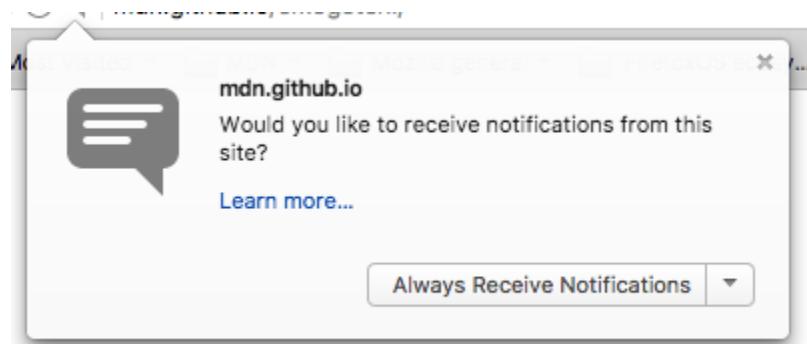
They have additional security mechanisms where appropriate

WebAPI features are subject to the same security considerations as JavaScript and other web technologies (for example [same-origin policy](#)), but they sometimes have additional security mechanisms in place. For example, some of the more modern WebAPIs will only work on pages served over HTTPS due to them transmitting potentially sensitive data (examples include [Service Workers](#) and [Push](#)).

In addition, some WebAPIs request permission to be enabled from the user once calls to them are made in your code. As an example, you may have noticed a dialog like the following when loading up our earlier [Geolocation](#) example:



The [Notifications API](#) asks for permission in a similar fashion:



These permission prompts are given to users for security — if they weren't in place, then sites could start secretly tracking your location without you knowing it, or spamming you with a lot of annoying notifications.

## Summary

At this point, you should have a good idea of what APIs are, how they work, and what you can do with them in your JavaScript code. You are probably excited to start actually doing some fun

things with specific APIs, so let's go! Next up, we'll look at manipulating documents with the Document Object Model (DOM).

## Manipulating documents

When writing web pages and apps, one of the most common things you'll want to do is manipulate the document structure in some way. This is usually done by using the Document Object Model (DOM), a set of APIs for controlling HTML and styling information that makes heavy use of the `Document` object. In this article we'll look at how to use the DOM in detail, along with some other interesting APIs that can alter your environment in interesting ways.

**Prerequisites:** Basic computer literacy, a basic understanding of HTML, CSS, and JavaScript — including JavaScript objects.

**Objective:** To gain familiarity with the core DOM APIs, and the other APIs commonly associated with DOM and document manipulation

## The important parts of a web browser

Web browsers are very complicated pieces of software with a lot of moving parts, many of which can't be controlled or manipulated by a web developer using JavaScript. You might think that such limitations are a bad thing, but browsers are locked down for good reasons, mostly centering around security. Imagine if a web site could get access to your stored passwords or other sensitive information, and log into websites as if it were you?

Despite the limitations, Web APIs still give us access to a lot of functionality that enable us to do a great many things with web pages. There are a few really obvious bits you'll reference regularly in your code — consider the following diagram, which represents the main parts of a browser directly involved in viewing web pages:



- The window is the browser tab that a web page is loaded into; this is represented in JavaScript by the `Window` object. Using methods available on this object you can do things like return the window's size (see `Window.innerWidth` and `Window.innerHeight`), manipulate the document loaded into that window, store data specific to that document on the client-side (for example using a local database or other storage mechanism), attach an [event handler](#) to the current window, and more.
- The navigator represents the state and identity of the browser (i.e. the user-agent) as it exists on the web. In JavaScript, this is represented by the `Navigator` object. You can use this object to retrieve things like geolocation information, the user's preferred language, a media stream from the user's webcam, etc.
- The document (represented by the DOM in browsers) is the actual page loaded into the window, and is represented in JavaScript by the `Document` object. You can use this object to return and manipulate information on the HTML and CSS that comprises the document, for example get a reference to an element in the DOM, change its text content, apply new styles to it, create new elements and add them to the current element as children, or even delete it altogether.

In this article we'll focus mostly on manipulating the document, but we'll show a few other useful bits besides.

## The document object model

The document currently loaded in each one of your browser tabs is represented by a document object model. This is a "tree structure" representation created by the browser that enables the HTML structure to be easily accessed by programming languages — for example the browser itself uses it to apply styling and other information to the correct elements as it renders a page, and developers like you can manipulate the DOM with JavaScript after the page has been rendered.

We have created a simple example page at [dom-example.html](#) ([see it live also](#)). Try opening this up in your browser — it is a very simple page containing a `<section>` element inside which you can find an image, and a paragraph with a link inside. The HTML source code looks like this:

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>Simple DOM example</title>
 </head>
 <body>
 <section>

 <p>Here we will add a link to the Mozilla homepage</p>
 </section>
 </body>
</html>
```

The DOM on the other hand looks like this:

```

└ DOCTYPE: html
 └ HTML
 └ HEAD
 └ #text:
 └ META charset="utf-8"
 └ #text:
 └ TITLE
 └ #text: Simple DOM example
 └ #text:
 └ #text:
 └ BODY
 └ #text:
 └ SECTION
 └ #text:
 └ IMG src="dinosaur.png" alt="A red Tyrannosaurus Rex: A two legged dinosaur standing upright like a human, with small arms, and a large head with lots of sharp teeth."
 └ #text:
 └ P
 └ #text: Here we will add a link to the
 └ A href="https://www.mozilla.org/"
 └ #text: Mozilla homepage
 └ #text:
 └ #text:

```

**Note:** This DOM tree diagram was created using Ian Hickson's [Live DOM viewer](#).

You can see here that each element and bit of text in the document has its own entry in the tree — each one is called a **node**. You will also encounter various terms used to describe the type of node, and their position in the tree in relation to one another:

- **Element node:** An element, as it exists in the DOM.
- **Root node:** The top node in the tree, which in the case of HTML is always the `HTML` node (other markup vocabularies like SVG and custom XML will have different root elements).
- **Child node:** A node *directly* inside another node. For example, `IMG` is a child of `SECTION` in the above example.
- **Descendant node:** A node *anywhere* inside another node. For example, `IMG` is a child of `SECTION` in the above example, and it is also a descendant. `IMG` is not a child of `BODY`, as it is two levels below it in the tree, but it is a descendant of `BODY`.
- **Parent node:** A node which has another node inside it. For example, `BODY` is the parent node of `SECTION` in the above example.
- **Sibling nodes:** Nodes that sit on the same level in the DOM tree. For example, `IMG` and `P` are siblings in the above example.
- **Text node:** A node containing a text string.

It is useful to familiarize yourself with this terminology before working with the DOM, as a number of the code terms you'll come across make use of them. You may have also come across them if you have studied CSS (e.g. descendant selector, child selector).

# Active learning: Basic DOM manipulation

To start learning about DOM manipulation, let's begin with a practical example.

1. Take a local copy of the [dom-example.html page](#) and the [image](#) that goes along with it.
2. Add a `<script></script>` element just above the closing `</body>` tag.
3. To manipulate an element inside the DOM, you first need to select it and store a reference to it inside a variable. Inside your script element, add the following line:

```
var link = document.querySelector('a');
```

- Now we have the element reference stored in a variable, we can start to manipulate it using properties and methods available to it (these are defined on interfaces like `HTMLAnchorElement` in the case of `<a>` element, its more general parent interface `HTMLElement`, and `Node` — which represents all nodes in a DOM). First of all, let's change the text inside the link by updating the value of the `Node.textContent` property. Add the following line below the previous one:

```
link.textContent = 'Mozilla Developer Network';
```

- We should also change the URL the link is pointing to, so that it doesn't go to the wrong place when it is clicked on. Add the following line, again at the bottom:

```
link.href = 'https://developer.mozilla.org';
```

5.

Note that, as with many things in JavaScript, there are many ways to select an element and store a reference to it in a variable. `Document.querySelector()` is the recommended modern approach, which is convenient because it allows you to select elements using CSS selectors. The above `querySelector()` call will match the first `<a>` element that appears in the document. If you wanted to match and do things to multiple elements, you could use `Document.querySelectorAll()`, which matches every element in the document that matches the selector, and stores references to them in an [array](#)-like object called a `NodeList`.

There are older methods available for grabbing element references, such as:

- `Document.getElementById()`, which selects an element with a given `id` attribute value, e.g. `<p id="myId">My paragraph</p>`. The ID is passed to the function as a parameter, i.e. `var elementRef = document.getElementById('myId');`.
- `Document.getElementsByTagName()`, which returns an array containing all the elements on the page of a given type, for example `<p>`s, `<a>`s, etc. The element type is passed to the function as a parameter, i.e. `var elementRefArray = document.getElementsByTagName('p');`.

These two work in older browsers than the modern methods like `querySelector()`, but are not as convenient. Have a look and see what others you can find!

## Creating and placing new nodes

The above has given you a little taste of what you can do, but let's go further and look at how we can create new elements.

1. Going back to the current example, let's start by grabbing a reference to the our `<section>` element — add the following code at the bottom of your existing script (do the same with the other lines too):

```
var sect = document.querySelector('section');
```

- • Now let's create a new paragraph using `Document.createElement()` and give it some text content in the same way as before:

```
var para = document.createElement('p');
para.textContent = 'We hope you enjoyed the ride.';
```

- • You can now append the new paragraph at the end of the section using `Node.appendChild()`:

```
sect.appendChild(para);
```

- • Finally for this part, let's add a text node to the paragraph the link sits inside, to round off the sentence nicely. First we will create the text node using `Document.createTextNode()`:

```
var text = document.createTextNode(' – the premier source for web development
knowledge.');
```

- • Now we'll grab a reference to the paragraph the link is inside, and append the text node to it:

```
var linkPara = document.querySelector('p');
linkPara.appendChild(text);
```

5.

That's most of what you need for adding nodes to the DOM — you'll make a lot of use of these methods when building dynamic interfaces (we'll look at some examples later).

## Moving and removing elements

There may be times when you want to move nodes, or delete them from the DOM altogether. This is perfectly possible.

If we wanted to move the paragraph with the link inside it to the bottom of the section, we could simply do this:

```
sect.appendChild(linkPara);
```

This moves the paragraph down to the bottom of the section. You might have thought it would make a second copy of it, but this is not the case — `linkPara` is a reference to the one and only copy of that paragraph. If you wanted to make a copy and add that as well, you'd need to use `Node.cloneNode()` instead.

Removing a node is pretty simple as well, at least when you have a reference to the node to be removed and its parent. In our current case, we just use `Node.removeChild()`, like this:

```
sect.removeChild(linkPara);
```

It gets slightly more complex when you want to remove a node based only on a reference to itself, which is fairly common. There is no method to tell a node to remove itself, so you'd have to do the following.

```
linkPara.parentNode.removeChild(linkPara);
```

Have a go at adding the above lines to your code.

## Manipulating styles

It is possible to manipulate CSS styles via JavaScript in a variety of ways.

To start with, you can get a list of all the stylesheets attached to a document using `Document.stylesheets`, which returns an array of `CSSStyleSheet` objects. You can then add/remove styles as wished. However, we're not going to expand on those features because they are a somewhat archaic and difficult way to manipulate style. There are much easier ways.

The first way is to add inline styles directly onto elements you want to dynamically style. This is done with the `HTMLElement.style` property, which contains inline styling information for each element in the document. You can set properties of this object to directly update element styles.

1. As an example, try adding these lines to our ongoing example:

```
2. para.style.color = 'white';
3. para.style.backgroundColor = 'black';
4. para.style.padding = '10px';
5. para.style.width = '250px';
 para.style.textAlign = 'center';
```

• • Reload the page and you'll see that the styles have been applied to the paragraph. If you look at that paragraph in your browser's [Page Inspector/DOM inspector](#), you'll see that these lines are indeed adding inline styles to the document:

```
<p style="color: white; background-color: black; padding: 10px; width: 250px;
text-align: center;">We hope you enjoyed the ride.</p>
```

- 2.

**Note:** Notice how the JavaScript property versions of the CSS styles are written in lower camel case whereas the CSS versions are hyphenated (e.g. `backgroundColor` versus `background-color`). Make sure you don't get these mixed up, otherwise it won't work.

There is another common way to dynamically manipulate styles on your document, which we'll look at now.

1. Delete the previous five lines you added to the JavaScript.

2. Add the following inside your HTML `<head>`:

```
3. <style>
4. .highlight {
5. color: white;
6. background-color: black;
7. padding: 10px;
8. width: 250px;
9. text-align: center;
10. }
</style>
```

• • Now we'll turn to a very useful method for general HTML manipulation —

`Element.setAttribute()` — this takes two arguments, the attribute you want to set on the element, and the value you want to set it to. In this case we will set a class name of `highlight` on our paragraph:

```
para.setAttribute('class', 'highlight');
```

3.

4. Refresh your page, and you'll see no change — the CSS is still applied to the paragraph, but this time by giving it a class that is selected by our CSS rule, not as inline CSS styles.

Which method you choose is up to you; both have their advantages and disadvantages. The first method takes less setup and is good for simple uses, whereas the second method is more purist (no mixing CSS and JavaScript, no inline styles, which are seen as a bad practice). As you start building larger and more involved apps, you will probably start using the second method more, but it is really up to you.

At this point, we haven't really done anything useful! There is no point using JavaScript to create static content — you might as well just write it into your HTML and not use JavaScript. It is more complex than HTML, and creating your content with JavaScript also has other issues attached to it (such as not being readable by search engines).

In the next couple of sections we will look at a couple of more practical uses of DOM APIs.

**Note:** You can find our [finished version of the dom-example.html](#) demo on GitHub ([see it live also](#)).

## Active learning: Getting useful information from the Window object

So far we've only really looked at using `Node` and `Document` features to manipulate documents, but there is no reason why you can't get data from other sources and use it in your UI. Think back to our simple [maps-example.html](#) demo from the last article — there we retrieved some location data and used it to display a map of your area. You just have to make sure your data is in the right format; JavaScript makes it easier than many other languages, being weakly typed —

for example numbers will convert to strings automatically when you want to print them to the screen.

In this example we will solve a common problem — making sure your application is as big as the window it is viewed in, whatever size it is. This is often useful in situations like games, where you want to use as much of the screen area as possible to play the game in.

To start with, make a local copy of our [window-resize-example.html](#) and [bgtile.png](#) demo files. Open it and have a look — you'll see that we've got a `<div>` element covering a small part of the screen, which has got a background tile applied to it. We'll use that to represent our app UI area.

1. First of all, let's grab a reference to the div, and then grab the width and height of the viewport (the inner window, where your document is displayed) and store them in variables — these two values are handily contained in the `Window.innerWidth` and `Window.innerHeight` properties. Add the following lines inside the existing `<script>` element:

```
2. var div = document.querySelector('div');
3. var WIDTH = window.innerWidth;
 var HEIGHT = window.innerHeight;
```

- • Next, we'll dynamically alter the width and height of the div to equal that of the viewport. Add the following two lines below your first ones:

```
div.style.width = WIDTH + 'px';
div.style.height = HEIGHT + 'px';
```

- • Save and try refreshing your browser — you should now see the div become as big as your viewport, whatever size of screen you are using. If you now try resizing your window to make it bigger, you'll see that the div stays the same size — we are only setting it once.

- How about we use an event so that the div resizes as we resize the window? The `Window` object has an event available on it called `resize`, which is fired every time the window is resized — let's access that via the `Window.onresize` event handler and rerun our sizing code each time it changes. Add the following to the bottom of your code:

```
window.onresize = function() {
 WIDTH = window.innerWidth;
 HEIGHT = window.innerHeight;
 div.style.width = WIDTH + 'px';
 div.style.height = HEIGHT + 'px';
}
```

4.

**Note:** If you get stuck, have a look at our [finished window resize example \(see it live also\)](#).

## Active learning: A dynamic shopping list

To round off the article, we'd like to set you a little challenge — we want to make a simple shopping list example that allows you to dynamically add items to the list using a form input and button. When you add an item to the input and press the button:

- The item should appear in the list.
- Each item should be given a button that can be pressed to delete that item off the list.
- The input should be emptied and focused ready for you to enter another item.

The finished demo will look something like this:

## My shopping list

Enter a new item:  [Add item](#)

- Eggs [Delete](#)
- Milk [Delete](#)
- Bread [Delete](#)
- Humous [Delete](#)

To complete the exercise, follow the steps below, and make sure that the list behaves as described above.

1. To start with, download a copy of our [shopping-list.html](#) starting file and make a copy of it somewhere. You'll see that it has some minimal CSS, a list with a label, input, and button, and an empty list and `<script>` element. You'll be making all your additions inside the script.
2. Create three variables that hold references to the list (`<ul>`), `<input>`, and `<button>` elements.
3. Create a [function](#) that will run in response to the button being clicked.
4. Inside the function body, start off by storing the current [value](#) of the input element in a variable.
5. Next, empty the input element by setting its value to an empty string — ''.
6. Create three new elements — a list item (`<li>`), `<span>`, and `<button>`, and store them in variables.
7. Append the span and the button as children of the list item.
8. Set the text content of the span to the input element value you saved earlier, and the text content of the button to 'Delete'.
9. Append the list item as a child of the list.
10. Attach an event handler to the delete button, so that when clicked it will delete the entire list item it is inside.
11. Finally, use the [`focus\(\)`](#) method to focus the input element ready for entering the next shopping list item.

**Note:** If you get really stuck, have a look at our [finished shopping list](#) ([see it running live also](#).)

## Summary

We have reached the end of our study of document and DOM manipulation. At this point you should understand what the important parts of a web browser are with respect to controlling documents and other aspects of the user's web experience. Most importantly, you should understand what the Document Object Model is, and how to manipulate it to create useful functionality.

## Fetching data from the server

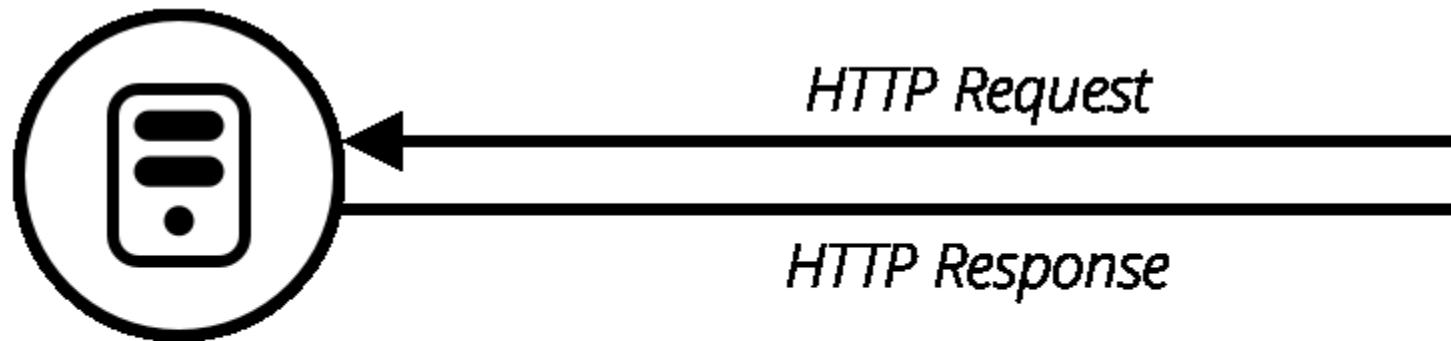
Another very common task in modern websites and applications is retrieving individual data items from the server to update sections of a webpage without having to load an entire new page. This seemingly small detail has had a huge impact on the performance and behavior of sites, so in this article we'll explain the concept and look at technologies that make it possible, such as XMLHttpRequest and the Fetch API.

**Prerequisites:** JavaScript basics (see [first steps](#), [building blocks](#), [JavaScript objects](#)), the [basics of Client-side APIs](#)

**Objective:** To learn how to fetch data from the server and use it to update the contents of a web page.

## What is the problem here?

Originally page loading on the web was simple — you'd send a request for a web site to a server, and as long as nothing went wrong, the assets that made the web page would be downloaded and displayed on your computer.



*Treatments*

*Data*

The trouble with this model is that whenever you want to update any part of the page, for example to display a new set of products or load a new page, you've got to load the entire page

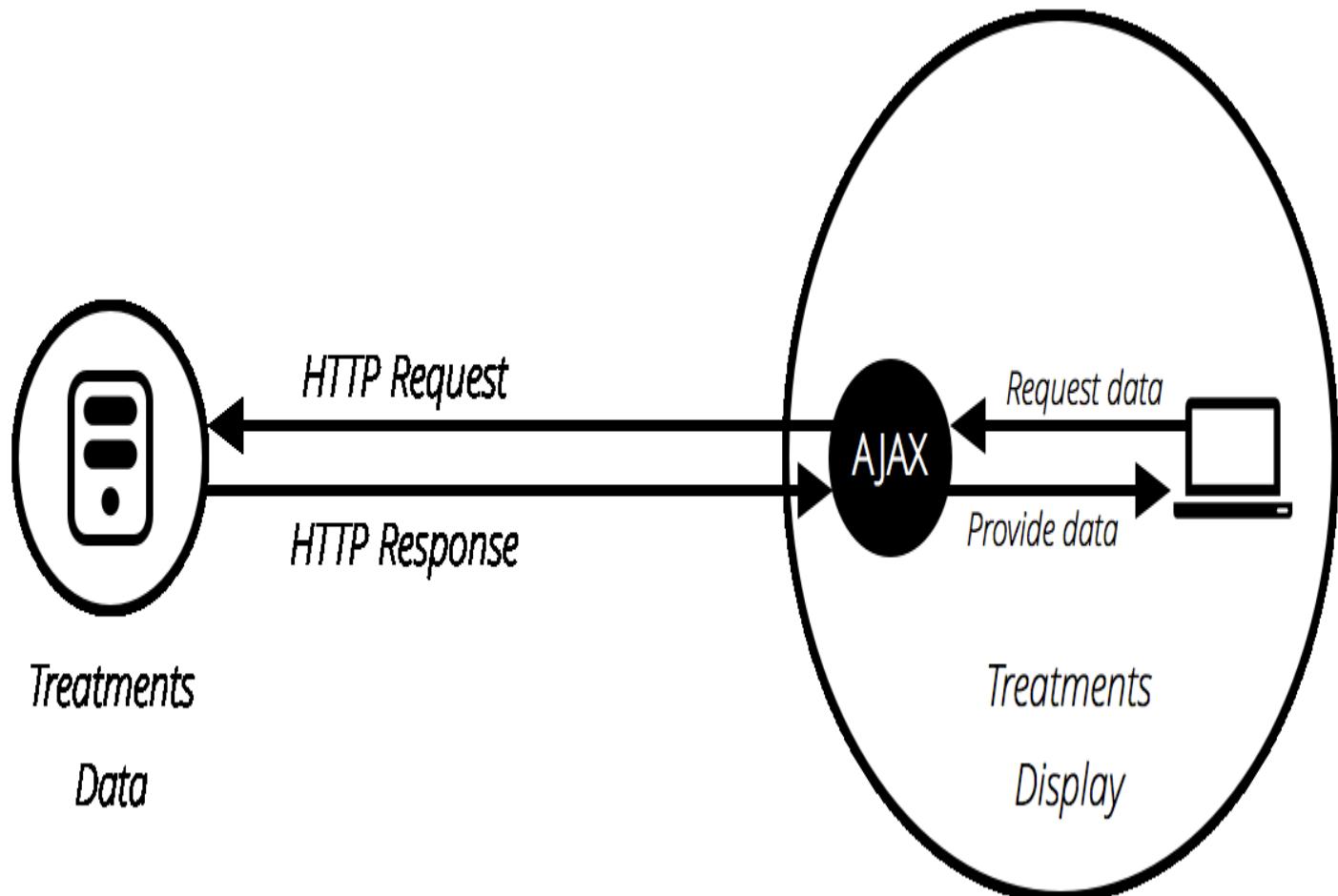
again. This is extremely wasteful and results in a poor user experience, especially as pages get larger and more complex.

## Enter Ajax

This led to the creation of technologies that allow web pages to request small chunks of data (such as [HTML](#), [XML](#), [JSON](#), or plain text) and display them only when needed, helping to solve the problem described above.

This is achieved by using APIs like [XMLHttpRequest](#) or — more recently — the [Fetch API](#). These technologies allow web pages to directly handle making [HTTP](#) requests for specific resources available on a server, and formatting the resulting data as needed, before it is displayed.

**Note:** In the early days, this general technique was known as Asynchronous JavaScript and XML (Ajax), because it tended to use [XMLHttpRequest](#) to request XML data. This is not normally the case these days (you'd be more likely to use [XMLHttpRequest](#) or Fetch to request JSON), but the result is still the same, and the term "Ajax" is still often used to describe the technique.



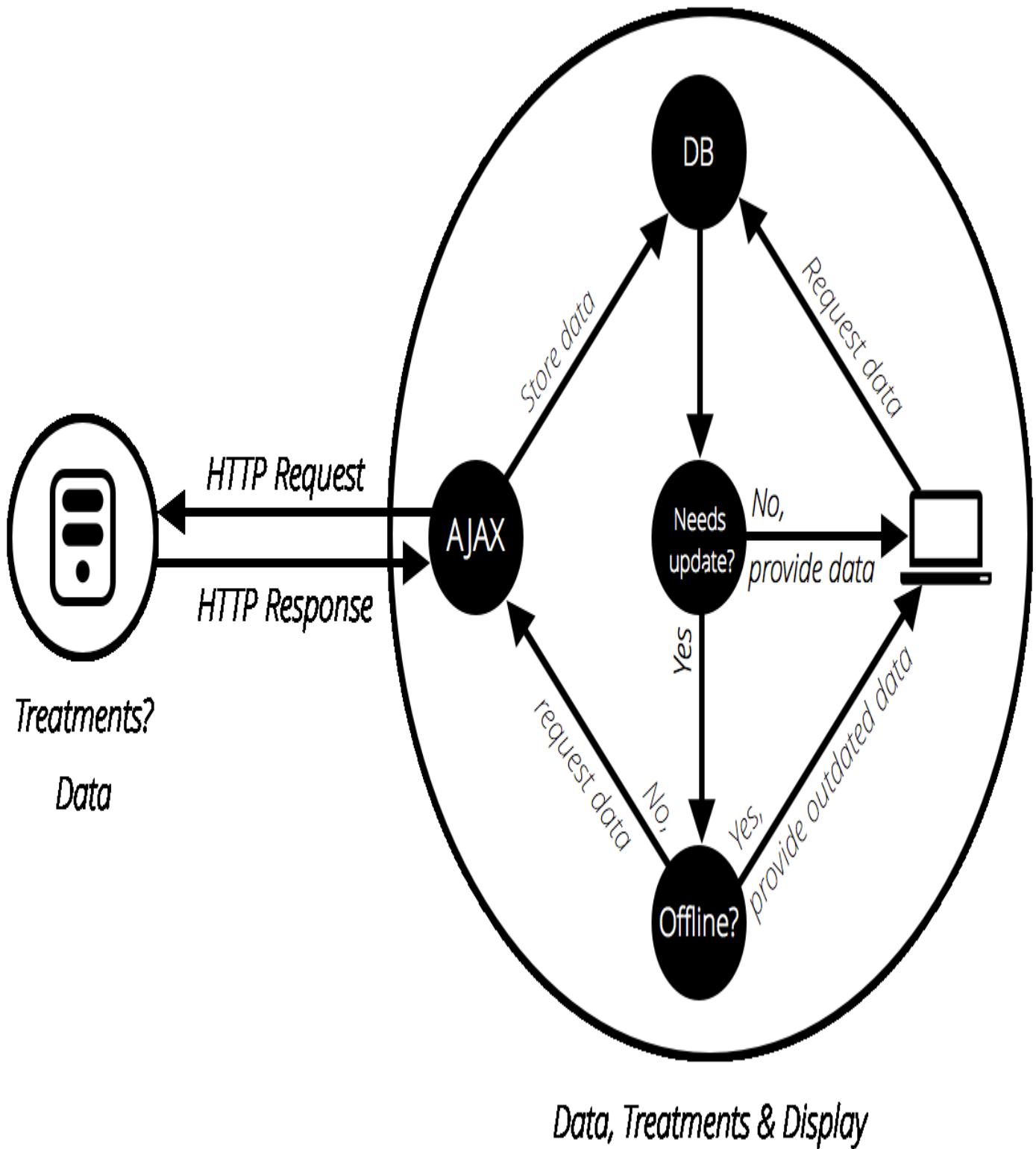
The Ajax model involves using a web API as a proxy to more intelligently request data rather than just having the browser to reload the entire page. Let's think about the significance of this:

1. Go to one of your favorite information-rich sites, like Amazon, YouTube, CNN, etc., and load it.
2. Now search for something, like a new product. The main content will change, but most of the surrounding information, like the header, footer, navigation menu, etc., will stay the same.

This is a really good thing because:

- Page updates are a lot quicker and you don't have to wait for the page to refresh, meaning that the site feels faster and more responsive.
- Less data is downloaded on each update, meaning less wasted bandwidth. This may not be such a big issue on a desktop on a broadband connection, but it's a major issue on mobile devices and in developing countries that don't have ubiquitous fast Internet service.

To speed things up even further, some sites also store assets and data on the user's computer when they are first requested, meaning that on subsequent visits they use the local versions instead of downloading fresh copies when the page is first loaded. The content is only reloaded from the server when it has been updated.



We won't cover such storage techniques in this article; we'll discuss it in a future article later in the module.

# A basic Ajax request

Let's look at how such a request is handled, using both [XMLHttpRequest](#) and [Fetch](#). For these examples, we'll request data out of a few different text files, and use them to populate a content area.

This series of files will act as our fake database; in a real application we'd be more likely to use a server-side language like PHP, Python, or Node to request our data from a database. Here however we want to keep it simple, and concentrate on the client-side part of this.

## XMLHttpRequest

`XMLHttpRequest` (which is frequently abbreviated to XHR) is a fairly old technology now — it was invented by Microsoft in the late 1990's, and has been standardized across browsers for quite a long time.

1. To begin this example, make a local copy of [ajax-start.html](#) and the four text files — [verse1.txt](#), [verse2.txt](#), [verse3.txt](#), and [verse4.txt](#) — in a new directory on your computer. In this example we will load a different verse of the poem (which you may well recognize) via XHR when it's selected in the drop down menu.
2. Just inside the `<script>` element, add the following code. This stores a reference to the `<select>` and `<pre>` elements in variables, and defines an `onchange` event handler function so that when the select's value is changed, its value is passed to an invoked function `updateDisplay()` as a parameter.

```
3. var verseChoose = document.querySelector('select');
4. var poemDisplay = document.querySelector('pre');
5.
6. verseChoose.onchange = function() {
7. var verse = verseChoose.value;
8. updateDisplay(verse);
};
```

- • Let's define our `updateDisplay()` function. First of all, put the following beneath your previous code block — this is the empty shell of the function:

```
function updateDisplay(verse) {
};
```

- • We'll start our function by constructing a relative URL pointing to the text file we want to load, as we'll need it later. The value of the `<select>` element at any time is the same as the text inside the selected `<option>` (unless you specify a different value in a `value` attribute) — so for example "Verse 1". The corresponding verse text file is "verse1.txt", and is in the same directory as the HTML file, therefore just the file name will do.

However, web servers tend to be case sensitive, and the file name hasn't got a space in it. To convert "Verse 1" to "verse1.txt" we need to convert the V to lower case, remove the space, and

add .txt on the end. This can be done with [replace\(\)](#), [toLowerCase\(\)](#), and simple [string concatenation](#). Add the following lines inside your `updateDisplay()` function:

```
verse = verse.replace(" ", "");
verse = verse.toLowerCase();
var url = verse + '.txt';
```

- • To begin creating an XHR request, you need to create a new request object using the [XMLHttpRequest\(\)](#) constructor. You can call this object anything you like, but we'll call it `request` to keep things simple. Add the following below your previous lines:

```
var request = new XMLHttpRequest();
```

- • Next, you need to use the [open\(\)](#) method to specify what [HTTP request method](#) to use to request the resource from the network, and what its URL is. We'll just use the [GET](#) method here, and set the URL as our `url` variable. Add this below your previous line:

```
request.open('GET', url);
```

- • Next, we'll set the type of response we are expecting — which is defined by the request's [responseType](#) property — as `text`. This isn't strictly necessary here — XHR returns text by default — but it is a good idea to get into the habit of setting this in case you want to fetch other types of data in the future. Add this next:

```
request.responseType = 'text';
```

- • Fetching a resource from the network is an [asynchronous](#) operation, meaning that you've got to wait for that operation to complete (e.g., the resource is returned from the network) before you can then do anything with that response, otherwise an error will be thrown. XHR allows you to handle this using its [onload](#) event handler — this is run when the [load](#) event fires (when the response has returned). When this has occurred, the response data will be available in the `response` property of the XHR request object.

Add the following below your last addition. you'll see that inside the `onload` event handler we are setting the [textContent](#) of the `poemDisplay` (the `<pre>` element) to the value of the `request.response` property.

