sitepoint

LEARN TO CODE

WITH JAVASCRIPT

BY **DARREN JONES**

START CODING TODAY!

Learn to Code with JavaScript

Ebook ISBN: 78-1-925836-41-7

- **Product Manager:** Simon Mackie
- **Technical Editor:** James Hibbard
- **English Editor:** Ralph Mason
- **Cover Designer:** Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

About Darren Jones

Darren has enjoyed coding since learning how to program in BASIC on his first Acorn Electron computer. Since then, he's taught himself Ruby and JavaScript and is the author of *JavaScript: Novice to Ninja* and *Jump Start Sinatra*. He also produced the "Getting Started With Ruby" video tutorials for SitePoint Premium and has written a number of articles on the SitePoint website. He was born in the city of Manchester in the UK, where he still lives, and he teaches Mathematics and Computing at a local high school. You can find him on Twitter @daz4126.

Preface

I still remember my first ever computer: it was an Acorn Electron, and I loved using it to play games such as Sphinx Adventure (a text-based adventure), Starship Command (a hard-as-nails shoot-em-up) and Chuckie Egg (pure platform action). But the real fun started when I realized that I could write my own code in a language called BASIC. My initial thoughts were that it had been named ironically, as it appeared to be anything but basic. It had line numbers that went up in multiples of ten and strange-sounding commands such as `GOTO`, `REM` and `CLS` that just looked like gobbledygook at first. But with lots of practice (and many mistakes), it started to make sense and I became fascinated by how the code allowed me to control what the computer did. I would spend hours copying code examples out of magazines (yes, it was that long ago!) and then play around making my own modifications. It was this experimentation and tinkering with code that helped me understand how it worked. I had been well and truly bitten by the coding bug.

My enjoyment of coding comes from the fact that it requires you to follow the precise rules of the programming language while also encouraging you to think creatively to achieve your desired outcome. The essence of coding is turning the abstract into the practical. When I set about writing this book, my aim was to introduce the basic concepts of coding and back the theory up with plenty of practical examples. Each chapter ends with coding challenges that will help consolidate your understanding. I encourage you to really dig into these challenges by not only trying to complete them, but also trying to extend them with your own modifications.

It was always games that I wanted to program, and I've tried to make as many of the challenges in the book as fun as possible, both to code and to play. And by the end of the book, you'll have coded a couple of fully playable games. You'll also have the foundations in place for creating interactive websites and be on the right path to writing full-scale applications.

Programming is a creative endeavor; it's fundamentally about creating things. You need your code to be precise and concise, but you also need to inject it with some flair to make it come alive. This mix of precision and ingenuity is what continues to make coding enjoyable for me, many years after starting out on my Acorn Electron. My hope is that this book will inspire you to start coding and that you'll continue to enjoy it for a long time after you've finished reading the last page.

Who Should Read This Book?

This book is for people with no prior programming experience who would like to learn how to code. We use JavaScript in this book to teach you, so at the end you'll have a good understanding of JavaScript, but you can apply the principles you've learned to other programing languages, too.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.
</p>
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

- https://www.sitepoint.com/community/ are SitePoint's forums, for help on any tricky problems.
- **books@sitepoint.com** is our email address, should you need to contact us to report a problem, or for any other reason.

Chapter 1: Press Start

So you want to learn to program? You've made a good decision. Programming is a fantastic skill to learn, and it's great fun. It can be used to build the next generation of apps, hack a Raspberry Pi or Arduino, write the latest blockbuster games—and a ton of other things. In fact, once you learn how to program, the only limit is your imagination.

In this chapter, we'll briefly survey the history of programming, look into what a computer program actually is, and then introduce the **JavaScript** programming language, which we'll be using to learn how to program.

We'll also be jumping right in and getting started with some programming, writing not one, but *two* programs in JavaScript!

Here's what this chapter will cover:

- what programming actually is
- algorithms and pseudocode
- a brief history of programming
- an introduction to the JavaScript language
- "Hello, World!"—your first JavaScript program
- JavaScript in the web browser
- "I Can Code a Rainbow"—your second JavaScript program
- the mindset of a programmer

We'll also finish the chapter with some programming challenges to help you to test your newfound skills—as we'll do at the end of every chapter in this book.

Programming

Programming is about making computers do what you want them to do. A computer program is basically a series of instructions that tell your computer

how to perform a task. Unfortunately, computers don't speak the same language as us.

For example, you can't just write "change the color of the circle to blue" and expect the computer to understand. A programming language acts as an intermediary: it's a language that can be understood by both computers and humans.

Learning to program is a bit like learning a foreign language, except computers can be *very* picky about grammar (even more so than my French teacher was!). You need to make sure you get everything in the right place, and the syntax needs to be just right. Computers are powerful, and you can get them to do some truly impressive stuff, but they'll also fall to pieces if just one bracket is out of place!

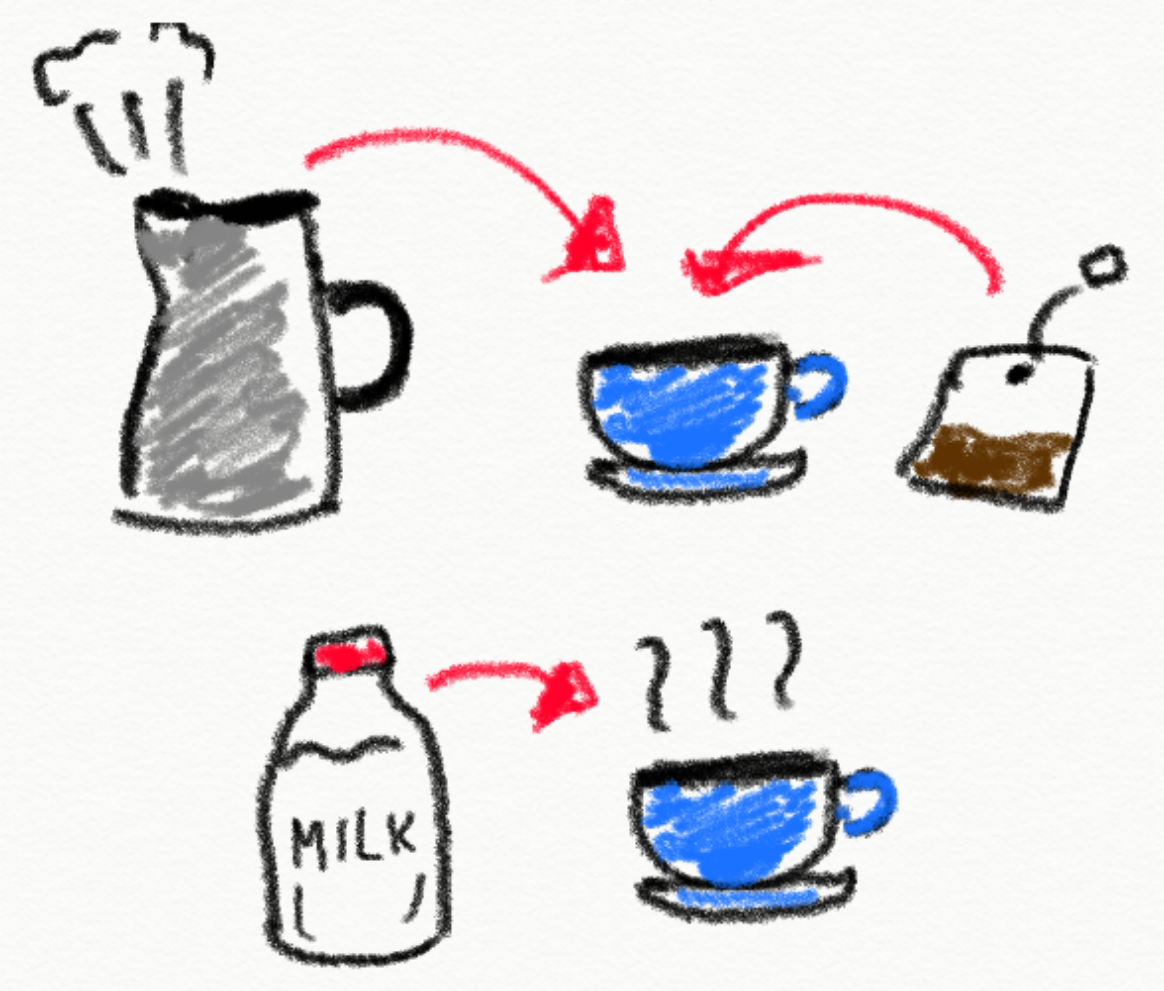
Writing a program is basically just writing a set of instructions for a computer to follow. The problem is, they have to be *very* precise instructions. Any slight ambiguity, and the computer will do something completely different from what you had in mind—or it might even crash.

Algorithms

The word **algorithm** is used quite frequently these days. You might have heard of the "Instagram algorithm" or the "Google search algorithm". But what exactly is an algorithm? The word "algorithm" is a Latinization of the name of a Persian mathematician, Al-Khwarizmi, who wrote the first algebraic textbook and liked to explain his methods using a step-by-step approach.

This led to the word *algorithm* being used to describe any step-by-step method. It should be clear what to do at each step and what each step does. For example, here's an algorithm for making a cup of tea:

1. Boil water in a kettle.
2. Get a cup.
3. Get a teabag.
4. Put the teabag in the cup.
5. When the kettle has boiled, pour the water into the cup.
6. Leave it to brew for two minutes.
7. Take the teabag out of the cup.
8. Get milk.
9. Add milk to the tea.
10. Stir the tea.



Milk after Tea

I realize my instruction to add milk *after* the tea is made might be controversial. If you think milk should be added first, I'm sorry, but you'll just have to accept that you're wrong on this one.

Those instructions seem fairly basic and straightforward, but there are quite a few assumptions made, and some steps are a bit ambiguous. How do you boil a kettle? Where do you get the cup and teabag from? How much water should I pour into the cup? These questions are left unanswered. This is usually okay, as most people are familiar with the process of making tea, but someone who had never seen a cup of tea being made might struggle to follow some of the steps (although I accept that it might be very hard to find such a person).

Pseudocode

Pseudocode is pretend code you can write to illustrate what a program does without having to worry about the complexities of an actual programming language. It still follows the conventions and structure of a program and uses precise commands, but without the specifics of a language.

For example, the instruction "display a message on the screen" might be written in pseudocode as `display(message)`, whereas in the Python programming language it would be written as `print(message)`. Notice that the Python code uses the very specific notation of `print`, which is the command it uses to display messages on the screen. Other languages use different commands to basically do the same thing.