```
request.onload = function() {
 poemDisplay.textContent = request.response;
};
```

- • The above is all setup for the XHR request — it won't actually run until we tell it to, which is done using the [send\(\)](#) method. Add the following below your previous addition to complete the function:

```
request.send();
```

- • One problem with the example as it stands is that it won't show any of the poem when it first loads. To fix this, add the following two lines at the bottom of your code (just above the closing `</script>` tag) to load verse 1 by default, and make sure the `<select>` element always shows the correct value:

```
updateDisplay('Verse 1');
verseChoose.value = 'Verse 1';
```

10.

## Serving your example from a server

Some browsers (including Chrome) will not run XHR requests if you just run the example from a local file. This is because of security restrictions (for more on web security, read [Website security](#)).

To get around this, we need to test the example by running it through a local web server. To find out how to do this, read [How do you set up a local testing server?](#)

## Fetch

The Fetch API is basically a modern replacement for XHR — it was introduced in browsers recently to make asynchronous HTTP requests easier to do in JavaScript, both for developers and other APIs that build on top of Fetch.

Let's convert the last example to use Fetch instead!

1. Make a copy of your previous finished example directory. (If you didn't work through the previous exercise, create a new directory, and inside it make copies of [xhr-basic.html](#) and the four text files — [verse1.txt](#), [verse2.txt](#), [verse3.txt](#), and [verse4.txt](#).)
2. Inside the `updateDisplay()` function, find the XHR code:

```
3. var request = new XMLHttpRequest();
4. request.open('GET', url);
5. request.responseType = 'text';
6.
7. request.onload = function() {
8. poemDisplay.textContent = request.response;
9. };
10.
request.send();
```

- • Replace all the XHR code with this:

```
fetch(url).then(function(response) {
 response.text().then(function(text) {
 poemDisplay.textContent = text;
 });
});
```

- 3.
4. Load the example in your browser (running it through a web server) and it should work just the same as the XHR version, provided you are running a modern browser.

## So what is going on in the Fetch code?

First of all, we invoke the `fetch()` method, passing it the URL of the resource we want to fetch. This is the modern equivalent of `request.open()` in XHR, plus you don't need any equivalent to `.send()`.

After that, you can see the `.then()` method chained onto the end of `fetch()` — this method is a part of [Promises](#), a modern JavaScript feature for performing asynchronous operations. `fetch()` returns a promise, which resolves to the response sent back from the server — we use `.then()` to run some follow-up code after the promise resolves, which is the function we've defined inside it. This is the equivalent of the `onload` event handler in the XHR version.

This function is automatically passed the response from the server as a parameter when the `fetch()` promise resolves. Inside the function we grab the response and run its `text()` method, which basically returns the response as raw text. This is the equivalent of `request.responseText = 'text'` in the XHR version.

You'll see that `text()` also returns a promise, so we chain another `.then()` onto it, inside of which we define a function to receive the raw text that the `text()` promise resolves to.

Inside the inner promise's function, we do much the same as we did in the XHR version — set the `<pre>` element's text content to the `text` value.

### Aside on promises

Promises are a bit confusing the first time you meet them, but don't worry too much about this for now. You'll get used to them after a while, especially as you learn more about modern JavaScript APIs — most of the newer ones are heavily based on promises.

Let's look at the promise structure from above again to see if we can make some more sense of it:

```
fetch(url).then(function(response) {
 response.text().then(function(text) {
 poemDisplay.textContent = text;
 });
});
```

The first line is saying "fetch the resource located at url" (`fetch(url)`) and "then run the specified function when the promise resolves" (`.then(function() { ... })`). "Resolve" means "finish performing the specified operation at some point in the future". The specified operation in this case is to fetch a resource from a specified URL (using an HTTP request), and return the response for us to do something with.

Effectively, the function passed into `then()` is a chunk of code that won't run immediately — instead, it will run at some point in the future when the response has been returned. Note that you could also choose to store your promise in a variable, and chain `.then()` onto that instead. the below code would do the same thing:

```
var myFetch = fetch(url);

myFetch.then(function(response) {
 response.text().then(function(text) {
 poemDisplay.textContent = text;
 });
});
```

Because the `fetch()` method returns a promise that resolves to the HTTP response, any function you define inside a `.then()` chained onto the end of it will automatically be given the response as a parameter. You can call the parameter anything you like — the below example would still work:

```
fetch(url).then(function(dogBiscuits) {
 dogBiscuits.text().then(function(text) {
 poemDisplay.textContent = text;
 });
});
```

But it makes more sense to call the parameter something that describes its contents!

Now let's focus just on the function:

```
function(response) {
 response.text().then(function(text) {
 poemDisplay.textContent = text;
 });
}
```

The response object has a method `text()`, which takes the raw data contained in the response body and turns it into plain text, which is the format we want it in. It also returns a promise (which resolves to the resulting text string), so here we use another `.then()`, inside of which we define another function that dictates what we want to do with that text string. We are just setting the `textContent` property of our poem's `<pre>` element to equal the text string, so this works out pretty simple.

It is also worth noting that you can directly chain multiple promise blocks (`.then()` blocks, but there are other types too) onto the end of one another, passing the result of each block to the next block as you travel down the chain. This makes promises very powerful.

The following block does the same thing as our original example, but is written in a different style:

```
fetch(url).then(function(response) {
```

```
 return response.text()
}).then(function(text) {
 poemDisplay.textContent = text;
});
```

Many developers like this style better, as it is flatter and arguably easier to read for longer promise chains — each subsequent promise comes after the previous one, rather than being inside the previous one (which can get unwieldy). The only other difference is that we've had to include a [return](#) statement in front of `response.text()`, to get it to pass its result on to the next link in the chain.

## Which mechanism should you use?

This really depends on what project you are working on. XHR has been around for a long time now and has very good cross-browser support. Fetch and Promises, on the other hand, are a more recent addition to the web platform, although they're supported well across the browser landscape, with the exception of Internet Explorer and Safari (which at the time of writing has Fetch available in its latest technology preview).

If you need to support older browsers, then an XHR solution might be preferable. If however you are working on a more progressive project and aren't as worried about older browsers, then Fetch could be a good choice.

You should really learn both — Fetch will become more popular as Internet Explorer declines in usage (IE is no longer being developed, in favor of Microsoft's new Edge browser), but you might need XHR for a while yet.

## A more complex example

To round off the article, we'll look at a slightly more complex example that shows some more interesting uses of Fetch. We have created a sample site called The Can Store — it's a fictional supermarket that only sells canned goods. You can find this [example live on GitHub](#), and [see the source code](#).

# The Can Store

Choose a category:

Enter search term:

e.g. beans

**Filter results**



Garden peas



Mushy peas

By default, the site displays all the products, but you can use the form controls in the left hand column to filter them by category, or search term, or both.

There is quite a lot of complex code that deals with filtering the products by category and search terms, manipulating strings so the data displays correctly in the UI, etc. We won't discuss all of it in the article, but you can find extensive comments in the code (see [can-script.js](#)).

We will however explain the Fetch code.

The first block that uses Fetch can be found at the start of the JavaScript:

```
fetch('products.json').then(function(response) {
 if(response.ok) {
```

```

 response.json().then(function(json) {
 products = json;
 initialize();
 });
 } else {
 console.log('Network request for products.json failed with response ' +
response.status + ': ' + response.statusText);
 }
});

```

This looks similar to what we saw before, except that the second promise is inside a conditional statement. In the condition we check to see if the response returned has been successful — the `response.ok` property contains a Boolean that is `true` if the response was OK (e.g. [200 meaning "OK"](#)), or `false` if it was unsuccessful.

If the response was successful, we run the second promise — this time however we use `json()`, not `text()`, as we want to return our response as structured JSON data, not plain text.

If the response was not successful, we print out an error to the console stating that the network request failed, which reports the network status and descriptive message of the response (contained in the `response.status` and `response.statusText` properties, respectively). Of course a complete web site would handle this error more gracefully, by displaying a message on the user's screen and perhaps offering options to remedy the situation.

You can test the fail case yourself:

1. Make a local copy of the example files (download and unpack [the can-store ZIP file](#))
2. Run the code through a web server (as described above, in [Serving your example from a server](#))
3. Modify the path to the file being fetched, to something like 'produc.json' (i.e. make sure it is misspelled)
4. Now load the index file in your browser (e.g. via `localhost:8000`) and look in your browser developer console. You'll see a message along the lines of "Network request for products.json failed with response 404: File not found"

The second Fetch block can be found inside the `fetchBlob()` function:

```

fetch(url).then(function(response) {
 if(response.ok) {
 response.blob().then(function(blob) {
 objectURL = URL.createObjectURL(blob);
 showProduct(objectURL, product);
 });
 } else {
 console.log('Network request for "' + product.name + '" image failed with
response ' + response.status + ': ' + response.statusText);
 }
});

```

This works in much the same way as the previous one, except that instead of using `json()`, we use `blob()` — in this case we want to return our response as an image file, and the data format

we use for that is [Blob](#) — the term is an abbreviation of "Binary Large Object", and can basically be used to represent large file-like objects — such as images or video files.

Once we've successfully received our blob, we create an object URL out of it, using [createObjectURL\(\)](#). This returns a temporary internal URL that points to an object referenced inside the browser. These are not very readable, but you can see what one looks like by opening up the Can Store app, Ctrl-/Right-clicking on an image, and selecting the "View image" option (which might vary slightly depending on what browser you are using). The object URL will be visible inside the address bar, and should be something like this:

```
blob:http://localhost:7800/9b75250e-5279-e249-884f-d03eb1fd84f4
```

### Challenge: An XHR version of the Can Store

We'd like you to have a go at converting the Fetch version of the app to use XHR, as a useful bit of practice. Take a [copy of the ZIP file](#), and try modifying the JavaScript as appropriate.

Some helpful hints:

- You might find the [XMLHttpRequest](#) reference material useful.
- You will basically need to use the same pattern as you saw earlier in the [XHR-basic.html](#) example.
- You will, however, need to add the error handling we showed you in the Fetch version of the Can Store:
  - The response is found in `request.response` after the `load` event has fired, not in a `promise then()`.
  - About the best equivalent to Fetch's `response.ok` in XHR is to check whether `request.status` is equal to 200, or if `request.readyState` is equal to 4.
  - The properties for getting the status and status message are the same, but they are found on the `request` (XHR) object, not the `response` object.

**Note:** If you have trouble with this, feel free to check your code against the finished version on GitHub ([see the source here](#), and also [see it running live](#)).

## Summary

That rounds off our article on fetching data from the server. By this point you should have an idea of how to start working with both XHR and Fetch.

## Third party APIs

The APIs we've covered so far are built into the browser, but not all APIs are. Many large websites and services such as Google Maps, Twitter, Facebook, PayPal, etc. provide APIs allowing developers to make use of their data (e.g. displaying your twitter stream on your blog)

or services (e.g. displaying custom Google Maps on your site, or using Facebook login to log in your users). This article looks at the difference between browser APIs and 3rd party APIs and shows some typical uses of the latter.

**Prerequisites:** JavaScript basics (see [first steps](#), [building blocks](#), [JavaScript objects](#)), the [basics of Client-side APIs](#)

**Objective:** To learn how third-party APIs work, and how to use them to enhance your websites.

## What are third party APIs?

Third party APIs are APIs provided by third parties — generally companies such as Facebook, Twitter, or Google — to allow you to access their functionality via JavaScript and use it on your own site. As we showed in our [introductory APIs article](#), one of the most obvious examples is using the [Google Maps APIs](#) to display custom maps on your pages.

Let's look at our map example again (see the [source code on GitHub](#); [see it live also](#)), and use it to illustrate how third-party APIs differ from browser APIs.

**Note:** You might want to just [get all our code examples](#) at once, in which case you can then just search the repo for the example files you need in each section.

### They are found on third-party servers

Browser APIs are built in to the browser — you can access them from JavaScript immediately. For example, the [Geolocation API](#) used in our example is accessed using the geolocation property of the [Navigator](#) object, which returns a [Geolocation](#) object. This example uses the [getCurrentPosition\(\)](#) method of this object to request the device's current position:

```
navigator.geolocation.getCurrentPosition(function(position) { ... });
```

Third party APIs, on the other hand, are located on third party servers. To access them from JavaScript you first need to connect to the API functionality and make it available on your page. This typically involves first linking to a JavaScript library available on the server via a [`<script>`](#) element, as seen in our example:

```
<script type="text/javascript"
src="https://maps.google.com/maps/api/js?key=AIzaSyDDuGt0E5IEGkcE6ZfrKfUtE9Ko
_de66pA"></script>
```

You can then start using the objects available in that library. For example:

```
var latlng = new
google.maps.LatLng(position.coords.latitude,position.coords.longitude);
var myOptions = {
 zoom: 8,
```

```

 center: latlng,
 mapTypeId: google.maps.MapTypeId.TERRAIN,
 disableDefaultUI: true
 }

var map = new google.maps.Map(document.getElementById("map_canvas"),
myOptions);

```

Here we are creating a new `LatLng` object using the `google.maps.LatLng()` constructor, which contains the latitude and longitude of the location we want to show, as returned from the Geolocation API. Next, we create an options object (`myOptions`) containing this, and other information related to displaying the map. Finally, we actually create the map using the `google.maps.Map()` constructor, which takes as its parameters the element that we want to draw the map onto, and the options object.

This is all the information the Google Maps API needs to plot a simple map. The server you are connecting to handles all the complex stuff, like displaying the correct map tiles for the area being displayed, etc.

**Note:** Some APIs handle access to their functionality slightly differently, requiring the developer to make an HTTP request (see [Fetching data from the server](#)) to a specific URL pattern to retrieve specific data. These are called RESTful APIs, and we'll show an example of this later in the article.

## Permissions are handled differently

Security for browser APIs tends to be handled by permission prompts, as [discussed in our first article](#). The purpose of these is so that the user knows what is going on in the websites they visit and is less likely to fall victim to someone using an API in a malicious way.

Third party APIs have a slightly different permissions system — they tend to use key codes to allow developers access to the API functionality. Look again at the URL of the Google Maps API library we linked to:

[https://maps.google.com/maps/api/js?key=AIzaSyDDuGt0E5IEGkcE6ZfrKfUtE9Ko\\_de66pA](https://maps.google.com/maps/api/js?key=AIzaSyDDuGt0E5IEGkcE6ZfrKfUtE9Ko_de66pA)

The URL parameter provided at the end of the URL is a developer key — the developer of the application must apply to get a key, and then include it in their code in a specific way to be allowed access to the API's functionality. In the case of Google Maps (and other Google APIs), you apply for a key at the [Google Cloud Platform](#).

Other APIs may require that you include the key in a slightly different way, but the pattern is fairly similar for most of them.

The point of requiring a key is so that not just anybody can use API functionality without any kind of accountability. When the developer has registered for a key, they are then known to the API provider, and action can be taken if they start to do anything malicious with the API (such as

tracking people's location or trying to spam the API with loads of requests to stop it working, for example). The easiest action would be to just revoke their API privileges.

## Extending the Google Maps example

Now we've examined the Google Maps API example and looked at how it works, let's add some more functionality to show how to use some other features of the API.

1. To start off this section, make yourself a copy of the [Google Maps starting file](#), in a new directory. If you've already [cloned the examples repository](#), you'll already have a copy of this file, which you can find in the `javascript/apis/third-party-apis/google-maps` directory.
2. Next, get your own developer key using the following steps:
  1. Go to the [Google Cloud Platform API Manager dashboard](#).
  2. Create a new project if you've not already got one.
  3. Click the *Enable API* button.
  4. Choose *Google Maps JavaScript API*.
  5. Click the *Enable* button.
  6. Click *Create credentials*, then choose *API key*.
  7. Copy your API key and replace the existing key in the example's first `<script>` element with your own (the bit between `?key=` and the attribute's closing quote mark `(")`.)

**Note:** Getting Google-related API keys can be a bit difficult — the Google Cloud Platform API Manager has a lot of different screens, and the workflow can differ slightly depending on things like whether you already have an account set up. If you have trouble with this step, we will be glad to help — [Contact us](#).

3. Open up your Google Maps starting file, find the string `INSERT-YOUR-API-KEY-HERE`, and replace it with the actual API key you got from the Google Cloud Platform API Manager dashboard.

### Adding a custom marker

Adding a marker (icon) at a certain point on the map is easy — you just need to create a new marker using the `google.maps.Marker()` constructor, passing it an options object containing the position to display the marker at (as a `LatLng` object), and the `Map` object to display it on.

1. Add the following just below the `var map ...` line:

```
2. var marker = new google.maps.Marker({
3. position: latlng,
4. map: map
});
```

- Now if you refresh your page, you'll see a nice little marker pop up in the center of the map. This is cool, but it is not exactly a custom marker — it is using the default marker icon.

- To use a custom icon, we need to specify it when we create the marker, using its URL. First of all, add the following line above the previous block you added:

```
var iconBase = 'https://maps.google.com/mapfiles/kml/shapes/';
```

- This defines the base URL where all the official Google Maps icons are stored (you could also specify your own icon location if you wished).
- The icon location is specified in the `icon` property of the options object. Update the constructor like so:

```
var marker = new google.maps.Marker({
 position: latlng,
 icon: iconBase + 'flag_maps.png',
 map: map
});
```

3. Here we specify the icon property value as the `iconBase` plus the icon filename, to create the complete URL. Now try reloading your example and you'll see a custom marker displayed on your map!

**Note:** See [Customizing a Google Map: Custom Markers](#) for more information.

**Note:** See [Map marker or Icon names](#) to find out what other icons are available, and see what their reference names are. Their file name will be the icon name they display when you click on them, with ".png" added on the end.

### Displaying a popup when the marker is clicked

Another common use case for Google Maps is displaying more information about a place when its name or marker is clicked (popups are called **info windows** in the Google Maps API). This is also very simple to achieve, so let's have a look at it.

1. First of all, you need to specify a JavaScript string containing HTML that will define the content of the popup. This will be injected into the popup by the API, and can contain just about any content you want. Add the following line below the `google.maps.Marker()` constructor definition:

```
var contentString = '<div id="content"><h2 id="firstHeading" class="firstHeading">Custom info window</h2><p>This is a cool custom info window.</p></div>';
```

- Next, you need to create a new info window object using the `google.maps.InfoWindow()` constructor. Add the following below your previous line:

```
var infowindow = new google.maps.InfoWindow({
 content: contentString
});
```

- There are other properties available (see [Info Windows](#)), but here we are just specifying the `content` property in the options object, which points to the source of the content.
- Finally, to get the popup to display when the marker is clicked, we use a simple click event handler. Add the following below the `google.maps.InfoWindow()` constructor:

```
marker.addListener('click', function() {
 infowindow.open(map, marker);
});
```

3. Inside the function we simply invoke the infowindow's `open()` function, which takes as parameters the map you want to display it on, and the marker you want it to appear next to.
4. Now try reloading the example, and clicking on the marker!

## Controlling what map controls are displayed

Inside the original `google.maps.Map()` constructor, you'll see the property `disableDefaultUI: true` specified. This disables all the standard UI controls you usually get on Google Maps.

1. Try setting its value to `false` (or just removing the line altogether) then reloading your example, and you'll see the map zoom buttons, scale indicator, etc.
2. Now undo your last change.
3. You can show or hide the controls in a more granular fashion by using other properties that specify single UI features. Try adding the following underneath the `disableDefaultUI: true` (remember to put a comma after `disableDefaultUI: true`, otherwise you'll get an error):
  4. `zoomControl: true,`
  5. `mapTypeControl: true,`
  - scaleControl: true,
- 3.
4. Now try reloading the example to see the effect these properties have. You can find more options to experiment with at the [MapOptions object reference page](#).

That's it for now — have a look around the [Google Maps APIs documentation](#), and have some more fun playing!

## A RESTful API — NYTimes

Now let's look at another API example — the [New York Times API](#). This API allows you to retrieve New York Times news story information and display it on your site. This type of API is known as a **RESTful API** — instead of getting data using the features of a JavaScript library like we did with Google Maps, we get data by making HTTP requests to specific URLs, with data like search terms and other properties encoded in the URL (often as URL parameters). This is a common pattern you'll encounter with APIs.

# An approach for using third party APIs

Below we'll take you through an exercise to show you how to use the NYTimes API, which also provides a more general set of steps to follow that you can use as an approach for working with new APIs.

## Find the documentation

When you want to use a third party API, it is essential to find out where the documentation is, so you can find out what features the API has, how you use them, etc. The New York Times API documentation is at <https://developer.nytimes.com/>.

## Get a developer key

Most APIs require you to use some kind of developer key, for reasons of security and accountability. To sign up for an NYTimes API key, you need to go to <https://developer.nytimes.com/signup>.

1. Let's request a key for the "Article Search API" — fill in the form, selecting this as the API you want to use.
2. Next, wait a few minutes, then get the key from your email.
3. Now, to start the example off, make copies of `nytimes_start.html` and `nytimes.css` in a new directory on your computer. If you've already [cloned the examples repository](#), you'll already have a copy of these files, which you can find in the `javascript/apis/third-party-apis/nytimes` directory. Initially the `<script>` element contains a number of variables needed for the setup of the example; below we'll fill in the required functionality.

The app will end up allowing you to type in a search term and optional start and end dates, which it will then use to query the Article Search API and display the search results.

## NY Times video search

Enter a SINGLE search term (required):  Previous 10 Next 10

Enter a start date (format YYYYMMDD):

Enter an end date (format YYYYMMDD):

**[Quarterback Josh McCown Signs One-Year Deal With Jets](#)**

McCown, who will turn 38 this summer, will most likely serve as much as a mentor as he will a placeholder at his position.



Keywords: Football New York Jets McCown, Josh (1979- )  
Petty, Bryce (1991- ) Hackenberg, Christian

### **[A Busy Day for the Redskins, but No Certainty at Quarterback](#)**

Washington signed wide receiver Terrelle Pryor, but there's no guarantee Kirk Cousins will be throwing to him even though Cousins signed a contract for the franchise.



### Connect the API to your app

First, you'll need to make a connection between the API, and your app. This is usually done either by connecting to the API's JavaScript (as we did in the Google Maps API), or by making requests to the correct URL(s).

In the case of this API, you need to include the API key as a [get](#) parameter every time you request data from it.

1. Find the following line:

```
var key = 'INSERT-YOUR-API-KEY-HERE';
```

- Replace INSERT-YOUR-API-KEY-HERE with the actual API key you got in the previous section.
- Add the following line to your JavaScript, below the " // Event listeners to control the functionality" comment. This runs a function called `fetchResults()` when the form is submitted (the button is pressed).

```
searchForm.addEventListener('submit', fetchResults);
```

- Now add the `fetchResults()` function definition, below the previous line:

```
function fetchResults(e) {
 // Use preventDefault() to stop the form submitting
```

```

e.preventDefault();

// Assemble the full URL
url = baseURL + '?api-key=' + key + '&page=' + pageNumber + '&q=' +
searchTerm.value;

if(startDate.value !== '') {
 url += '&begin_date=' + startDate.value;
};

if(endDate.value !== '') {
 url += '&end_date=' + endDate.value;
};

}

```

### 3.

This first calls `preventDefault()` on the event object, to stop the form actually submitting (which would break the example). Next, we use some string manipulation to assemble the full URL that we will make the request to. We start off by assembling the parts we deem as mandatory for this demo:

- The base URL (taken from the `baseURL` variable).
- The API key, which has to be specified in the `api-key` URL parameter (the value is taken from the `key` variable).
- The page number, which has to be specified in the `page` URL parameter (the value is taken from the `pageNumber` variable).
- The search term, which has to be specified in the `q` URL parameter (the value is taken from the value of the `searchTerm` text `<input>`).

Next, we use a couple of `if()` statements to check whether the `startDate` and `endDate` `<input>`s have had values filled in on them. If they do, we append their values to the URL, specified in `begin_date` and `end_date` URL parameters respectively.

So, a complete URL would end up looking something like this:

```
https://api.nytimes.com/svc/search/v2/articlesearch.json?api-
key=4f3c267e125943d79b0a3e679f608a78&page=0&q=cats
&begin_date=20170301&end_date=20170312
```

**Note:** You can find more details of what URL parameters can be included in the [Article Search API reference](#).

**Note:** The example has rudimentary form data validation — the search term field has to be filled in before the form can be submitted (achieved using the `required` attribute), and the date fields have `pattern` attributes specified, which means they won't submit unless their values consist of 8 numbers (`pattern="[0-9]{8}"`). See [Form data validation](#) for more details on how these work.

## Requesting data from the api

Now we've constructed our URL, let's make a request to it. We'll do this using the [Fetch API](#).

Add the following code block inside the `fetchResults()` function, just above the closing curly brace:

```
// Use fetch() to make the request to the API
fetch(url).then(function(result) {
 return result.json();
}).then(function(json) {
 displayResults(json);
});
```

Here we run the request by passing our `url` variable to `fetch()`, convert the response body to JSON using the `json()` function, then pass the resulting JSON to the `displayResults()` function so the data can be displayed in our UI.

## Displaying the data

OK, let's look at how we'll display the data. Add the following function below your `fetchResults()` function.

```
function displayResults(json) {
 while (section.firstChild) {
 section.removeChild(section.firstChild);
 }

 var articles = json.response.docs;

 if(articles.length === 10) {
 nav.style.display = 'block';
 } else {
 nav.style.display = 'none';
 }

 if(articles.length === 0) {
 var para = document.createElement('p');
 para.textContent = 'No results returned.'
 section.appendChild(para);
 } else {
 for(var i = 0; i < articles.length; i++) {
 var article = document.createElement('article');
 var heading = document.createElement('h2');
 var link = document.createElement('a');
 var img = document.createElement('img');
 var para1 = document.createElement('p');
 var para2 = document.createElement('p');
 var clearfix = document.createElement('div');

 var current = articles[i];
 console.log(current);

 heading.textContent = current.title;
 link.href = current.url;
 link.textContent = current.url;
 img.src = current.image;
 para1.textContent = current.summary;
 para2.textContent = current.description;

 article.appendChild(heading);
 article.appendChild(link);
 article.appendChild(img);
 article.appendChild(para1);
 article.appendChild(para2);
 article.appendChild(clearfix);
 }
 }
}
```

```

link.href = current.web_url;
link.textContent = current.headline.main;
para1.textContent = current.lead_paragraph;
para2.textContent = 'Keywords: ';
for(var j = 0; j < current.keywords.length; j++) {
 var span = document.createElement('span');
 span.textContent += current.keywords[j].value + ' ';
 para2.appendChild(span);
}

if(current.multimedia.length > 0) {
 img.src = 'http://www.nytimes.com/' + current.multimedia[0].url;
 img.alt = current.headline.main;
}

clearfix.setAttribute('class','clearfix');

article.appendChild(heading);
heading.appendChild(link);
article.appendChild(img);
article.appendChild(para1);
article.appendChild(para2);
article.appendChild(clearfix);
section.appendChild(article);
}
}
};


```

There's a lot of code here; let's explain it step by step:

- The `while` loop is a common pattern used to delete all of the contents of a DOM element, in this case the `<section>` element. We keep checking to see if the `<section>` has a first child, and if it does, we remove the first child. The loop ends when `<section>` no longer has any children.
- Next, we set the `articles` variable to equal `json.response.docs` — this is the array holding all the objects that represent the articles returned by the search. This is done purely to make the following code a bit simpler.
- The first `if()` block checks to see if 10 articles are returned (the API returns up to 10 articles at a time.) If so, we display the `<nav>` that contains the *Previous 10/Next 10* pagination buttons. If less than 10 articles are returned, they will all fit on one page, so we don't need to show the pagination buttons. We will wire up the pagination functionality in the next section.
- The next `if()` block checks to see if no articles are returned. If so, we don't try to display any — we just create a `<p>` containing the text "No results returned." and insert it into the `<section>`.
- If some articles are returned, we first of all create all the elements that we want to use to display each news story, insert the right contents into each one, and then insert them into the DOM at the appropriate places. To work out which properties in the article objects contained the right data to show, we consulted the [Article Search API reference](#). Most of these operations are fairly obvious, but a few are worth calling out:
  - We used a `for loop` (`for(var j = 0; j < current.keywords.length; j++) { ... }`) to loop through all the keywords associated with each article, and insert each

- one inside its own `<span>`, inside a `<p>`. This was done to make it easy to style each one.
- o We used an `if()` block (`if(current.multimedia.length > 0) { ... }`) to check whether each article actually has any images associated with it (some stories don't.) We display the first image only if it actually exists (otherwise an error would be thrown).
- o We gave our `<div>` element a class of "clearfix", so we can easily apply clearing to it (this technique is needed at the time of writing to stop floated layouts from breaking.)

If you try the example now, it should work, although the pagination buttons won't work yet.

### Wiring up the pagination buttons

To make the pagination buttons work, we will increment (or decrement) the value of the `pageNumber` variable, and then re-run the fetch request with the new value included in the page URL parameter. This works because the NYTimes API only returns 10 results at a time — if more than 10 results are available, it will return the first 10 (0-9) if the `page` URL parameter is set to 0 (or not included at all — 0 is the default value), the next 10 (10-19) if it is set to 1, and so on.

This allows us to easily write a simplistic pagination function.

1. Below the existing `addEventListener()` call, add these two new ones, which cause the `nextPage()` and `previousPage()` functions to be invoked when the relevant buttons are clicked:
2. 

```
nextBtn.addEventListener('click', nextPage);
previousBtn.addEventListener('click', previousPage);
```
- • Below your previous addition, let's define the two functions — add this code now:

```
function nextPage(e) {
 pageNumber++;
 fetchResults(e);
}

function previousPage(e) {
 if(pageNumber > 0) {
 pageNumber--;
 } else {
 return;
 }
 fetchResults(e);
};
```

2. The first function is simple — we increment the `pageNumber` variable, then run the `fetchResults()` function again to display the next page's results.

The second function works nearly exactly the same way in reverse, but we also have to take the extra step of checking that `pageNumber` is not already zero before decrementing

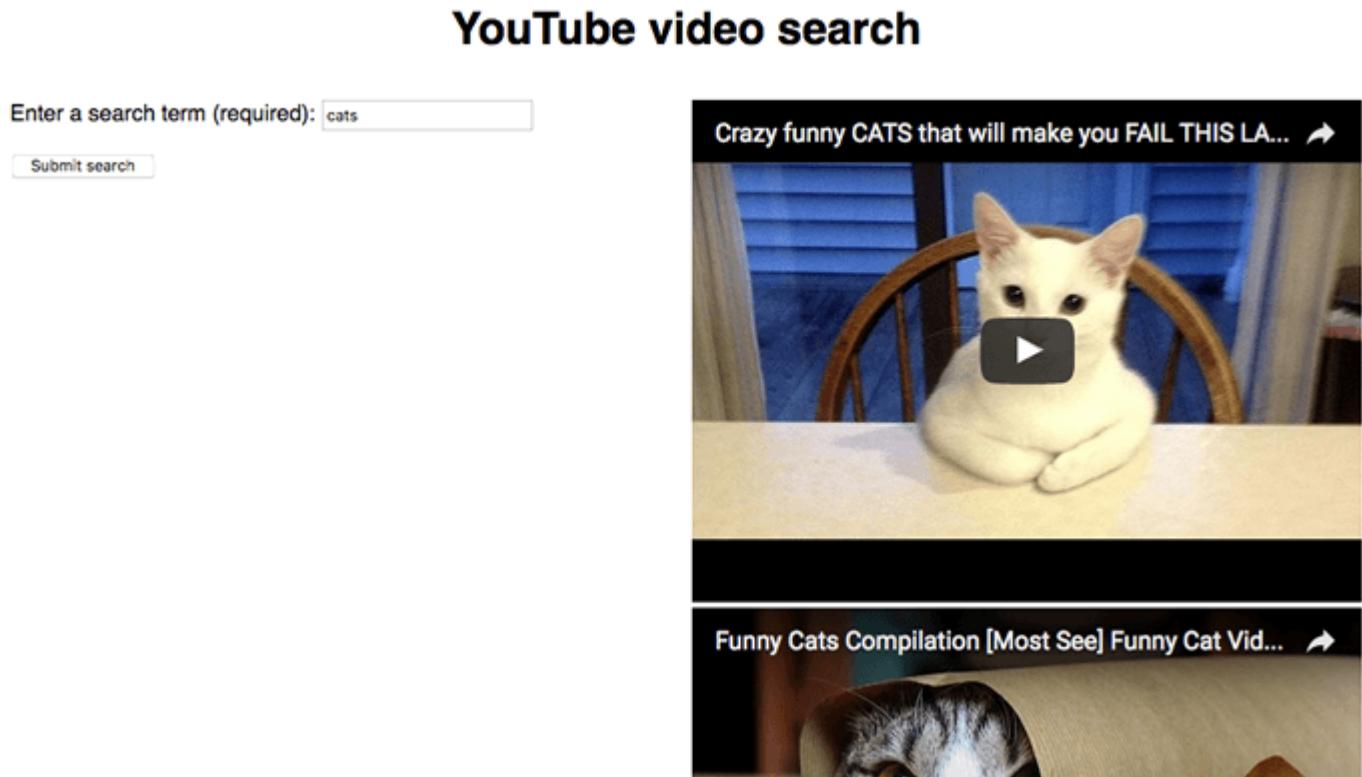
it — if the fetch request runs with a minus `page` URL parameter, it could cause errors. If the `pageNumber` is already 0, we simply `return` out of the function, to avoid wasting processing power (If we are already at the first page, we don't need to load the same results again).

## YouTube example

We also built another example for you to study and learn from — see our [YouTube video search example](#). This uses two related APIs:

- The [YouTube Data API](#) to search YouTube videos and return results.
- The [YouTube IFrame Player API](#) to display the returned video examples inside IFrame video players so you can watch them.

This example is interesting because it shows two related third-party APIs being used together to build an app. The first one is a RESTful API, while the second one works more like Google Maps (with constructors, etc.). It is worth noting however that both of the APIs require a JavaScript library to be applied to the page. The RESTful API has functions available to handle making the HTTP requests and returning the results, so you don't have to write them out yourself using say `fetch` or `XHR`.



We are not going to say too much more about this example in the article — [the source code](#) has detailed comments inserted inside it to explain how it works.

## Summary

This article has given you a useful introduction to using third party APIs to add functionality to your websites.

### Drawing graphics

The browser contains some very powerful graphics programming tools, from the Scalable Vector Graphics ([SVG](#)) language, to APIs for drawing on HTML `<canvas>` elements, (see [The Canvas API](#) and [WebGL](#)). This article provides an introduction to canvas, and further resources to allow you to learn more.

**Prerequisites:** JavaScript basics (see [first steps](#), [building blocks](#), [JavaScript objects](#)), the [basics of Client-side APIs](#)

**Objective:** To learn the basics of drawing on `<canvas>` elements using JavaScript.

## Graphics on the Web

As we talked about in our HTML [Multimedia and embedding](#) module, the Web was originally just text, which was very boring, so images were introduced — first via the `<img>` element and later via CSS properties such as `background-image`, and [SVG](#).

This however was still not enough. While you could use [CSS](#) and [JavaScript](#) to animate (and otherwise manipulate) SVG vector images — as they are represented by markup — there was still no way to do the same for bitmap images, and the tools available were rather limited. The Web still had no way to effectively create animations, games, 3D scenes, and other requirements commonly handled by lower level languages such as C++ or Java.

The situation started to improve when browsers began to support the `<canvas>` element and associated [Canvas API](#) — Apple invented it in around 2004, and other browsers followed by implementing it in the years that followed. As you'll see below, canvas provides many useful tools for creating 2D animations, games, data visualizations, and other types of app, especially when combined with some of the other APIs the web platform provides.

The below example shows a simple 2D canvas-based bouncing balls animation that we originally met in our [Introducing JavaScript objects](#) module:

Around 2006–2007, Mozilla started work on an experimental 3D canvas implementation. This became [WebGL](#), which gained traction among browser vendors, and was standardized around 2009–2010. WebGL allows you to create real 3D graphics inside your web browser; the below example shows a simple rotating WebGL cube:

This article will focus mainly on 2D canvas, as raw WebGL code is very complex. We will however show how to use a WebGL library to create a 3D scene more easily, and you can find a tutorial covering raw WebGL code at [WebGL](#).

**Note:** Basic canvas functionality is supported well across browsers, with the exception of IE 8 and below for 2D canvas, and IE 11 and below for WebGL.

## Active learning: Getting started with a <canvas>

If you want to create a 2D *or* 3D scene on a web page, you need to start with an HTML `<canvas>` element. This element is used to define the area on the page into which the image will be drawn. This is as simple as including the element on the page:

```
<canvas width="320" height="240"></canvas>
```

This will create a canvas on the page with a size of 320 by 240 pixels.

Inside the canvas tags, you can put some fallback content, which is shown if the user's browser doesn't support canvas.

```
<canvas width="320" height="240">
 <p>Your browser doesn't support canvas. Boo hoo!</p>
</canvas>
```

Of course, the above message is really unhelpful! In a real example you'd want to relate the fallback content to the canvas content. For example, if you were rendering a constantly updating graph of stock prices, the fallback content could be a static image of the latest stock graph, with alt text saying what the prices are in text.

### Creating and sizing our canvas

Let's start by creating our own canvas that we draw future experiments on to.

1. First make a local copy of our [0\\_canvas\\_start.html](#) file, and open it in your text editor.
2. Add the following code into it, just below the opening `<body>` tag:
  3.   `<canvas class="myCanvas">`
  4.     `<p>Add suitable fallback here.</p>`
  - `</canvas>`
- We have added a `class` to the `<canvas>` element so it will be easier to select if we have multiple canvases on the page, but we have removed the `width` and `height` attributes for now

(you could add them back in if you wanted, but we will set them using JavaScript in a below section). Canvases with no explicit width and height set default to 300 by 150 pixels.

- Now add the following lines of JavaScript inside the `<script>` element:

```
var canvas = document.querySelector('.myCanvas');
var width = canvas.width = window.innerWidth;
var height = canvas.height = window.innerHeight;
```

- Here we have stored a reference to the canvas in the `canvas` variable. In the second line we set both a new variable `width` and the `canvas`' `width` property equal to `window.innerWidth` (which gives us the viewport width). In the third line we set both a new variable `height` and the `canvas`' `height` property equal to `window.innerHeight` (which gives us the viewport height). So now we have a canvas that fills the entire width and height of the browser window!

You'll also see that we are chaining variable assignments together with multiple equals signs — this is allowed in JavaScript, and it is a good technique if you want to make multiple variables all equal to the same value. We wanted to make the canvas width and height easily accessible in the `width/height` variables, as they are useful values to have available for later (for example, if you want to draw something exactly halfway across the width of the canvas).

- If you save and load your example in a browser now, you'll see nothing, which is fine, but you'll also see scrollbars — this is a problem for us, which happens because the `<body>` element has a `margin` that, added to our full-window-size canvas, results in a document that's wider than the window. To get rid of the scrollbars, we need to remove the `margin` and also set `overflow` to `hidden`. Add the following into the `<head>` of your document:

```
<style>
 body {
 margin: 0;
 overflow: hidden;
 }
</style>
```

4. The scrollbars should now be gone.

**Note:** You should generally set the size of the image using HTML attributes or DOM properties, as explained above. You could use CSS, but the trouble then is that the sizing is done after the canvas has rendered, and just like any other image (the rendered canvas is just an image), the image could become pixellated/distorted.

## Getting the canvas context and final setup

We need to do one final thing before we can consider our canvas template finished. To draw onto the canvas we need to get a special reference to the drawing area called a context. This is done using the `HTMLCanvasElement.getContext()` method, which for basic usage takes a single string as a parameter representing the type of context you want to retrieve.

In this case we want a 2d canvas, so add the following JavaScript line below the others inside the `<script>` element:

```
var ctx = canvas.getContext('2d');
```

**Note:** other context values you could choose include `webgl` for WebGL, `webgl2` for WebGL 2, etc., but we won't need those in this article.

So that's it — our canvas is now primed and ready for drawing on! The `ctx` variable now contains a `CanvasRenderingContext2D` object, and all drawing operations on the canvas will involve manipulating this object.

Let's do one last thing before we move on. We'll color the canvas background black to give you a first taste of the canvas API. Add the following lines at the bottom of your JavaScript:

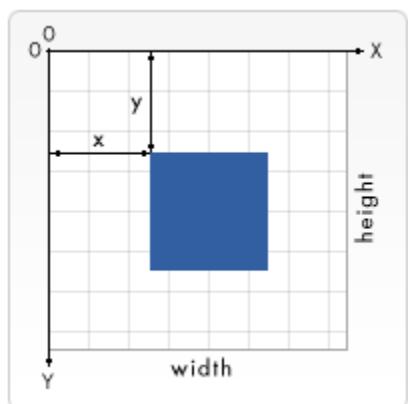
```
ctx.fillStyle = 'rgb(0, 0, 0)';
ctx.fillRect(0, 0, width, height);
```

Here we are setting a fill color using the canvas' `fillStyle` property (this takes [color values](#) just like CSS properties do), then drawing a rectangle that covers the entire area of the canvas with the `fillRect` method (the first two parameters are the coordinates of the rectangle's top left hand corner; the last two are the width and height you want the rectangle drawn at — we told you those `width` and `height` variables would be useful)!

OK, our template is done and it's time to move on.

## 2D canvas basics

As we said above, all drawing operations are done by manipulating a `CanvasRenderingContext2D` object (in our case, `ctx`). Many operations need to be given coordinates to pinpoint exactly where to draw something — the top left of the canvas is point  $(0, 0)$ , the horizontal (`x`) axis runs from left to right, and the vertical (`y`) axis runs from top to bottom.



Drawing shapes tends to be done using the rectangle shape primitive, or by tracing a line along a certain path and then filling in the shape. Below we'll show how to do both.

## Simple rectangles

Let's start with some simple rectangles.

1. First of all, take a copy of your newly coded canvas template (or make a local copy of [1\\_canvas\\_template.html](#) if you didn't follow the above steps).

2. Next, add the following lines to the bottom of your JavaScript:

```
ctx.fillStyle = 'rgb(255, 0, 0)';
ctx.fillRect(50, 50, 100, 150);
```

- If you save and refresh, you should see a red rectangle has appeared on your canvas. Its top left corner is 50 pixels away from the top and left of the canvas edge (as defined by the first two parameters), and it is 100 pixels wide and 150 pixels tall (as defined by the third and fourth parameters).
- Let's add another rectangle into the mix — a green one this time. Add the following at the bottom of your JavaScript:

```
ctx.fillStyle = 'rgb(0, 255, 0)';
ctx.fillRect(75, 75, 100, 100);
```

- Save and refresh, and you'll see your new rectangle. This raises an important point: graphics operations like drawing rectangles, lines, and so forth are performed in the order in which they occur. Think of it like painting a wall, where each coat of paint overlaps and may even hide what's underneath. You can't do anything to change this, so you have to think carefully about the order in which you draw the graphics.

- Note that you can draw semi-transparent graphics by specifying a semi-transparent color, for example by using `rgba()`. The `a` value defines what's called the "alpha channel," or the amount of transparency the color has. The higher its value, the more it will obscure whatever's behind it. Add the following to your code:

```
ctx.fillStyle = 'rgba(0, 0, 0, 0.25)';
ctx.fillRect(25, 100, 50, 50);
```

4.

5. Now try drawing some more rectangles of your own; have fun!

## Strokes and line widths

So far we've looked at drawing filled rectangles, but you can also draw rectangles that are just outlines (called **strokes** in graphic design). To set the color you want for your stroke, you use the `strokeStyle` property; drawing a stroke rectangle is done using `strokeRect`.

1. Add the following to the previous example, again below the previous JavaScript lines:
2. 

```
ctx.strokeStyle = 'rgb(255, 255, 255)';
ctx.strokeRect(25, 25, 175, 200);
```

- • The default width of strokes is 1 pixel; you can adjust the `lineWidth` property value to change this (it takes a number representing the number of pixels wide the stroke is). Add the following line in between the previous two lines:

```
ctx.lineWidth = 5;
```

2.

Now you should see that your white outline has become much thicker! That's it for now. At this point your example should look like this:

**Note:** The finished code is available on GitHub as [2\\_canvas\\_rectangles.html](#).

## Drawing paths

If you want to draw anything more complex than a rectangle, you need to draw a path. Basically, this involves writing code to specify exactly what path the pen should move along on your canvas to trace the shape you want to draw. Canvas includes functions for drawing straight lines, circles, Bézier curves, and more.

Let's start the section off by making a fresh copy of our canvas template ([1\\_canvas\\_template.html](#)), in which to draw the new example.

We'll be using some common methods and properties across all of the below sections:

- `beginPath()` — start drawing a path at the point where the pen currently is on the canvas. On a new canvas, the pen starts out at (0, 0).
- `moveTo()` — move the pen to a different point on the canvas, without recording or tracing the line; the pen simply "jumps" to the new position.
- `fill()` — draw a filled shape by filling in the path you've traced so far.
- `stroke()` — draw an outline shape by drawing a stroke along the path you've drawn so far.
- You can also use features like `lineWidth` and `fillStyle/strokeStyle` with paths as well as rectangles.

A typical simple path drawing option would look something like so:

```
ctx.fillStyle = 'rgb(255, 0, 0)';
ctx.beginPath();
ctx.moveTo(50, 50);
// draw your path
ctx.fill();
```

## Drawing lines

Let's draw an equilateral triangle on the canvas.

1. First of all, add the following helper function to the bottom of your code. This converts degree values to radians, which is useful because whenever you need to provide an angle value in JavaScript, it will nearly always be in radians, but humans usually think in degrees.

```
2. function degToRad(degrees) {
3. return degrees * Math.PI / 180;
 };
```

- • Next, start off your path by adding the following below your previous addition; here we set a color for our triangle, start drawing a path, and then move the pen to (50, 50) without drawing anything. That's where we'll start drawing our triangle.

```
ctx.fillStyle = 'rgb(255, 0, 0)';
ctx.beginPath();
ctx.moveTo(50, 50);
```

- • Now add the following lines at the bottom of your script:

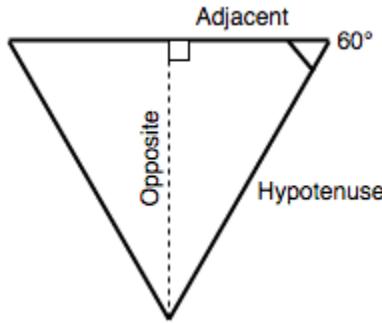
```
ctx.lineTo(150, 50);
var triHeight = 50 * Math.tan(degToRad(60));
ctx.lineTo(100, 50+triHeight);
ctx.lineTo(50, 50);
ctx.fill();
```

3. Let's run through this in order:

First we draw a line across to (150, 50) — our path now goes 100 pixels to the right along the x axis.

Second, we work out the height of our equalateral triangle, using a bit of simple trigonometry. Basically, we are drawing the triangle pointing downwards. The angles in an equalateral triangle are always 60 degrees; to work out the height we can split it down the middle into two right-angled triangles, which will each have angles of 90 degrees, 60 degrees, and 30 degrees. In terms of the sides:

- The longest side is called the **hypotenuse**
- The side next to the 60 degree angle is called the **adjacent** — which we know is 50 pixels, as it is half of the line we just drew.
- The side opposite the 60 degree angle is called the **opposite**, which is the height of the triangle we want to calculate.



One of the basic trigonometric formulae states that the length of the adjacent multiplied by the tangent of the angle is equal to the opposite, hence we come up with  $50 * \text{Math.tan}(\text{degToRad}(60))$ . We use our `degToRad()` function to convert 60 degrees to radians, as `Math.tan()` expects an input value in radians.

4. With the height calculated, we draw another line to `(100, 50+triHeight)`. The X coordinate is simple; it must be halfway between the previous two X values we set. The Y value on the other hand must be 50 plus the triangle height, as we know the top of the triangle is 50 pixels from the top of the canvas.
5. The next line draws a line back to the starting point of the triangle.
6. Last of all, we run `ctx.fill()` to end the path and fill in the shape.

## Drawing circles

Now let's look at how to draw a circle in canvas. This is accomplished using the `arc()` method, which draws all or part of a circle at a specified point.

1. Let's add an arc to our canvas — add the following to the bottom of your code:
2. `ctx.fillStyle = 'rgb(0, 0, 255)';`
3. `ctx.beginPath();`
4. `ctx.arc(150, 106, 50, degToRad(0), degToRad(360), false);`  
`ctx.fill();`

- `arc()` takes six parameters. The first two specify the position of the arc's center (X and Y, respectively). The third is the circle's radius, the fourth and fifth are the start and end angles at which to draw the circle (so specifying 0 and 360 degrees gives us a full circle), and the sixth parameter defines whether the circle should be drawn counterclockwise (anticlockwise) or clockwise (`false` is clockwise).

**Note:** 0 degrees is horizontally to the right.

- Let's try adding another arc:

```
ctx.fillStyle = 'yellow';
ctx.beginPath();
ctx.arc(200, 106, 50, degToRad(-45), degToRad(45), true);
ctx.lineTo(200, 106);
```

```
ctx.fill();
```

2. The pattern here is very similar, but with two differences:

- We have set the last parameter of `arc()` to `true`, meaning that the arc is drawn counterclockwise, which means that even though the arc is specified as starting at -45 degrees and ending at 45 degrees, we draw the arc around the 270 degrees not inside this portion. If you were to change `true` to `false` and then re-run the code, only the 90 degree slice of the circle would be drawn.
- Before calling `fill()`, we draw a line to the center of the circle. This means that we get the rather nice Pac-Man-style cutout rendered. If you removed this line (try it!) then re-ran the code, you'd get just an edge of the circle chopped off between the start and end point of the arc. This illustrates another important point of the canvas — if you try to fill an incomplete path (i.e. one that is not closed), the browser fills in a straight line between the start and end point and then fills it in.

That's it for now; your final example should look like this:

**Note:** The finished code is available on GitHub as [3\\_canvas\\_paths.html](#).

**Note:** To find out more about advanced path drawing features such as Bézier curves, check out our [Drawing shapes with canvas](#) tutorial.

## Text

Canvas also has features for drawing text. Let's explore these briefly. Start by making another fresh copy of our canvas template ([1\\_canvas\\_template.html](#)) in which to draw the new example.

Text is drawn using two methods:

- `fillText()` — draws filled text.
- `strokeText()` — draws outline (stroke) text.

Both of these take three properties in their basic usage: the text string to draw and the X and Y coordinates of the top left corner of the **text box** (literally, the box surrounding the text you draw).

There are also a number of properties to help control text rendering such as `font`, which lets you specify font family, size, etc. It takes as its value the same syntax as the CSS `font` property.

Try adding the following block to the bottom of your JavaScript:

```
ctx.strokeStyle = 'white';
ctx.lineWidth = 1;
ctx.font = '36px arial';
ctx.strokeText('Canvas text', 50, 50);

ctx.fillStyle = 'red';
ctx.font = '48px georgia';
```

```
ctx.fillText('Canvas text', 50, 150);
```

Here we draw two lines of text, one outline and the other stroke. The final example should look like so:

**Note:** The finished code is available on GitHub as [4\\_canvas\\_text.html](#).

Have a play and see what you can come up with! You can find more information on the options available for canvas text at [Drawing text](#).

## Drawing images onto canvas

It is possible to render external images onto your canvas. These can be simple images, frames from videos, or the content of other canvases. For the moment we'll just look at the case of using some simple images on our canvas.

1. As before, make another fresh copy of our canvas template ([1\\_canvas\\_template.html](#)) in which to draw the new example. In this case you'll also need to save a copy of our sample image — [firefox.png](#) — in the same directory.

Images are drawn onto canvas using the `drawImage()` method. The simplest version takes three parameters — a reference to the image you want to render, and the X and Y coordinates of the image's top left corner.

2. Let's start by getting an image source to embed in our canvas. Add the following lines to the bottom of your JavaScript:

```
3. var image = new Image();
image.src = 'firefox.png';
```

- Here we create a new `HTMLImageElement` object using the `Image()` constructor. The returned object is the same type as that which is returned when you grab a reference to an existing `<img>` element). We then set its `src` attribute to equal our Firefox logo image. At this point, the browser starts loading the image.

- We could now try to embed the image using `drawImage()`, but we need to make sure the image file has been loaded first, otherwise the code will fail. We can achieve this using the `onload` event handler, which will only be invoked when the image has finished loading. Add the following block below the previous one:

```
image.onload = function() {
 ctx.drawImage(image, 50, 50);
}
```

- If you load your example in the browser now, you should see the image embeded in the canvas.

- But there's more! What if we want to display only a part of the image, or to resize it? We can do both with the more complex version of `drawImage()`. Update your `ctx.drawImage()` line like so:

```
ctx.drawImage(image, 20, 20, 185, 175, 50, 50, 185, 175);
```

4.

- The first parameter is the image reference, as before.
- Parameters 2 and 3 define the coordinates of the top left corner of the area you want to cut out of the loaded image, relative to the top-left corner of the image itself. Nothing to the left of the first parameter or above the second will be drawn.
- Parameters 4 and 5 define the width and height of the area we want to cut out from the original image we loaded.
- Parameters 6 and 7 define the coordinates at which you want to draw the top-left corner of the cut-out portion of the image, relative to the top-left corner of the canvas.
- Parameters 8 and 9 define the width and height to draw the cut-out area of the image. In this case, we have specified the same dimensions as the original slice, but you could resize it by specifying different values.

The final example should look like so:

**Note:** The finished code is available on GitHub as [5\\_canvas\\_images.html](#).

## Loops and animations

We have so far covered some very basic uses of 2D canvas, but really you won't experience the full power of canvas unless you update or animate it in some way. After all, canvas does provide scriptable images! If you aren't going to change anything, then you might as well just use a static images and save yourself all the work.

### Creating a loop

Playing with loops in canvas is rather fun — you can run canvas commands inside a `for` (or other type of) loop just like any other JavaScript code.

Let's build a simple example.

1. Make another fresh copy of our canvas template ([1\\_canvas\\_template.html](#)) and open it in your code editor.
2. Add the following line to the bottom of your JavaScript. This contains a new method, `translate()`, which moves the origin point of the canvas:

```
ctx.translate(width/2, height/2);
```

- This causes the coordinate origin (0, 0) to be moved to the center of the canvas, rather than being at the top left corner. This is very useful in many situations, like this one, where we want our design to be drawn relative to the center of the canvas.

- Now add the following code to the bottom of the JavaScript:

```
function degToRad(degrees) {
 return degrees * Math.PI / 180;
}

function rand(min, max) {
 return Math.floor(Math.random() * (max-min+1)) + (min);
}

var length = 250;
var moveOffset = 20;

for(var i = 0; i < length; i++) {

}
```

- Here we are implementing the same `degToRad()` function we saw in the triangle example above, a `rand()` function that returns a random number between given lower and upper bounds, `length` and `moveOffset` variables (which we'll find out more about later), and an empty `for` loop.
- The idea here is that we'll draw something on the canvas inside the `for` loop, and iterate on it each time so we can create something interesting. Add the following code inside your `for` loop:

```
ctx.fillStyle = 'rgba(' + (255-length) + ', 0, ' + (255-length) + ', 0.9)';
ctx.beginPath();
ctx.moveTo(moveOffset, moveOffset);
ctx.lineTo(moveOffset+length, moveOffset);
var triHeight = length/2 * Math.tan(degToRad(60));
ctx.lineTo(moveOffset+(length/2), moveOffset+triHeight);
ctx.lineTo(moveOffset, moveOffset);
ctx.fill();

length--;
moveOffset += 0.7;
ctx.rotate(degToRad(5));
```

#### 4. So on each iteration, we:

- Set the `fillStyle` to be a shade of slightly transparent purple, which changes each time based on the value of `length`. As you'll see later the `length` gets smaller each time the loop runs, so the effect here is that the color gets brighter with each successive triangle drawn.
- Begin the path.
- Move the pen to a coordinate of `(moveOffset, moveOffset)`; This variable defines how far we want to move each time we draw a new triangle.

- Draw a line to a coordinate of `(moveOffset+length, moveOffset)`. This draws a line of length `length` parallel to the X axis.
- Calculate the triangle's height, as before.
- Draw a line to the downward-pointing corner of the triangle, then draw a line back to the start of the triangle.
- Call `fill()` to fill in the triangle.
- Update the variables that describe the sequence of triangles, so we can be ready to draw the next one. We decrease the `length` value by 1, so the triangles get smaller each time; increase `moveOffset` by a small amount so each successive triangle is slightly further away, and use another new function, `rotate()`, which allows us to rotate the entire canvas! We rotate it by 5 degrees before drawing the next triangle.

That's it! The final example should look like so:

At this point, we'd like to encourage you to play with the example and make it your own! For example:

- Draw rectangles or arcs instead of triangles, or even embed images.
- Play with the `length` and `moveOffset` values.
- Introduce some random numbers using that `rand()` function we included above but didn't use.

**Note:** The finished code is available on GitHub as [6\\_canvas\\_for\\_loop.html](#).

## Animations

The loop example we built above was fun, but really you need a constant loop that keeps going and going for any serious canvas applications (such as games and real time visualizations). If you think of your canvas as being like a movie, you really want the display to update on each frame to show the updated view, with an ideal refresh rate of 60 frames per second so that movement appears nice and smooth to the human eye.

There are a few JavaScript functions that will allow you to run functions repeatedly, several times a second, the best one for our purposes here being `window.requestAnimationFrame()`. It takes one parameter — the name of the function you want to run for each frame. The next time the browser is ready to update the screen, your function will get called. If that function draws the new update to your animation, then calls `requestAnimationFrame()` again just before the end of the function, the animation loop will continue to run. The loop ends when you stop calling `requestAnimationFrame()` or if you call `window.cancelAnimationFrame()` after calling `requestAnimationFrame()` but before the frame is called.

**Note:** It's good practice to call `cancelAnimationFrame()` from your main code when you're done using the animation, to ensure that no updates are still waiting to be run.

The browser works out complex details such as making the animation run at a consistent speed, and not wasting resources animating things that can't be seen.

To see how it works, let's quickly look again at our Bouncing Balls example ([see it live](#), and also see [the source code](#)). The code for the loop that keeps everything moving looks like this:

```
function loop() {
 ctx.fillStyle = 'rgba(0, 0, 0, 0.25)';
 ctx.fillRect(0, 0, width, height);

 while(balls.length < 25) {
 var ball = new Ball();
 balls.push(ball);
 }

 for(i = 0; i < balls.length; i++) {
 balls[i].draw();
 balls[i].update();
 balls[i].collisionDetect();
 }

 requestAnimationFrame(loop);
}

loop();
```

We run the `loop()` function once at the bottom of the code to start the cycle, drawing the first animation frame; the `loop()` function then takes charge of calling `requestAnimationFrame(loop)` to run the next frame of the animation, again and again.

Note that on each frame we are completely clearing the canvas and redrawing everything. (We create a new ball on each frame (up to a maximum of 25), then for every ball draw it, update its position, and check to see if it is colliding with any other balls). Once you've drawn a graphic to a canvas, there's no way to manipulate that graphic individually like you can with DOM elements. You can't move each ball around on the canvas, because once it's drawn, it's part of the canvas, and is no longer a ball. Instead, you have to erase and redraw, either by erasing the entire frame and redrawing everything, or by having code that knows exactly what parts need to be erased and only erases and redraws the minimum area of the canvas necessary.

Optimizing animation of graphics is an entire specialty of programming, with lots of clever techniques available. Those are beyond what we need for our example, though!

In general, the process of doing a canvas animation involves the following steps:

1. Clear the canvas contents (e.g. with `fillRect()` or `clearRect()`).
2. Save state (if necessary) using `save()` — this is needed when you want to save settings you've updated on the canvas before continuing, which is useful for more advanced applications.
3. Draw the graphics you are animating.
4. Restore the settings you saved in step 2, using `restore()`
5. Call `requestAnimationFrame()` to schedule drawing of the next frame of the animation.

**Note:** We won't cover `save()` and `restore()` here, but they are explained nicely in our [Transformations](#) tutorial (and the ones that follow it).

## A simple character animation

Now let's create our own simple animation — we'll get a character from a certain rather awesome retro computer game to walk across the screen.

1. Make another fresh copy of our canvas template ([1\\_canvas\\_template.html](#)) and open it in your code editor. Make a copy of [walk-right.png](#) in the same directory.
2. At the bottom of the JavaScript, add the following line to once again make the coordinate origin sit in the middle of the canvas:

```
ctx.translate(width/2, height/2);
```

- Now let's create a new `HTMLImageElement` object, set its `src` to the image we want to load, and add an `onload` event handler that will cause the `draw()` function to fire when the image is loaded:

```
var image = new Image();
image.src = 'walk-right.png';
image.onload = draw;
```

- Now we'll add some variables to keep track of the position the sprite is to be drawn on the screen, and the sprite number we want to display.

```
var sprite = 0;
var posX = 0;
```

- Let's explain the spritesheet image (which we have respectfully borrowed from Mike Thomas' [Create a sprite sheet walk cycle using CSS animation](#)). The image looks like this:



It contains six sprites that make up the whole walking sequence — each one is 102 pixels wide and 148 pixels high. To display each sprite cleanly we will have to use `drawImage()` to chop out a single sprite image from the spritesheet and display only that part, like we did above with the Firefox logo. The X coordinate of the slice will have to be a multiple of 102, and the Y coordinate will always be 0. The slice size will always be 102 by 148 pixels.

- Now let's insert an empty `draw()` function at the bottom of the code, ready for filling up with some code:

```
function draw() {
};
```

- • the rest of the code in this section goes inside `draw()`. First, add the following line, which clears the canvas to prepare for drawing each frame. Notice that we have to specify the top-left corner of the rectangle as `-(width/2)`, `-(height/2)` because we specified the origin position as `width/2`, `height/2` earlier on.

```
ctx.fillRect(-(width/2), -(height/2), width, height);
```

- • Next, we'll draw our image using `drawImage` — the 9-parameter version. Add the following:

```
ctx.drawImage(image, (sprite*102), 0, 102, 148, 0+posX, -74, 102, 148);
```

- As you can see:

- We specify `image` as the image to embed.
- Parameters 2 and 3 specify the top-left corner of the slice to cut out of the source image, with the X value as `sprite` multiplied by 102 (where `sprite` is the sprite number between 0 and 5) and the Y value always 0.
- Parameters 4 and 5 specify the size of the slice to cut out — 102 pixels by 148 pixels.
- Parameters 6 and 7 specify the top-left corner of the box into which to draw the slice on the canvas — the X position is `0 + posX`, meaning that we can alter the drawing position by altering the `posX` value.
- Parameters 8 and 9 specify the size of the image on the canvas. We just want to keep its original size, so we specify 102 and 148 as the width and height.

- Now we'll alter the `sprite` value after each draw — well, after some of them anyway. Add the following block to the bottom of the `draw()` function:

```
if (posX % 13 === 0) {
 if (sprite === 5) {
 sprite = 0;
 } else {
 sprite++;
 }
}
```

- We are wrapping the whole block in `if (posX % 13 === 0) { ... }`. We use the modulo (%) operator (also known as the [remainder operator](#)) to check whether the `posX` value can be exactly divided by 13 with no remainder. If so, we move on to the next sprite by incrementing `sprite` (wrapping to 0 after we're done with sprite #5). This effectively means that we are only updating the sprite on every 13th frame, or roughly about 5 frames a second (`requestAnimationFrame()` calls us at up to 60 frames per second if possible). We are deliberately slowing down the frame rate because we only have six sprites to work with, and if we display one every 60th of a second, our character will move way too fast!

Inside the outer block we use an `if ... else` statement to check whether the `sprite` value is at 5 (the last sprite, given that the sprite numbers run from 0 to 5). If we are showing the last sprite already, we reset `sprite` back to 0; if not we just increment it by 1.

- Next we need to work out how to change the `posX` value on each frame — add the following code block just below your last one.

```
if(posX > width/2) {
 newStartPos = -((width/2) + 102);
 posX = Math.ceil(newStartPos / 13) * 13;
 console.log(posX);
} else {
 posX += 2;
}
```

- We are using another `if ... else` statement to see if the value of `posX` has become greater than `width/2`, which means our character has walked off the right edge of the screen. If so, we calculate a position that would put the character just to the left of the left side of the screen, and then set `posX` to equal the multiple of 13 closest to that number. This has to be a multiple of 13 because otherwise the previous code block won't work, since `posX` would never equal a multiple of 13!

If our character hasn't yet walked off the edge of the screen, we simply increment `posX` by 2. This will make him move a little bit to the right the next time we draw him.

- Finally, we need to make the animation loop by calling `requestAnimationFrame()` at the bottom of the `draw()` function:

```
window.requestAnimationFrame(draw);
```

10.

That's it! The final example should look like so:

**Note:** The finished code is available on GitHub as [7\\_canvas\\_walking\\_animation.html](#).

## A simple drawing application

As a final animation example, we'd like to show you a very simple drawing application, to illustrate how the animation loop can be combined with user input (like mouse movement, in this case). We won't get you to walk through and build this one; we'll just explore the most interesting parts of the code.

The example can be found on GitHub as [8\\_canvas\\_drawing\\_app.html](#), and you can play with it live below:

Let's look at the most interesting parts. First of all, we keep track of the mouse's X and Y coordinates and whether it is being clicked or not with three variables: `curX`, `curY`, and `pressed`. When the mouse moves, we fire a function set as the `onmousemove` event handler, which captures the current X and Y values. We also use `onmousedown` and `onmouseup` event handlers to change the value of `pressed` to `true` when the mouse button is pressed, and back to `false` again when it is released.

```
var curX;
var curY;
var pressed = false;

document.onmousemove = function(e) {
 curX = (window.Event) ? e.pageX : e.clientX +
 (document.documentElement.scrollLeft ? document.documentElement.scrollLeft :
 document.body.scrollLeft);
 curY = (window.Event) ? e.pageY : e.clientY +
 (document.documentElement.scrollTop ? document.documentElement.scrollTop :
 document.body.scrollTop);
}

canvas.onmousedown = function() {
 pressed = true;
};

canvas.onmouseup = function() {
 pressed = false;
}
```

When the "Clear canvas" button is pressed, we run a simple function that clears the whole canvas back to black, the same way we've seen before:

```
clearBtn.onclick = function() {
 ctx.fillStyle = 'rgb(0, 0, 0)';
 ctx.fillRect(0, 0, width, height);
}
```

The drawing loop is pretty simple this time around — if `pressed` is `true`, we draw a circle with a fill style equal to the value in the color picker, and a radius equal to the value set in the range input.