The example of making a cup of tea could be written in pseudocode like so:

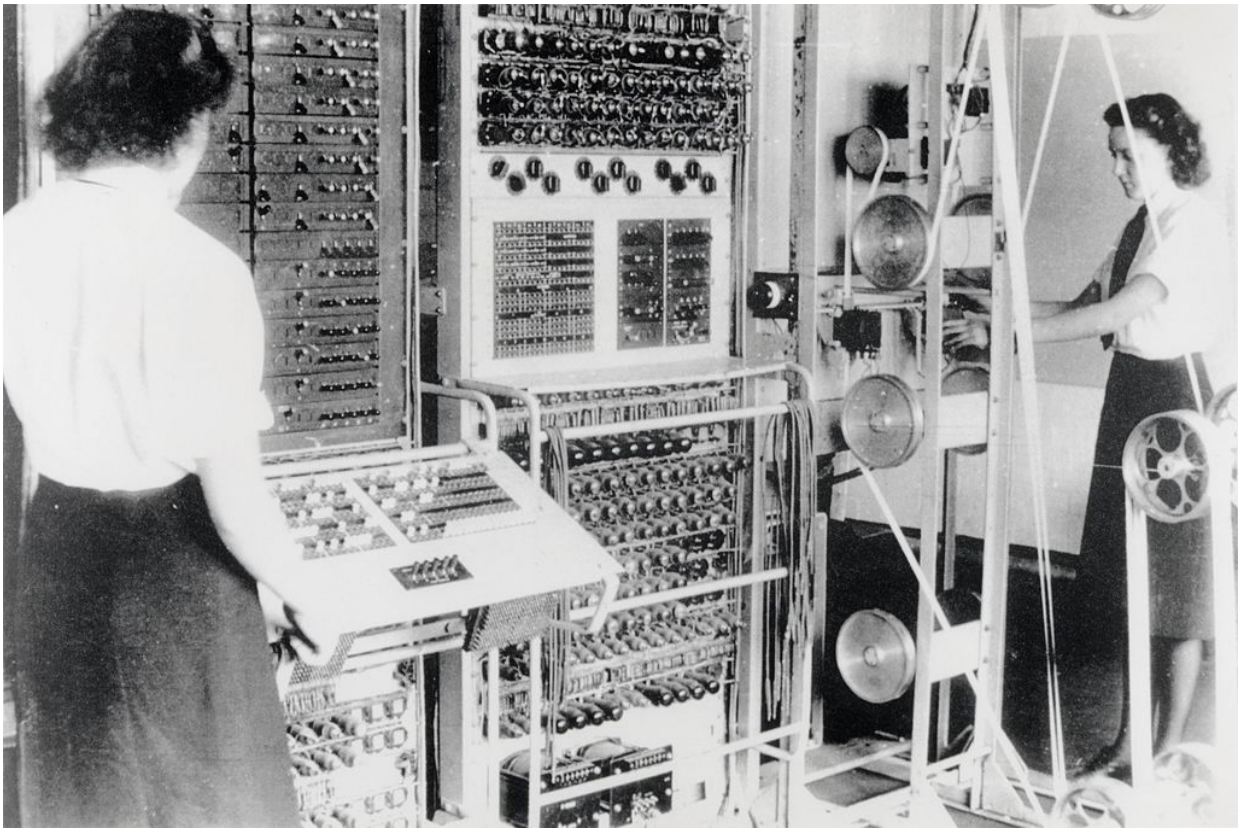
```
boil(water,250ml) in kettle
get(cup) from cupboard
get(teabag) from box
put(teabag) in cup
when(kettle boils)
  { pour(water, 250ml) in cup }
wait(2 mins)
remove(teabag) from cup
get(milk, 20ml)
add(milk, 20 ml) to tea
stir(tea)
```

A programmer who knows a how to program in a specific language should be able to follow a pseudocode example and then write it in their language of choice. Pseudocode is useful for planning out a program before you start coding, and it also makes it easier to share ideas between programmers who use different programming languages.

A Brief History of Programming

The earliest computers were programmed using punched cards to represent a **binary number system**, the number system that computers use. The binary system is made up entirely of 1s and 0s that loosely translate to "on" and

"off". On the cards, a hole represented `1` and no hole `0`. After this, people started to develop languages that could be used to "speak" to the computer.



The first computer programs were written in **machine code** and **assembly language**. These are **low-level programming languages** that are closely associated with a computer's hardware. This means they can be difficult languages to program in because they involve writing abstract code that's heavily tied to a computer's architecture. In fact, it's rare to find anyone who programs in machine code or assembly nowadays, but those who do will work closely with a computer's hardware—for example, writing device drivers, or working on embedded systems.

High-level programming languages use abstractions that make the code easier for humans to read and write. Programs are written in a language such as Swift, C# or Java, which is then compiled into machine code and executed. The programs written using these languages are very fast, making high-level languages suited to writing games and professional business software where speed is important.

Scripting languages are another type of high-level language, but they're **interpreted**, instead of **compiled**, which means that they're translated into machine code when the program runs, rather than beforehand. This means that they're often slower than compiled languages, although interpreters are becoming more sophisticated, making some interpreted languages almost as fast as compiled languages. Some common scripting languages that you might hear about are Python, Ruby and, of course, JavaScript.

JavaScript

The language we'll be learning in this book is JavaScript, often referred to as the language of the Web. Pretty much every web browser can run JavaScript, making it one of the most popular programming languages in the world.

JavaScript is a great language to use when learning how to program. It has a low barrier to entry: all you need to be able to program in JavaScript is a web browser. It's easy to get started, and the basics are easy to pick up. It's also a flexible and expressive language that can create a variety of powerful programs.

JavaScript is a scripting language that's interpreted and compiled at runtime. This means that it requires an engine to interpret and run a program. This is usually done by a web browser, but there are JavaScript engines that can run programs without a browser. JavaScript is also a **dynamic** language, which means that elements of a program can change while it's running, unlike a compiled language such as C++.

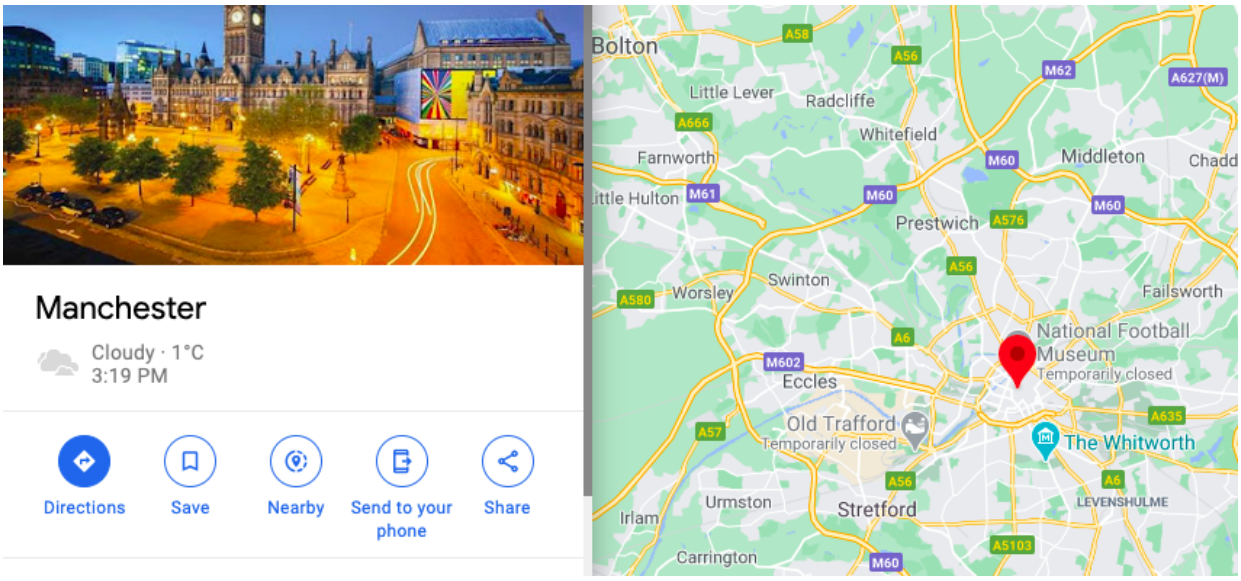
The History of JavaScript

The World Wide Web started life as a bunch of pages linked by hyperlinks. Users soon wanted more interaction with these pages, so Netscape (an early browser vendor) asked one of their employees, Brendan Eich, to develop a new language for their Navigator browser. He came up with a prototype scripting language called Mocha in just ten days. The new language was renamed LiveScript, but was then hastily rebranded as JavaScript so that it could benefit from the publicity that Sun Microsystems' Java language was attracting at the time.

The name “JavaScript” has often caused some unfortunate confusion, with JavaScript often thought of as “Java lite”, even though the two languages are completely unrelated.

JavaScript made its debut in 1995 and ushered in an exciting new era of being able to program a web browser to do stuff. (Unfortunately, in the early days, the most common use was to create pop-up ads and spinning logos!)

By 2005, sites such as Google Maps and Gmail started to appear, and they demonstrated that JavaScript was capable of creating rich internet applications that looked and behaved like native desktop applications. This progress has continued at pace, and almost every website today will use JavaScript in some way. JavaScript has also found its way into a variety of other things such as smartphone apps, wearables and databases.



JavaScript Versions

In 1996, a decision was made to standardize JavaScript with the help of the European Computer Manufacturers Association, who would host the standard. This standardized language was called **ECMAScript** to avoid infringing on the “Java” trademark. This caused even more name-based confusion, but eventually ECMAScript was used to refer to the specification, and JavaScript was used to refer to the language itself. This is still the case, although there’s constant debate about changing the name. One suggestion is

to officially rename JavaScript just as "JS"—which it is often called anyway (just as Michael Jordan is often referred to as "MJ").

When JavaScript was standardized in 1997, the specification was known as ECMAScript version 1. In 2015, it was decided to publish a new specification every year, with the version named after the year it was published. As a result of this change, ECMAScript version 6 was renamed ES2015 when it was published, and since then there's been a new version of JavaScript every year. In this book, we'll use the most up-to-date version of JavaScript, but it's always worth keeping up to date with the latest additions and changes to the language each year.

Backward Compatibility

An important concept in the development of the JavaScript language is that of **backward compatibility**. This means that all old code must work the same way when interpreted by an engine running a new specification. (It's a bit like saying that a PlayStation 5 must still be able to run any games created for all the previous PlayStations). This is to prevent JavaScript from "breaking the Web" by making drastic changes that would mean old, legacy code used on some websites might not run as expected in modern browsers.

In this book, we'll assume you're using a modern browser that's capable of running the latest version of JS. (Try to update to the latest version of whichever is your favorite browser.)

That's enough talk about JavaScript. Let's write your first program!

Hello, World! Your First JavaScript Program

It's a tradition when learning programming languages to start with a **Hello, World!** program. This is a simple program that outputs the phrase "Hello, World!" to announce your arrival to the world of programming.

We're going to stick to this tradition and write a Hello, World! program in JavaScript. It will be a single statement that logs the phrase "Hello, World!" to the console.

The Console?

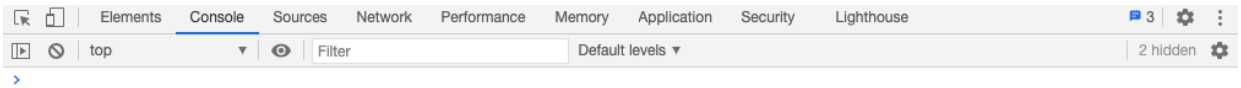
A **console** is basically a command prompt used to run code. You can enter a snippet of code and it will be evaluated and the output logged to the screen. A console is a great way to test and experiment with code. You can get JS consoles to run in your computer terminal as well as JS console apps for smartphones and tablets. There are also many websites that run a console in the browser, and most browsers also have a built-in console as well.