```
function draw() {
 if(pressed) {
 ctx.fillStyle = colorPicker.value;
 ctx.beginPath();
 ctx.arc(curX, curY-85, sizePicker.value, degToRad(0), degToRad(360),
false);
 ctx.fill();
 }

 requestAnimationFrame(draw);
}

draw();
```

**Note:** The `<input>` `range` and `color` types are supported fairly well across browsers, with the exception of Internet Explorer versions less than 10; also Safari doesn't yet support `color`. If your browser doesn't support these inputs, they will fall back to simple text fields and you'll just have to enter valid color/number values yourself.

## WebGL

It's now time to leave 2D behind, and take a quick look at 3D canvas. 3D canvas content is specified using the [WebGL API](#), which is a completely separate API from the 2D canvas API, even though they both render onto `<canvas>` elements.

WebGL is based on the [OpenGL](#) graphics programming language, and allows you to communicate directly with the computer's [GPU](#). As such, writing raw WebGL is closer to low level languages such as C++ than regular JavaScript; it is quite complex but incredibly powerful.

### Using a library

Because of its complexity, most people write 3D graphics code using a third party JavaScript library such as [Three.js](#), [PlayCanvas](#) or [Babylon.js](#). Most of these work in a similar way, providing functionality to create primitive and custom shapes, position viewing cameras and lighting, covering surfaces with textures, and more. They handle the WebGL for you, letting you work on a higher level.

Yes, using one of these means learning another new API (a third party one, in this case), but they are a lot simpler than coding raw WebGL.

### Recreating our cube

Let's look at a simple example of how to create something with a WebGL library. We'll choose [Three.js](#), as it is one of the most popular ones. In this tutorial we'll create the 3D spinning cube we saw earlier.

1. To start with, make a local copy of [index.html](#) in a new folder, then save a copy of [metal003.png](#) in the same folder. This is the image we'll use as a surface texture for the cube later on.
2. Next, create a new file called `main.js`, again in the same folder as before.
3. If you open `index.html` in your code editor, you'll see that it has two `<script>` elements — the first one attaching `three.min.js` to the page, and the second one attaching our `main.js` file to the page. You need to [download the three.min.js library](#) and save it in the same directory as before.
4. Now we've got `three.js` attached to our page, we can start to write JavaScript that makes use of it into `main.js`. Let's start by creating a new scene — add the following into your `main.js` file:

```
var scene = new THREE.Scene();
```

- The [Scene\(\)](#) constructor creates a new scene, which represents the whole 3D world we are trying to display.
- Next, we need a **camera** so we can see the scene. In 3D imagery terms, the camera represents a viewer's position in the world. To create a camera, add the following lines next:

```
var camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);
camera.position.z = 5;
```

- The [PerspectiveCamera\(\)](#) constructor takes four arguments:
  - The field of view: How wide the area in front of the camera is that should be visible onscreen, in degrees.
  - The aspect ratio: Usually, this is the ratio of the scene's width divided by the scene's height. Using another value will distort the scene (which might be what you want, but usually isn't).
  - The near plane: How close to the camera objects can be before we stop rendering them to the screen. Think about how when you move your fingertip closer and closer to the space between your eyes, eventually you can't see it anymore.
  - The far plane: How far away things are from the camera before they are no longer rendered.

We also set the camera's position to be 5 distance units out of the Z axis, which, like in CSS, is out of the screen towards you, the viewer.

- The third vital ingredient is a renderer. This is an object that renders a given scene, as viewed through a given camera. We'll create one for now using the [WebGLRenderer\(\)](#) constructor, but we'll not use it till later. Add the following lines next:

```
var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

- The first line creates a new renderer, the second line sets the size at which the renderer will draw the camera's view, and the third line appends the <canvas> element created by the renderer to the document's <body>. Now anything the renderer draws will be displayed in our window.
- Next, we want to create the cube we'll display on the canvas. Add the following chunk of code at the bottom of your JavaScript:

```
var cube;

var loader = new THREE.TextureLoader();

loader.load('metal003.png', function (texture) {
 texture.wrapS = THREE.RepeatWrapping;
 texture.wrapT = THREE.RepeatWrapping;
 texture.repeat.set(2, 2);

 var geometry = new THREE.BoxGeometry(2.4, 2.4, 2.4);
```

```

var material = new THREE.MeshLambertMaterial({ map: texture, shading:
THREE.FlatShading });
cube = new THREE.Mesh(geometry, material);
scene.add(cube);

draw();
});

```

- There's a bit more to take in here, so let's go through it in stages:

- We first create a `cube` global variable so we can access our cube from anywhere in the code.
- Next, we create a new [TextureLoader](#) object, then call `load()` on it. `load()` takes two parameters in this case (although it can take more): the texture we want to load (our PNG), and a function that will run when the texture has loaded.
- Inside this function we use properties of the [texture](#) object to specify that we want a  $2 \times 2$  repeat of the image wrapped around all sides of the cube. Next, we create a new [BoxGeometry](#) object and a new [MeshLambertMaterial](#) object, and bring them together in a [Mesh](#) to create our cube. An object typically requires a geometry (what shape it is) and a material (what its surface looks like).
- Last of all, we add our cube to the scene, then call our `draw()` function to start off the animation.

- Before we get to defining `draw()`, we'll add a couple of lights to the scene, to liven things up a bit; add the following blocks next:

```

var light = new THREE.AmbientLight('rgb(255, 255, 255)'); // soft white light
scene.add(light);

var spotLight = new THREE.SpotLight('rgb(255, 255, 255)');
spotLight.position.set(100, 1000, 1000);
spotLight.castShadow = true;
scene.add(spotLight);

```

- An [AmbientLight](#) object is a kind of soft light that lightens the whole scene a bit, like the sun when you are outside. The [SpotLight](#) object, on the other hand, is a directional beam of light, more like a flashlight/torch (or a spotlight, in fact).

- Last of all, let's add our `draw()` function to the bottom of the code:

```

function draw() {
 cube.rotation.x += 0.01;
 cube.rotation.y += 0.01;
 renderer.render(scene, camera);

 requestAnimationFrame(draw);
}

```

9. This is fairly intuitive; on each frame, we rotate our cube slightly on its X and Y axes, then render the scene as viewed by our camera, then finally call `requestAnimationFrame()` to schedule drawing our next frame.

Let's have another quick look at what the finished product should look like:

You can [find the finished code on GitHub](#).

**Note:** In our GitHub repo you can also find another interesting 3D cube example — [Three.js Video Cube](#) (see it live also). This uses `getUserMedia()` to take a video stream from a computer web cam and project it onto the side of the cube as a texture!

## Summary

At this point, you should have a useful idea of the basics of graphics programming using Canvas and WebGL and what you can do with these APIs, as well as a good idea of where to go for further information. Have fun!

## Video and audio APIs

HTML5 comes with elements for embedding rich media in documents — `<video>` and `<audio>` — which in turn come with their own APIs for controlling playback, seeking, etc. This article shows you how to do common tasks such as creating custom playback controls.

**Prerequisites:** JavaScript basics (see [first steps](#), [building blocks](#), [JavaScript objects](#)), the [basics of Client-side APIs](#)

**Objective:** To learn how to use browser APIs to control video and audio playback.

## HTML5 video and audio

The `<video>` and `<audio>` elements allow us to embed video and audio into web pages. As we showed in [Video and audio content](#), a typical implementation looks like this:

```
<video controls>
 <source src="rabbit320.mp4" type="video/mp4">
 <source src="rabbit320.webm" type="video/webm">
 <p>Your browser doesn't support HTML5 video. Here is a link to the video instead.</p>
</video>
```

This creates a video player inside the browser like so:

You can review what all the HTML features do in the article linked above; for our purposes here, the most interesting attribute is `controls`, which enables the default set of playback controls. If you don't specify this, you get no playback controls:

This is not as immediately useful for video playback, but it does have advantages. One big issue with the native browser controls is that they are different in each browser — not very good for cross-browser support! Another big issue is that the native controls in most browsers aren't very keyboard-accessible.

You can solve both these problems by hiding the native controls (by removing the `controls` attribute), and programming your own with HTML, CSS, and JavaScript. In the next section we'll look at the basic tools we have available to do this.

## The `HTMLMediaElement` API

Part of the HTML5 spec, the `HTMLMediaElement` API provides features to allow you to control video and audio players programmatically — for example `HTMLMediaElement.play()`, `HTMLMediaElement.pause()`, etc. This interface is available to both `<audio>` and `<video>` elements, as the features you'll want to implement are nearly identical. Let's go through an example, adding features as we go.

Our finished example will look (and function) something like the following:

### Getting started

To get started with this example, [download our media-player-start.zip](#) and unzip it into a new directory on your hard drive. If you [downloaded our examples repo](#), you'll find it in `javascript/apis/video-audio/start/`

At this point, if you load the HTML you should see a perfectly normal HTML5 video player, with the native controls rendered.

### Exploring the HTML

Open the HTML index file. You'll see a number of features; the HTML is dominated by the video player and its controls:

```
<div class="player">
 <video controls>
 <source src="video/sintel-short.mp4" type="video/mp4">
 <source src="video/sintel-short.mp4" type="video/webm">
 <!-- fallback content here -->
 </video>
 <div class="controls">
 <button class="play" data-icon="P" aria-label="play pause
toggle"></button>
 <button class="stop" data-icon="S" aria-label="stop"></button>
 <div class="timer">
 <div></div>
 00:00
 </div>
 <button class="rwd" data-icon="B" aria-label="rewind"></button>
```

```
<button class="fwd" data-icon="F" aria-label="fast forward"></button>
</div>
</div>
```

- The whole player is wrapped in a `<div>` element, so it can all be styled as one unit if needed.
- The `<video>` element contains two `<source>` elements so that different formats can be loaded depending on the browser viewing the site.
- The controls HTML is probably the most interesting:
  - We have four `<button>`s — play/pause, stop, rewind, and fast forward.
  - Each `<button>` has a `class` name, a `data-icon` attribute for defining what icon should be shown on each button (we'll show how this works in the below section), and an `aria-label` attribute to provide an understandable description of each button, since we're not providing a human-readable label inside the tags. The contents of `aria-label` attributes are read out by screenreaders when their users focus on the elements that contain them.
  - There is also a timer `<div>`, which will report the elapsed time when the video is playing. Just for fun, we are providing two reporting mechanisms — a `<span>` containing the elapsed time in minutes and seconds, and an extra `<div>` that we will use to create a horizontal indicator bar that gets longer as the time elapses. To get an idea of what the finished product will look like, [check out our finished version](#).

## Exploring the CSS

Now open the CSS file and have a look inside. The CSS for the example is not too complicated, but we'll highlight the most interesting bits here. First of all, notice the `.controls` styling:

```
.controls {
 visibility: hidden;
 opacity: 0.5;
 width: 400px;
 border-radius: 10px;
 position: absolute;
 bottom: 20px;
 left: 50%;
 margin-left: -200px;
 background-color: black;
 box-shadow: 3px 3px 5px black;
 transition: 1s all;
 display: flex;
}

.player:hover .controls, player:focus .controls {
 opacity: 1;
}
```

- We start off with the `visibility` of the custom controls set to `hidden`. In our JavaScript later on, we will set the `controls` to `visible`, and remove the `controls` attribute from the `<video>` element. This is so that, if the JavaScript doesn't load for some reason, users can still use the video with the native controls.

- We give the controls an `opacity` of 0.5 by default, so that they are less distracting when you are trying to watch the video. Only when you are hovering/focusing over the player do the controls appear at full opacity.
- We lay out the buttons inside the control bar out using Flexbox (`display: flex`), to make things easier.

Next, let's look at our button icons:

```
@font-face {
 font-family: 'HeydingsControlsRegular';
 src: url('fonts/heydings_controls-webfont.eot');
 src: url('fonts/heydings_controls-webfont.eot?#iefix') format('embedded-opentype'),
 url('fonts/heydings_controls-webfont.woff') format('woff'),
 url('fonts/heydings_controls-webfont.ttf') format('truetype');
 font-weight: normal;
 font-style: normal;
}

button:before {
 font-family: HeydingsControlsRegular;
 font-size: 20px;
 position: relative;
 content: attr(data-icon);
 color: #aaa;
 text-shadow: 1px 1px 0px black;
}
```

First of all, at the top of the CSS we use a `@font-face` block to import a custom web font. This is an icon font — all the characters of the alphabet equate to common icons you might want to use in an application.

Next we use generated content to display an icon on each button:

- We use the `::before` selector to display the content before each `<button>` element.
- We use the `content` property to set the content to be displayed in each case to be equal to the contents of the `data-icon` attribute. In the case of our play button, `data-icon` contains a capital "P".
- We apply the custom web font to our buttons using `font-family`. In this font, "P" is actually a "play" icon, so therefore the play button has a "play" icon displayed on it.

Icon fonts are very cool for many reasons — cutting down on HTTP requests because you don't need to download those icons as image files, great scalability, and the fact that you can use text properties to style them — like `color` and `text-shadow`.

Last but not least, let's look at the CSS for the timer:

```
.timer {
 line-height: 38px;
 font-size: 10px;
 font-family: monospace;
```

```

text-shadow: 1px 1px 0px black;
color: white;
flex: 5;
position: relative;
}

.timer div {
position: absolute;
background-color: rgba(255,255,255,0.2);
left: 0;
top: 0;
width: 0;
height: 38px;
z-index: 2;
}

.timer span {
position: absolute;
z-index: 3;
left: 19px;
}

```

- We set the outer `.timer <div>` to have `flex: 5`, so it takes up most of the width of the controls bar. We also give it `{cssxref("position")}`: `relative`, so that we can position elements inside it conveniently according to its boundaries, and not the boundaries of the `<body>` element.
- The inner `<div>` is absolutely positioned to sit directly on top of the outer `<div>`. It is also given an initial width of 0, so you can't see it at all. As the video plays, the width will be increased via JavaScript as the video elapses.
- The `<span>` is also absolutely positioned to sit near the left hand side of the timer bar.
- We also give our inner `<div>` and `<span>` the right amount of `z-index` so that the timer will be displayed on top, and the inner `<div>` below that. This way, we make sure we can see all the information — one box is not obscuring another.

## Implementing the JavaScript

We've got a fairly complete HTML and CSS interface already; now we just need to wire up all the buttons to get the controls working.

1. Create a new JavaScript file in the same directory level as your `index.html` file. Call it `custom-player.js`.

2. At the top of this file, insert the following code:

```

3. var media = document.querySelector('video');
4. var controls = document.querySelector('.controls');
5.
6. var play = document.querySelector('.play');
7. var stop = document.querySelector('.stop');
8. var rwd = document.querySelector('.rwd');
9. var fwd = document.querySelector('.fwd');
10.
11. var timerWrapper = document.querySelector('.timer');
12. var timer = document.querySelector('.timer span');
 var timerBar = document.querySelector('.timer div');

```

- Here we are creating variables to hold references to all the objects we want to manipulate. We have three groups:

- The <video> element, and the controls bar.
  - The play/pause, stop, rewind, and fast forward buttons.
  - The outer timer wrapper <div>, the digital timer readout <span>, and the inner <div> that gets wider as the time elapses.
- Next, insert the following at the bottom of your code:

```
media.removeAttribute('controls');
controls.style.visibility = 'visible';
```

3. These two lines remove the default browser controls from the video, and make the custom controls visible.

## Playing and pausing the video

Let's implement probably the most important control — the play/pause button.

1. First of all, add the following to the bottom of your code, so that the `playPauseMedia()` function is invoked when the play button is clicked:

```
play.addEventListener('click', playPauseMedia);
```

- Now to define `playPauseMedia()` — add the following, again at the bottom of your code:

```
function playPauseMedia() {
 if(media.paused) {
 play.setAttribute('data-icon', 'u');
 media.play();
 } else {
 play.setAttribute('data-icon', 'P');
 media.pause();
 }
}
```

2. He we use an `if` statement to check whether the video is paused. The `HTMLMediaElement.paused` property returns true if the media is paused, which is any time the video is not playing, including when it is sat at 0 duration after it first loads. If it is paused, we set the `data-icon` attribute value on the play button to "u", which is a "paused" icon, and invoke the `HTMLMediaElement.play()` method to play the media.

On the second click, the button will be toggled back again — the "play" icon will be shown again, and the video will be paused with `HTMLMediaElement.paused()`.

## Stopping the video

1. Next, let's add functionality to handle stopping the video. Add the following `addEventListener()` lines below the previous one you added:

```
stop.addEventListener('click', stopMedia);
media.addEventListener('ended', stopMedia);
```
- The `click` event is obvious — we want to stop the video by running our `stopMedia()` function when the stop button is clicked. We do however also want to stop the video when it finishes playing — this is marked by the `ended` event firing, so we also set up a listener to run the function on that event firing too.
- Next, let's define `stopMedia()` — add the following function below `playPauseMedia()`:

```
function stopMedia() {
 media.pause();
 media.currentTime = 0;
 play.setAttribute('data-icon', 'P');
}
```

2. there is no `stop()` method on the `HTMLMediaElement` API — the equivalent is to `pause()` the video, and set its `currentTime` property to 0. Setting `currentTime` to a value (in seconds) immediately jumps the media to that position.

All there is left to do after that is to set the displayed icon to the "play" icon. Regardless of whether the video was paused or playing when the stop button is pressed, you want it to be ready to play afterwards.

## Seeking back and forth

There are many ways that you can implement rewind and fast forward functionality; here we are showing you a relatively complex way of doing it, which doesn't break when the different buttons are pressed in an unexpected order.

1. First of all, add the following two `addEventListener()` lines below the previous ones:

```
rwd.addEventListener('click', mediaBackward);
fwd.addEventListener('click', mediaForward);
```
- Now on to the event handler functions — add the following code below your previous functions to define `mediaBackward()` and `mediaForward()`:

```
var intervalFwd;
var intervalRwd;

function mediaBackward() {
 clearInterval(intervalFwd);
 fwd.classList.remove('active');

 intervalRwd = setInterval(function() {
 media.currentTime -= 0.1;
 }, 100);
}
```

```

if(rwd.classList.contains('active')) {
 rwd.classList.remove('active');
 clearInterval(intervalRwd);
 media.play();
} else {
 rwd.classList.add('active');
 media.pause();
 intervalRwd = setInterval(windBackward, 200);
}
}

function mediaForward() {
 clearInterval(intervalRwd);
 rwd.classList.remove('active');

 if(fwd.classList.contains('active')) {
 fwd.classList.remove('active');
 clearInterval(intervalFwd);
 media.play();
 } else {
 fwd.classList.add('active');
 media.pause();
 intervalFwd = setInterval(windForward, 200);
 }
}

```

- You'll notice that first we initialize two variables — `intervalFwd` and `intervalRwd` — you'll find out what they are for later on.

Let's step through `mediaBackward()` (the functionality for `mediaForward()` is exactly the same, but in reverse):

1. We clear any classes and intervals that are set on the fast forward functionality — we do this because if we press the `rwd` button after pressing the `fwd` button, we want to cancel any fast forward functionality and replace it with the rewind functionality. If we tried to do both at one, the player would break.
2. We use an `if` statement to check whether the `active` class has been set on the `rwd` button, indicating that it has already been pressed. The `classList` is a rather handy property that exists on every element — it contains a list of all the classes set on the element, as well as methods for adding/removing classes, etc. We use the `classList.contains()` method to check whether the list contains the `active` class. This returns a boolean `true/false` result.
3. If `active` has been set on the `rwd` button, we remove it using `classList.remove()`, clear the interval that has been set when the button was first pressed (see below for more explanation), and use `HTMLMediaElement.play()` to cancel the rewind and start the video playing normally.
4. If it hasn't yet been set, we add the `active` class to the `rwd` button using `classList.add()`, pause the video using `HTMLMediaElement.pause()`, then set the `intervalRwd` variable to equal a `setInterval()` call. When invoked, `setInterval()` creates an active interval, meaning that it runs the function given as the first parameter every x milliseconds, where x is the value of the 2nd parameter. So here we are running the `windBackward()` function every 200 milliseconds — we'll use this function to wind the video backwards constantly. To stop a

`setInterval()` running, you have to call `clearInterval()`, giving it the identifying name of the interval to clear, which in this case is the variable name `intervalRwd` (see the `clearInterval()` call earlier on in the function).

- last of all for this section, we need to define the `windBackward()` and `windForward()` functions invoked in the `setInterval()` calls. Add the following below your two previous functions:

```
function windBackward() {
 if(media.currentTime <= 3) {
 rwd.classList.remove('active');
 clearInterval(intervalRwd);
 stopMedia();
 } else {
 media.currentTime -= 3;
 }
}

function windForward() {
 if(media.currentTime >= media.duration - 3) {
 fwd.classList.remove('active');
 clearInterval(intervalFwd);
 stopMedia();
 } else {
 media.currentTime += 3;
 }
}
```

3. Again, we'll just run through the first one of these functions as they work almost identically, but in reverse to one another. In `windBackward()` we do the following — bear in mind that when the interval is active, this function is being run once every 200 milliseconds.

1. We start off with an `if` statement that checks to see whether the current time is less than 3 seconds, i.e., if rewinding by another three seconds would take it back past the start of the video. This would cause strange behaviour, so if this is the case we stop the video playing by calling `stopMedia()`, remove the `active` class from the rewind button, and clear the `intervalRwd` interval to stop the rewind functionality. If we didn't do this last step, the video would just keep rewinding forever.
2. If the current time is not within 3 seconds of the start of the video, we simply remove three seconds from the current time by executing `media.currentTime -= 3`. So in effect, we are rewinding the video by 3 seconds, once every 200 milliseconds.

## Updating the elapsed time

The very last piece of our media player to implement is the time elapsed displays. To do this we'll run a function to update the time displays every time the `timeupdate` event is fired on the `<video>` element. This frequency with which this event fires depends on your browser, CPU power, etc ([see this stackoverflow post](#)).

Add the following `addEventListener()` line just below the others:

```
media.addEventListener('timeupdate', setTime);
```

Now to define the `setTime()` function. Add the following at the bottom of your file:

```
function setTime() {
 var minutes = Math.floor(media.currentTime / 60);
 var seconds = Math.floor(media.currentTime - minutes * 60);
 var minuteValue;
 var secondValue;

 if (minutes < 10) {
 minuteValue = '0' + minutes;
 } else {
 minuteValue = minutes;
 }

 if (seconds < 10) {
 secondValue = '0' + seconds;
 } else {
 secondValue = seconds;
 }

 var mediaTime = minuteValue + ':' + secondValue;
 timer.textContent = mediaTime;

 var barLength = timerWrapper.clientWidth *
(media.currentTime/media.duration);
 timerBar.style.width = barLength + 'px';
}
```

This is a fairly long function, so let's go through it step by step:

1. First of all, we work out the number of minutes and seconds in the `HTMLMediaElement.currentTime` value.
2. Then we initialize two more variables — `minuteValue` and `secondValue`.
3. The two `if` statements work out whether the number of minutes and seconds are less than 10. If so, they add a leading zero to the values, in the same way that a digital clock display works.
4. The actual time value to display is set as `minuteValue` plus a colon character plus `secondValue`.
5. The `Node.textContent` value of the timer is set to the time value, so it displays in the UI.
6. The length we should set the inner `<div>` to is worked out by first working out the width of the outer `<div>` (any element's `clientWidth` property will contain its length), and then multiplying it by the `HTMLMediaElement.currentTime` divided by the total `HTMLMediaElement.duration` of the media.
7. We set the width of the inner `<div>` to equal the calculated bar length, plus "px", so it will be set to that number of pixels.

## Fixing play and pause

There is one problem left to fix. If the play/pause or stop buttons are pressed while the rewind or fast forward functionality is active, they just don't work. How can we fix it so that they cancel the `rwd`/`fwd` button functionality and play/stop the video as you'd expect? This is fairly easy to fix.

First of all, add the following lines inside the `stopMedia()` function — anywhere will do:

```
rwd.classList.remove('active');
fwd.classList.remove('active');
clearInterval(intervalRwd);
clearInterval(intervalFwd);
```

Now add the same lines again, at the very start of the `playPauseMedia()` function (just before the start of the `if` statement).

At this point, you could delete the equivalent lines from the `windBackward()` and `windForward()` functions, as that functionality has been implemented in the `stopMedia()` function instead.

Note: You could also further improve the efficiency of the code by creating a separate function that runs these lines, then calling that anywhere it is needed, rather than repeating the lines multiple times in the code. But we'll leave that one up to you.

## Summary

I think we've taught you enough in this article. The `HTMLMediaElement` API makes a wealth of functionality available for creating simple video and audio players, and that's only the tip of the iceberg. See the "See also" section below for links to more complex and interesting functionality.

Here are some suggestions for ways you could enhance the existing example we've built up:

1. The time display currently breaks if the video is an hour long or more (well, it won't display hours; just minutes and seconds). Can you figure out how to change the example to make it display hours?
2. Because `<audio>` elements have the same `HTMLMediaElement` functionality available to them, you could easily get this player to work for an `<audio>` element too. Try doing so.
3. Can you work out a way to turn the timer inner `<div>` element into a true seek bar/scrobbler — i.e., when you click somewhere on the bar, it jumps to that relative position in the video playback? As a hint, you can find out the X and Y values of the element's left/right and top/bottom sides via the `getBoundingClientRect()` method, and you can find the coordinates of a mouse click via the event object of the click event, called on the Document object. For example:
  4. `document.onclick = function(e) {`
  5.  `console.log(e.x) + ',' + console.log(e.y)`

```
}
```

## Client-side storage

Modern web browsers support a number of ways for web sites to store data on the user's computer — with the user's permission — then retrieve it when necessary. This lets you persist data for long-term storage, save sites or documents for offline use, retain user-specific settings for your site, and more. This article explains the very basics of how these work.

**Prerequisites:** JavaScript basics (see [first steps](#), [building blocks](#), [JavaScript objects](#)), the [basics of Client-side APIs](#)

**Objective:** To learn how to use client-side storage APIs to store application data.

## Client-side storage?

Elsewhere in the MDN learning area we talked about the difference between [static sites](#) and [dynamic sites](#). Most major modern web sites are dynamic — they store data on the server using some kind of database (server-side storage), then run [server-side](#) code to retrieve needed data, insert it into static page templates, and serve the resulting HTML to the client to be displayed by the user's browser.

Client-side storage works on similar principles, but has different uses. It consists of JavaScript APIs that allow you to store data on the client (i.e. on the user's machine) and then retrieve it when needed. This has many distinct uses, such as:

- Personalizing site preferences (e.g. showing a user's choice of custom widgets, color scheme, or font size).
- Persisting previous site activity (e.g. storing the contents of a shopping cart from a previous session, remembering if a user was previously logged in).
- Saving data and assets locally so a site will be quicker (and potentially less expensive) to download, or be usable without a network connection.
- Saving web application generated documents locally for use offline

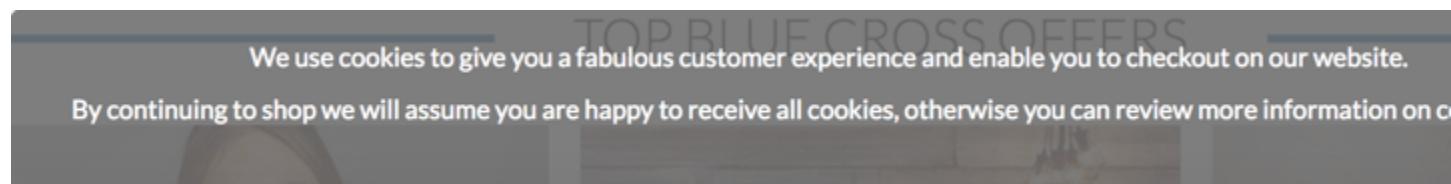
Often client-side and server-side storage are used together. For example, you could download from a database a batch of music files used by a web game or music player app store them inside a client-side database, and play them as needed. The user would only have to download the music files once — on subsequent visits they would be retrieved from the database instead.

**Note:** There are limits to the amount of data you can store using client-side storage APIs (possibly both per individual API and cumulatively); the exact limit varies depending on the browser and possibly based on user settings. See [Browser storage limits and eviction criteria](#) for more information.

## Old fashioned: cookies

The concept of client-side storage has been around for a long time. Since the early days of the web, sites have used [cookies](#) to store information to personalize user experience on websites. They're the earliest form of client-side storage commonly used on the web.

Because of that age, there are a number of problems — both technical and user experience-wise — afflicting cookies. These problems are significant enough that upon visiting a site for the first time, people living in Europe are shown messages informing them if they will use cookies to store data about them. This is due to a piece of European Union legislation known as the [EU Cookie directive](#).



For these reasons, we won't be teaching you how to use cookies in this article. Between being outdated, their assorted [security problems](#), and inability to store complex data, there are better, more modern ways to store a wider variety of data on the user's computer.

The only advantage cookies have is that they're supported by extremely old browsers, so if your project requires that you support browsers that are already obsolete (such as Internet Explorer 8 or earlier), cookies may still be useful, but for most projects you shouldn't need to resort to them anymore.

Why are there still new sites being created using cookies? This is mostly because of developers' habits, use of older libraries that still use cookies, and the existence of many web sites providing out-of-date reference and training materials to learn how to store data.

## New school: Web Storage and IndexedDB

Modern browsers have much easier, more effective APIs for storing client-side data than by using cookies.

- The [Web Storage API](#) provides a very simple syntax for storing and retrieving smaller, data items consisting of a name and a corresponding value. This is useful when you just need to store some simple data, like the user's name, whether they are logged in, what color to use for the background of the screen, etc.
- The [IndexedDB API](#) provides the browser with a complete database system for storing complex data. This can be used for things from complete sets of customer records to even complex data types like audio or video files.

You'll learn more about these APIs below.

## The future: Cache API

Some modern browsers support the new [Cache](#) API. This API is designed for storing HTTP responses to specific requests, and is very useful for doing things like storing website assets offline so the site can subsequently be used without a network connection. Cache is usually used in combination with the [Service Worker API](#), although it doesn't have to be.

Use of Cache and Service Workers is an advanced topic, and we won't be covering it in great detail in this article, although we will show a simple example in the [Offline asset storage](#) section below.

## Storing simple data — web storage

The [Web Storage API](#) is very easy to use — you store simple name/value pairs of data (limited to strings, numbers, etc.) and retrieve these values when needed.

### Basic syntax

Let's show you how:

1. First, go to our [web storage blank template](#) on GitHub (open this in a new tab).
2. Open the JavaScript console of your browser's developer tools.
3. All of your web storage data is contained within two object-like structures inside the browser: [sessionStorage](#) and [localStorage](#). The first one persists data for as long as the browser is open (the data is lost when the browser is closed) and the second one persists data even after the browser is closed and then opened again. We'll use the second one in this article as it is generally more useful.

The [Storage.setItem\(\)](#) method allows you to save a data item in storage — it takes two parameters: the name of the item, and its value. Try typing this into your JavaScript console (change the value to your own name, if you wish!):

```
localStorage.setItem('name', 'Chris');
```

- • The [storage.getItem\(\)](#) method takes one parameter — the name of a data item you want to retrieve — and returns the item's value. Now type these lines into your JavaScript console:

```
var myName = localStorage.getItem('name');
myName
```

- Upon typing in the second line, you should see that the `myName` variable now contains the value of the `name` data item.

- The `Storage.removeItem()` method takes one parameter — the name of a data item you want to remove — and removes that item out of web storage. Type the following lines into your JavaScript console:

```
localStorage.removeItem('name');
var myName = localStorage.getItem('name');
myName
```

5. The third line should now return `null` — the `name` item no longer exists in the web storage.

## The data persists!

One key feature of web storage is that the data persists between page loads (and even when the browser is shut down, in the case of `localStorage`). Let's look at this in action.

1. Open our web storage blank template again, but this time in a different browser to the one you've got this tutorial open in! This will make it easier to deal with.
2. Type these lines into the browser's JavaScript console:
  3. `localStorage.setItem('name', 'Chris');`
  4. `var myName = localStorage.getItem('name');`  
`myName`
- You should see the `name` item returned.
- Now close down the browser and open it up again.
- Enter the following lines again:

```
var myName = localStorage.getItem('name');
myName
```

4. You should see that the value is still available, even though the browser has been closed and then opened again.

## Separate storage for each domain

There is a separate data store for each domain (each separate web address loaded in the browser). You will see that if you load two websites (say `google.com` and `amazon.com`) and try storing an item on one website, it won't be available to the other website.

This makes sense — you can imagine the security issues that would arise if websites could see each other's data!

## A more involved example

Let's apply this new-found knowledge by writing a simple working example to give you an idea of how web storage can be used. Our example will allow you enter a name, after which the page will update to give you a personalized greeting. This state will also persist across page/browser reloads, because the name is stored in web storage.

You can find the example HTML at [personal-greeting.html](#) — this contains a simple website with a header, content, and footer, and a form for entering your name.

# Welcome, Billy Bob

**Welcome to our website, Billy Bob! We hope you have fun while you are here.**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam malesuada, metus ut mollis volutpat risus malesuada dui. Phasellus tempus elit at purus vestibulum suscipit. Donec quis est nec dui prefinibus, ipsum non semper dignissim, massa magna sagittis est, vitae vehicula nunc magna vitae dictum tempor, at mattis justo sagittis. Nunc ut nulla et erat viverra tincidunt. Interdum et malesuada fames Integer vitae bibendum justo. Vestibulum porta velit sit amet libero accumsan fermentum.

Ut id mauris urna. In sed porttitor lectus. Suspendisse dignissim dolor id lectus pellentesque, eu bibendum volutpat sollicitudin suscipit. Donec id libero nisl. Praesent gravida purus vel euismod facilisis. Maecenas dictum vitae eu augue. Donec euismod enim elementum elit laoreet sodales. Lorem ipsum dolor sit amet,

Integer vulputate, libero sed vulputate eleifend, magna libero malesuada ligula, sit amet tincidunt duis turpis mauris, in sagittis orci rutrum efficitur. Sed vel purus fringilla, pretium sapien sed, accumsan enim faucibus. Nunc fringilla nulla ut aliquam posuere. Vivamus id lectus eleifend, bibendum urna non, ornare mollis.

Copyright nobody. Use the code as you like.

Let's build up the example, so you can understand how it works.