In this book, we'll mostly use the online jsconsole.com as our console. I'd recommend that you use it to try running the code snippets that appear throughout this book to get a feel for how they work. It's always better to get a feel for typing in the code rather than just reading it. And it also means you can experiment with the code by making changes and checking the results.

Another option for a console is to use the one built into your browser. To open it, simply follow the instructions below, depending on your browser:

- **Firefox**: hold `Shift` + `Ctrl` + `J` (or `Option` + ⌘ + `J` on a Mac) or press `F12`
- **Safari**: hold `Option` + ⌘ + `C` (note that you need to enable the Developer Menu in preferences first)
- **Chrome**: hold `Shift` + `Ctrl` + `J` (or `Option` + ⌘ + `J` on a Mac) or press `F12`
- **Edge/Internet Explorer**: press `F12`

Once you open the console in your browser, it will act exactly the same as jsconsole.com. The image below shows Chrome's built-in console.



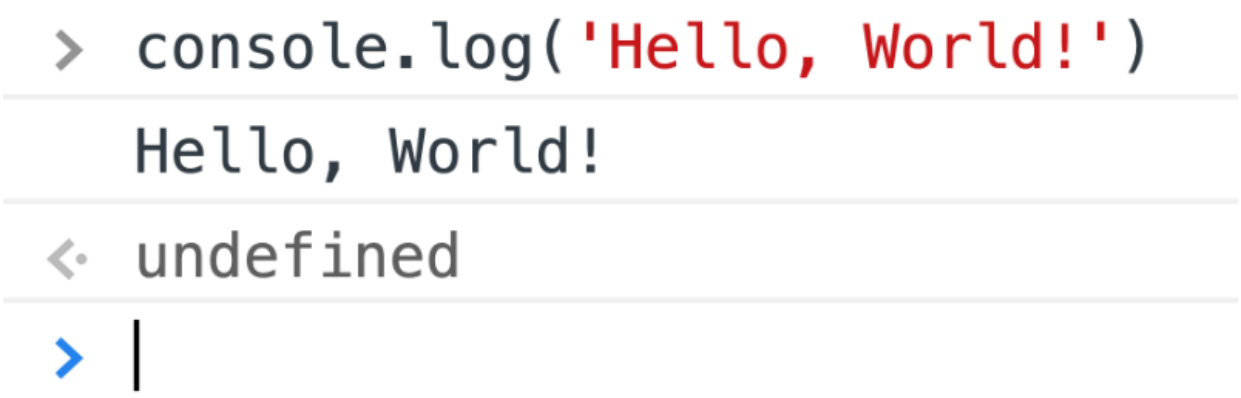
Okay, let's write our first program. Go to jsconsole.com or open up your browser's console and type the following code, then press the `Enter` key:

```
console.log('Hello, World!');
```

Enter? Return?

I told you to press the `Enter` key on your keyboard above. Just to be clear, on most PC keyboards you'll see an `Enter` key towards the right-hand side. (Sometimes it will just be an arrow symbol: ↵.) On Mac keyboards you'll mostly see a `return` key instead. For simplicity, I'll just refer to it as `Enter` from here on.

If everything went to plan, you should see "Hello, World!" displayed on the screen—just as in the screenshot below.



Congratulations! You've just written your first JavaScript program! It might not look like much, but a wise person once said that every programmer's journey begins with a single line of code—or something like that, anyway!

JavaScript in the Browser

I said earlier that JavaScript is an interpreted language and needs a host environment to run. Because of its origins, the main environment that JavaScript runs in is the browser, although it can be run in other environments as well.

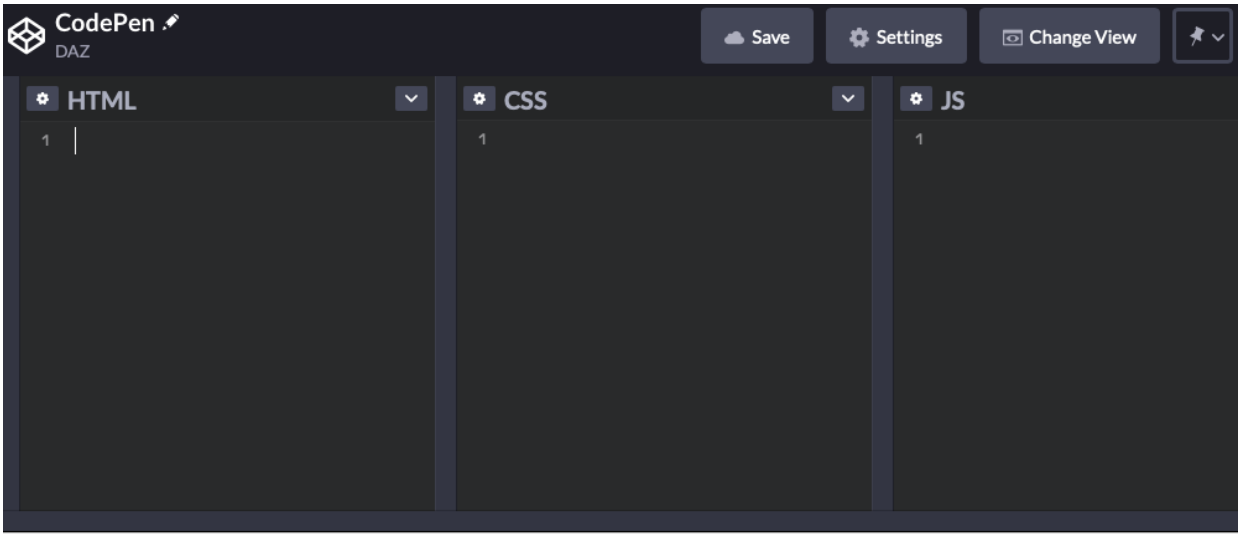
By far the most common use of JavaScript is still to make web pages interactive. Because of this, we should have a look at what makes up a web page before we go any further.

The Structure of a Web Page

Nearly all web pages are made up of three key ingredients: HTML, CSS, and JavaScript. HTML is used to mark up the content of the page, CSS is the presentation layer that dictates what the page will look like, and JavaScript adds the interactivity that makes the page do stuff. Together, these three technologies are known as the *three layers of the Web*. It's possible to add CSS and JavaScript in HTML, but it's good practice to keep the code for each layer separate.

We won't be focusing on the HTML and CSS parts of a web page in this book, as we're here to learn how to program. Some HTML and CSS will occasionally be required, though, to provide the graphical elements that our programs will need to interact with.

To save you spending a long time setting up and coding web pages, we'll be using the CodePen website for many of the examples in this book. This site sets everything up for you so that you only need to add any HTML, CSS and JS code to create a functioning web page. It also keeps the three layers of the Web separate in three sections at the top of the editor, as you can see in the screenshot below.



CodePen also offers a console. At various points throughout this book, you'll find it convenient to enter JavaScript code in the **JS** section of CodePen and then see how that code performs behind the scenes by opening the CodePen console. You can access it by clicking the **Console** tab at the bottom left of the CodePen interface.

Online Editors and Working Offline

There are many other options for running JavaScript code alongside HTML and CSS. Just search for "online JavaScript editor" to see more online options. It's also possible to work offline (that is, just on your computer) by writing the code in a text editor and opening it up in a browser. See "HTML5 Template: A Basic Boilerplate for Any Project" for instructions on how to set this up.

Let's try writing our second program.

I Can Code a Rainbow: Your Second JavaScript Program

We're going to finish this chapter with a second JavaScript program that will run in the browser. This example is much more complicated than the previous one and includes a lot of concepts that will be covered in more depth in later chapters, so don't worry if you don't fully understand

everything that's going on at this stage! The idea is to show you what JavaScript is capable of doing and interactively introduce some of the important concepts that will be covered in the upcoming chapters.

Head over to [the CodePen website] and start a new Pen.

Add the following to the **HTML** section:

```
<button id='button'>click me</button>
```

This will display a button with an ID of `button`. The ID attribute is a useful way for our JavaScript program to identify, and manipulate, certain elements on the page. It should look similar to the one in the image below.

The actual JavaScript code goes in the **JS** section. Add the following lines of code:

```
const button = document.getElementById('button');

const colors =
['red','orange','yellow','green','blue','rebeccapurple','violet'
];

function change() {
  document.body.style.background =
colors[Math.floor(7*Math.random())];
}

button.addEventListener('click', change);
```

The first line of our program creates a variable called `button`. (We cover variables in Chapter 2.)

We then use the `document.getElementById` function to find the HTML element with the ID of `button`. This is the button element we created in the HTML section. This is assigned to the variable `button`, so from now on, whenever we refer to `button`, the program knows we're talking about that button. (Finding HTML elements is covered in Chapter 10.)

We now create another variable called `colors` that's assigned to an array containing a list of strings that represent different colors of the rainbow. (We cover strings in Chapter 3 and arrays in Chapter 5.)

Then we create a function called `change`. (We cover functions in Chapter 8.) This sets the background color of the web page's body element to one of the colors in the `colors` array. (Changing the look of a page is covered in Chapter 10.)

The `change` function also uses a random number to pick a color at random from the array. (Numbers, including generating random numbers, are covered in Chapter 4.)

The program ends with an **event handler**, which listens out for when the button is clicked. When this happens, it runs the function we just defined. (Events and how to handle them are covered in Chapter 11.)

Try clicking on the button a few times. If everything's working correctly, the background should change to every color of the rainbow!

You can see my code on CodePen.

Rebecca Purple

`rebeccapurple` is the official name for the color with a hex code of `#663399`. It's named after the daughter of the web designer Eric Meyer, who tragically died, aged just six years old. This was her favorite color and it was added to the official list of CSS colors as a tribute to her.

A Programmer's Mindset

Before we finish the chapter, we need to have a word about *programming style*. Obviously the point of writing a computer program is to make it do what it's supposed to do, but the style of the program is also important. Your code should be clear and concise, and also be consistent in the way it follows conventions—from the names you choose for variables to the amount of whitespace you leave between blocks of code. Your code should also be easy to read and include comments that explain what the code does.

You should always be looking to improve your code to make it more efficient and easier to follow. This process is called **refactoring**, and should be done regularly so that your code stays up to date and doesn't become stale.

An important principle in programming is the rule "Don't Repeat Yourself", or **DRY**. This means that you should always be looking to avoid repeating lots of code. Following this principle will help make your code more flexible and easier to maintain and update. (It's always easier to change one bit of code than hundreds!) If this sounds a bit like programmers are lazy, then you're absolutely right! A good programmer will always be looking to write code that does the job in the most efficient and elegant way possible. Less code means less chance of bugs, and it's easier to maintain.

A good program doesn't just appear instantly. It starts as a plan, and grows organically into its finished state. Developing a program involves a lot of problem solving, and sometimes you'll take some wrong turns and get stuck. This is normal, and you should be prepared to make changes and adapt. A good plan and well-commented code that follows the DRY principle will stop your code becoming a complex mess that's impossible to follow and comprehend.

Challenges

1. Try writing pseudocode to describe some common tasks such as brushing your teeth or riding a bike.

2. Write pseudocode that will find the largest number in a list of numbers.

3. Try logging some different messages to the console using `console.log()`. You can see my code on CodePen.

4. Add some more colors to the array in the "I Can Code a Rainbow" program. You can see my code on CodePen.

Summary

- An algorithm is a set of step-by-step instructions that complete a specific task.
- Pseudocode is used to write programs without using a specific programming language.
- JavaScript was created in 1995 and is considered to be the language of the Web.
- JavaScript's main environment is the browser, but it can also run in other environments.
- Each version of JavaScript has to be backward compatible with older versions.
- Code should be easy to read and regularly refactored to keep it running smoothly.
- Remember the DRY principle when coding. Don't Repeat Yourself!

In the next chapter, we're going to start looking at some programming fundamentals. Let's get going!

Chapter 2: Programming Basics

In the last chapter, we were introduced to programming and we even got our hands dirty writing a few programs.

In this chapter, we're going to look at some of the basics of programming and write some more programs.

We'll cover the following topics:

- commenting your code
- programming grammar
- data types
- variables
- pop-up boxes

Comments

The first task on your journey to becoming a programmer is learning how to "comment" your code. **Comments** in programming are similar to marginal notes in a book: they're notes that explain the purpose and rationale of the code. This may seem a strange place to start, because comments are ignored by the programming language. They don't do anything. Despite this, comments are extremely important. Well-commented code is the hallmark of a skilled programmer. Comments make it easier for anybody reading your code to understand what's going on—including you! Believe me, you'll be thankful you commented your code when you come back to read it after a few weeks. You don't need to write an essay, though, but just enough so that it's clear what the code is supposed to do.

How to add comments varies between programming languages. In JavaScript, there are two types of comment:

- Single-line comments start with `//` and finish at the end of the line:

```
// this is a short comment
```

- Multi-line comments start with `/*` and finishing with `*/`:

```
/* This is a longer comment
anything here will be ignored
This is a useful place to put notes
*/
```

It's good practice to write comments in your code, but make sure that the comments are useful and not just describing what's fairly obvious from the code itself. For example, the following comment doesn't really add anything that isn't obvious from the code itself:

```
// log the message to the console
console.log(message);
```

It's useful to think of comments as notes to your future self, to remind you about what your thinking process was when you wrote that particular piece of code. For example, here's a comment explaining that the code is activated by a user clicking the left arrow key:

```
// user has clicked on the left arrow key so move left
character.style.left = left - speed + 'px';
```

Programming Grammar

As I mentioned in the last chapter, a programming language is similar to a spoken language, in that each has its own grammatical rules and quirks. In this section, we're going to take a look into how a programming language is written, with some specific JavaScript examples.

Statements

A program is made up of a series of **statements**. For example, here are two statements in pseudocode:

```
store response to 'Please enter your name.' as name
print name to screen
```

In JavaScript, this would be:

```
const name = prompt('Please enter your name.');
alert(name);
```

Ending Lines with Semicolons

I finished each line above with a semicolon. Strictly speaking, it isn't necessary to end a statement with a semicolon *when it's on its own line*, but it's considered to be good practice, and it's what I'll be doing throughout this book. I'd encourage you to do the same so it becomes habit. Believe me, it will be useful by the time you're writing larger, more complex programs!

If you have more than one statement per line, you do need to separate them with a semicolon. But it makes your code neater and more readable to have each statement on its own line.

Blocks

A **block** is a series of statements that are collected together. Different programming languages have various ways of collecting statements inside blocks. Some do it by using keywords to signify the start and end of blocks, while others use indentation to signify a block. JavaScript uses curly braces to enclose a block of code, as can be seen in the example below:

```
{
  // this is a block containing a comment and two statements
  const name = prompt('Please enter your name.');
  alert(message);
}
```

Note that blocks don't need to be terminated by a semicolon.

Whitespace

Whitespace (such as spaces, tabs, and new lines) is used to separate different parts of your code. JavaScript allows you to use as much whitespace as required to format your code so that it's neat and easy to read. The whitespace is basically all ignored by the program itself. Examples of this include using spaces to indent nested code, and multiple lines to separate

blocks of code. In fact, this is highly recommended, since good code should be easy to follow.

In some languages, whitespace is used as part of the program. For example, Python explicitly uses new lines to start a new statement, and it uses spaces at the start of a line to start a new block of code.

Data Types

Every value in a programming language has a **type** that describes the data it contains. This determines how the language will interpret and use the value. The data type determines what values it can take and what it can do.

Primitive data types, or just **primitives**, are implemented at the lowest level of a programming language, essentially making them the basic building blocks of any program. Primitives are **immutable**, which means they can't be changed or altered. Most languages include some variation of the following primitive data types:

- **Character** (or **char**): a single character such as letters, numbers or symbols—such as `T`,`3`,`@`
- **String**: a collection of characters inside quote marks—such as `'Hello'`, `"123!"`
- **Integer**: a whole number that can be positive or negative—such as `7`, `-100`, `0`
- **Float**: numbers with a fractional part, usually expressed as a decimal—such as `2.5`, `3.14159`, `-7.0`
- **Boolean**: a Boolean value can either be "true" or "false"

JavaScript doesn't have a char primitive data type, but it does have "strings" (which we'll cover in the next chapter), and a char could be represented by a single-character string.

JavaScript doesn't differentiate between integers and floats either. It just has a single primitive data type of `Number`, which we'll cover in Chapter 4.

Booleans are one of the primitives included in JavaScript, and we'll go through them in Chapter 6.

`symbol`, `bigint` and `undefined`

JavaScript also has three other primitive values: `symbol`, `bigint` and `undefined`. We won't be covering `symbol` and `bigint` in this book, as they don't tend to get used as much as the other primitive values. However, `undefined` is briefly covered later in this chapter, since you'll probably see it every now and then in your code. You can read more about all the different primitive data types used in JavaScript in the Mozilla docs.

Composite data types are made up, or *composed*, of primitive data types in a structured format. JavaScript has two main composite data types: "arrays" (which we'll cover in Chapter 5), and "objects" (which we'll introduce in Chapter 9). Both of these composite data types are **mutable**, which means they can be updated after they've been created.

`Map` and `Set`

JavaScript also has two other composite data types called "Map" and "Set", but we won't be covering them in this book.

What Type Are You?

Let's do some coding to explore the different types that JavaScript uses. There's a special operator called `typeof` that can be used to find out what data type a value is. You simply enter `typeof`, followed by the value, and its type will be returned.

If a value isn't one of the primitive data types, JavaScript will return either `object` or `function`. (These will both be covered in later chapters.)

Let's open up the console again to do some tests. (Reminder: once you've typed the first line—such as `typeof 'hello'`—press the `Enter` key to get the result, `<< 'string'`.) Try entering the following code—one line at a time—to investigate the type of various expressions:

```
typeof 'hello'; // see Chapter 3
<< 'string'
```

```
typeof 10; // see Chapter 4
<< 'number'

typeof true; // see Chapter 6
<< 'boolean'

typeof { name: 'JavaScript' }; // see Chapter 9
<< 'object'

typeof [ 1, 2, 3 ]; // this is an array, covered in Chapter 5,
it's also considered an object
<< 'object'

typeof function(){ }; // this is an empty function, more
interesting functions are covered in Chapter 8
<< 'function'
```

Try a few more of your own, until you're confident about what constitutes a string, number and Boolean value.

Operators

An **operator** applies an operation to a value, which is known as the **operand**. A unary operator only requires one operand. `typeof` is an example of a JavaScript operator that we've already used:

```
typeof 'hello';
<< 'string'
```

The operator is `typeof` and the string `'hello'` is the operand.

A **binary operator** requires two operands. For instance:

```
3 + 5
```

The operator is `+` and the numbers `3` and `5` are the operands.

There's also a **ternary operator** that's used to evaluate logical statements and requires three operands. This is covered in Chapter 6.

Variables

Variables are used in programming languages to refer to a value stored in memory. They give us a convenient way to refer to different values in a program. They can be thought of as labels that point to different values. Variables are a useful way of dealing with long expressions, as they save us from typing these expressions over and over again.

Declaring and Assigning Variables

In most programming languages, variables have to be **declared** before they can be used. That is, they need to be explicitly referred to in the code, and possibly assigned a value.

In a **strongly typed** language, the type of the variable has to be declared with the variable. For example, if we were going to use the variable `name` for the string `'Homer Simpson'`, we might use the following code in a strongly typed language:

```
let name:string = "Homer Simpson";
```

This not only sets the variable `name` to point to the string `'Homer Simpson'`, but also declares that this variable will be a string. This will result in an error if you try to assign the variable to another value that is *not* a string later in the program.

Weakly typed programming languages don't insist on explicitly stating what type a variable is when it's declared. The type is said to be implicit from the actual value that's assigned to it, although you could theoretically assign the variable to a different type later in the program without any problems. In a weakly typed language, the following code might be used to set the variable `name` to be the string `'Homer Simpson'`:

```
name = "Homer Simpson";
```

This still has the same effect of pointing the variable `name` to the string `'Homer Simpson'`, but it doesn't explicitly say that this variable is a string, since this is implicit in the fact that it has been assigned to a string.

Duck Typing

A common concept used by dynamic programming languages—such as Ruby and Python—is that of **duck typing**. This is based on the phrase "If it walks like a duck and quacks like a duck, then it's a duck".

The essence of this phrase is that there's no need to worry about the type of a value or object. If it does what it's supposed to do, it doesn't matter what type it is. The focus is on checking that the output of the code works as expected, rather than on the type of input.

JavaScript is a weakly typed language, so you don't need to specify the type of a variable when you declare it. This might seem to be a benefit at first, although it can make debugging a program difficult when it isn't clear what type a variable should be.

TypeScript

TypeScript is an extension of JavaScript that provides the option to specify the type of variables explicitly when they're declared—effectively making it a strongly typed version of JavaScript. It also adds some other features, and is designed to make building large-scale applications in JavaScript easier.

Constants

Many languages also include **constants**, which work in a similar way to variables, except that their value never changes. It's common practice to use all capital letters for constants. An example is the value of the constant pi (the ratio of the circumference and diameter of a circle):

```
PI = 3.14159;
```

JavaScript originally used the keyword `var` to declare all variables. More modern versions of JavaScript have introduced the keywords `const` and `let` —`const` to declare variables that can't be reassigned to a new value, and `let` to declare variables that will change during the program. You might still see `var` used in some code examples, and it works in *almost* the same way as using `let`. The biggest difference between `var` and `let` is to do with block scope, which is discussed in Chapter 13.

Constants in JavaScript

Despite the introduction of the keyword `const`, it's important to note that JavaScript still doesn't support constants in the strictest sense. Even though you can't alter the value of a variable created using `const`, if that variable contains an array or an object, you can change its properties and values (because they are mutable). We'll cover this in more detail in later chapters, so there's no need to worry about it yet. For now, you can assume that the value of any variables declared using `const` can't be changed.