1. First, make a local copy of our [personal-greeting.html](#) file in a new directory on your computer.

2. Next, note how our HTML references a JavaScript file called `index.js` (see line 40). We need to create this and write our JavaScript code into it. Create an `index.js` file in the same directory as your HTML file.
3. We'll start off by creating references to all the HTML features we need to manipulate in this example — we'll create them all as constants, as these references do not need to change in the lifecycle of the app. Add the following lines to your JavaScript file:

```

4. // create needed constants
5. const rememberDiv = document.querySelector('.remember');
6. const forgetDiv = document.querySelector('.forget');
7. const form = document.querySelector('form');
8. const nameInput = document.querySelector('#entername');
9. const submitBtn = document.querySelector('#submitname');
10. const forgetBtn = document.querySelector('#forgetname');
11.
12. const h1 = document.querySelector('h1');
 const personalGreeting = document.querySelector('.personal-greeting');

```

- • Next up, we need to include a small event listener to stop the form from actually submitting itself when the submit button is pressed, as this is not the behavior we want. Add this snippet below your previous code:

```
// Stop the form from submitting when a button is pressed
form.addEventListener('submit', function(e) {
 e.preventDefault();
});
```

- • Now we need to add an event listener, the handler function of which will run when the "Say hello" button is clicked. The comments explain in detail what each bit does, but in essence here we are taking the name the user has entered into the text input box and saving it in web storage using `setItem()`, then running a function called `nameDisplayCheck()` that will handle updating the actual website text. Add this to the bottom of your code:

```
// run function when the 'Say hello' button is clicked
submitBtn.addEventListener('click', function() {
 // store the entered name in web storage
 localStorage.setItem('name', nameInput.value);
 // run nameDisplayCheck() to sort out displaying the
 // personalized greetings and updating the form display
 nameDisplayCheck();
});
```

- • At this point we also need an event handler to run a function when the "Forget" button is clicked — this is only displayed after the "Say hello" button has been clicked (the two form states toggle back and forth). In this function we remove the `name` item from web storage using `removeItem()`, then again run `nameDisplayCheck()` to update the display. Add this to the bottom:

```
// run function when the 'Forget' button is clicked
forgetBtn.addEventListener('click', function() {
 // Remove the stored name from web storage
 localStorage.removeItem('name');
```

```
// run nameDisplayCheck() to sort out displaying the
// generic greeting again and updating the form display
nameDisplayCheck();
});
```

- • It is now time to define the `nameDisplayCheck()` function itself. Here we check whether the name item has been stored in web storage by using `localStorage.getItem('name')` as a conditional test. If it has been stored, this call will evaluate to `true`; if not, it will be `false`. If it is `true`, we display a personalized greeting, display the "forget" part of the form, and hide the "Say hello" part of the form. If it is `false`, we display a generic greeting and do the opposite. Again, put the following code at the bottom:

```
// define the nameDisplayCheck() function
function nameDisplayCheck() {
 // check whether the 'name' data item is stored in web Storage
 if(localStorage.getItem('name')) {
 // If it is, display personalized greeting
 let name = localStorage.getItem('name');
 h1.textContent = 'Welcome, ' + name;
 personalGreeting.textContent = 'Welcome to our website, ' + name + '! We
hope you have fun while you are here.';
 // hide the 'remember' part of the form and show the 'forget' part
 forgetDiv.style.display = 'block';
 rememberDiv.style.display = 'none';
 } else {
 // if not, display generic greeting
 h1.textContent = 'Welcome to our website ';
 personalGreeting.textContent = 'Welcome to our website. We hope you have
fun while you are here.';
 // hide the 'forget' part of the form and show the 'remember' part
 forgetDiv.style.display = 'none';
 rememberDiv.style.display = 'block';
 }
}
```

- • Last but not least, we need to run the `nameDisplayCheck()` function every time the page is loaded. If we don't do this, then the personalized greeting will not persist across page reloads. Add the following to the bottom of your code:

```
document.body.onload = nameDisplayCheck;
```

8.

Your example is finished — well done! All that remains now is to save your code and test your HTML page in a browser. You can see our [finished version running live here](#).

**Note:** There is another, slightly more complex example to explore at [Using the Web Storage API](#).

## Storing complex data — IndexedDB

The [IndexedDB API](#) (sometimes abbreviated IDB) is a complete database system available in the browser in which you can store complex related data, the types of which aren't limited to simple values like strings or numbers. You can store videos, images, and pretty much anything else in an IndexedDB instance.

However, this does come at a cost: IndexedDB is much more complex to use than the Web Storage API. In this section, we'll really only scratch the surface of what it is capable of, but we will give you enough to get started.

### Working through a note storage example

Here we'll run you through an example that allows you to store notes in your browser and view and delete them whenever you like, getting you to build it up for yourself and explaining the most fundamental parts of IDB as we go along.

The app looks something like this:

# IndexedDB notes demo

## Notes

### Buy milk

Need both cows milk and soya.

[Delete](#)

### post mum's birthday card

Need to do it tomorrow otherwise it will be late.

[Delete](#)

## Enter a new note

Note title

Note text

[Create new note](#)

**Copyright nobody. Use the code as you like.**

Each note has a title and some body text, each individually editable. The JavaScript code we'll go through below has detailed comments to help you understand what's going on.

## Getting started

1. First of all, make local copies of our `index.html`, `style.css`, and `index-start.js` files into a new directory on your local machine.
2. Have a look at the files. You'll see that the HTML is pretty simple: a web site with a header and footer, as well as a main content area that contains a place to display notes, and a form for entering new notes into the database. The CSS provides some simple styling to make it clearer what is going on. The JavaScript file contains five declared constants containing references to the `<ul>` element the notes will be displayed in, the title and body `<input>` elements, the `<form>` itself, and the `<button>`.
3. Rename your JavaScript file to `index.js`. You are now ready to start adding code to it.

## Database initial set up

Now let's look at what we have to do in the first place, to actually set up a database.

1. Below the constant declarations, add the following lines:

```
2. // Create an instance of a db object for us to store the open database
 in
 let db;
```

- Here we are declaring a variable called `db` — this will later be used to store an object representing our database. We will use this in a few places, so we've declared it globally here to make things easier.
- Next, add the following to the bottom of your code:

```
window.onload = function() {
};
```

- We will write all of our subsequent code inside this `window.onload` event handler function, called when the window's `load` event fires, to make sure we don't try to use IndexedDB functionality before the app has completely finished loading (it could fail if we don't).
- Inside the `window.onload` handler, add the following:

```
// Open our database; it is created if it doesn't already exist
// (see onupgradeneeded below)
let request = window.indexedDB.open('notes', 1);
```

- This line creates a `request` to open version 1 of a database called `notes`. If this doesn't already exist, it will be created for you by subsequent code. You will see this request pattern used very often throughout IndexedDB. Database operations take time. You don't want to hang the browser while you wait for the results, so database operations are [asynchronous](#), meaning that instead of happening immediately, they will happen at some point in the future, and you get notified when they're done.

To handle this in IndexedDB, you create a request object (which can be called anything you like — we called it `request` so it is obvious what it is for). You then use event handlers to run code when the request completes, fails, etc., which you'll see in use below.

**Note:** The version number is important. If you want to upgrade your database (for example, by changing the table structure), you have to run your code again with an increased version number, different schema specified inside the `onupgradeneeded` handler (see below), etc. We won't cover upgrading databases in this simple tutorial.

- Now add the following event handlers just below your previous addition — again inside the `window.onload` handler:

```
// onerror handler signifies that the database didn't open successfully
request.onerror = function() {
 console.log('Database failed to open');
};

// onsuccess handler signifies that the database opened successfully
request.onsuccess = function() {
 console.log('Database opened successfully');

 // Store the opened database object in the db variable. This is used a lot
 // below
 db = request.result;

 // Run the displayData() function to display the notes already in the IDB
 displayData();
};
```

- The `request.onerror` handler will run if the system comes back saying that the request failed. This allows you to respond to this problem. In our simple example, we just print a message to the JavaScript console.

The `request.onsuccess` handler on the other hand will run if the request returns successfully, meaning the database was successfully opened. If this is the case, an object representing the opened database becomes available in the `request.result` property, allowing us to manipulate the database. We store this in the `db` variable we created earlier for later use. We also run a custom function called `displayData()`, which displays the data in the database inside the `<ul>`. We run it now so that the notes already in the database are displayed as soon as the page loads. You'll see this defined later on.

- Finally for this section, we'll add probably the most important event handler for setting up the database: `request.onupgradeneeded`. This handler runs if the database has not already been set up, or if the database is opened with a bigger version number than the existing stored database (when performing an upgrade). Add the following code, below your previous handler:

```
// Setup the database tables if this has not already been done
request.onupgradeneeded = function(e) {
 // Grab a reference to the opened database
```

```

let db = e.target.result;

// Create an objectStore to store our notes in (basically like a single
table)
// including a auto-incrementing key
let objectStore = db.createObjectStore('notes', { keyPath: 'id',
autoIncrement:true });

// Define what data items the objectStore will contain
objectStore.createIndex('title', 'title', { unique: false });
objectStore.createIndex('body', 'body', { unique: false });

console.log('Database setup complete');
};

```

5. This is where we define the schema (structure) of our database; that is, the set of columns (or fields) it contains. Here we first grab a reference to the existing database from `e.target.result` (the event target's `result` property), which is the `request` object. This is equivalent to the line `db = request.result;` inside the `onsuccess` handler, but we need to do this separately here because the `onupgradeneeded` handler (if needed) will run before the `onsuccess` handler, meaning that the `db` value wouldn't be available if we didn't do this.

We then use [IDBDatabase.createObjectStore\(\)](#) to create a new object store inside our opened database. This is equivalent to a single table in a conventional database system. We've given it the name `notes`, and also specified an `autoIncrement` key field called `id` — in each new record this will automatically be given an incremented value — the developer doesn't need to set this explicitly. Being the key, the `id` field will be used to uniquely identify records, such as when deleting or displaying a record.

We also create two other indexes (fields) using the [IDBObjectStore.createIndex\(\)](#) method: `title` (which will contain a title for each note), and `body` (which will contain the body text of the note).

So with this simple database schema set up, when we start adding records to the database each one will be represented as an object along these lines:

```
{
 title: "Buy milk",
 body: "Need both cows milk and soya.",
 id: 8
}
```

## Adding data to the database

Now let's look at how we can add records to the database. This will be done using the form on our page.

Below your previous event handler (but still inside the `window.onload` handler), add the following line, which sets up an `onsubmit` handler that runs a function called `addData()` when the form is submitted (when the submit `<button>` is pressed leading to a successful form submission):

```
// Create an onsubmit handler so that when the form is submitted the
// addData() function is run
form.onsubmit = addData;
```

Now let's define the `addData()` function. Add this below your previous line:

```
// Define the addData() function
function addData(e) {
 // prevent default - we don't want the form to submit in the conventional
 // way
 e.preventDefault();

 // grab the values entered into the form fields and store them in an object
 // ready for being inserted into the DB
 let newItem = { title: titleInput.value, body: bodyInput.value };

 // open a read/write db transaction, ready for adding the data
 let transaction = db.transaction(['notes'], 'readwrite');

 // call an object store that's already been added to the database
 let objectStore = transaction.objectStore('notes');

 // Make a request to add our newItem object to the object store
 var request = objectStore.add(newItem);
 request.onerror = function() {
 // Clear the form, ready for adding the next entry
 titleInput.value = '';
 bodyInput.value = '';
 };

 // Report on the success of the transaction completing, when everything is
 // done
 transaction.oncomplete = function() {
 console.log('Transaction completed: database modification finished.');

 // update the display of data to show the newly added item, by running
 // displayData() again.
 displayData();
 };

 transaction.onerror = function() {
 console.log('Transaction not opened due to error');
 };
}
```

This is quite complex; breaking it down, we:

- Run `Event.preventDefault()` on the event object to stop the form actually submitting in the conventional manner (this would cause a page refresh and spoil the experience).

- Create an object representing a record to enter into the database, populating it with values from the form inputs. note that we don't have to explicitly include an `id` value — as we explained earlier, this is auto-populated.
- Open a `readwrite` transaction against the `notes` object store using the `IDBDatabase.transaction()` method. This transaction object allows us to access the object store so we can do something to it, e.g. add a new record.
- Access the object store using the `IDBTransaction.objectStore()` method, saving the result in the `objectStore` variable.
- Add the new record to the database using `IDBObjectStore.add()`. This creates a request object, in the same fashion as we've seen before.
- Add a bunch of event handlers to the `request` and the `transaction` to run code at critical points in the lifecycle. Once the request has succeeded, we clear the form inputs ready for entering the next note. Once the transaction has completed, we run the `displayData()` function again to update the display of notes on the page.

## Displaying the data

We've referenced `displayData()` twice in our code already, so we'd probably better define it. Add this to your code, below the previous function definition:

```
// Define the displayData() function
function displayData() {
 // Here we empty the contents of the list element each time the display is
 updated
 // If you didn't do this, you'd get duplicates listed each time a new note
 is added
 while (list.firstChild) {
 list.removeChild(list.firstChild);
 }

 // Open our object store and then get a cursor - which iterates through all
 the
 // different data items in the store
 let objectStore = db.transaction('notes').objectStore('notes');
 objectStore.openCursor().onsuccess = function(e) {
 // Get a reference to the cursor
 let cursor = e.target.result;

 // If there is still another data item to iterate through, keep running
 this code
 if(cursor) {
 // Create a list item, h3, and p to put each data item inside when
 displaying it
 // structure the HTML fragment, and append it inside the list
 let listItem = document.createElement('li');
 let h3 = document.createElement('h3');
 let para = document.createElement('p');

 listItem.appendChild(h3);
 listItem.appendChild(para);
 list.appendChild(listItem);
 }
 }
}
```

```

 // Put the data from the cursor inside the h3 and para
 h3.textContent = cursor.value.title;
 para.textContent = cursor.value.body;

 // Store the ID of the data item inside an attribute on the listItem,
 so we know
 // which item it corresponds to. This will be useful later when we want
 to delete items
 listItem.setAttribute('data-note-id', cursor.value.id);

 // Create a button and place it inside each listItem
 let deleteBtn = document.createElement('button');
 listItem.appendChild(deleteBtn);
 deleteBtn.textContent = 'Delete';

 // Set an event handler so that when the button is clicked, the
 deleteItem()
 // function is run
 deleteBtn.onclick = deleteItem;

 // Iterate to the next item in the cursor
 cursor.continue();
} else {
 // Again, if list item is empty, display a 'No notes stored' message
 if(!list.firstChild) {
 let listItem = document.createElement('li');
 listItem.textContent = 'No notes stored.';
 list.appendChild(listItem);
 }
 // if there are no more cursor items to iterate through, say so
 console.log('Notes all displayed');
}
};

}
}

```

Again, let's break this down:

- First we empty out the `<ul>` element's content, before then filling it with the updated content. If you didn't do this, you'd end up with a huge list of duplicated content being added to with each update.
- Next, we get a reference to the `notes` object store using `IDBDatabase.transaction()` and `IDBTransaction.objectStore()` like we did in `addData()`, except here we are chaining them together in one line.
- The next step is to use `IDBObjectStore.openCursor()` method to open a request for a cursor — this is a construct that can be used to iterate over the records in an object store. We chain an `onsuccess` handler on to the end of this line to make the code more concise — when the cursor is successfully returned, the handler is run.
- We get a reference to the cursor itself (an `IDBCursor` object) using `let cursor = e.target.result`.
- Next, we check to see if the cursor contains a record from the datastore (`if(cursor) { ... }`) — if so, we create a DOM fragment, populate it with the data from the record, and insert it into the page (inside the `<ul>` element). We also include a delete button that, when clicked, will

delete that note by running the `deleteItem()` function, which we will look at in the next section.

- At the end of the `if` block, we use the [`IDBCursor.continue\(\)`](#) method to advance the cursor to the next record in the datastore, and run the content of the `if` block again. If there is another record to iterate to, this causes it to be inserted into the page, and then `continue()` is run again, and so on.
- When there are no more records to iterate over, `cursor` will return `undefined`, and therefore the `else` block will run instead of the `if` block. This block checks whether any notes were inserted into the `<ul>` — if not, it inserts a message to say no note was stored.

## Deleting a note

As stated above, when a note's delete button is pressed, the note is deleted. This is achieved by the `deleteItem()` function, which looks like so:

```
// Define the deleteItem() function
function deleteItem(e) {
 // retrieve the name of the task we want to delete. We need
 // to convert it to a number before trying it use it with IDB; IDB key
 // values are type-sensitive.
 let noteId = Number(e.target.parentNode.getAttribute('data-note-id'));

 // open a database transaction and delete the task, finding it using the id
 // we retrieved above
 let transaction = db.transaction(['notes'], 'readwrite');
 let objectStore = transaction.objectStore('notes');
 let request = objectStore.delete(noteId);

 // report that the data item has been deleted
 transaction.oncomplete = function() {
 // delete the parent of the button
 // which is the list item, so it is no longer displayed
 e.target.parentNode.parentNode.removeChild(e.target.parentNode);
 console.log('Note ' + noteId + ' deleted.');

 // Again, if list item is empty, display a 'No notes stored' message
 if(!list.firstChild) {
 let listItem = document.createElement('li');
 listItem.textContent = 'No notes stored.';
 list.appendChild(listItem);
 }
 };
}
```

- The first part of this could use some explaining — we retrieve the ID of the record to be deleted using `Number(e.target.parentNode.getAttribute('data-note-id'))` — recall that the ID of the record was saved in a `data-note-id` attribute on the `<li>` when it was first displayed. We do however need to pass the attribute through the global built-in [`Number\(\)`](#) object, as it is currently a string, and otherwise won't be recognized by the database.
- We then get a reference to the object store using the same pattern we've seen previously, and use the [`IDBObjectStore.delete\(\)`](#) method to delete the record from the database, passing it the ID.

- When the database transaction is complete, we delete the note's `<li>` from the DOM, and again do the check to see if the `<ul>` is now empty, inserting a note as appropriate.

So that's it! Your example should now work.

If you are having trouble with it, feel free to [check it against our live example](#) (see the [source code](#) also).

## Storing complex data via IndexedDB

As we mentioned above, IndexedDB can be used to store more than just simple text strings. You can store just about anything you want, including complex objects such as video or image blobs. And it isn't much more difficult to achieve than any other type of data.

To demonstrate how to do it, we've written another example called [IndexedDB video store](#) (see it [running live here also](#)). When you first run the example, it downloads all the videos from the network, stores them in an IndexedDB database, and then displays the videos in the UI inside `<video>` elements. The second time you run it, it finds the videos in the database and gets them from there instead before displaying them — this makes subsequent loads much quicker and less bandwidth-hungry.

Let's walk through the most interesting parts of the example. We won't look at it all — a lot of it is similar to the previous example, and the code is well-commented.

1. For this simple example, we've stored the names of the videos to fetch in an array of objects:

```
2. const videos = [
3. { 'name' : 'crystal' },
4. { 'name' : 'elf' },
5. { 'name' : 'frog' },
6. { 'name' : 'monster' },
7. { 'name' : 'pig' },
8. { 'name' : 'rabbit' }
];
```

- To start with, once the database is successfully opened we run an `init()` function. This loops through the different video names, trying to load a record identified by each name from the `videos` database.

If each video is found in the database (easily checked by seeing whether `request.result` evaluates to `true` — if the record is not present, it will be `undefined`), its video files (stored as blobs) and the video name are passed straight to the `displayVideo()` function to place them in the UI. If not, the video name is passed to the `fetchVideoFromNetwork()` function to ... you guessed it — fetch the video from the network.

```
function init() {
 // Loop through the video names one by one
 for(let i = 0; i < videos.length; i++) {
```

```

// Open transaction, get object store, and get() each video by name
let objectStore = db.transaction('videos').objectStore('videos');
let request = objectStore.get(videos[i].name);
request.onsuccess = function() {
 // If the result exists in the database (is not undefined)
 if(request.result) {
 // Grab the videos from IDB and display them using displayVideo()
 console.log('taking videos from IDB');
 displayVideo(request.result.mp4, request.result.webm,
request.result.name);
 } else {
 // Fetch the videos from the network
 fetchVideoFromNetwork(videos[i]);
 }
};

}

}

```

- • The following snippet is taken from inside `fetchVideoFromNetwork()` — here we fetch MP4 and WebM versions of the video using two separate [WindowOrWorkerGlobalScope.fetch\(\)](#) requests. We then use the [Body.blob\(\)](#) method to extract each response's body as a blob, giving us an object representation of the videos that can be stored and displayed later on.

We have a problem here though — these two requests are both asynchronous, but we only want to try to display or store the video when both promises have fulfilled. Fortunately there is a built-in method that handles such a problem — [Promise.all\(\)](#). This takes one argument — references to all the individual promises you want to check for fulfillment placed in an array — and is itself promise-based.

When all those promises have fulfilled, the `all()` promise fulfills with an array containing all the individual fulfillment values. Inside the `all()` block, you can see that we then call the `displayVideo()` function like we did before to display the videos in the UI, then we also call the `storeVideo()` function to store those videos inside the database.

```

let mp4Blob = fetch('videos/' + video.name + '.mp4').then(response =>
 response.blob()
);
let webmBlob = fetch('videos/' + video.name + '.webm').then(response =>
 response.blob()
);

// Only run the next code when both promises have fulfilled
Promise.all([mp4Blob, webmBlob]).then(function(values) {
 // display the video fetched from the network with displayVideo()
 displayVideo(values[0], values[1], video.name);
 // store it in the IDB using storeVideo()
 storeVideo(values[0], values[1], video.name);
});

```

- • Let's look at `storeVideo()` first. This is very similar to the pattern you saw in the previous example for adding data to the database — we open a `readwrite` transaction and get an object

store reference to our `videos`, create an object representing the record to add to the database, then simply add it using [`IDBObjectStore.add\(\)`](#).

```
function storeVideo(mp4Blob, webmBlob, name) {
 // Open transaction, get object store; make it a readwrite so we can write
 to the IDB
 let objectStore = db.transaction(['videos'],
 'readwrite').objectStore('videos');
 // Create a record to add to the IDB
 let record = {
 mp4 : mp4Blob,
 webm : webmBlob,
 name : name
 }

 // Add the record to the IDB using add()
 let request = objectStore.add(record);

 ...
}

};
```

- • Last but not least, we have `displayVideo()`, which creates the DOM elements needed to insert the video in the UI and then appends them to the page. The most interesting parts of this are those shown below — to actually display our video blobs in a `<video>` element, we need to create object URLs (internal URLs that point to the video blobs stored in memory) using the [`URL.createObjectURL\(\)`](#) method. Once that is done, we can set the object URLs to be the values of our `<source>` element's `src` attributes, and it works fine.

```
function displayVideo(mp4Blob, webmBlob, title) {
 // Create object URLs out of the blobs
 let mp4URL = URL.createObjectURL(mp4Blob);
 let webmURL = URL.createObjectURL(webmBlob);

 ...

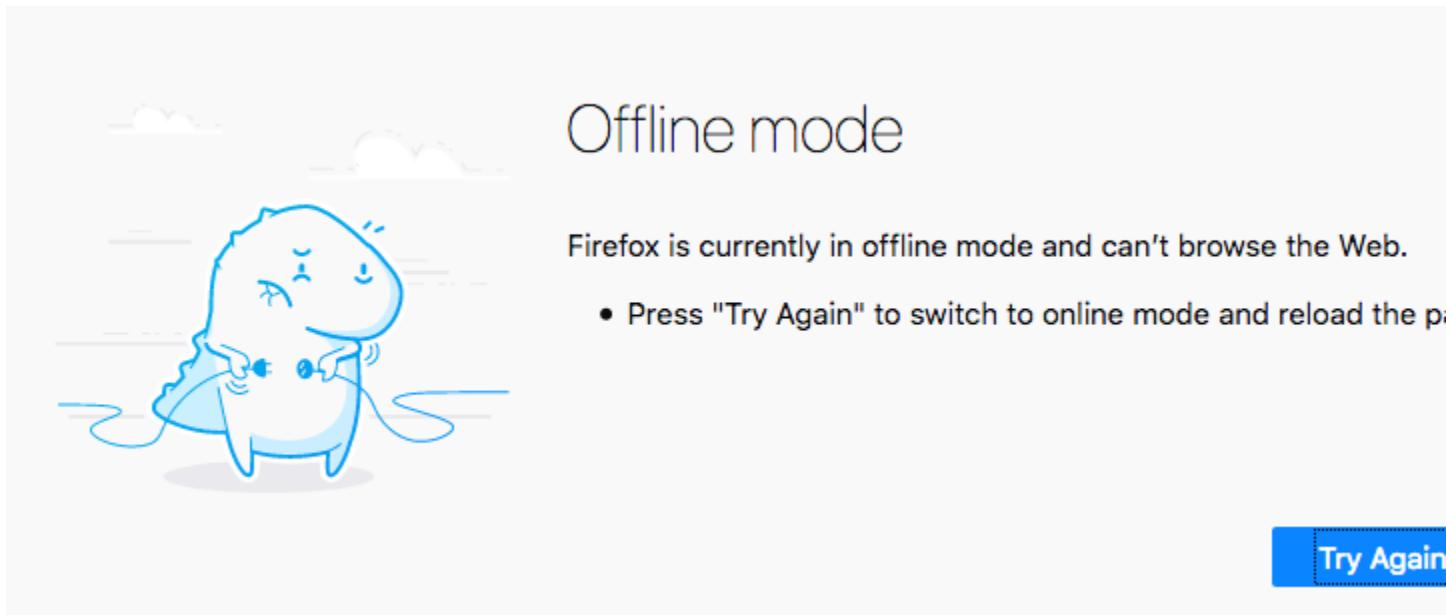
 let video = document.createElement('video');
 video.controls = true;
 let source1 = document.createElement('source');
 source1.src = mp4URL;
 source1.type = 'video/mp4';
 let source2 = document.createElement('source');
 source2.src = webmURL;
 source2.type = 'video/webm';

 ...
}
```

5.

## Offline asset storage

The above example already shows how to create an app that will store large assets in an IndexedDB database, avoiding the need to download them more than once. This is already a great improvement to the user experience, but there is still one thing missing — the main HTML, CSS, and JavaScript files still need to be downloaded each time the site is accessed, meaning that it won't work when there is no network connection.



This is where [Service workers](#) and the closely-related [Cache API](#) come in.

A service worker is a JavaScript file that, simply put, is registered against a particular origin (web site, or part of a web site at a certain domain) when it is accessed by a browser. When registered, it can control pages available at that origin. It does this by sitting between a loaded page and the network and intercepting network requests aimed at that origin.

When it intercepts a request, it can do anything you wish to it (see [use case ideas](#)), but the classic example is saving the network responses offline and then providing those in response to a request instead of the responses from the network. In effect, it allows you to make a web site work completely offline.

The Cache API is another client-side storage mechanism, with a bit of a difference — it is designed to save HTTP responses, and so works very well with service workers.

**Note:** Service workers and Cache are supported in most modern browsers now. At the time of writing, Safari was still busy implementing it, but it should be there soon.

### A service worker example

Let's look at an example, to give you a bit of an idea of what this might look like. We have created another version of the video store example we saw in the previous section — this

functions identically, except that it also saves the HTML, CSS, and JavaScript in the Cache API via a service worker, allowing the example to run offline!

See [IndexedDB video store with service worker running live](#), and also [see the source code](#).

## Registering the service worker

The first thing to note is that there's an extra bit of code placed in the main JavaScript file (see [index.js](#)). First we do a feature detection test to see if the `serviceWorker` member is available in the [Navigator](#) object. If this returns true, then we know that at least the basics of service workers are supported. Inside here we use the [ServiceWorkerContainer.register\(\)](#) method to register a service worker contained in the `sw.js` file against the origin it resides at, so it can control pages in the same directory as it, or subdirectories. When its promise fulfills, the service worker is deemed registered.

```
// Register service worker to control making site work offline

if('serviceWorker' in navigator) {
 navigator.serviceWorker
 .register('/learning-area/javascript/apis/client-side-
storage/cache-sw/video-store-offline/sw.js')
 .then(function() { console.log('Service Worker Registered'); });
}
```

**Note:** The given path to the `sw.js` file is relative to the site origin, not the JavaScript file that contains the code. The service worker is at <https://mdn.github.io/learning-area/javascript/apis/client-side-storage/cache-sw/video-store-offline/sw.js>. The origin is <https://mdn.github.io>, and therefore the given path has to be `/learning-area/javascript/apis/client-side-storage/cache-sw/video-store-offline/sw.js`. If you wanted to host this example on your own server, you'd have to change this accordingly. This is rather confusing, but it has to work this way for security reasons.

## Installing the service worker

The next time any page under the service worker's control is accessed (e.g. when the example is reloaded), the service worker is installed against that page, meaning that it will start controlling it. When this occurs, an `install` event is fired against the service worker; you can write code inside the service worker itself that will respond to the installation.

Let's look at an example, in the [sw.js](#) file (the service worker). You'll see that the `install` listener is registered against `self`. This `self` keyword is a way to refer to the global scope of the service worker from inside the service worker file.

Inside the `install` handler we use the [ExtendableEvent.waitUntil\(\)](#) method, available on the event object, to signal that the browser shouldn't complete installation of the service worker until after the promise inside it has fulfilled successfully.

Here is where we see the Cache API in action. We use the [CacheStorage.open\(\)](#) method to open a new cache object in which responses can be stored (similar to an IndexedDB object store). This promise fulfills with a [Cache](#) object representing the video-store cache. We then use the [Cache.addAll\(\)](#) method to fetch a series of assets and add their responses to the cache.

```
self.addEventListener('install', function(e) {
 e.waitUntil(
 caches.open('video-store').then(function(cache) {
 return cache.addAll([
 '/learning-area/javascript/apis/client-side-storage/cache-sw/video-store-offline/',
 '/learning-area/javascript/apis/client-side-storage/cache-sw/video-store-offline/index.html',
 '/learning-area/javascript/apis/client-side-storage/cache-sw/video-store-offline/index.js',
 '/learning-area/javascript/apis/client-side-storage/cache-sw/video-store-offline/style.css'
]);
 })
);
});
```

That's it for now, installation done.

## Responding to further requests

With the service worker registered and installed against our HTML page, and the relevant assets all added to our cache, we are nearly ready to go. There is only one more thing to do, write some code to respond to further network requests.

This is what the second bit of code in `sw.js` does. We add another listener to the service worker global scope, which runs the handler function when the `fetch` event is raised. This happens whenever the browser makes a request for an asset in the directory the service worker is registered against.

Inside the handler we first log the URL of the requested asset. We then provide a custom response to the request, using the [FetchEvent.respondWith\(\)](#) method.

Inside this block we use [CacheStorage.match\(\)](#) to check whether a matching request (i.e. matches the URL) can be found in any cache. This promise fulfills with the matching response if a match is not found, or `undefined` if it isn't.

If a match is found, we simply return it as the custom response. If not, we [fetch\(\)](#) the response from the network and return that instead.

```
self.addEventListener('fetch', function(e) {
 console.log(e.request.url);
 e.respondWith(
 caches.match(e.request).then(function(response) {
 return response || fetch(e.request);
 })
);
});
```

```
 })
);
}) ;
```

And that is it for our simple service worker. There is a whole load more you can do with them — for a lot more detail, see the [service worker cookbook](#). And thanks to Paul Kinlan for his article [Adding a Service Worker and Offline into your Web App](#), which inspired this simple example.

## Testing the example offline

To test our [service worker example](#), you'll need to load it a couple of times to make sure it is installed. Once this is done, you can:

- Try unplugging your network/turning your Wifi off.
- Select *File > Work Offline* if you are using Firefox.
- Go to the devtools, then choose *Application > Service Workers*, then check the *Offline* checkbox if you are using Chrome.

If you refresh your example page again, you should still see it load just fine. Everything is stored offline — the page assets in a cache, and the videos in an IndexedDB database.

## Summary

That's it for now. We hope you've found our rundown of client-side storage technologies useful.

## A re-introduction to JavaScript (JS tutorial)

Why a re-introduction? Because [JavaScript](#) is notorious for being [the world's most misunderstood programming language](#). It is often derided as being a toy, but beneath its layer of deceptive simplicity, powerful language features await. JavaScript is now used by an incredible number of high-profile applications, showing that deeper knowledge of this technology is an important skill for any web or mobile developer.

It's useful to start with an overview of the language's history. JavaScript was created in 1995 by Brendan Eich while he was an engineer at Netscape. JavaScript was first released with Netscape 2 early in 1996. It was originally going to be called LiveScript, but it was renamed in an ill-fated marketing decision that attempted to capitalize on the popularity of Sun Microsystem's Java language — despite the two having very little in common. This has been a source of confusion ever since.

Several months later, Microsoft released JScript with Internet Explorer 3. It was a mostly-compatible JavaScript work-alike. Several months after that, Netscape submitted JavaScript to [Ecma International](#), a European standards organization, which resulted in the first edition of the [ECMAScript](#) standard that year. The standard received a significant update as [ECMAScript](#)

[edition 3](#) in 1999, and it has stayed pretty much stable ever since. The fourth edition was abandoned, due to political differences concerning language complexity. Many parts of the fourth edition formed the basis for ECMAScript edition 5, published in December of 2009, and for the 6th major edition of the standard, published in June of 2015.

Because it is more familiar, we will refer to ECMAScript as "JavaScript" from this point on.

Unlike most programming languages, the JavaScript language has no concept of input or output. It is designed to run as a scripting language in a host environment, and it is up to the host environment to provide mechanisms for communicating with the outside world. The most common host environment is the browser, but JavaScript interpreters can also be found in a huge list of other places, including Adobe Acrobat, Adobe Photoshop, SVG images, Yahoo's Widget engine, server-side environments such as [Node.js](#), NoSQL databases like the open source [Apache CouchDB](#), embedded computers, complete desktop environments like [GNOME](#) (one of the most popular GUIs for GNU/Linux operating systems), and others.

## Overview

JavaScript is a multi-paradigm, dynamic language with types and operators, standard built-in objects, and methods. Its syntax is based on the Java and C languages — many structures from those languages apply to JavaScript as well. JavaScript supports object-oriented programming with object prototypes, instead of classes (see more about [prototypical inheritance](#) and [ES2015 classes](#)). JavaScript also supports functional programming — functions are objects, giving functions the capacity to hold executable code and be passed around like any other object.

Let's start off by looking at the building blocks of any language: the types. JavaScript programs manipulate values, and those values all belong to a type. JavaScript's types are:

- [Number](#)
- [String](#)
- [Boolean](#)
- [Function](#)
- [Object](#)
- [Symbol](#) (new in ES2015)

... oh, and [undefined](#) and [null](#), which are ... slightly odd. And [Array](#), which is a special kind of object. And [Date](#) and [RegExp](#), which are objects that you get for free. And to be technically accurate, functions are just a special type of object. So the type diagram looks more like this:

- [Number](#)
- [String](#)
- [Boolean](#)
- [Symbol](#) (new in ES2015)
- [Object](#)
  - [Function](#)
  - [Array](#)
  - [Date](#)

- `null`
- `undefined`
- `RegExp`

And there are some built-in `Error` types as well. Things are a lot easier if we stick with the first diagram, however, so we'll discuss the types listed there for now.