Assignment

Assignment is the process of assigning a value to a constant or variable. Most languages use the `=` operator to do this.

Let's try declaring and assigning some variables in JavaScript, using the console.

This example shows how we would declare a variable called `name` and assign the value `'JavaScript'` to it. Copy the following code into the console:

```
const name = 'JavaScript';
```

The variable `name` now has a value of `'JavaScript'`, so any reference to the variable `name` will behave in exactly the same way as the string `'JavaScript'`. We can see this by checking the type of the variable name:

```
typeof name;
<< 'string'
```

To see the value of a variable, simply type its name into the console and press return:

```
name;
<< 'JavaScript';
```

Using `const` means that you can't *reassign* the variable `name` to another value. You'll get an error message if you try. We can see this if we try to reassign the value of `name` to `'JS'`:

```
name = 'JS';
<< TypeError: Assignment to constant variable.
```

As you can see, we get an error message and the `name` variable is still pointing to the string `'JavaScript'`:

```
name;
<< 'JavaScript'
```

It may seem like a restriction to use `const`, but it actually helps make your programs more predictable if values can't change, and it helps to avoid any bugs caused by unexpected changes in assignment.

Now let's try declaring and assigning a value to a variable that *can* change. The next example shows how we would declare the variable `score` and assign it a value of `0`:

```
let score = 0;
```

The value of the variable `score` would initially start as `0`, but it would be able to change throughout the program.

Variables that have been declared using the `let` keyword can be reassigned to another value at any point later in the program. This is done by simply assigning them to a new value. For example, we could update the `score` variable to a value of `5` like so:

```
score = 5;
```

Note that you don't need to use `let` when you reassign a variable. It's only needed when you're declaring it for the first time.

Undefined

`undefined` is a primitive data type that JavaScript assigns to any variable that hasn't been explicitly assigned a value. It's basically JavaScript's way of saying "I can't find a value for this". For example, try declaring a new variable called `score` in the console, but don't assign a value to it:

```
let score;
```

Now take a look at that variable. You'll see that JavaScript has assigned it a value of `undefined`:

```
score;
<< undefined
```

Naming Variables

When naming constants and variables, you should try to give them sensible names that describe what the variable represents. For example, `characterSpeed` is a better variable name than `ac12`.

In JavaScript, variable names can start with any uppercase or lowercase letter, an underscore (_), or dollar character (`$`). They can also *contain* numbers, but they can't *start* with them.

Here are some valid variable name examples:

```
$name
_answer
firstName
last_name
address_line1
```

Variable names are case sensitive, so `ANSWER`, `Answer` and `answer` are all different variables.

When using multiple words for constant and variable names, there are two conventions that can be used:

- **camelCase** starts with a lowercase letter and then the first letter of each new word is capitalized:

  ```
  firstNameAndLastName
  ```

- **underscore** separates each new word with an underscore:

  ```
  first_name_and_last_name
  ```

JavaScript's built-in functions use the camelCase notation, and this is probably the best convention to follow when naming the variables in your

code. The most important thing to remember is to *be consistent*.

Symbols in Variable Names

It's generally not a good idea to use symbols such as `$` and _ at the beginning of your variable names, as a number of JavaScript libraries do this, which could end up making things really confusing. (A **library** is code written by someone else, for a specific purpose, which you can use along with your own code.)

Pop-up Interaction

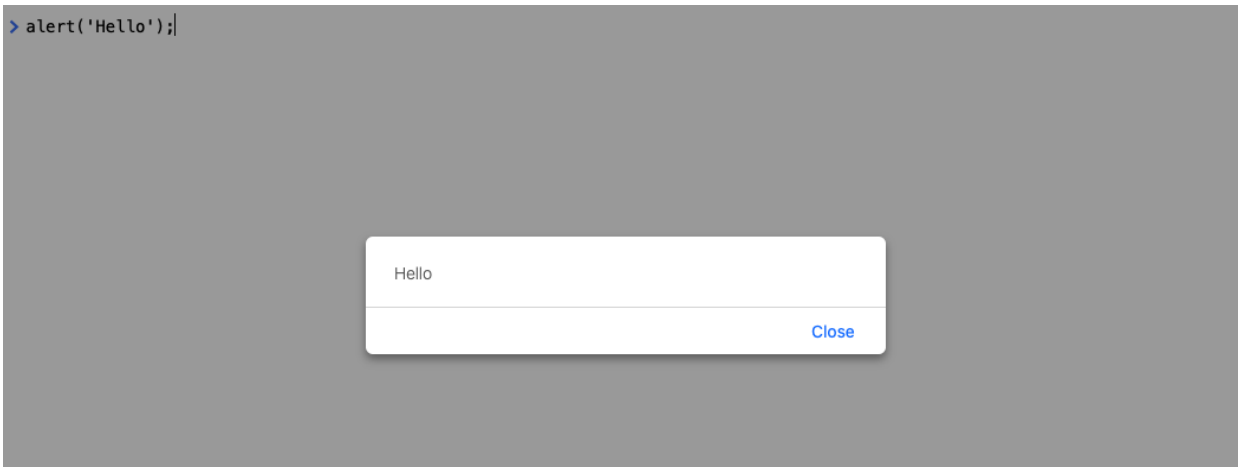
JavaScript provides three different types of pop-up dialog boxes that provide some simplistic interaction between the browser and the user.

Alert Box

An **alert** box can be used to display a message. The user has to click a button to remove it.

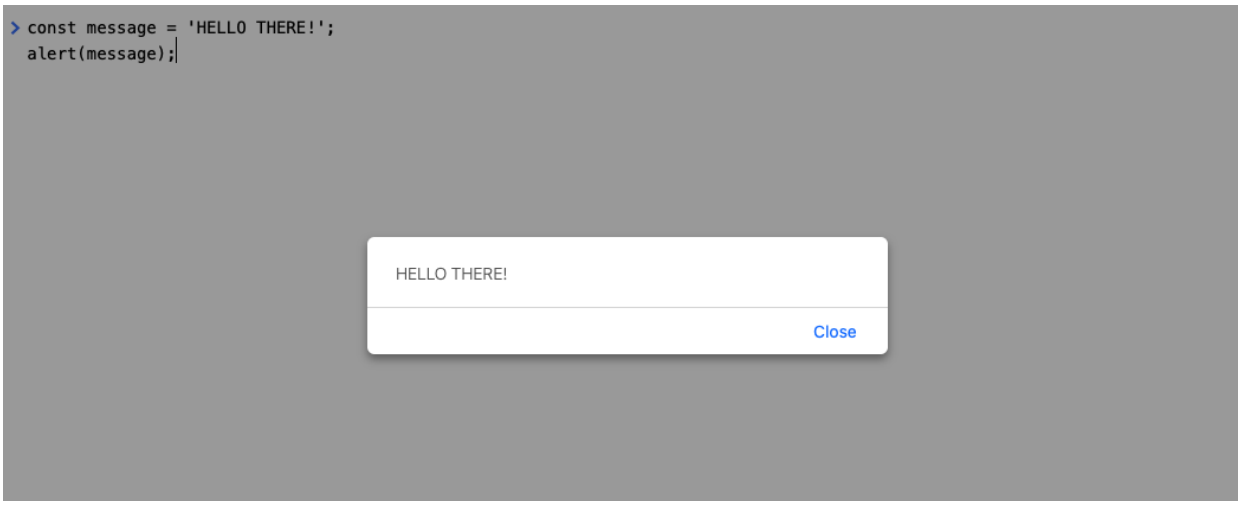
Here's an example that will display the message "Hello" in a dialog box:

```
alert('Hello');
```



You can also use `alert()` to display the value of a variable:

```
const message = 'HELLO THERE!';
alert(message);
```



Prompt Box

A **prompt** box allows the user to enter a response, which can then be stored in a variable. In the following example, the user is asked to enter their name and store their response in a variable called `name`:

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

Whatever is entered in the prompt box will now be stored in the variable `name`. We can type `name` into the console to see that it now holds the value that was entered into the prompt box.

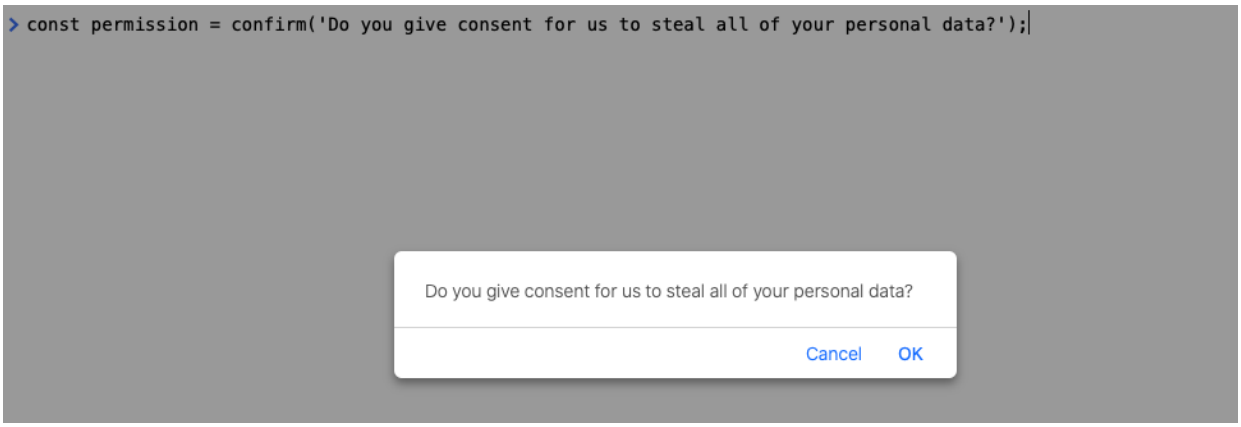
```
> const name = prompt('What is your name?');
<· undefined
> name
<· "JavaScript"
>
```

Confirm Box

A **confirm** box allows the user to confirm or cancel a request. The response can be stored in a variable as `true` if the user clicks “okay” or `false` if they click “cancel”.

For example, the following code could be used to confirm if a user gives consent to use their personal data:

```
const permission = confirm('Do you give consent for us to steal all of your personal data?');
```



The response will be stored in the variable `permission`, which can be seen by typing this into the console.

In practice, most users find these pop-up boxes annoying, and their use is definitely not recommended in most practical situations. But they do give us a convenient way to provide some interactivity and get information from a user—at least until we learn about more advanced techniques later in this book.

Hello `name`

Now that we've learned all about variables and how to use pop-up boxes to collect information from a user and store it in a variable, let's try writing some code that will deliver a personalized greeting to a user.