## Numbers

Numbers in JavaScript are "double-precision 64-bit format IEEE 754 values", according to the spec. This has some interesting consequences. There's no such thing as an integer in JavaScript, so you have to be a little careful with your arithmetic if you're used to math in C or Java.

Also, watch out for stuff like:

```
0.1 + 0.2 == 0.30000000000000004;
```

In practice, integer values are treated as 32-bit ints, and some implementations even store it that way until they are asked to perform an instruction that's valid on a Number but not on a 32-bit integer. This can be important for bit-wise operations.

The standard [arithmetic operators](#) are supported, including addition, subtraction, modulus (or remainder) arithmetic, and so forth. There's also a built-in object that we forgot to mention earlier called `Math` that provides advanced mathematical functions and constants:

```
Math.sin(3.5);
var circumference = 2 * Math.PI * r;
```

You can convert a string to an integer using the built-in `parseInt()` function. This takes the base for the conversion as an optional second argument, which you should always provide:

```
parseInt('123', 10); // 123
parseInt('010', 10); // 10
```

In older browsers, strings beginning with a "0" are assumed to be in octal (radix 8), but this hasn't been the case since 2013 or so. Unless you're certain of your string format, you can get surprising results on those older browsers:

```
parseInt('010'); // 8
parseInt('0x10'); // 16
```

Here, we see the `parseInt()` function treat the first string as octal due to the leading 0, and the second string as hexadecimal due to the leading "0x". The *hexadecimal notation is still in place*; only octal has been removed.

If you want to convert a binary number to an integer, just change the base:

```
parseInt('11', 2); // 3
```

Similarly, you can parse floating point numbers using the built-in `parseFloat()` function. Unlike its `parseInt()` cousin, `parseFloat()` always uses base 10.

You can also use the unary + operator to convert values to numbers:

```
+ '42'; // 42
+ '010'; // 10
+ '0x10'; // 16
```

A special value called `NaN` (short for "Not a Number") is returned if the string is non-numeric:

```
parseInt('hello', 10); // NaN
```

`NaN` is toxic: if you provide it as an operand to any mathematical operation the result will also be `NaN`:

```
NaN + 5; // NaN
```

You can test for `NaN` using the built-in `isNaN()` function:

```
isNaN(NaN); // true
```

JavaScript also has the special values `Infinity` and `-Infinity`:

```
1 / 0; // Infinity
-1 / 0; // -Infinity
```

You can test for `Infinity`, `-Infinity` and `NaN` values using the built-in `isFinite()` function:

```
isFinite(1 / 0); // false
isFinite(-Infinity); // false
isFinite(NaN); // false
```

The `parseInt()` and `parseFloat()` functions parse a string until they reach a character that isn't valid for the specified number format, then return the number parsed up to that point. However the "+" operator simply converts the string to `NaN` if there is an invalid character contained within it. Just try parsing the string "10.2abc" with each method by yourself in the console and you'll understand the differences better.

## Strings

Strings in JavaScript are sequences of [Unicode characters](#). This should be welcome news to anyone who has had to deal with internationalization. More accurately, they are sequences of UTF-16 code units; each code unit is represented by a 16-bit number. Each Unicode character is represented by either 1 or 2 code units.

If you want to represent a single character, you just use a string consisting of that single character.

To find the length of a string (in code units), access its [length](#) property:

```
'hello'.length; // 5
```

There's our first brush with JavaScript objects! Did we mention that you can use strings like [objects](#) too? They have [methods](#) as well that allow you to manipulate the string and access information about the string:

```
'hello'.charAt(0); // "h"
'hello, world'.replace('hello', 'goodbye'); // "goodbye, world"
'hello'.toUpperCase(); // "HELLO"
```

## Other types

JavaScript distinguishes between [null](#), which is a value that indicates a deliberate non-value (and is only accessible through the [null](#) keyword), and [undefined](#), which is a value of type [undefined](#) that indicates an uninitialized value — that is, a value hasn't even been assigned yet. We'll talk about variables later, but in JavaScript it is possible to declare a variable without assigning a value to it. If you do this, the variable's type is [undefined](#). [undefined](#) is actually a constant.

JavaScript has a boolean type, with possible values [true](#) and [false](#) (both of which are keywords.) Any value can be converted to a boolean according to the following rules:

1. [false](#), 0, empty strings (""), [NaN](#), [null](#), and [undefined](#) all become [false](#).
2. All other values become [true](#).

You can perform this conversion explicitly using the `Boolean()` function:

```
Boolean(''); // false
Boolean(234); // true
```

However, this is rarely necessary, as JavaScript will silently perform this conversion when it expects a boolean, such as in an `if` statement (see below). For this reason, we sometimes speak simply of "true values" and "false values," meaning values that become [true](#) and [false](#), respectively, when converted to booleans. Alternatively, such values can be called "truthy" and "falsy", respectively.

Boolean operations such as `&&` (logical *and*), `||` (logical *or*), and `!` (logical *not*) are supported; see below.

## Variables

New variables in JavaScript are declared using one of three keywords: [let](#), [const](#), or [var](#).

**let** allows you to declare block-level variables. The declared variable is available from the *block* it is enclosed in.

```
let a;
let name = 'Simon';
```

The following is an example of scope with a variable declared with **let**:

```
// myLetVariable is *not* visible out here

for (let myLetVariable = 0; myLetVariable < 5; myLetVariable++) {
 // myLetVariable is only visible in here
}

// myLetVariable is *not* visible out here
```

**const** allows you to declare variables whose values are never intended to change. The variable is available from the *block* it is declared in.

```
const Pi = 3.14; // variable Pi is set
Pi = 1; // will throw an error because you cannot change a constant variable.
```

**var** is the most common declarative keyword. It does not have the restrictions that the other two keywords have. This is because it was traditionally the only way to declare a variable in JavaScript. A variable declared with the **var** keyword is available from the *function* it is declared in.

```
var a;
var name = 'Simon';
```

An example of scope with a variable declared with **var**:

```
// myVarVariable *is* visible out here

for (var myVarVariable = 0; myVarVariable < 5; myVarVariable++) {
 // myVarVariable is visible to the whole function
}

// myVarVariable *is* visible out here
```

If you declare a variable without assigning any value to it, its type is `undefined`.

An important difference between JavaScript and other languages like Java is that in JavaScript, blocks do not have scope; only functions have a scope. So if a variable is defined using **var** in a compound statement (for example inside an **if** control structure), it will be visible to the entire function. However, starting with ECMAScript 2015, [let](#) and [const](#) declarations allow you to create block-scoped variables.

# Operators

JavaScript's numeric operators are `+`, `-`, `*`, `/` and `%` which is the remainder operator ([which is not the same as modulo](#).) Values are assigned using `=`, and there are also compound assignment statements such as `+=` and `-=`. These extend out to `x = x operator y`.

```
x += 5;
x = x + 5;
```

You can use `++` and `--` to increment and decrement respectively. These can be used as a prefix or postfix operators.

The [+ operator](#) also does string concatenation:

```
'hello' + ' world'; // "hello world"
```

If you add a string to a number (or other value) everything is converted in to a string first. This might catch you up:

```
'3' + 4 + 5; // "345"
3 + 4 + '5'; // "75"
```

Adding an empty string to something is a useful way of converting it to a string itself.

[Comparisons](#) in JavaScript can be made using `<`, `>`, `<=` and `>=`. These work for both strings and numbers. Equality is a little less straightforward. The double-equals operator performs type coercion if you give it different types, with sometimes interesting results:

```
123 == '123'; // true
1 == true; // true
```

To avoid type coercion, use the triple-equals operator:

```
123 === '123'; // false
1 === true; // false
```

There are also `!=` and `!==` operators.

JavaScript also has [bitwise operations](#). If you want to use them, they're there.

# Control structures

JavaScript has a similar set of control structures to other languages in the C family. Conditional statements are supported by `if` and `else`; you can chain them together if you like:

```
var name = 'kittens';
if (name == 'puppies') {
```

```
name += ' woof';
} else if (name == 'kittens') {
 name += ' meow';
} else {
 name += ' !';
}
name == 'kittens meow';
```

JavaScript has `while` loops and `do-while` loops. The first is good for basic looping; the second for loops where you wish to ensure that the body of the loop is executed at least once:

```
while (true) {
 // an infinite loop!
}

var input;
do {
 input = get_input();
} while (inputIsNotValid(input));
```

JavaScript's [for loop](#) is the same as that in C and Java: it lets you provide the control information for your loop on a single line.

```
for (var i = 0; i < 5; i++) {
 // Will execute 5 times
}
```

JavaScript also contains two other prominent for loops: [for...of](#)

```
for (let value of array) {
 // do something with value
}
```

and [for...in](#):

```
for (let property in object) {
 // do something with object property
}
```

The `&&` and `||` operators use short-circuit logic, which means whether they will execute their second operand is dependent on the first. This is useful for checking for null objects before accessing their attributes:

```
var name = o && o.getName();
```

Or for caching values (when falsy values are invalid):

```
var name = cachedName || (cachedName = getName());
```

JavaScript has a ternary operator for conditional expressions:

```
var allowed = (age > 18) ? 'yes' : 'no';
```

The `switch` statement can be used for multiple branches based on a number or string:

```
switch (action) {
 case 'draw':
 drawIt();
 break;
 case 'eat':
 eatIt();
 break;
 default:
 doNothing();
}
```

If you don't add a `break` statement, execution will "fall through" to the next level. This is very rarely what you want — in fact it's worth specifically labeling deliberate fallthrough with a comment if you really meant it to aid debugging:

```
switch (a) {
 case 1: // fallthrough
 case 2:
 eatIt();
 break;
 default:
 doNothing();
}
```

The default clause is optional. You can have expressions in both the switch part and the cases if you like; comparisons take place between the two using the `==` operator:

```
switch (1 + 3) {
 case 2 + 2:
 yay();
 break;
 default:
 neverhappens();
}
```

## Objects

JavaScript objects can be thought of as simple collections of name-value pairs. As such, they are similar to:

- Dictionaries in Python.
- Hashes in Perl and Ruby.
- Hash tables in C and C++.
- HashMaps in Java.
- Associative arrays in PHP.

The fact that this data structure is so widely used is a testament to its versatility. Since everything (bar core types) in JavaScript is an object, any JavaScript program naturally involves a great deal of hash table lookups. It's a good thing they're so fast!

The "name" part is a JavaScript string, while the value can be any JavaScript value — including more objects. This allows you to build data structures of arbitrary complexity.

There are two basic ways to create an empty object:

```
var obj = new Object();
```

And:

```
var obj = {};
```

These are semantically equivalent; the second is called object literal syntax and is more convenient. This syntax is also the core of JSON format and should be preferred at all times.

Object literal syntax can be used to initialize an object in its entirety:

```
var obj = {
 name: 'Carrot',
 for: 'Max', // 'for' is a reserved word, use '_for' instead.
 details: {
 color: 'orange',
 size: 12
 }
};
```

Attribute access can be chained together:

```
obj.details.color; // orange
obj['details']['size']; // 12
```

The following example creates an object prototype, `Person` and an instance of that prototype, `You`.

```
function Person(name, age) {
 this.name = name;
 this.age = age;
}

// Define an object
var you = new Person('You', 24);
// We are creating a new person named "You" aged 24.
```

**Once created**, an object's properties can again be accessed in one of two ways:

```
//dot notation
obj.name = 'Simon';
```

```
var name = obj.name;
```

And...

```
// bracket notation
obj['name'] = 'Simon';
var name = obj['name'];
// can use a variable to define a key
var user = prompt('what is your key?')
obj[user] = prompt('what is its value?')
```

These are also semantically equivalent. The second method has the advantage that the name of the property is provided as a string, which means it can be calculated at run-time. However, using this method prevents some JavaScript engine and minifier optimizations being applied. It can also be used to set and get properties with names that are [reserved words](#):

```
obj.for = 'Simon'; // Syntax error, because 'for' is a reserved word
obj['for'] = 'Simon'; // works fine
```

Starting in ECMAScript 5, reserved words may be used as object property names "in the buff". This means that they don't need to be "clothed" in quotes when defining object literals. See the [ES5 Spec](#).

For more on objects and prototypes see [Object.prototype](#). For an explanation of object prototypes and the object prototype chains see [Inheritance and the prototype chain](#).

Starting in ECMAScript 2015, object keys can be defined by the variable using bracket notation upon being created. `{ [phoneType]: 12345 }` is possible instead of just `var userPhone = {};` `userPhone[phoneType] = 12345.`

## Arrays

Arrays in JavaScript are actually a special type of object. They work very much like regular objects (numerical properties can naturally be accessed only using `[]` syntax) but they have one magic property called '`length`'. This is always one more than the highest index in the array.

One way of creating arrays is as follows:

```
var a = new Array();
a[0] = 'dog';
a[1] = 'cat';
a[2] = 'hen';
a.length; // 3
```

A more convenient notation is to use an array literal:

```
var a = ['dog', 'cat', 'hen'];
a.length; // 3
```

Note that `array.length` isn't necessarily the number of items in the array. Consider the following:

```
var a = ['dog', 'cat', 'hen'];
a[100] = 'fox';
a.length; // 101
```

Remember — the length of the array is one more than the highest index.

If you query a non-existent array index, you'll get a value of `undefined` in return:

```
typeof a[90]; // undefined
```

If you take the above about `[]` and `length` into account, you can iterate over an array using the following `for` loop:

```
for (var i = 0; i < a.length; i++) {
 // Do something with a[i]
}
```

ECMAScript introduced the more concise [for...of](#) loop for iterable objects such as arrays:

```
for (const currentValue of a) {
 // Do something with currentValue
}
```

You could also iterate over an array using a [for...in](#) loop. But if someone added new properties to `Array.prototype`, they would also be iterated over by this loop. Therefore this loop type is not recommended for arrays.

Another way of iterating over an array that was added with ECMAScript 5 is [forEach\(\)](#):

```
['dog', 'cat', 'hen'].forEach(function(currentValue, index, array) {
 // Do something with currentValue or array[index]
});
```

If you want to append an item to an array simply do it like this:

```
a.push(item);
```

Arrays come with a number of methods. See also the [full documentation for array methods](#).

| Method name                                 | Description                                                                                  |
|---------------------------------------------|----------------------------------------------------------------------------------------------|
| <code>a.toString()</code>                   | Returns a string with the <code>toString()</code> of each element separated by commas.       |
| <code>a.toLocaleString()</code>             | Returns a string with the <code>toLocaleString()</code> of each element separated by commas. |
| <code>a.concat(item1[, item2[, ...[,</code> | Returns a new array with the items added on to it.                                           |

| Method name                                        | Description                                                                          |
|----------------------------------------------------|--------------------------------------------------------------------------------------|
| itemN] ]])                                         |                                                                                      |
| a.join(sep)                                        | Converts the array to a string — with values delimited by the <code>sep</code> param |
| a.pop()                                            | Removes and returns the last item.                                                   |
| a.push(item1, ..., itemN)                          | Appends items to the end of the array.                                               |
| a.reverse()                                        | Reverses the array.                                                                  |
| a.shift()                                          | Removes and returns the first item.                                                  |
| a.slice(start[, end])                              | Returns a sub-array.                                                                 |
| a.sort([cmpfn])                                    | Takes an optional comparison function.                                               |
| a.splice(start, delcount[, item1[, ...[, itemN]]]) | Lets you modify an array by deleting a section and replacing it with more items.     |
| a.unshift(item1[, item2[, ...[, itemN]]])          | Prepends items to the start of the array.                                            |

## Functions

Along with objects, functions are the core component in understanding JavaScript. The most basic function couldn't be much simpler:

```
function add(x, y) {
 var total = x + y;
 return total;
}
```

This demonstrates a basic function. A JavaScript function can take 0 or more named parameters. The function body can contain as many statements as you like and can declare its own variables which are local to that function. The `return` statement can be used to return a value at any time, terminating the function. If no `return` statement is used (or an empty `return` with no value), JavaScript returns `undefined`.

The named parameters turn out to be more like guidelines than anything else. You can call a function without passing the parameters it expects, in which case they will be set to `undefined`.

```
add(); // NaN
// You can't perform addition on undefined
```

You can also pass in more arguments than the function is expecting:

```
add(2, 3, 4); // 5
// added the first two; 4 was ignored
```

That may seem a little silly, but functions have access to an additional variable inside their body called `arguments`, which is an array-like object holding all of the values passed to the function. Let's re-write the `add` function to take as many values as we want:

```

function add() {
 var sum = 0;
 for (var i = 0, j = arguments.length; i < j; i++) {
 sum += arguments[i];
 }
 return sum;
}

add(2, 3, 4, 5); // 14

```

That's really not any more useful than writing `2 + 3 + 4 + 5` though. Let's create an averaging function:

```

function avg() {
 var sum = 0;
 for (var i = 0, j = arguments.length; i < j; i++) {
 sum += arguments[i];
 }
 return sum / arguments.length;
}

avg(2, 3, 4, 5); // 3.5

```

This is pretty useful, but it does seem a little verbose. To diminish this code a bit more we can look at substituting the use of the `arguments` array through [Rest parameter syntax](#). In this way, we can pass in any number of arguments into the function while keeping our code minimal. The **rest parameter operator** is used in function parameter lists with the format: `...variable` and it will include within that variable the entire list of uncaptured arguments that the function was called with. We will also replace the `for` loop with a `for...of` loop to return the values within our variable.

```

function avg(...args) {
 var sum = 0;
 for (let value of args) {
 sum += value;
 }
 return sum / args.length;
}

avg(2, 3, 4, 5); // 3.5

```

In the above code, the variable `args` holds all the values that were passed into the function.

It is important to note that wherever the rest parameter operator is placed in a function declaration it will store all arguments *after* its declaration, but not before. *i.e.* `function avg(firstValue, ...args)` will store the first value passed into the function in the `firstValue` variable and the remaining arguments in `args`. Another useful function but it does lead us to a new problem. The `avg()` function takes a comma-separated list of arguments — but what if you want to find the average of an array? You could just rewrite the function as follows:

```

function avgArray(arr) {
 var sum = 0;
 for (var i = 0, j = arr.length; i < j; i++) {
 sum += arr[i];
 }
}

```

```
 }
 return sum / arr.length;
 }

avgArray([2, 3, 4, 5]); // 3.5
```

But it would be nice to be able to reuse the function that we've already created. Luckily, JavaScript lets you call a function with an arbitrary array of arguments, using the `apply()` method of any function object.

```
avg.apply(null, [2, 3, 4, 5]); // 3.5
```

The second argument to `apply()` is the array to use as arguments; the first will be discussed later on. This emphasizes the fact that functions are objects too.

You can achieve the same result using the [spread operator](#) in the function call.

For instance: `avg(...numbers)`

JavaScript lets you create anonymous functions.

```
var avg = function() {
 var sum = 0;
 for (var i = 0, j = arguments.length; i < j; i++) {
 sum += arguments[i];
 }
 return sum / arguments.length;
};
```

This is semantically equivalent to the `function avg()` form. It's extremely powerful, as it lets you put a full function definition anywhere that you would normally put an expression. This enables all sorts of clever tricks. Here's a way of "hiding" some local variables — like block scope in C:

```
var a = 1;
var b = 2;

(function() {
 var b = 3;
 a += b;
})();

a; // 4
b; // 2
```

JavaScript allows you to call functions recursively. This is particularly useful for dealing with tree structures, such as those found in the browser DOM.

```
function countChars(elm) {
 if (elm.nodeType == 3) { // TEXT_NODE
 return elm.nodeValue.length;
```

```

 }
 var count = 0;
 for (var i = 0, child; child = elm.childNodes[i]; i++) {
 count += countChars(child);
 }
 return count;
 }
}

```

This highlights a potential problem with anonymous functions: how do you call them recursively if they don't have a name? JavaScript lets you name function expressions for this. You can use named IIFEs (Immediately Invoked Function Expressions) as shown below:

```

var charsInBody = (function counter(elm) {
 if (elm.nodeType == 3) { // TEXT_NODE
 return elm.nodeValue.length;
 }
 var count = 0;
 for (var i = 0, child; child = elm.childNodes[i]; i++) {
 count += counter(child);
 }
 return count;
})(document.body);

```

The name provided to a function expression as above is only available to the function's own scope. This allows more optimizations to be done by the engine and results in more readable code. The name also shows up in the debugger and some stack traces, which can save you time when debugging.

Note that JavaScript functions are themselves objects — like everything else in JavaScript — and you can add or change properties on them just like we've seen earlier in the Objects section.

## Custom objects

For a more detailed discussion of object-oriented programming in JavaScript, see [Introduction to Object-Oriented JavaScript](#).

In classic Object Oriented Programming, objects are collections of data and methods that operate on that data. JavaScript is a prototype-based language that contains no class statement, as you'd find in C++ or Java (this is sometimes confusing for programmers accustomed to languages with a class statement). Instead, JavaScript uses functions as classes. Let's consider a person object with first and last name fields. There are two ways in which the name might be displayed: as "first last" or as "last, first". Using the functions and objects that we've discussed previously, we could display the data like this:

```

function makePerson(first, last) {
 return {
 first: first,
 last: last
 };
}

```

```

function personFullName(person) {
 return person.first + ' ' + person.last;
}
function personFullNameReversed(person) {
 return person.last + ', ' + person.first;
}

s = makePerson('Simon', 'Willison');
personFullName(s); // "Simon Willison"
personFullNameReversed(s); // "Willison, Simon"

```

This works, but it's pretty ugly. You end up with dozens of functions in your global namespace. What we really need is a way to attach a function to an object. Since functions are objects, this is easy:

```

function makePerson(first, last) {
 return {
 first: first,
 last: last,
 fullName: function() {
 return this.first + ' ' + this.last;
 },
 fullNameReversed: function() {
 return this.last + ', ' + this.first;
 }
 };
}

s = makePerson('Simon', 'Willison');
s.fullName(); // "Simon Willison"
s.fullNameReversed(); // "Willison, Simon"

```

There's something here we haven't seen before: the `this` keyword. Used inside a function, `this` refers to the current object. What that actually means is specified by the way in which you called that function. If you called it using [dot notation or bracket notation](#) on an object, that object becomes `this`. If dot notation wasn't used for the call, `this` refers to the global object.

Note that `this` is a frequent cause of mistakes. For example:

```

s = makePerson('Simon', 'Willison');
var fullName = s.fullName;
fullName(); // undefined undefined

```

When we call `fullName()` alone, without using `s.fullName()`, `this` is bound to the global object. Since there are no global variables called `first` or `last` we get `undefined` for each one.

We can take advantage of the `this` keyword to improve our `makePerson` function:

```

function Person(first, last) {
 this.first = first;
 this.last = last;
 this.fullName = function() {

```

```

 return this.first + ' ' + this.last;
 };
 this.fullNameReversed = function() {
 return this.last + ', ' + this.first;
 };
}
var s = new Person('Simon', 'Willison');

```

We have introduced another keyword: `new`. `new` is strongly related to `this`. It creates a brand new empty object, and then calls the function specified, with `this` set to that new object. Notice though that the function specified with `this` does not return a value but merely modifies the `this` object. It's `new` that returns the `this` object to the calling site. Functions that are designed to be called by `new` are called constructor functions. Common practice is to capitalize these functions as a reminder to call them with `new`.

The improved function still has the same pitfall with calling `fullName()` alone.

Our person objects are getting better, but there are still some ugly edges to them. Every time we create a person object we are creating two brand new function objects within it — wouldn't it be better if this code was shared?

```

function personFullName() {
 return this.first + ' ' + this.last;
}
function personFullNameReversed() {
 return this.last + ', ' + this.first;
}
function Person(first, last) {
 this.first = first;
 this.last = last;
 this.fullName = personFullName;
 this.fullNameReversed = personFullNameReversed;
}

```

That's better: we are creating the method functions only once, and assigning references to them inside the constructor. Can we do any better than that? The answer is yes:

```

function Person(first, last) {
 this.first = first;
 this.last = last;
}
Person.prototype.fullName = function() {
 return this.first + ' ' + this.last;
};
Person.prototype.fullNameReversed = function() {
 return this.last + ', ' + this.first;
};

```

`Person.prototype` is an object shared by all instances of `Person`. It forms part of a lookup chain (that has a special name, "prototype chain"): any time you attempt to access a property of `Person` that isn't set, JavaScript will check `Person.prototype` to see if that property exists there

instead. As a result, anything assigned to `Person.prototype` becomes available to all instances of that constructor via the `this` object.

This is an incredibly powerful tool. JavaScript lets you modify something's prototype at any time in your program, which means you can add extra methods to existing objects at runtime:

```
s = new Person('Simon', 'Willison');
s.firstNameCaps(); // TypeError on line 1: s.firstNameCaps is not a function

Person.prototype.firstNameCaps = function() {
 return this.firstName.toUpperCase();
};

s.firstNameCaps(); // "SIMON"
```

Interestingly, you can also add things to the prototype of built-in JavaScript objects. Let's add a method to `String` that returns that string in reverse:

```
var s = 'Simon';
s.reversed(); // TypeError on line 1: s.reversed is not a function

String.prototype.reversed = function() {
 var r = '';
 for (var i = this.length - 1; i >= 0; i--) {
 r += this[i];
 }
 return r;
};

s.reversed(); // nomiS
```

Our new method even works on string literals!

```
'This can now be reversed'.reversed(); // desrever eb won nac sihT
```

As mentioned before, the prototype forms part of a chain. The root of that chain is `Object.prototype`, whose methods include `toString()` — it is this method that is called when you try to represent an object as a string. This is useful for debugging our `Person` objects:

```
var s = new Person('Simon', 'Willison');
s.toString(); // [object Object]

Person.prototype.toString = function() {
 return '<Person: ' + this.fullName() + '>';
}

s.toString(); // "<Person: Simon Willison>"
```

Remember how `avg.apply()` had a null first argument? We can revisit that now. The first argument to `apply()` is the object that should be treated as `'this'`. For example, here's a trivial implementation of `new`:

```
function trivialNew(constructor, ...args) {
 var o = {}; // Create an object
 constructor.apply(o, args);
 return o;
}
```

This isn't an exact replica of `new` as it doesn't set up the prototype chain (it would be difficult to illustrate). This is not something you use very often, but it's useful to know about. In this snippet, `...args` (including the ellipsis) is called the "[rest arguments](#)" — as the name implies, this contains the rest of the arguments.

## Calling

```
var bill = trivialNew(Person, 'William', 'Orange');
```

is therefore almost equivalent to

```
var bill = new Person('William', 'Orange');
```

`apply()` has a sister function named `call`, which again lets you set `this` but takes an expanded argument list as opposed to an array.

```
function lastNameCaps() {
 return this.last.toUpperCase();
}
var s = new Person('Simon', 'Willison');
lastNameCaps.call(s);
// Is the same as:
s.lastNameCaps = lastNameCaps;
s.lastNameCaps(); // WILLISON
```

## Inner functions

JavaScript function declarations are allowed inside other functions. We've seen this once before, with an earlier `makePerson()` function. An important detail of nested functions in JavaScript is that they can access variables in their parent function's scope:

```
function parentFunc() {
 var a = 1;

 function nestedFunc() {
 var b = 4; // parentFunc can't use this
 return a + b;
 }
 return nestedFunc(); // 5
}
```

This provides a great deal of utility in writing more maintainable code. If a function relies on one or two other functions that are not useful to any other part of your code, you can nest those utility

functions inside the function that will be called from elsewhere. This keeps the number of functions that are in the global scope down, which is always a good thing.

This is also a great counter to the lure of global variables. When writing complex code it is often tempting to use global variables to share values between multiple functions — which leads to code that is hard to maintain. Nested functions can share variables in their parent, so you can use that mechanism to couple functions together when it makes sense without polluting your global namespace — "local globals" if you like. This technique should be used with caution, but it's a useful ability to have.

## Closures

This leads us to one of the most powerful abstractions that JavaScript has to offer — but also the most potentially confusing. What does this do?

```
function makeAdder(a) {
 return function(b) {
 return a + b;
 };
}
var x = makeAdder(5);
var y = makeAdder(20);
x(6); // ?
y(7); // ?
```

The name of the `makeAdder()` function should give it away: it creates new 'adder' functions, each of which, when called with one argument, adds it to the argument that it was created with.

What's happening here is pretty much the same as was happening with the inner functions earlier on: a function defined inside another function has access to the outer function's variables. The only difference here is that the outer function has returned, and hence common sense would seem to dictate that its local variables no longer exist. But they *do* still exist — otherwise, the adder functions would be unable to work. What's more, there are two different "copies" of `makeAdder()`'s local variables — one in which `a` is 5 and the other one where `a` is 20. So the result of that function calls is as follows:

```
x(6); // returns 11
y(7); // returns 27
```

Here's what's actually happening. Whenever JavaScript executes a function, a 'scope' object is created to hold the local variables created within that function. It is initialized with any variables passed in as function parameters. This is similar to the global object that all global variables and functions live in, but with a couple of important differences: firstly, a brand new scope object is created every time a function starts executing, and secondly, unlike the global object (which is accessible as `this` and in browsers as `window`) these scope objects cannot be directly accessed from your JavaScript code. There is no mechanism for iterating over the properties of the current scope object, for example.

So when `makeAdder()` is called, a scope object is created with one property: `a`, which is the argument passed to the `makeAdder()` function. `makeAdder()` then returns a newly created function. Normally JavaScript's garbage collector would clean up the scope object created for `makeAdder()` at this point, but the returned function maintains a reference back to that scope object. As a result, the scope object will not be garbage-collected until there are no more references to the function object that `makeAdder()` returned.

Scope objects form a chain called the scope chain, similar to the prototype chain used by JavaScript's object system.

A **closure** is the combination of a function and the scope object in which it was created. Closures let you save state — as such, they can often be used in place of objects. You can find [several excellent introductions to closures](#).

## JavaScript data types and data structures

Programming languages all have built-in data structures, but these often differ from one language to another. This article attempts to list the built-in data structures available in JavaScript and what properties they have; these can be used to build other data structures. When possible, comparisons with other languages are drawn.

## Dynamic typing

JavaScript is a *loosely typed* or a *dynamic* language. Variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned (and re-assigned) values of all types:

```
var foo = 42; // foo is now a Number
var foo = 'bar'; // foo is now a String
var foo = true; // foo is now a Boolean
```

## Data types

The latest ECMAScript standard defines seven data types:

- Six data types that are [primitives](#):
  - [Boolean](#)
  - [Null](#)
  - [Undefined](#)
  - [Number](#)
  - [String](#)
  - [Symbol](#) (new in ECMAScript 6)
- and [Object](#)

# Primitive values

All types except objects define immutable values (values, which are incapable of being changed). For example and unlike to C, Strings are immutable. We refer to values of these types as "primitive values".

## Boolean type

Boolean represents a logical entity and can have two values: `true`, and `false`.

## Null type

The Null type has exactly one value: `null`. See [null](#) and [Null](#) for more details.

## Undefined type

A variable that has not been assigned a value has the value `undefined`. See [undefined](#) and [Undefined](#) for more details.

## Number type

According to the ECMAScript standard, there is only one number type: the [double-precision 64-bit binary format IEEE 754 value](#) (numbers between  $-(2^{53}-1)$  and  $2^{53}-1$ ). **There is no specific type for integers**. In addition to being able to represent floating-point numbers, the number type has three symbolic values: `+Infinity`, `-Infinity`, and `NaN` (not-a-number).

To check for the largest available value or smallest available value within `+/-Infinity`, you can use the constants [Number.MAX\\_VALUE](#) or [Number.MIN\\_VALUE](#) and starting with ECMAScript 6, you are also able to check if a number is in the double-precision floating-point number range using [Number.isSafeInteger\(\)](#) as well as [Number.MAX\\_SAFE\\_INTEGER](#) and [Number.MIN\\_SAFE\\_INTEGER](#). Beyond this range, integers in JavaScript are not safe anymore and will be a double-precision floating point approximation of the value.

The number type has only one integer that has two representations: 0 is represented as `-0` and `+0`. ("0" is an alias for `+0`). In the praxis, this has almost no impact. For example `+0 === -0` is `true`. However, you are able to notice this when you divide by zero:

```
> 42 / +0
Infinity
> 42 / -0
-Infinity
```

Although a number often represents only its value, JavaScript provides [some binary operators](#). These can be used to represent several Boolean values within a single number using [bit masking](#). However, this is usually considered a bad practice, since JavaScript offers other means to

represent a set of Booleans (like an array of Booleans or an object with Boolean values assigned to named properties). Bit masking also tends to make the code more difficult to read, understand, and maintain. It may be necessary to use such techniques in very constrained environments, like when trying to cope with the storage limitation of local storage or in extreme cases when each bit over the network counts. This technique should only be considered when it is the last measure that can be taken to optimize size.

## String type

JavaScript's [String](#) type is used to represent textual data. It is a set of "elements" of 16-bit unsigned integer values. Each element in the String occupies a position in the String. The first element is at index 0, the next at index 1, and so on. The length of a String is the number of elements in it.

Unlike in languages like C, JavaScript strings are immutable. This means that once a string is created, it is not possible to modify it. However, it is still possible to create another string based on an operation on the original string. For example:

- A substring of the original by picking individual letters or using [String.substr\(\)](#).
- A concatenation of two strings using the concatenation operator (+) or [String.concat\(\)](#).