Open up CodePen and add the following code in the **JS** section:

```
// ask the user for their name
const name = prompt('Please enter your name.');
// say hello
alert('Hello');
// then personalize it!
alert(name);
```

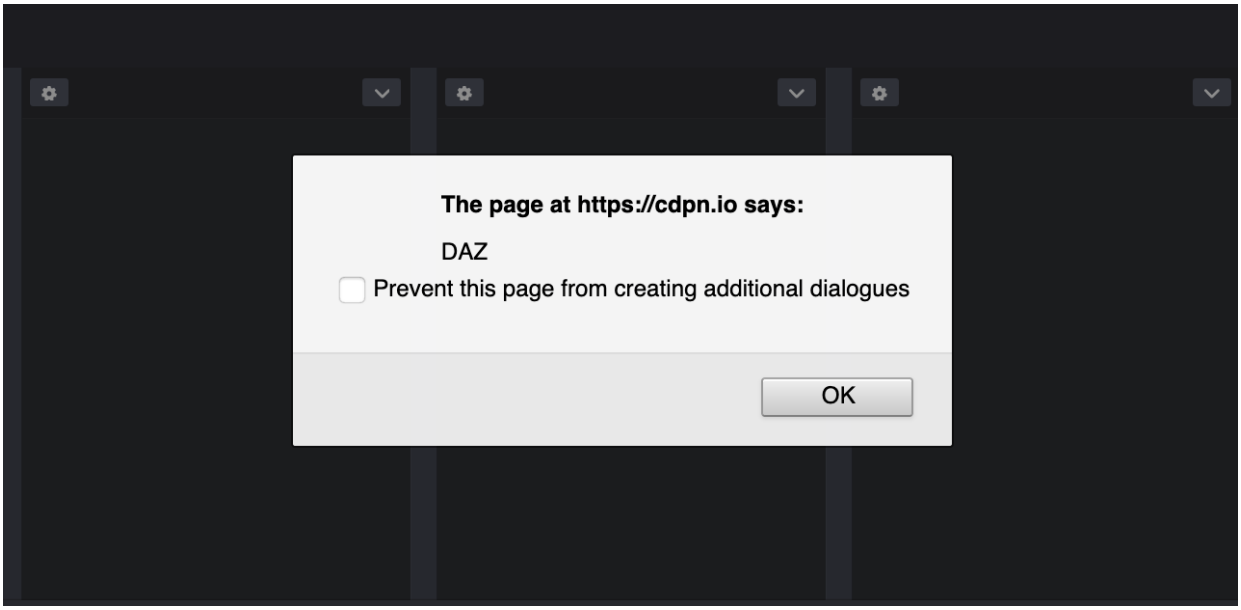
Hopefully the comments explain what's happening in each line of the code: we use a prompt box to ask for the user's name, then we use an alert box to say hello, followed by a second alert box that personalizes the greeting.

Auto-updating Preview

CodePen has a feature called "Auto-Updating preview" that constantly runs the code as you write it. This can get annoying if your code contains pop-up boxes, as you need to close them to continue coding … and they won't stop

popping up! There's an easy fix for this: when in a Pen, just click on the **Settings** tab (top right) > **Behavior** and turn off **Auto-Updating preview**. You'll then get a **Run** button that you can press to run your code once you've finished writing it all.

Try running the code. You should see something similar to this:



You can see my code for this here.

Challenges

1. Add comments to the "I Can Code a Rainbow" program from the last chapter to explain what happens in each step. You can see my code on CodePen.

2. Find out the type of the following values. Can you guess them in advance?

```
42
'Twenty20'
'123'
-68
true
FALSE
```

```
'true'
0
```

You can see my code on CodePen.

3. Use a prompt or confirm box to collect some information from the user and store it in a suitably named variable. Then use an alert box to display a message that uses the variable in some way. You can see my code on CodePen.

Summary

- Comments are ignored by the program, but make your program easier to read and understand.
- Primitive types are the basic building blocks of a program.
- Primitive types in JavaScript include strings, numbers and Booleans.
- Composite data types are structured collections of primitive data types.
- Composite data types in JavaScript include arrays and objects.
- Variables point to values stored in memory and are declared using the `const`,`let` or `var` keywords.
- Values are assigned to variables using the `=` operator.
- The value of any variables declared using `const` can't be changed.
- You can reassign values to variables declared using `let` or `var`.
- Alert, prompt and confirm boxes can be used to add some interaction between a web page and the user.

In the next chapter, we'll be looking more closely at how strings are used to display letters and words.

Chapter 3: Letters and Words

In this chapter, we'll be looking at how letters and words are represented and used in programming languages. We'll also look at how JavaScript represents them using strings. We're going to cover the following:

- chars and strings
- strings in JavaScript
- escaping values
- finding characters
- string length
- string arithmetic
- string methods
- template literals and interpolation

Chars and Strings

As we saw in the previous chapter, most programming languages have primitive data types of char (a single character) and strings (a collection of characters). In fact, you can almost think of a string as a collection of characters joined together by a piece of string:

JavaScript only has a primitive data type of String, but this can be used to represent both single characters as well as words and longer blocks of text. Here are a few examples of some strings:

```
'M'
'@'
`Hello`
"Easy as 1,2,3"
```

```
`What The F*!#?`
"A long time ago, in a galaxy far, far away."
'   '
''
```

Empty Strings

The last example is just an empty string, with nothing inside it, not even a space. This is still considered a string.

Strings can be used to represent words, paragraphs of text and even markup such as HTML. Nearly all data that's entered by a user will initially be stored as a string.

Creating Strings in JavaScript

It's really easy to create a string. You simply write it in quotation marks. Open up a console and try entering the following string:

```
'Hello';
<< "Hello"
```

This is called a **string literal**, because a literal representation of the value is written out in the code. You can use double or single quote marks to create a string literal. Try the following example in the console:

```
"I'm also a string literal."
<< "I'm also a string literal."
```

Double quote marks are useful if you want to use single quote marks as apostrophes in the string. If you were to use single quotes around the string above, the apostrophe would terminate the string, causing an error, as can be seen in the following code:

```
'Don't do this.'
<< SyntaxError { Unexpected token, expected ; (1:19) }
```

But using double quote marks will fix it:

```
"That's the way to do it."
<< "That's the way to do it."
```

Escaping Values

It's also possible to **escape** quotation marks. This is done by placing a backslash before the apostrophe so that it appears as an apostrophe inside the string instead of terminating the string. Try entering the following into the console:

```
'It\'s okay if you escape the apostrophe.'
<< "It's okay if you escape the apostrophe."
```

The backslash can be used to insert special whitespace into strings, such as the following:

- `\n`: end of line
- `\r`: carriage return
- `\t`: tab

If you want to actually write a backslash, you need to escape it with another backslash:

```
"This is a backslash \\"
<< "This is a backslash \"
```

Find the Char

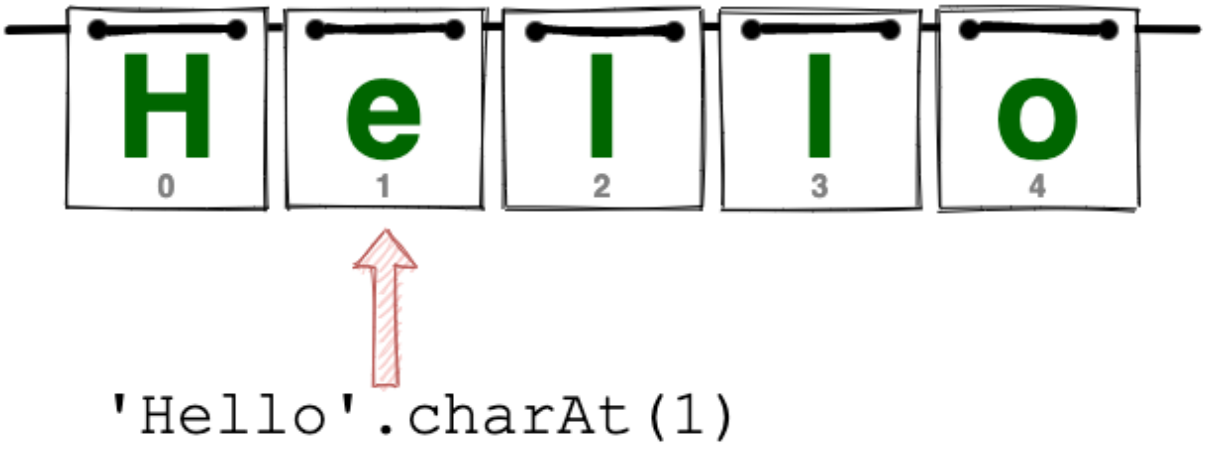
Even though JavaScript doesn't have a char data type, it can still separate each string into individual characters.

You can find out a particular character in a string by using the `charAt()` method. We can apply methods to strings using **dot notation**. This involves writing a dot followed by the method we want to use. For example, the following code will tell us which character is at position 1 in the string `'Hello'`. Try entering it into a console:

```
'Hello'.chartAt(1);
<< "e"
```

In this example, the number 1 in the parentheses is used to indicate the position of the character we want to know about.

The result above tells us that the character “e” is at position 1. If you’re wondering why it isn’t “H”, this is because the first character in a string is classed as being at position 0. (You’ll find that counting usually starts at zero in programming!)



There’s also a shortcut notation for referring to individual characters in a string using square brackets. For example, the following code will tell us the first letter of the string:

```
'Hello'[0];
<< "H"
```

This notation gives us a convenient way of referencing each individual character inside a string.

Finding Chars

If you want to find where a certain character or substring appears in a string, we can use the `indexOf()` method. This will return the position of the first occurrence of a character in the string. In the following example, we use it to find that the character “l” first appears at position 2 (remember that counting starts at zero, so this means it’s the third character in the string):

```
'Hello'.indexOf('l');
<< 2
```

If a character doesn’t appear in the string, `-1` will be returned:

```
'Hello'.indexOf('z');
<< -1
```

If we want the last occurrence of a character or substring, we can use the `lastIndexOf()` method instead. For example, this shows us that the character "l" last appears at position 3 in the string `'Hello'`:

```
'Hello'.lastIndexOf('l');
<< 3
```

If we only need to know if a string contains a certain character, we can use the `includes()` method. This will return the Boolean values of `true` or `false`, depending on whether the character is in the string or not, as can be seen in the code examples below:

```
'Hello'.includes('e');
<< true

'Hello'.includes('z');
<< false
```

We can also check to see if a string *starts* with a certain character. To do so, we can use the `startsWith()` method. Be careful, though, as it's case-sensitive:

```
'Hello'.startsWith('H');
<< true

'Hello'.startsWith('h');
<< false
```

And we can use the similar `endsWith()` method to check if a string ends with a particular character:

```
'Hello'.endsWith('O');
<< false

'Hello'.endsWith('o');
<< true
```

How Long Is a String?

Every string has a `length` property that tells us how many characters it contains. This property is also accessed using the dot notation we used with the `charAt()` method. For example, the following code will tell us how many characters are in the string `'Hello'`:

```
'Hello'.length;
<< 5
```

You can also assign a string to a variable and then apply the dot notation to the variable:

```
const myString = "Hello, is it me you're looking for?";
myString.length;
<< 35
```

As you can see, this tells us that there are 35 characters in the string that has been assigned to the `myString` variable.