## Beware of "stringly-typing" your code!

It can be tempting to use strings to represent complex data. Doing this comes with short-term benefits:

- It is easy to build complex strings with concatenation.
- Strings are easy to debug (what you see printed is always what is in the string).
- Strings are the common denominator of a lot of APIs ([input fields](#), [local storage](#) values, [XMLHttpRequest](#) responses when using `responseText`, etc.) and it can be tempting to only work with strings.

With conventions, it is possible to represent any data structure in a string. This does not make it a good idea. For instance, with a separator, one could emulate a list (while a JavaScript array would be more suitable). Unfortunately, when the separator is used in one of the "list" elements, then, the list is broken. An escape character can be chosen, etc. All of this requires conventions and creates an unnecessary maintenance burden.

Use strings for textual data. When representing complex data, parse strings and use the appropriate abstraction.

## Symbol type

Symbols are new to JavaScript in ECMAScript Edition 6. A Symbol is a **unique** and **immutable** primitive value and may be used as the key of an Object property (see below). In some

programming languages, Symbols are called atoms. For more details see [Symbol](#) and the [Symbol](#) object wrapper in JavaScript.

# Objects

In computer science, an object is a value in memory which is possibly referenced by an [identifier](#).

## Properties

In JavaScript, objects can be seen as a collection of properties. With the [object literal syntax](#), a limited set of properties are initialized; then properties can be added and removed. Property values can be values of any type, including other objects, which enables building complex data structures. Properties are identified using key values. A key value is either a String or a Symbol value.

There are two types of object properties which have certain attributes: The data property and the accessor property.

### Data property

Associates a key with a value and has the following attributes:

Attributes of a data property

| Attribute                     | Type                | Description                                                                                                                                                     | Default value      |
|-------------------------------|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|
| <code>[[Value]]</code>        | Any JavaScript type | The value retrieved by a get access of the property.                                                                                                            | undefined          |
| <code>[[Writable]]</code>     | Boolean             | If <code>false</code> , the property's <code>[[Value]]</code> can't be changed.                                                                                 | <code>false</code> |
| <code>[[Enumerable]]</code>   | Boolean             | If <code>true</code> , the property will be enumerated in <a href="#">for...in</a> loops.<br>See also <a href="#">Enumerability and ownership of properties</a> | <code>false</code> |
| <code>[[Configurable]]</code> | Boolean             | If <code>false</code> , the property can't be deleted and attributes other than <code>[[Value]]</code> and <code>[[Writable]]</code> can't be changed.          | <code>false</code> |

Obsolete attributes (as of ECMAScript 3, renamed in ECMAScript 5)

| Attribute | Type    | Description                                                    |
|-----------|---------|----------------------------------------------------------------|
| Read-only | Boolean | Reversed state of the ES5 <code>[[Writable]]</code> attribute. |

DontEnum Boolean Reversed state of the ES5 [[Enumerable]] attribute.

DontDelete Boolean Reversed state of the ES5 [[Configurable]] attribute.

## Accessor property

Associates a key with one or two accessor functions (get and set) to retrieve or store a value and has the following attributes:

| Attributes of an accessor property |                              |                                                                                                                                                                                   |               |
|------------------------------------|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| Attribute                          | Type                         | Description                                                                                                                                                                       | Default value |
| [[Get]]                            | Function object or undefined | The function is called with an empty argument list and retrieves the property value whenever a get access to the value is performed. See also <a href="#">get</a> .               | undefined     |
| [[Set]]                            | Function object or undefined | The function is called with an argument that contains the assigned value and is executed whenever a specified property is attempted to be changed. See also <a href="#">set</a> . | undefined     |
| [[Enumerable]]                     | Boolean                      | If <code>true</code> , the property will be enumerated in <a href="#">for...in</a> loops.                                                                                         | false         |
| [[Configurable]]                   | Boolean                      | If <code>false</code> , the property can't be deleted and can't be changed to a data property.                                                                                    | false         |

**Note:** Attribute is usually used by JavaScript engine, so you can't directly access it(see more about [Object.defineProperty\(\)](#)).That's why the attribute is put in double square brackets instead of single.

## "Normal" objects, and functions

A JavaScript object is a mapping between keys and values. Keys are strings (or [Symbol](#)s) and values can be anything. This makes objects a natural fit for [hashmaps](#).

Functions are regular objects with the additional capability of being callable.

## Dates

When representing dates, the best choice is to use the built-in [Date utility](#) in JavaScript.

## Indexed collections: Arrays and typed Arrays

[Arrays](#) are regular objects for which there is a particular relationship between integer-key-ed properties and the 'length' property. Additionally, arrays inherit from `Array.prototype` which provides to them a handful of convenient methods to manipulate arrays. For example, [indexOf](#) (searching a value in the array) or [push](#) (adding an element to the array), etc. This makes Arrays a perfect candidate to represent lists or sets.

[Typed Arrays](#) are new to JavaScript with ECMAScript Edition 6 and present an array-like view of an underlying binary data buffer. The following table helps you to find the equivalent C data types:

### TypedArray objects

| Type                              | Value Range                                     | Size in bytes | Description                                                               | Web IDL type        | Equivalent C type |
|-----------------------------------|-------------------------------------------------|---------------|---------------------------------------------------------------------------|---------------------|-------------------|
| <a href="#">Int8Array</a>         | -128 to 127                                     | 1             | 8-bit two's complement signed integer                                     | byte                | int8_t            |
| <a href="#">Uint8Array</a>        | 0 to 255                                        | 1             | 8-bit unsigned integer                                                    | octet               | uint8_t           |
| <a href="#">Uint8ClampedArray</a> | 0 to 255                                        | 1             | 8-bit unsigned integer (clamped)                                          | octet               | uint8_t           |
| <a href="#">Int16Array</a>        | -32768 to 32767                                 | 2             | 16-bit two's complement signed integer                                    | short               | int16_t           |
| <a href="#">Uint16Array</a>       | 0 to 65535                                      | 2             | 16-bit unsigned integer                                                   | unsigned short      | uint16_t          |
| <a href="#">Int32Array</a>        | -2147483648 to 2147483647                       | 4             | 32-bit two's complement signed integer                                    | long                | int32_t           |
| <a href="#">Uint32Array</a>       | 0 to 4294967295                                 | 4             | 32-bit unsigned integer                                                   | unsigned long       | uint32_t          |
| <a href="#">Float32Array</a>      | $1.2 \times 10^{-38}$ to $3.4 \times 10^{38}$   | 4             | 32-bit IEEE floating point number (7 significant digits e.g. 1.1234567)   | unrestricted float  | float             |
| <a href="#">Float64Array</a>      | $5.0 \times 10^{-324}$ to $1.8 \times 10^{308}$ | 8             | 64-bit IEEE floating point number (16 significant digits e.g. 1.123...15) | unrestricted double | double            |

## Keyed collections: Maps, Sets, WeakMaps, WeakSets

These data structures take object references as keys and are introduced in ECMAScript Edition 6. [Set](#) and [WeakSet](#) represent a set of objects, while [Map](#) and [WeakMap](#) associate a value to an object. The difference between Maps and WeakMaps is that in the former, object keys can be enumerated over. This allows garbage collection optimizations in the latter case.

One could implement Maps and Sets in pure ECMAScript 5. However, since objects cannot be compared (in the sense of "less than" for instance), look-up performance would necessarily be linear. Native implementations of them (including WeakMaps) can have look-up performance that is approximately logarithmic to constant time.

Usually, to bind data to a DOM node, one could set properties directly on the object or use `data-*` attributes. This has the downside that the data is available to any script running in the same context. Maps and WeakMaps make it easy to privately bind data to an object.

## Structured data: JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format, derived from JavaScript but used by many programming languages. JSON builds universal data structures. See [JSON](#) and [JSON](#) for more details.

## More objects in the standard library

JavaScript has a standard library of built-in objects. Please have a look at the [reference](#) to find out about more objects.

## Determining types using the `typeof` operator

The `typeof` operator can help you to find the type of your variable. Please read the [reference page](#) for more details and edge cases.

## Equality comparisons and sameness

There are four equality algorithms in ES2015:

- Abstract Equality Comparison (`==`)
- Strict Equality Comparison (`===`): used by `Array.prototype.indexOf`, `Array.prototype.lastIndexOf`, and case-matching

- `SameValueZero`: used by `%TypedArray%` and `ArrayBuffer` constructors, as well as `Map` and `Set` operations, and also `String.prototype.includes` and `Array.prototype.includes` since ES2016
- `SameValue`: used in all other places

JavaScript provides three different value-comparison operations:

- strict equality (or "triple equals" or "identity") using `====`,
- loose equality ("double equals") using `==`,
- and `Object.is` (new in ECMAScript 2015).

The choice of which operation to use depends on what sort of comparison you are looking to perform.

Briefly, double equals will perform a type conversion when comparing two things; triple equals will do the same comparison without type conversion (by simply always returning false if the types differ); and `Object.is` will behave the same way as triple equals, but with special handling for `NaN` and `-0` and `+0` so that the last two are not said to be the same, while `Object.is(NaN, NaN)` will be `true`. (Comparing `NaN` with `NaN` ordinarily—i.e., using either double equals or triple equals—evaluates to `false`, because IEEE 754 says so.) Do note that the distinction between these all have to do with their handling of primitives; none of them compares whether the parameters are conceptually similar in structure. For any non-primitive objects `x` and `y` which have the same structure but are distinct objects themselves, all of the above forms will evaluate to `false`.

## Strict equality using `====`

Strict equality compares two values for equality. Neither value is implicitly converted to some other value before being compared. If the values have different types, the values are considered unequal. Otherwise, if the values have the same type and are not numbers, they're considered equal if they have the same value. Finally, if both values are numbers, they're considered equal if they're both not `NaN` and are the same value, or if one is `+0` and one is `-0`.

```
var num = 0;
var obj = new String('0');
var str = '0';

console.log(num === num); // true
console.log(obj === obj); // true
console.log(str === str); // true

console.log(num === obj); // false
console.log(num === str); // false
console.log(obj === str); // false
console.log(null === undefined); // false
console.log(obj === null); // false
console.log(obj === undefined); // false
```

Strict equality is almost always the correct comparison operation to use. For all values except numbers, it uses the obvious semantics: a value is only equal to itself. For numbers it uses slightly different semantics to gloss over two different edge cases. The first is that floating point zero is either positively or negatively signed. This is useful in representing certain mathematical solutions, but as most situations don't care about the difference between  $+0$  and  $-0$ , strict equality treats them as the same value. The second is that floating point includes the concept of a not-a-number value, `NaN`, to represent the solution to certain ill-defined mathematical problems: negative infinity added to positive infinity, for example. Strict equality treats `NaN` as unequal to every other value -- including itself. (The only case in which `(x !== x)` is `true` is when `x` is `NaN`.)

## Loose equality using `==`

Loose equality compares two values for equality, *after* converting both values to a common type. After conversions (one or both sides may undergo conversions), the final equality comparison is performed exactly as `==` performs it. Loose equality is *symmetric*: `A == B` always has identical semantics to `B == A` for any values of `A` and `B` (except for the order of applied conversions).

The equality comparison is performed as follows for operands of the various types:

|               |           | Operand B |                                  |                                            |                                            |                                            |        |
|---------------|-----------|-----------|----------------------------------|--------------------------------------------|--------------------------------------------|--------------------------------------------|--------|
|               |           | Undefined | Null                             | Number                                     | String                                     | Boolean                                    | Object |
| Operan<br>d A | Undefined | true      | true                             | false                                      | false                                      | false                                      | false  |
|               | Null      | true      | true                             | false                                      | false                                      | false                                      | false  |
| Number        | false     | false     | <code>A === B</code>             | <code>A === ToNumber(B)</code>             | <code>A === ToNumber(B)</code>             | <code>A == ToPrimitive(B)</code>           |        |
| String        | false     | false     | <code>ToNumber(A) === B</code>   | <code>A === B</code>                       | <code>ToNumber(A) === ToNumber(B)</code>   | <code>A == ToPrimitive(B)</code>           |        |
| Boolean       | false     | false     | <code>ToNumber(A) === B</code>   | <code>ToNumber(A) === ToNumber(B)</code>   | <code>A === B</code>                       | <code>ToNumber(A) == ToPrimitive(B)</code> |        |
| Object        | false     | false     | <code>ToPrimitive(A) == B</code> | <code>ToPrimitive(A) == ToNumber(B)</code> | <code>ToPrimitive(A) == ToNumber(B)</code> | <code>A === B</code>                       |        |

In the above table, `ToNumber(A)` attempts to convert its argument to a number before comparison. Its behavior is equivalent to `+A` (the unary `+` operator). `ToPrimitive(A)` attempts to convert its object argument to a primitive value, by attempting to invoke varying sequences of `A.toString` and `A.valueOf` methods on `A`.

Traditionally, and according to ECMAScript, all objects are loosely unequal to `undefined` and `null`. But most browsers permit a very narrow class of objects (specifically, the `document.all` object for any page), in some contexts, to act as if they *emulate* the value `undefined`. Loose equality is one such context: `null == A` and `undefined == A` evaluate to true if, and only if, `A` is an object that *emulates* `undefined`. In all other cases an object is never loosely equal to `undefined` or `null`.

```
var num = 0;
var obj = new String('0');
var str = '0';

console.log(num == num); // true
console.log(obj == obj); // true
console.log(str == str); // true

console.log(num == obj); // true
console.log(num == str); // true
console.log(obj == str); // true
console.log(null == undefined); // true

// both false, except in rare cases
console.log(obj == null);
console.log(obj == undefined);
```

Some developers consider that it is pretty much never a good idea to use loose equality. The result of a comparison using strict equality is easier to predict, and as no type coercion takes place the evaluation may be faster.

## Same-value equality

Same-value equality addresses a final use case: determining whether two values are *functionally identical* in all contexts. (This use case demonstrates an instance of the [Liskov substitution principle](#).) One instance occurs when an attempt is made to mutate an immutable property:

```
// Add an immutable NEGATIVE_ZERO property to the Number constructor.
Object.defineProperty(Number, 'NEGATIVE_ZERO',
 { value: -0, writable: false, configurable: false,
enumerable: false });

function attemptMutation(v) {
 Object.defineProperty(Number, 'NEGATIVE_ZERO', { value: v });
}
```

`Object.defineProperty` will throw an exception when attempting to change an immutable property would actually change it, but it does nothing if no actual change is requested. If `v` is `-0`,

no change has been requested, and no error will be thrown. But if  $v$  is  $+0$ , `Number.NEGATIVE_ZERO` would no longer have its immutable value. Internally, when an immutable property is redefined, the newly-specified value is compared against the current value using same-value equality.

Same-value equality is provided by the `Object.is` method.

## Same-value-zero equality

Similar to same-value equality, but considered  $+0$  and  $-0$  equal.

## Abstract equality, strict equality, and same value in the specification

In ES5, the comparison performed by `==` is described in [Section 11.9.3, The Abstract Equality Algorithm](#). The `===` comparison is [11.9.6, The Strict Equality Algorithm](#). (Go look at these. They're brief and readable. Hint: read the strict equality algorithm first.) ES5 also describes, in [Section 9.12, The SameValue Algorithm](#) for use internally by the JS engine. It's largely the same as the Strict Equality Algorithm, except that 11.9.6.4 and 9.12.4 differ in handling `Numbers`. ES2015 simply proposes to expose this algorithm through `Object.is`.

We can see that with double and triple equals, with the exception of doing a type check upfront in 11.9.6.1, the Strict Equality Algorithm is a subset of the Abstract Equality Algorithm, because 11.9.6.2–7 correspond to 11.9.3.1.a–f.

## A model for understanding equality comparisons?

Prior to ES2015, you might have said of double equals and triple equals that one is an "enhanced" version of the other. For example, someone might say that double equals is an extended version of triple equals, because the former does everything that the latter does, but with type conversion on its operands. E.g., `6 == "6"`. (Alternatively, someone might say that double equals is the baseline, and triple equals is an enhanced version, because it requires the two operands to be the same type, so it adds an extra constraint. Which one is the better model for understanding depends on how you choose to view things.)

However, this way of thinking about the built-in sameness operators is not a model that can be stretched to allow a place for ES2015's `Object.is` on this "spectrum". `Object.is` isn't simply "looser" than double equals or "stricter" than triple equals, nor does it fit somewhere in between (i.e., being both stricter than double equals, but looser than triple equals). We can see from the sameness comparisons table below that this is due to the way that `Object.is` handles `NaN`. Notice that if `Object.is(NaN, NaN)` evaluated to `false`, we *could* say that it fits on the loose/strict spectrum as an even stricter form of triple equals, one that distinguishes between  $-0$  and  $+0$ . The `NaN` handling means this is untrue, however. Unfortunately, `Object.is` simply has

to be thought of in terms of its specific characteristics, rather than its looseness or strictness with regard to the equality operators.

#### Sameness Comparisons

| <b>x</b>                       | <b>y</b>                       | <b>==</b> | <b>===</b> | <b>Object.is</b> |
|--------------------------------|--------------------------------|-----------|------------|------------------|
| undefined                      | undefined                      | true      | true       | true             |
| null                           | null                           | true      | true       | true             |
| true                           | true                           | true      | true       | true             |
| false                          | false                          | true      | true       | true             |
| 'foo'                          | 'foo'                          | true      | true       | true             |
| 0                              | 0                              | true      | true       | true             |
| +0                             | -0                             | true      | true       | false            |
| +0                             | 0                              | true      | true       | true             |
| -0                             | 0                              | true      | true       | false            |
| 0                              | false                          | true      | false      | false            |
| ""                             | false                          | true      | false      | false            |
| ""                             | 0                              | true      | false      | false            |
| '0'                            | 0                              | true      | false      | false            |
| '17'                           | 17                             | true      | false      | false            |
| [1, 2]                         | '1,2'                          | true      | false      | false            |
| <code>new String('foo')</code> | 'foo'                          | true      | false      | false            |
| null                           | undefined                      | true      | false      | false            |
| null                           | false                          | false     | false      | false            |
| undefined                      | false                          | false     | false      | false            |
| { foo: 'bar' }                 | { foo: 'bar' }                 | false     | false      | false            |
| <code>new String('foo')</code> | <code>new String('foo')</code> | false     | false      | false            |
| 0                              | null                           | false     | false      | false            |

## Sameness Comparisons

| <b>x</b> | <b>y</b> | <b>==</b> | <b>===</b> | <b>Object.is</b> |
|----------|----------|-----------|------------|------------------|
| 0        | NaN      | false     | false      | false            |
| 'foo'    | NaN      | false     | false      | false            |
| NaN      | NaN      | false     | false      | true             |

## When to use [Object.is](#) versus triple equals

Aside from the way it treats [NaN](#), generally, the only time [Object.is](#)'s special behavior towards zeros is likely to be of interest is in the pursuit of certain meta-programming schemes, especially regarding property descriptors when it is desirable for your work to mirror some of the characteristics of [Object.defineProperty](#). If your use case does not require this, it is suggested to avoid [Object.is](#) and use [==](#) instead. Even if your requirements involve having comparisons between two [NaN](#) values evaluate to [true](#), generally it is easier to special-case the [NaN](#) checks (using the [isNaN](#) method available from previous versions of ECMAScript) than it is to work out how surrounding computations might affect the sign of any zeros you encounter in your comparison.

Here's an in-exhaustive list of built-in methods and operators that might cause a distinction between  $-0$  and  $+0$  to manifest itself in your code:

### [-](#) (unary negation)

It's obvious that negating  $0$  produces  $-0$ . But the abstraction of an expression can cause  $-0$  to creep in when you don't realize it. For example, consider:

```
let stoppingForce = obj.mass * -obj.velocity;
```

If `obj.velocity` is  $0$  (or computes to  $0$ ), a  $-0$  is introduced at that place and propagates out into `stoppingForce`.

[Math.atan2](#)

[Math.ceil](#)

[Math.pow](#)

[Math.round](#)

It's possible for a  $-0$  to be introduced into an expression as a return value of these methods in some cases, even when no  $-0$  exists as one of the parameters. E.g., using [Math.pow](#) to raise [-Infinity](#) to the power of any negative, odd exponent evaluates to  $-0$ . Refer to the documentation for the individual methods.

[Math.floor](#)

[Math.max](#)

[Math.min](#)

[Math.sin](#)

[Math.sqrt](#)

[Math.tan](#)

It's possible to get a `-0` return value out of these methods in some cases where a `-0` exists as one of the parameters. E.g., `Math.min(-0, +0)` evaluates to `-0`. Refer to the documentation for the individual methods.

`~`

`<<`

`>>`

Each of these operators uses the `ToInt32` algorithm internally. Since there is only one representation for `0` in the internal 32-bit integer type, `-0` will not survive a round trip after an inverse operation. E.g., both `Object.is(~(-0), -0)` and `Object.is(-0 << 2 >> 2, -0)` evaluate to `false`.

Relying on [Object.is](#) when the signedness of zeros is not taken into account can be hazardous. Of course, when the intent is to distinguish between `-0` and `+0`, it does exactly what's desired.

## Caveat: [Object.is](#) and NaN

The [Object.is](#) specification treats all instances of [NaN](#) as same, but since typed arrays are present, we can distinct instances, which behave not identical in all contexts. Example:

```
var f2b = x => new Uint8Array(new Float64Array([x]).buffer);
var b2f = x => new Float64Array(x.buffer)[0];
var n = f2b(NaN);
n[0] = 1;
var nan2 = b2f(n);
nan2
> NaN
Object.is(nan2, NaN)
> true
f2b(NaN)
> Uint8Array(8) [0, 0, 0, 0, 0, 0, 248, 127]
f2b(nan2)
> Uint8Array(8) [1, 0, 0, 0, 0, 0, 248, 127]
```

## Inheritance and the prototype chain

JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not provide a `class` implementation per se (the `class` keyword is introduced in ES2015, but is syntactical sugar, JavaScript remains prototype-based).

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype, and acts as the final link in this **prototype chain**.

Nearly all objects in JavaScript are instances of [Object](#) which sits on the top of a prototype chain.

While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model.

## Inheritance with the prototype chain

### Inheriting properties

JavaScript objects are dynamic "bags" of properties (referred to as **own properties**). JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

Following the ECMAScript standard, the notation `someObject.[[Prototype]]` is used to designate the prototype of `someObject`. Since ECMAScript 2015, the `[[Prototype]]` is accessed using the accessors [Object.getPrototypeOf\(\)](#) and [Object.setPrototypeOf\(\)](#). This is equivalent to the JavaScript property `__proto__` which is non-standard but de-facto implemented by many browsers.

It should not be confused with the `func.prototype` property of functions, which instead specifies the `[ [ Prototype ] ]` to be assigned to all *instances* of objects created by the given function when used as a constructor. The `Object.prototype` property represents the [Object](#) prototype object.

Here is what happens when trying to access a property:

```
// Let's create an object o from function f with its own properties a and b:
let f = function () {
 this.a = 1;
 this.b = 2;
}
let o = new f(); // {a: 1, b: 2}

//add properties in f function's prototype
f.prototype.b = 3;
f.prototype.c = 4;

// do not set the prototype f.prototype = {b:3,c:4}; this will break the
prototype chain
// o.[[Prototype]] has properties b and c:
// Finally, o.[[Prototype]].[[Prototype]] is null.
// This is the end of the prototype chain, as null,
// by definition, has no [[Prototype]].
// Thus, the full prototype chain looks like:
// {a: 1, b: 2} ---> {b: 3, c: 4} ---> null

console.log(o.a); // 1
// Is there an 'a' own property on o? Yes, and its value is 1.

console.log(o.b); // 2
// Is there a 'b' own property on o? Yes, and its value is 2.
// The prototype also has a 'b' property, but it's not visited.
// This is called "property shadowing."

console.log(o.c); // 4
// Is there a 'c' own property on o? No, check its prototype.
// Is there a 'c' own property on o.[[Prototype]]? Yes, its value is 4.

console.log(o.d); // undefined
// Is there a 'd' own property on o? No, check its prototype.
// Is there a 'd' own property on o.[[Prototype]]? No, check its prototype.
// o.[[Prototype]].[[Prototype]] is null, stop searching,
// no property found, return undefined.
```

## [Code Link](#)

Setting a property to an object creates an own property. The only exception to the getting and setting behavior rules is when there is an inherited property with a [getter or a setter](#).

## Inheriting "methods"

JavaScript does not have "methods" in the form that class-based languages define them. In JavaScript, any function can be added to an object in the form of a property. An inherited function acts just as any other property, including property shadowing as shown above (in this case, a form of *method overriding*).

When an inherited function is executed, the value of `this` points to the inheriting object, not to the prototype object where the function is an own property.

```
var o = {
 a: 2,
 m: function() {
 return this.a + 1;
 }
};

console.log(o.m()); // 3
// When calling o.m in this case, 'this' refers to o

var p = Object.create(o);
// p is an object that inherits from o

p.a = 4; // creates a property 'a' on p
console.log(p.m()); // 5
// when p.m is called, 'this' refers to p.
// So when p inherits the function m of o,
// 'this.a' means p.a, the property 'a' of p
```

## Different ways to create objects and the resulting prototype chain

### Objects created with syntax constructs

```
var o = {a: 1};

// The newly created object o has Object.prototype as its [[Prototype]]
// o has no own property named 'hasOwnProperty'
// hasOwnProperty is an own property of Object.prototype.
// So o inherits hasOwnProperty from Object.prototype
// Object.prototype has null as its prototype.
// o ---> Object.prototype ---> null

var b = ['yo', 'whadup', '?'];

// Arrays inherit from Array.prototype
// (which has methods indexOf, forEach, etc.)
// The prototype chain looks like:
// b ---> Array.prototype ---> Object.prototype ---> null

function f() {
 return 2;
}

// Functions inherit from Function.prototype
// (which has methods call, bind, etc.)
// f ---> Function.prototype ---> Object.prototype ---> null
```

## With a constructor

A "constructor" in JavaScript is "just" a function that happens to be called with the [new operator](#).

```
function Graph() {
 this.vertices = [];
 this.edges = [];
}

Graph.prototype = {
 addVertex: function(v) {
 this.vertices.push(v);
 }
};

var g = new Graph();
// g is an object with own properties 'vertices' and 'edges'.
// g.[[Prototype]] is the value of Graph.prototype when new Graph() is
executed.
```

## With `Object.create`

ECMAScript 5 introduced a new method: [Object.create\(\)](#). Calling this method creates a new object. The prototype of this object is the first argument of the function:

```
var a = {a: 1};
// a ---> Object.prototype ---> null

var b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
console.log(b.a); // 1 (inherited)

var c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null

var d = Object.create(null);
// d ---> null
console.log(d.hasOwnProperty());
// undefined, because d doesn't inherit from Object.prototype
```

## With the `class` keyword

ECMAScript 2015 introduced a new set of keywords implementing [classes](#). Although these constructs look like those familiar to developers of class-based languages, they are not the same. JavaScript remains prototype-based. The new keywords include [class](#), [constructor](#), [static](#), [extends](#), and [super](#).

```
'use strict';

class Polygon {
 constructor(height, width) {
 this.height = height;
```

```

 this.width = width;
 }
}

class Square extends Polygon {
 constructor(sideLength) {
 super(sideLength, sideLength);
 }
 get area() {
 return this.height * this.width;
 }
 set sideLength(newLength) {
 this.height = newLength;
 this.width = newLength;
 }
}

var square = new Square(2);

```

## Performance

The lookup time for properties that are high up on the prototype chain can have a negative impact on the performance, and this may be significant in the code where performance is critical. Additionally, trying to access nonexistent properties will always traverse the full prototype chain.

Also, when iterating over the properties of an object, **every** enumerable property that is on the prototype chain will be enumerated. To check whether an object has a property defined on *itself* and not somewhere on its prototype chain, it is necessary to use the [hasOwnProperty](#) method which all objects inherit from `Object.prototype`. To give you a concrete example, let's take the above graph example code to illustrate it:

```

console.log(g.hasOwnProperty('vertices'));
// true

console.log(g.hasOwnProperty('nope'));
// false

console.log(g.hasOwnProperty('addVertex'));
// false

console.log(g.__proto__.hasOwnProperty('addVertex'));
// true

```

[hasOwnProperty](#) is the only thing in JavaScript which deals with properties and does **not** traverse the prototype chain.

Note: It is **not** enough to check whether a property is [undefined](#). The property might very well exist, but its value just happens to be set to `undefined`.

## Bad practice: Extension of native prototypes

One misfeature that is often used is to extend `Object.prototype` or one of the other built-in prototypes.

This technique is called monkey patching and breaks *encapsulation*. While used by popular frameworks such as Prototype.js, there is still no good reason for cluttering built-in types with additional *non-standard* functionality.

The **only** good reason for extending a built-in prototype is to backport the features of newer JavaScript engines, like `Array.forEach`.

## Example

B shall inherit from A:

```
function A(a) {
 this.varA = a;
}

// What is the purpose of including varA in the prototype when
A.prototype.varA will always be shadowed by
// this.varA, given the definition of function A above?
A.prototype = {
 varA: null, // Shouldn't we strike varA from the prototype as doing
nothing?
 // perhaps intended as an optimization to allocate space in hidden
classes?
 // https://developers.google.com/speed/articles/optimizing-
javascript#Initializing-instance-variables
 // would be valid if varA wasn't being initialized uniquely for each
instance
 doSomething: function() {
 // ...
 }
};

function B(a, b) {
 A.call(this, a);
 this.varB = b;
}
B.prototype = Object.create(A.prototype, {
 varB: {
 value: null,
 enumerable: true,
 configurable: true,
 writable: true
 },
 doSomething: {
 value: function() { // override
 A.prototype.doSomething.apply(this, arguments); // call super
 // ...
 }
 }
});
```

```

 },
 enumerable: true,
 configurable: true,
 writable: true
 }
});
B.prototype.constructor = B;

var b = new B();
b.doSomething();

```

The important parts are:

- Types are defined in `.prototype`.
- You use `Object.create()` to inherit.

### **prototype and `Object.getPrototypeOf`**

JavaScript is a bit confusing for developers coming from Java or C++, as it's all dynamic, all runtime, and it has no classes at all. It's all just instances (objects). Even the "classes" we simulate are just a function object.

You probably already noticed that our function `A` has a special property called `prototype`. This special property works with the JavaScript `new` operator. The reference to the `prototype` object is copied to the internal `[[Prototype]]` property of the new instance. For example, when you do `var a1 = new A()`, JavaScript (after creating the object in memory and before running function `A()` with `this` defined to it) sets `a1.[[Prototype]] = A.prototype`. When you then access properties of the instance, JavaScript first checks whether they exist on that object directly, and if not, it looks in `[[Prototype]]`. This means that all the stuff you define in `prototype` is effectively shared by all instances, and you can even later change parts of `prototype` and have the changes appear in all existing instances, if you wanted to.

If, in the example above, you do `var a1 = new A(); var a2 = new A();` then `a1.doSomething` would actually refer to `Object.getPrototypeOf(a1).doSomething`, which is the same as the `A.prototype.doSomething` you defined, i.e.  
`Object.getPrototypeOf(a1).doSomething ==`  
`Object.getPrototypeOf(a2).doSomething == A.prototype.doSomething.`

In short, `prototype` is for types, while `Object.getPrototypeOf()` is the same for instances.

`[[Prototype]]` is looked at *recursively*, i.e. `a1.doSomething`, `Object.getPrototypeOf(a1).doSomething`, `Object.getPrototypeOf(Object.getPrototypeOf(a1)).doSomething` etc., until it's found or `Object.getPrototypeOf` returns `null`.

So, when you call

```
var o = new Foo();
```

JavaScript actually just does

```
var o = new Object();
o.[[Prototype]] = Foo.prototype;
Foo.call(o);
```

(or something like that) and when you later do

```
o.someProp;
```

it checks whether `o` has a property `someProp`. If not, it checks `Object.getPrototypeOf(o).someProp`, and if that doesn't exist it checks `Object.getPrototypeOf(Object.getPrototypeOf(o)).someProp`, and so on.

## In conclusion

It is **essential** to understand the prototypal inheritance model before writing complex code that makes use of it. Also, be aware of the length of the prototype chains in your code and break them up if necessary to avoid possible performance problems. Further, the native prototypes should **never** be extended unless it is for the sake of compatibility with newer JavaScript features.

## Strict mode

[ECMAScript 5](#)'s strict mode is a way to *opt in* to a restricted variant of JavaScript. Strict mode isn't just a subset: it *intentionally* has different semantics from normal code. Browsers not supporting strict mode will run strict mode code with different behavior from browsers that do, so don't rely on strict mode without feature-testing for support for the relevant aspects of strict mode. Strict mode code and non-strict mode code can coexist, so scripts can opt into strict mode incrementally.

Strict mode makes several changes to normal JavaScript semantics:

1. Eliminates some JavaScript silent errors by changing them to throw errors.
2. Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
3. Prohibits some syntax likely to be defined in future versions of ECMAScript.

See [transitioning to strict mode](#), if you want to change your code to work in the restricted variant of JavaScript.

Sometimes, you'll see the default, non-strict, mode referred to as "sloppy mode". This isn't an official term, but be aware of it, just in case.

## Invoking strict mode

Strict mode applies to *entire scripts* or to *individual functions*. It doesn't apply to block statements enclosed in {} braces; attempting to apply it to such contexts does nothing. eval code, Function code, event handler attributes, strings passed to [WindowTimers.setTimeout\(\)](#), and the like are entire scripts, and invoking strict mode in them works as expected.

## Strict mode for scripts

To invoke strict mode for an entire script, put the *exact* statement "use strict"; (or 'use strict';) before any other statements.

```
// Whole-script strict mode syntax
'use strict';
var v = "Hi! I'm a strict mode script!";
```

This syntax has a trap that has [already bitten a major site](#): it isn't possible to blindly concatenate non-conflicting scripts. Consider concatenating a strict mode script with a non-strict mode script: the entire concatenation looks strict! The inverse is also true: non-strict plus strict looks non-strict. Concatenation of strict mode scripts with each other is fine, and concatenation of non-strict mode scripts is fine. Only concatenating strict and non-strict scripts is problematic. It is thus recommended that you enable strict mode on a function-by-function basis (at least during the transition period).

You can also take the approach of wrapping the entire contents of a script in a function and having that outer function use strict mode. This eliminates the concatenation problem but it means that you have to explicitly export any global variables out of the function scope.

## Strict mode for functions

Likewise, to invoke strict mode for a function, put the *exact* statement "use strict"; (or 'use strict';) in the function's body before any other statements.

```
function strict() {
 // Function-level strict mode syntax
 'use strict';
 function nested() { return 'And so am I!'; }
 return "Hi! I'm a strict mode function! " + nested();
}
function notStrict() { return "I'm not strict."; }
```

# Changes in strict mode

Strict mode changes both syntax and runtime behavior. Changes generally fall into these categories: changes converting mistakes into errors (as syntax errors or at runtime), changes simplifying how the particular variable for a given use of a name is computed, changes simplifying eval and arguments, changes making it easier to write "secure" JavaScript, and changes anticipating future ECMAScript evolution.

## Converting mistakes into errors

Strict mode changes some previously-accepted mistakes into errors. JavaScript was designed to be easy for novice developers, and sometimes it gives operations which should be errors non-error semantics. Sometimes this fixes the immediate problem, but sometimes this creates worse problems in the future. Strict mode treats these mistakes as errors so that they're discovered and promptly fixed.

First, strict mode makes it impossible to accidentally create global variables. In normal JavaScript mistyping a variable in an assignment creates a new property on the global object and continues to "work" (although future failure is possible: likely, in modern JavaScript).

Assignments which would accidentally create global variables instead throw in strict mode:

```
'use strict';
// Assuming a global variable mistypedVariable exists
mistypeVariable = 17; // this line throws a ReferenceError due to the
// misspelling of variable
```

Second, strict mode makes assignments which would otherwise silently fail to throw an exception. For example, `NaN` is a non-writable global variable. In normal code assigning to `NaN` does nothing; the developer receives no failure feedback. In strict mode assigning to `NaN` throws an exception. Any assignment that silently fails in normal code (assignment to a non-writable global or property, assignment to a getter-only property, assignment to a new property on a [non-extensible](#) object) will throw in strict mode:

```
'use strict';

// Assignment to a non-writable global
var undefined = 5; // throws a TypeError
var Infinity = 5; // throws a TypeError

// Assignment to a non-writable property
var obj1 = {};
Object.defineProperty(obj1, 'x', { value: 42, writable: false });
obj1.x = 9; // throws a TypeError

// Assignment to a getter-only property
var obj2 = { get x() { return 17; } };
obj2.x = 5; // throws a TypeError

// Assignment to a new property on a non-extensible object
var fixed = {};
Object.preventExtensions(fixed);
fixed.newProp = 'ohai'; // throws a TypeError
```

Third, strict mode makes attempts to delete undeletable properties throw (where before the attempt would simply have no effect):

```
'use strict';
delete Object.prototype; // throws a TypeError
```

Fourth, strict mode prior to Gecko 34 requires that all properties named in an object literal be unique. The normal code may duplicate property names, with the last one determining the property's value. But since only the last one does anything, the duplication is simply a vector for bugs, if the code is modified to change the property value other than by changing the last instance. Duplicate property names are a syntax error in strict mode:

This is no longer the case in ECMAScript 2015 ([bug 1041128](#)).

```
'use strict';
var o = { p: 1, p: 2 }; // !!! syntax error
```

Fifth, strict mode requires that function parameter names be unique. In normal code the last duplicated argument hides previous identically-named arguments. Those previous arguments remain available through `arguments[i]`, so they're not completely inaccessible. Still, this hiding makes little sense and is probably undesirable (it might hide a typo, for example), so in strict mode duplicate argument names are a syntax error:

```
function sum(a, a, c) { // !!! syntax error
 'use strict';
 return a + a + c; // wrong if this code ran
}
```

Sixth, a strict mode in ECMAScript 5 forbids octal syntax. The octal syntax isn't part of ECMAScript 5, but it's supported in all browsers by prefixing the octal number with a zero: `0644` === `420` and `"\045"` === `"%"`. In ECMAScript 2015 Octal number is supported by prefixing a number with `"0o"`. i.e.

```
var a = 0o10; // ES2015: Octal
```

Novice developers sometimes believe a leading zero prefix has no semantic meaning, so they use it as an alignment device — but this changes the number's meaning! A leading zero syntax for the octals is rarely useful and can be mistakenly used, so strict mode makes it a syntax error:

```
'use strict';
var sum = 015 + // !!! syntax error
 197 +
 142;

var sumWithOctal = 0o10 + 8;
console.log(sumWithOctal); // 16
```

Seventh, strict mode in ECMAScript 2015 forbids setting properties on [primitive](#) values. Without strict mode, setting properties is simply ignored (no-op), with strict mode, however, a [TypeError](#) is thrown.

```
(function() {
 'use strict';

 false.true = '';
 // TypeError
 (14).sailing = 'home';
 // TypeError
```

```
'with'.you = 'far away'; // TypeError
})();
```

## Simplifying variable uses

Strict mode simplifies how variable names map to particular variable definitions in the code. Many compiler optimizations rely on the ability to say that variable *X* is stored in *that* location: this is critical to fully optimizing JavaScript code. JavaScript sometimes makes this basic mapping of name to variable definition in the code impossible to perform until runtime. Strict mode removes most cases where this happens, so the compiler can better optimize strict mode code.

First, strict mode prohibits `with`. The problem with `with` is that any name inside the block might map either to a property of the object passed to it, or to a variable in surrounding (or even global) scope, at runtime: it's impossible to know which beforehand. Strict mode makes `with` a syntax error, so there's no chance for a name in a `with` to refer to an unknown location at runtime:

```
'use strict';
var x = 17;
with (obj) { // !!! syntax error
 // If this weren't strict mode, would this be var x, or
 // would it instead be obj.x? It's impossible in general
 // to say without running the code, so the name can't be
 // optimized.
 x;
}
```

The simple alternative of assigning the object to a short name variable, then accessing the corresponding property on that variable, stands ready to replace `with`.

Second, [eval of strict mode code does not introduce new variables into the surrounding scope](#).

In normal code `eval("var x;")` introduces a variable `x` into the surrounding function or the global scope. This means that, in general, in a function containing a call to `eval` every name not referring to an argument or local variable must be mapped to a particular definition at runtime (because that `eval` might have introduced a new variable that would hide the outer variable). In strict mode `eval` creates variables only for the code being evaluated, so `eval` can't affect whether a name refers to an outer variable or some local variable:

```
var x = 17;
var evalX = eval("'use strict'; var x = 42; x;");
console.assert(x === 17);
console.assert(evalX === 42);
```

If the function `eval` is invoked by an expression of the form `eval(...)` in strict mode code, the code will be evaluated as strict mode code. The code may explicitly invoke strict mode, but it's unnecessary to do so.

```
function strict1(str) {
```

```

'use strict';
return eval(str); // str will be treated as strict mode code
}
function strict2(f, str) {
 'use strict';
 return f(str); // not eval(...): str is strict if and only
 // if it invokes strict mode
}
function nonstrict(str) {
 return eval(str); // str is strict if and only
 // if it invokes strict mode
}

strict1("'Strict mode code!'");
strict1("'use strict'; 'Strict mode code!'");
strict2(eval, "'Non-strict code.'");
strict2(eval, "'use strict'; 'Strict mode code!'");
nonstrict("'Non-strict code.'");
nonstrict("'use strict'; 'Strict mode code!'");

```

Thus names in strict mode `eval` code behave identically to names in strict mode code not being evaluated as the result of `eval`.

Third, strict mode forbids deleting plain names. `delete name` in strict mode is a syntax error:

```

'use strict';

var x;
delete x; // !!! syntax error

eval('var y; delete y;'); // !!! syntax error

```

### Making `eval` and `arguments` simpler

Strict mode makes `arguments` and `eval` less bizarrely magical. Both involve a considerable amount of magical behavior in normal code: `eval` to add or remove bindings and to change binding values, and `arguments` by its indexed properties aliasing named arguments. Strict mode makes great strides toward treating `eval` and `arguments` as keywords, although full fixes will not come until a future edition of ECMAScript.

First, the names `eval` and `arguments` can't be bound or assigned in language syntax. All these attempts to do so are syntax errors:

```

'use strict';
eval = 17;
arguments++;
++eval;
var obj = { set p(arguments) { } };
var eval;
try { } catch (arguments) { }
function x(eval) { }
function arguments() { }

```

```
var y = function eval() { };
var f = new Function('arguments', "'use strict'; return 17;");
```

Second, strict mode code doesn't alias properties of `arguments` objects created within it. In normal code within a function whose first argument is `arg`, setting `arg` also sets `arguments[0]`, and vice versa (unless no arguments were provided or `arguments[0]` is deleted). `arguments` objects for strict mode functions store the original arguments when the function was invoked. `arguments[i]` does not track the value of the corresponding named argument, nor does a named argument track the value in the corresponding `arguments[i]`.

```
function f(a) {
 'use strict';
 a = 42;
 return [a, arguments[0]];
}
var pair = f(17);
console.assert(pair[0] === 42);
console.assert(pair[1] === 17);
```

Third, `arguments.callee` is no longer supported. In normal code `arguments.callee` refers to the enclosing function. This use case is weak: simply name the enclosing function! Moreover, `arguments.callee` substantially hinders optimizations like inlining functions, because it must be made possible to provide a reference to the un-inlined function if `arguments.callee` is accessed. `arguments.callee` for strict mode functions is a non-deletable property which throws when set or retrieved:

```
'use strict';
var f = function() { return arguments.callee; };
f(); // throws a TypeError
```

## "Securing" JavaScript

Strict mode makes it easier to write "secure" JavaScript. Some websites now provide ways for users to write JavaScript which will be run by the website *on behalf of other users*. JavaScript in browsers can access the user's private information, so such JavaScript must be partially transformed before it is run, to censor access to forbidden functionality. JavaScript's flexibility makes it effectively impossible to do this without many runtime checks. Certain language functions are so pervasive that performing runtime checks has a considerable performance cost. A few strict mode tweaks, plus requiring that user-submitted JavaScript be strict mode code and that it be invoked in a certain manner, substantially reduce the need for those runtime checks.

First, the value passed as `this` to a function in strict mode is not forced into being an object (a.k.a. "boxed"). For a normal function, `this` is always an object: either the provided object if called with an object-valued `this`; the value, boxed, if called with a Boolean, string, or number `this`; or the global object if called with an `undefined` or `null` `this`. (Use `call`, `apply`, or `bind` to specify a particular `this`.) Not only is automatic boxing a performance cost, but exposing the global object in browsers is a security hazard because the global object provides access to

functionality that "secure" JavaScript environments must restrict. Thus for a strict mode function, the specified `this` is not boxed into an object, and if unspecified, `this` will be `undefined`:

```
'use strict';
function fun() { return this; }
console.assert(fun() === undefined);
console.assert(fun.call(2) === 2);
console.assert(fun.apply(null) === null);
console.assert(fun.call(undefined) === undefined);
console.assert(fun.bind(true) () === true);
```

That means, among other things, that in browsers it's no longer possible to reference the `window` object through `this` inside a strict mode function.

Second, in strict mode it's no longer possible to "walk" the JavaScript stack via commonly-implemented extensions to ECMAScript. In normal code with these extensions, when a function `fun` is in the middle of being called, `fun.caller` is the function that most recently called `fun`, and `fun.arguments` is the `arguments` for that invocation of `fun`. Both extensions are problematic for "secure" JavaScript because they allow "secured" code to access "privileged" functions and their (potentially unsecured) arguments. If `fun` is in strict mode, both `fun.caller` and `fun.arguments` are non-deletable properties which throw when set or retrieved:

```
function restricted() {
 'use strict';
 restricted.caller; // throws a TypeError
 restricted.arguments; // throws a TypeError
}
function privilegedInvoker() {
 return restricted();
}
privilegedInvoker();
```

Third, `arguments` for strict mode functions no longer provide access to the corresponding function call's variables. In some old ECMAScript implementations `arguments.caller` was an object whose properties aliased variables in that function. This is a [security hazard](#) because it breaks the ability to hide privileged values via function abstraction; it also precludes most optimizations. For these reasons no recent browsers implement it. Yet because of its historical functionality, `arguments.caller` for a strict mode function is also a non-deletable property which throws when set or retrieved:

```
'use strict';
function fun(a, b) {
 'use strict';
 var v = 12;
 return arguments.caller; // throws a TypeError
}
fun(1, 2); // doesn't expose v (or a or b)
```

## Paving the way for future ECMAScript versions

Future ECMAScript versions will likely introduce new syntax, and strict mode in ECMAScript 5 applies some restrictions to ease the transition. It will be easier to make some changes if the foundations of those changes are prohibited in strict mode.

First, in strict mode, a short list of identifiers become reserved keywords. These words are `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, and `yield`. In strict mode, then, you can't name or use variables or arguments with these names.

```
function package(protected) { // !!!
 'use strict';
 var implements; // !!!

 interface: // !!!
 while (true) {
 break interface; // !!!
 }

 function private() { } // !!!
}
function fun(static) { 'use strict'; } // !!!
```

Two Mozilla-specific caveats: First, if your code is JavaScript 1.7 or greater (for example in chrome code or when using the right `<script type="">`) and is strict mode code, `let` and `yield` have the functionality they've had since those keywords were first introduced. But strict mode code on the web, loaded with `<script src="">` or `<script>...</script>`, won't be able to use `let/yield` as identifiers. Second, while ES5 unconditionally reserves the words `class`, `enum`, `export`, `extends`, `import`, and `super`, before Firefox 5 Mozilla reserved them only in strict mode.

Second, [strict mode prohibits function statements, not at the top level of a script or function](#). In normal mode in browsers, function statements are permitted "everywhere". *This is not part of ES5 (or even ES3)!* It's an extension with incompatible semantics in different browsers. Note that function statements outside top level are permitted in ES2015.

```
'use strict';
if (true) {
 function f() { } // !!! syntax error
 f();
}

for (var i = 0; i < 5; i++) {
 function f2() { } // !!! syntax error
 f2();
}

function baz() { // kosher
 function eit() { } // also kosher
}
```

This prohibition isn't strict mode proper because such function statements are an extension of basic ES5. But it is the recommendation of the ECMAScript committee, and browsers will implement it.

## Strict mode in browsers

The major browsers now implement strict mode. However, don't blindly depend on it since there still are numerous [Browser versions used in the wild that only have partial support for strict mode](#) or do not support it at all (e.g. Internet Explorer below version 10!). *Strict mode changes semantics*. Relying on those changes will cause mistakes and errors in browsers which don't implement strict mode. Exercise caution in using strict mode, and back up reliance on strict mode with feature tests that check whether relevant parts of strict mode are implemented. Finally, make sure to *test your code in browsers that do and don't support strict mode*. If you test only in browsers that don't support strict mode, you're very likely to have problems in browsers that do, and vice versa.

## JavaScript typed arrays

JavaScript typed arrays are array-like objects and provide a mechanism for accessing raw binary data. As you may already know, `Array` objects grow and shrink dynamically and can have any JavaScript value. JavaScript engines perform optimizations so that these arrays are fast. However, as web applications become more and more powerful, adding features such as audio and video manipulation, access to raw data using WebSockets, and so forth, it has become clear that there are times when it would be helpful for JavaScript code to be able to quickly and easily manipulate raw binary data in typed arrays.

However, typed arrays are not to be confused with normal arrays, as calling `Array.isArray()` on a typed array returns `false`. Moreover, not all methods available for normal arrays are supported by typed arrays (e.g. `push` and `pop`).

## Buffers and views: typed array architecture

To achieve maximum flexibility and efficiency, JavaScript typed arrays split the implementation into **buffers** and **views**. A buffer (implemented by the `ArrayBuffer` object) is an object representing a chunk of data; it has no format to speak of and offers no mechanism for accessing its contents. In order to access the memory contained in a buffer, you need to use a view. A view provides a context — that is, a data type, starting offset, and the number of elements — that turns the data into a typed array.

### ArrayBuffer (16 bytes)

|                     |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---------------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| <b>Uint8Array</b>   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| <b>Uint16Array</b>  | 0 |   | 1 |   | 2 |   | 3 |   | 4 |   | 5  |    | 6  |    | 7  |    |
| <b>Uint32Array</b>  |   | 0 |   |   | 1 |   |   |   | 2 |   |    |    | 3  |    |    |    |
| <b>Float64Array</b> |   |   | 0 |   |   |   |   |   |   |   |    | 1  |    |    |    |    |

## ArrayBuffer

The `ArrayBuffer` is a data type that is used to represent a generic, fixed-length binary data buffer. You can't directly manipulate the contents of an `ArrayBuffer`; instead, you create a typed array view or a `DataView` which represents the buffer in a specific format, and use that to read and write the contents of the buffer.

### Typed array views

Typed array views have self-descriptive names and provide views for all the usual numeric types like `Int8`, `Uint32`, `Float64` and so forth. There is one special typed array view, the `Uint8ClampedArray`. It clamps the values between 0 and 255. This is useful for [Canvas data processing](#), for example.

| Type                           | Value Range     | Size in bytes | Description                            | Web IDL type | Equivalent C type    |
|--------------------------------|-----------------|---------------|----------------------------------------|--------------|----------------------|
| <code>Int8Array</code>         | -128 to 127     | 1             | 8-bit two's complement signed integer  | byte         | <code>int8_t</code>  |
| <code>Uint8Array</code>        | 0 to 255        | 1             | 8-bit unsigned integer                 | octet        | <code>uint8_t</code> |
| <code>Uint8ClampedArray</code> | 0 to 255        | 1             | 8-bit unsigned integer (clamped)       | octet        | <code>uint8_t</code> |
| <code>Int16Array</code>        | -32768 to 32767 | 2             | 16-bit two's complement signed integer | short        | <code>int16_t</code> |

|              |                                                 |   |                                                                           |                     |          |
|--------------|-------------------------------------------------|---|---------------------------------------------------------------------------|---------------------|----------|
| Uint16Array  | 0 to 65535                                      | 2 | 16-bit unsigned integer                                                   | unsigned short      | uint16_t |
| Int32Array   | -2147483648 to 2147483647                       | 4 | 32-bit two's complement signed integer                                    | long                | int32_t  |
| Uint32Array  | 0 to 4294967295                                 | 4 | 32-bit unsigned integer                                                   | unsigned long       | uint32_t |
| Float32Array | $1.2 \times 10^{-38}$ to $3.4 \times 10^{38}$   | 4 | 32-bit IEEE floating point number (7 significant digits e.g. 1.1234567)   | unrestricted float  | float    |
| Float64Array | $5.0 \times 10^{-324}$ to $1.8 \times 10^{308}$ | 8 | 64-bit IEEE floating point number (16 significant digits e.g. 1.123...15) | unrestricted double | double   |

## DataView

The `DataView` is a low-level interface that provides a getter/setter API to read and write arbitrary data to the buffer. This is useful when dealing with different types of data, for example. Typed array views are in the native byte-order (see [Endianness](#)) of your platform. With a `DataView` you are able to control the byte-order. It is big-endian by default and can be set to little-endian in the getter/setter methods.

## Web APIs using typed arrays

`FileReader.prototype.readAsArrayBuffer()`

The `FileReader.prototype.readAsArrayBuffer()` method starts reading the contents of the specified `Blob` or `File`.

`XMLHttpRequest.prototype.send()`

`XMLHttpRequest` instances' `send()` method now supports typed arrays and `ArrayBuffer` objects as argument.

[`ImageData.data`](#)

Is a `Uint8ClampedArray` representing a one-dimensional array containing the data in the RGBA order, with integer values between 0 and 255 inclusive.

## Examples

## Using views with buffers

First of all, we will need to create a buffer, here with a fixed length of 16-bytes:

```
var buffer = new ArrayBuffer(16);
```

At this point, we have a chunk of memory whose bytes are all pre-initialized to 0. There's not a lot we can do with it, though. We can confirm that it is indeed 16 bytes long, and that's about it:

```
if (buffer.byteLength === 16) {
 console.log("Yes, it's 16 bytes.");
} else {
 console.log("Oh no, it's the wrong size!");
}
```

Before we can really work with this buffer, we need to create a view. Let's create a view that treats the data in the buffer as an array of 32-bit signed integers:

```
var int32View = new Int32Array(buffer);
```

Now we can access the fields in the array just like a normal array:

```
for (var i = 0; i < int32View.length; i++) {
 int32View[i] = i * 2;
}
```

This fills out the 4 entries in the array (4 entries at 4 bytes each makes 16 total bytes) with the values 0, 2, 4, and 6.

## Multiple views on the same data

Things start to get really interesting when you consider that you can create multiple views onto the same data. For example, given the code above, we can continue like this:

```
var int16View = new Int16Array(buffer);

for (var i = 0; i < int16View.length; i++) {
 console.log('Entry ' + i + ': ' + int16View[i]);
}
```

Here we create a 16-bit integer view that shares the same buffer as the existing 32-bit view and we output all the values in the buffer as 16-bit integers. Now we get the output 0, 0, 2, 0, 4, 0, 6, 0.

You can go a step farther, though. Consider this:

```
int16View[0] = 32;
console.log('Entry 0 in the 32-bit array is now ' + int32View[0]);
```

The output from this is "Entry 0 in the 32-bit array is now 32". In other words, the two arrays are indeed simply viewed on the same data buffer, treating it as different formats. You can do this with any [view types](#).

## Working with complex data structures

By combining a single buffer with multiple views of different types, starting at different offsets into the buffer, you can interact with data objects containing multiple data types. This lets you, for example, interact with complex data structures from [WebGL](#), data files, or C structures you need to use while using [js-ctypes](#).

Consider this C structure:

```
struct someStruct {
 unsigned long id;
 char username[16];
 float amountDue;
};
```

You can access a buffer containing data in this format like this:

```
var buffer = new ArrayBuffer(24);

// ... read the data into the buffer ...

var idView = new Uint32Array(buffer, 0, 1);
var usernameView = new Uint8Array(buffer, 4, 16);
var amountDueView = new Float32Array(buffer, 20, 1);
```

Then you can access, for example, the amount due with `amountDueView[0]`.

**Note:** The [data structure alignment](#) in a C structure is platform-dependent. Take precautions and considerations for these padding differences.

## Conversion to normal arrays

After processing a typed array, it is sometimes useful to convert it back to a normal array in order to benefit from the `Array` prototype. This can be done using `Array.from`, or using the following code where `Array.from` is unsupported.

```
var typedArray = new Uint8Array([1, 2, 3, 4]),
 normalArray = Array.prototype.slice.call(typedArray);
normalArray.length === 4;
normalArray.constructor === Array;
```

# Memory Management

## Introduction

Low-level languages, like C, have low-level memory management primitives like `malloc()` and `free()`. On the other hand, JavaScript values are allocated when things (objects, strings, etc.) are created and "automatically" freed when they are not used anymore. The latter process is called *garbage collection*. This "automatically" is a source of confusion and gives JavaScript (and other high-level languages) developers the impression they can decide not to care about memory management. This is a mistake.

## Memory life cycle

Regardless of the programming language, memory life cycle is pretty much always the same:

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

The second part is explicit in all languages. The first and last parts are explicit in low-level languages but are mostly implicit in high-level languages like JavaScript.

### Allocation in JavaScript

#### Value initialization

In order to not bother the programmer with allocations, JavaScript does it alongside with declaring values.

```
var n = 123; // allocates memory for a number
var s = 'azerty'; // allocates memory for a string

var o = {
 a: 1,
 b: null
}; // allocates memory for an object and contained values

// (like object) allocates memory for the array and
// contained values
var a = [1, null, 'abra'];

function f(a) {
 return a + 2;
} // allocates a function (which is a callable object)

// function expressions also allocate an object
someElement.addEventListener('click', function() {
 someElement.style.backgroundColor = 'blue';
}, false);
```

#### Allocation via function calls

Some function calls result in object allocation.

```
var d = new Date(); // allocates a Date object
var e = document.createElement('div'); // allocates a DOM element
```

Some methods allocate new values or objects:

```
var s = 'azerty';
var s2 = s.substr(0, 3); // s2 is a new string
// Since strings are immutable value,
// JavaScript may decide to not allocate memory,
// but just store the [0, 3] range.

var a = ['ouais ouais', 'nan nan'];
var a2 = ['generation', 'nan nan'];
var a3 = a.concat(a2);
// new array with 4 elements being
// the concatenation of a and a2 elements
```

## Using values

Using values basically means reading and writing in allocated memory. This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

### Release when the memory is not needed anymore

Most memory management issues come at this phase. The hardest task here is to find when "the allocated memory is not needed any longer". It often requires the developer to determine where in the program such piece of memory is not needed anymore and free it.

High-level languages embed a piece of software called "garbage collector" whose job is to track memory allocation and use in order to find when a piece of allocated memory is not needed any longer in which case, it will automatically free it. This process is an approximation since the general problem of knowing whether some piece of memory is needed is [undecidable](#) (can't be solved by an algorithm).

## Garbage collection

As stated above, the general problem of automatically finding whether some memory "is not needed anymore" is undecidable. As a consequence, garbage collections implement a restriction of a solution to the general problem. This section will explain the necessary notions to understand the main garbage collection algorithms and their limitations.

## References

The main notion garbage collection algorithms rely on is the notion of *reference*. Within the context of memory management, an object is said to reference another object if the former has an

access to the latter (either implicitly or explicitly). For instance, a JavaScript object has a reference to its [prototype](#) (implicit reference) and to its properties values (explicit reference).

In this context, the notion of "object" is extended to something broader than regular JavaScript objects and also contains function scopes (or the global lexical scope).

### Reference-counting garbage collection

This is the most naive garbage collection algorithm. This algorithm reduces the definition of "an object is not needed anymore" to "an object has no other object referencing to it". An object is considered garbage collectible if there is zero reference pointing at this object.

### Example

```
var o = {
 a: {
 b: 2
 }
};

// 2 objects are created. One is referenced by the other as one of its
// properties.
// The other is referenced by virtue of being assigned to the 'o' variable.
// Obviously, none can be garbage-collected

var o2 = o; // the 'o2' variable is the second thing that
 // has a reference to the object
o = 1; // now, the object that was originally in 'o' has a unique
reference
 // embodied by the 'o2' variable

var oa = o2.a; // reference to 'a' property of the object.
 // This object has now 2 references: one as a property,
 // the other as the 'oa' variable

o2 = 'yo'; // The object that was originally in 'o' has now zero
 // references to it. It can be garbage-collected.
 // However its 'a' property is still referenced by
 // the 'oa' variable, so it cannot be freed

oa = null; // The 'a' property of the object originally in o
 // has zero references to it. It can be garbage collected.
```

### Limitation: cycles

There is a limitation when it comes to cycles. In the following example, two objects are created and reference one another, thus creating a cycle. They will go out of scope after the function call, so they are effectively useless and could be freed. However, the reference-counting algorithm considers that since each of the two objects is referenced at least once, neither can be garbage-collected.

```
function f() {
```

```

var o = {};
var o2 = {};
o.a = o2; // o references o2
o2.a = o; // o2 references o

return 'azerty';
}

f();

```

## Real-life example

Internet Explorer 6 and 7 are known to have reference-counting garbage collectors for DOM objects. Cycles are a common mistake that can generate memory leaks:

```

var div;
window.onload = function() {
 div = document.getElementById('myDivElement');
 div.circularReference = div;
 div.lotsOfData = new Array(10000).join('*');
};

```

In the above example, the DOM element "myDivElement" has a circular reference to itself in the "circularReference" property. If the property is not explicitly removed or nulled, a reference-counting garbage collector will always have at least one reference intact and will keep the DOM element in memory even if it was removed from the DOM tree. If the DOM element holds lots of data (illustrated in the above example with the "lotsOfData" property), the memory consumed by this data will never be released.

## Mark-and-sweep algorithm

This algorithm reduces the definition of "an object is not needed anymore" to "an object is unreachable".

This algorithm assumes the knowledge of a set of objects called *roots* (In JavaScript, the root is the global object). Periodically, the garbage-collector will start from these roots, find all objects that are referenced from these roots, then all objects referenced from these, etc. Starting from the roots, the garbage collector will thus find all *reachable* objects and collect all non-reachable objects.

This algorithm is better than the previous one since "an object has zero references" leads to this object being unreachable. The opposite is not true as we have seen with cycles.

As of 2012, all modern browsers ship a mark-and-sweep garbage-collector. All improvements made in the field of JavaScript garbage collection (generational/incremental/concurrent/parallel garbage collection) over the last few years are implementation improvements of this algorithm, but not improvements over the garbage collection algorithm itself nor its reduction of the definition of when "an object is not needed anymore".

## Cycles are not a problem anymore

In the first above example, after the function call returns, the 2 objects are not referenced anymore by something reachable from the global object. Consequently, they will be found unreachable by the garbage collector.

## Limitation: objects need to be made explicitly unreachable

Although this is marked as a limitation, it is one that is rarely reached in practice which is why no one usually cares that much about garbage collection.

## Concurrency model and Event Loop

JavaScript has a concurrency model based on an "event loop". This model is quite different from models in other languages like C and Java.

## Runtime concepts

The following sections explain a theoretical model. Modern JavaScript engines implement and heavily optimize the described semantics.

### Visual representation

### Stack

Function calls form a stack of *frames*.

```
function foo(b) {
 var a = 10;
 return a + b + 11;
}

function bar(x) {
 var y = 3;
 return foo(x * y);
}

console.log(bar(7)); //returns 42
```

When calling `bar`, a first frame is created containing `bar`'s arguments and local variables. When `bar` calls `foo`, a second frame is created and pushed on top of the first one containing `foo`'s arguments and local variables. When `foo` returns, the top frame element is popped out of the stack (leaving only `bar`'s call frame). When `bar` returns, the stack is empty.

## Heap

Objects are allocated in a heap which is just a name to denote a large mostly unstructured region of memory.

## Queue

A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function which gets called in order to handle the message.

At some point during the [event loop](#), the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty; then the event loop will process the next message in the queue (if there is one).

## Event loop

The **event loop** got its name because of how it's usually implemented, which usually resembles:

```
while (queue.waitForMessage()) {
 queue.processNextMessage();
}
```

`queue.waitForMessage()` waits synchronously for a message to arrive if there is none currently.

### "Run-to-completion"

Each message is processed completely before any other message is processed. This offers some nice properties when reasoning about your program, including the fact that whenever a function runs, it cannot be pre-empted and will run entirely before any other code runs (and can modify data the function manipulates). This differs from C, for instance, where if a function runs in a thread, it can be stopped at any point to run some other code in another thread.

A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog. A good practice to follow is to make message processing short and if possible cut down one message into several messages.

## Adding messages

In web browsers, messages are added anytime an event occurs and there is an event listener attached to it. If there is no listener, the event is lost. So a click on an element with a click event handler will add a message--likewise with any other event.

The function `setTimeout` is called with 2 arguments: a message to add to the queue, and a time value (optional; defaults to 0). The time value represents the (minimum) delay after which the message will actually be executed. If there is no other message in the queue, the message is processed right after the delay; however, if there are messages, the `setTimeout` message will have to wait for other messages to be processed. For that reason, the second argument indicates a minimum time and not a guaranteed time.

Here is an example that demonstrates this concept (`setTimeout` does not run immediately after its timer expires):

```
const s = new Date().getSeconds();

setTimeout(function() {
 // prints out "2", meaning that the callback is not called immediately
 // after 500 milliseconds.
 console.log("Ran after " + (new Date().getSeconds() - s) + " seconds");
}, 500);

while(true) {
 if(new Date().getSeconds() - s >= 2) {
 console.log("Good, looped for 2 seconds");
 break;
 }
}
```

## Zero delays

Zero delay doesn't actually mean the call back will fire-off after zero milliseconds. Calling `setTimeout` with a delay of 0 (zero) milliseconds doesn't execute the callback function after the given interval.

The execution depends on the number of waiting tasks in the queue. In the example below, the message "this is just a message" will be written to the console before the message in the callback gets processed, because the delay is the minimum time required for the runtime to process the request, but not a guaranteed time.

Basically, the `setTimeout` needs to wait for all the codes to complete even though you specified a particular time limit for your `setTimeout`.

```
(function() {

 console.log('this is the start');
```

```

setTimeout(function cb() {
 console.log('this is a msg from call back');
});

console.log('this is just a message');

setTimeout(function cb1() {
 console.log('this is a msg from call back1');
}, 0);

console.log('this is the end');

})();

// "this is the start"
// "this is just a message"
// "this is the end"
// note that function return, which is undefined, happens here
// "this is a msg from call back"
// "this is a msg from call back1"

```

## Several runtimes communicating together

A web worker or a cross-origin `iframe` has its own stack, heap, and message queue. Two distinct runtimes can only communicate through sending messages via the [postMessage](#) method. This method adds a message to the other runtime if the latter listens to `message` events.

## Never blocking

A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks. Handling I/O is typically performed via events and callbacks, so when the application is waiting for an [IndexedDB](#) query to return or an [XHR](#) request to return, it can still process other things like user input.

Legacy exceptions exist like `alert` or synchronous XHR, but it is considered as a good practice to avoid them. Beware, [exceptions to the exception do exist](#) (but are usually implementation bugs rather than anything else).

## Solve common problems in your JavaScript code

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Howto>

## JavaScript guide

A much more detailed guide to the JavaScript language, aimed at those with previous programming experience either in JavaScript or another language.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>

## JavaScript reference

If you need exhaustive information about a language feature, have a look at the [JavaScript reference](#).

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>