All properties of primitive data types are **immutable**. This means that they can't be changed, so it's impossible to change the length property of a string by reassigning it to another value. You can try, but your efforts will be futile:

```
myString.length = 36; // try to change the length property
<< 36
```

Although it looks like the length property of the `myString` variable has been changed to 36, this isn't the case. We can see this by having another look at the value of the `length` property:

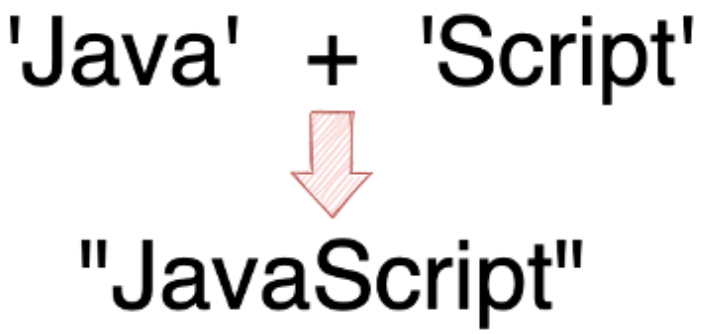
```
myString.length; // check to see if it's changed
<< 35
```

String Arithmetic

Most programming languages let you add two strings together to produce a longer string. JavaScript is no exception and even makes it look just like a mathematical calculation by using the `+` symbol! Try the following example in a console:

```
'Java' + 'Script';
<< "JavaScript"
```

As you can see, the result combines the two strings together into a single string. This is known as string **concatenation**.



Watch out, though. Concatenation doesn't insert spaces for you, so the following code won't quite work as you might expect it to:

```
'Hello' + 'World';
<< "HelloWorld"
```

To add a space, you could either add a space to the end of the first word or to the beginning of the second, like so:

```
'Hello ' + 'World';
<< "Hello World"

'Hello' + ' World';
<< "Hello World"
```

You could also add a string containing a single space in the middle, like so:

```
'Hello' + ' ' + 'World';
<< "Hello World"
```

The `concat()` Method

There's also a method called `concat` that will also concatenate two strings:

```
'Java'.concat('Script');
<< "JavaScript"
```

Finding the Last Character in a String

There are times when we'll want to know what the last character of a string is. That's easy enough if we know what the string is. For example, we can find the last character of the string `'Hello'` using the following code:

```
'Hello'[4];
<< "o"
```

Remember that counting starts at zero, so the fifth letter "o" is at position 4. If we want to find the last character of the string `'Goodbye'`, we can use the following code:

```
'Goodbye'[6];
<< "e"
```

So the value we place in the square brackets changes depending on the length of the string.

As said, this is all fine if we know the value of the string. But in real life, we *often won't know the value of the string*. For example, the string value might be collected from a user via a form, or created on the fly during the operation of the program.

Thankfully, we can use the `length` property to help us find the last character of a string, as we'll see next.

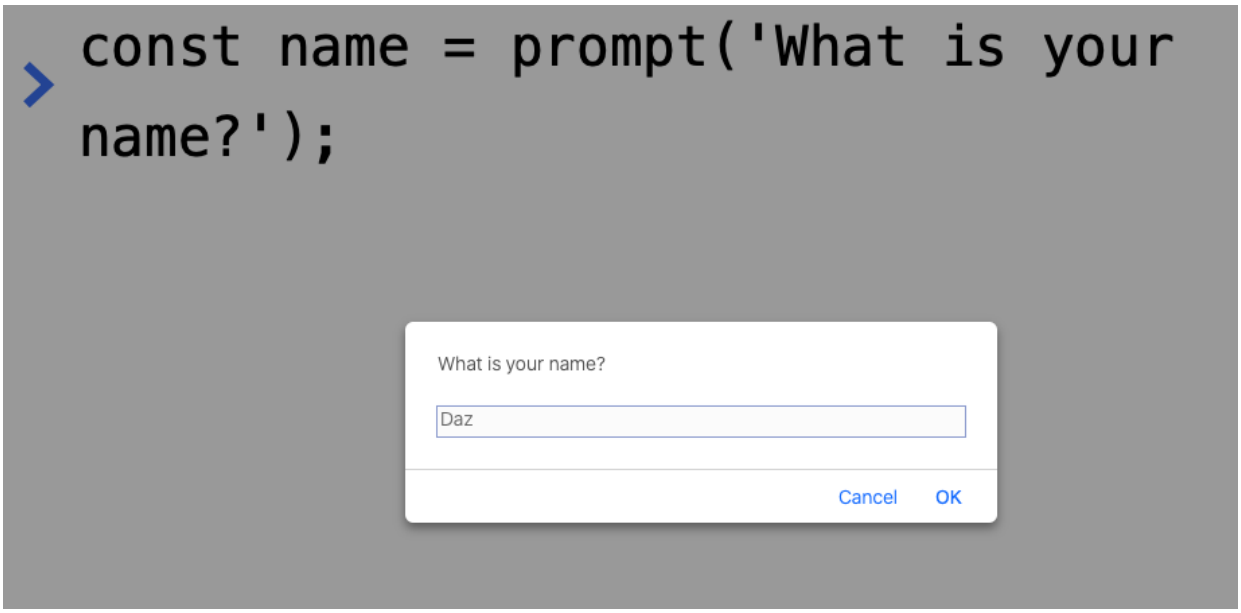
What's In a Name?

Let's try writing some code that will ask a user for their name and then tell them the first and last letter of their name.

At the end of the last chapter, we learned how we can collect information from a user using a prompt box and store it in a variable. Our first line of code will create a prompt box, asking for their name:

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

If you enter the code above into a console, you should see something similar to the following appear on your screen.

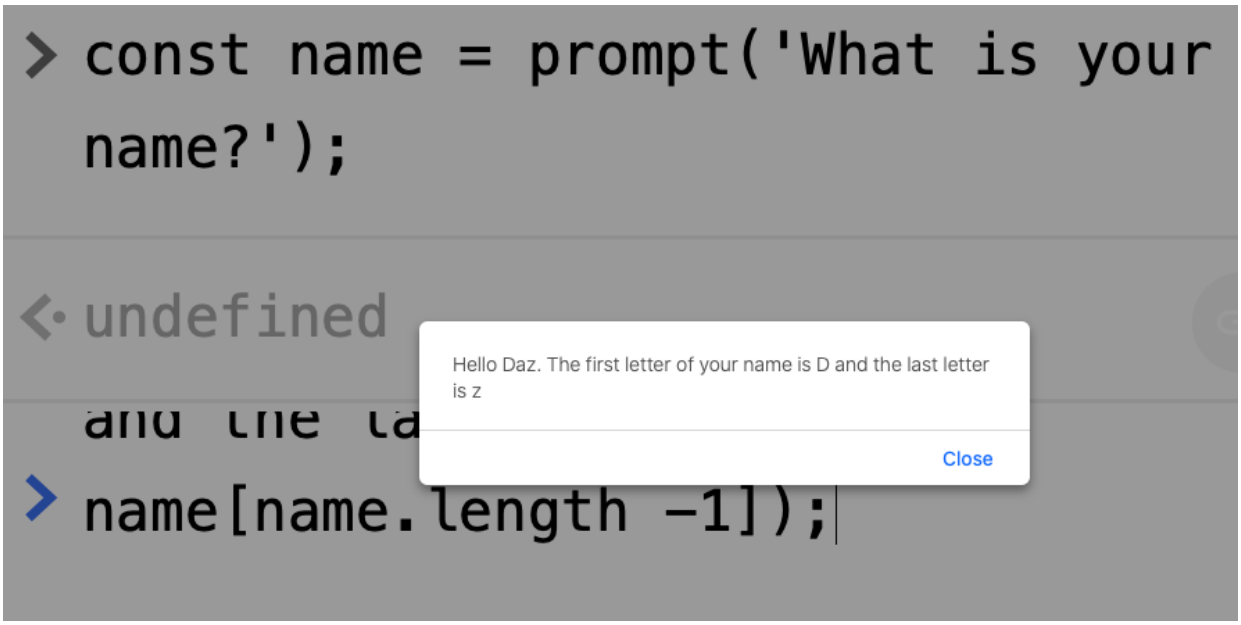


Enter your name into the text box and click **OK**. Whatever you entered will now be stored in the variable `name` as a string. (You can check this is the case by typing `name` into the console, then pressing `Enter`.)

Now let's create an alert box that will show off our string manipulation skills:

```
alert('Hello ' + name + '. The first letter of your name is ' + name[0] + ' and the last letter is ' + name[name.length -1]);
```

If you run this code, you should get something similar to what's pictured below.



Notice how we found the last letter of the `name` variable using `name[name.length - 1]`. This works out the length of the string (remember, we don't know how long it is until the user enters their name) and then subtracts `1` (to take account of the fact that the position of characters starts at zero).

You can see my code on CodePen.

Changing Cases

It's possible to change the case of a string to all uppercase letters or all lowercase letters. To demonstrate this, let's assign the string `'JavaScript'` to the variable `name` in the console:

```
const name = 'JavaScript';
```

Now we can return a string of all uppercase letters using the `toUpperCase()` method:

```
name.toUpperCase();
<< 'JAVASCRIPT'
```

Or we can return a string of all lowercase letters using the `toLowerCase()` method:

```
name.toLowerCase();
<< 'javascript'
```

The More Things Change …

These two methods don't actually change the string stored in the `name` variable. They just return a value with the case changed. If you check the value of the `name` variable, it should still be the same as when we declared it:

```
name;
<< 'JavaScript'
```

Trimming Space

The `trim()` method can be used to remove any whitespace from the beginning and end of a string. This is useful when users entering information into a form inadvertently add spaces at the beginning or end. The example below shows that the spaces at the beginning and end of the string are removed, but the space in the middle is preserved:

```
'    Hello World     '.trim();
<< 'Hello World'
```

It will even get rid of any tabs or carriage returns, as can be seen in the following example:

```
'   \t\t  JavaScript! \r'.trim(); // escaped tabs and carriage
returns are also removed
<< 'JavaScript!'
```

If you only want to remove the whitespace from the beginning or just from the end, there are more specific methods for doing just that:

```
'    Hello World     '.trimStart(); // removes whitespace from
the beginning of a string
<< 'Hello World     '

'    Hello World     '.trimEnd(); // removes whitespace from the
end of a string
<< '    Hello World'
```

More Methods

There are loads more things that strings can do. A full list of properties and methods can be found on the Mozilla Developer Network.

Template Literals

Template literals are special types of strings in JavaScript that use the backtick character (`) to delineate the string, as shown in the example below:

```
`Hello!`;
```

One advantage of using template literals is that you can then use both types of quote mark within the string:

```
`She said, "It's Me!"`
```

More importantly than this, though, they also allow **interpolation** of JavaScript code. This means that a piece of code can be inserted inside a template literal and the result will be displayed in the resulting string.

In the example below, we use interpolation to insert a variable called `name` into a template literal:

```
const name = `World`;
`Hello ${ name }!`;
<< "Hello World!"
```

Notice that the variable is replaced with the value of "World" in the resulting string.

To use interpolation, a JavaScript expression needs to be placed inside curly braces with a `$` character in front of them: `${ // JS expression here }`.

The expression is then evaluated, and the result is **interpolated** (that is, placed) into the resulting string. It doesn't just have to be a variable that goes inside the curly braces; you can use any code. In the example below, we also use the `toUppercase()` method:

```
const name = `World`;
`Hello ${ name.toUpperCase() }!`;
<< 'Hello WORLD!'
```

Template literals can also contain line breaks (by just pressing `Enter`), which are all converted into a line feed character (`\n`). Try entering the example below with some line breaks in the middle:

```
`This is the start ...


.... and this is the end`
<< 'This is the start ...\n\n\n.... and this is the end'
```

If you instead try to insert line breaks by just pressing `Enter` in a normal string, you'll get an error:

```
"This is the start ...


.... and this is the end"
<< SyntaxError { Unterminated string constant (1:14) }
```

If you want to place a backtick inside a template literal, it needs to be escaped in the usual way, using a backslash:

```
`This character, \`, is a backtick`
<< 'This character, `, is a backtick'
```

Template literals can be thought of as superpowered strings, as they behave in the same way as normal string literals, but with the extra superpower of string interpolation. For this reason, it's not uncommon to see backticks used to create *all* strings in modern JS code.

Mad Libs

We'll use the power of template literals to finish this chapter with a simple Mad Libs game. **Mad Libs** is a fun party game where you choose random words that get inserted into a predefined sentence, often with humorous or just plain crazy results! The adult game Cards Against Humanity uses a similar concept. (Feel free to change the code to be more like this, if that's how you roll!)

Fire up CodePen and add the following code into the JS section:

```
// ask the user for some words....
const animal = prompt('Please enter an animal.');
const color = prompt('Please enter a color.');
const verb = prompt('Please enter a verb.');
const job = prompt('Please enter a job.');
```

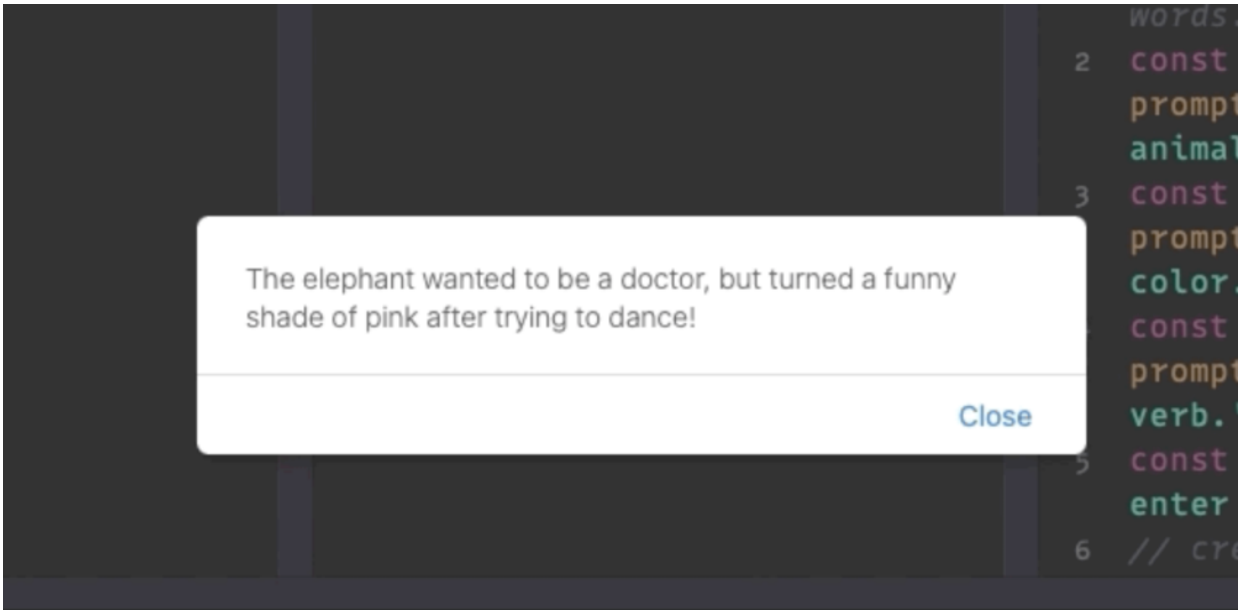
This uses prompt boxes to ask different questions and store the user's responses in relevant variables. Now we need to create the actual Mad Lib. We'll use a template literal to insert the variables we've just created into a string that's assigned to the variable `madlib`:

```
// create the Mad Lib
const madlib = `The ${animal} wanted to be a ${job}, but turned
a funny shade of ${color} after trying to ${verb}!`
```

Last of all, we need to show our hilarious creation to the user, using an alert box:

```
// Show the Mad Lib
alert(madlib);
```

Try running the code and see what sort of wacky sentences you can come up with. Below is a screenshot of my effort.



You can see my CodePen code here.

Challenges

Now that we've been introduced to strings and seen a little bit of what they can do, it's time for some coding challenges.

1. Use a prompt box to ask for a user's name and then use a template literal to insert their name into a personalized "hello" message in an alert box. You can see my code on CodePen.

2. Now try changing the code so that it tells the user how many letters their name contains. You can see my code on CodePen.

3. Try changing the Mad Libs code to ask for more words and create a funnier result. You must be able to do better than mine!

4. Try writing some code that asks for a user's name, then tells them their "swappy name" by swapping the first and last letter around. So if I enter "Daz" as my name into the prompt box, the alert box should say, "Hello Daz, your swappy name is Zad." (Note: this is harder than it sounds at first. You should probably investigate the `slice()` method for this challenge.) You can see my code on CodePen.

Summary

- Strings are collections of characters that are used to show blocks of text in JavaScript.
- Special values can be *escaped* by placing a backslash (`\`) in front of them.
- Strings have various properties and methods that provide information about them.
- The `length` property tells us how many characters there are in a string.
- Strings can be *concatenated* (joined together) using the `+` operator.
- Template literals are like superpowered strings, allowing JavaScript code to be inserted into a string.

Now that we've learned all about strings, it's time to learn all about numbers in the next chapter.

Chapter 4: Numbers

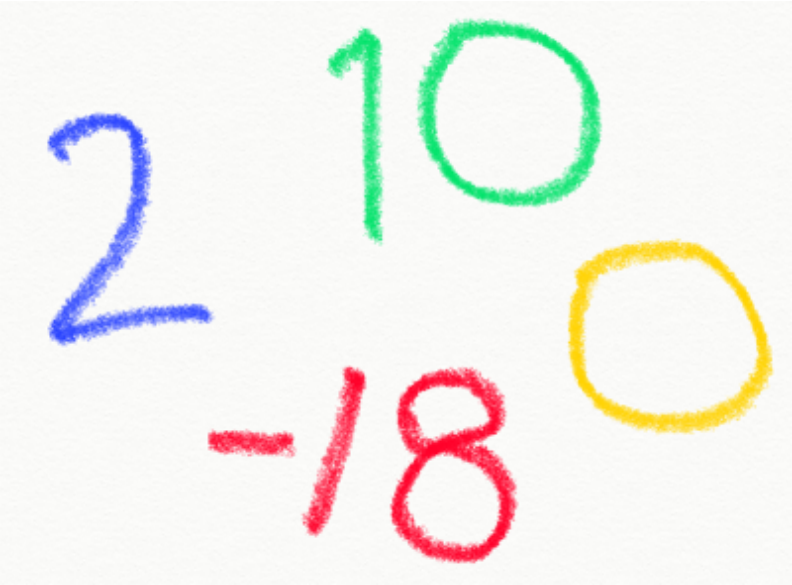
In Chapter 3, we learned all about strings. Now it's time to learn about numbers! In this chapter, we're going to cover the following:

- integers and floats
- numeric literals
- exponential notation
- arithmetical operations
- varying variables
- converting between numbers and strings
- random numbers

Integers and Floats

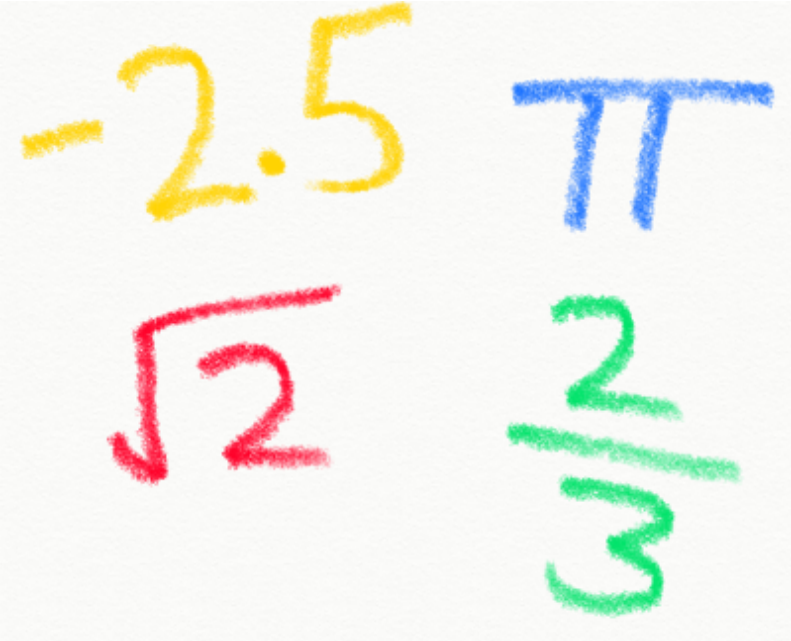
Numbers can be classified into various types.

Numbers can be **integers**, which are whole numbers, such as 3, -4 or 0.



Numbers can also be **floating-point numbers**—often referred to as just **reals**, **decimals** or **floats**. These include all numbers that have fractional or

decimal parts, such as 1.5, -3.8 or pi.



Most programming languages will have separate primitive data types for integers and floats, but JavaScript doesn't distinguish between them and just calls them both "numbers", as we can see if we use the `typeof` operator:

```
typeof 42; // this is an integer
<< 'number'

typeof 3.14159; // this is a floating-point decimal
<< 'number'
```

Numeric Literals

A **numeric literal** is a sequence of digits that form a decimal number and don't start with a leading zero. Creating a numeric literal in JavaScript is as simple as just writing the number in the console:

```
3;
<< 3

-4.67;
<< -4.67
```

Exponential Notation

Numbers can also be represented in **exponential notation**, which is shorthand for "multiply by 10 to the power of" (you may have heard this referred to as "scientific notation" or "standard form"). The following example shows how to write 1 multiplied by 10 to the power of 6, which is a million:

```
1e6;
<< 1000000
```

The next example returns 2 multiplied by 10 to the power of 3, or 1000:

```
2E3;
<< 2000
```

Fractional decimal values can be created by using a negative index. The following example returns 2.5 multiplied by 10 to the power of negative 3, which is the same as 2.5 multiplied by 0.001:

```
2.5e-3;
<< 0.0025
```

When Is a Number Not a Number?

`NaN` is a special error value that's short for "Not a Number". It's used when an operation is attempted and the result isn't numerical, like if you try to multiply a string by a number, for example:

```
'hello' * 5;
<< NaN
```

Here's a little JavaScript joke, just for some light relief. Let's see what data type `NaN` is. Enter the following code in a console:

```
typeof NaN;
<< 'number'
```

Somewhat ironically, given that `NaN` stands for "Not a Number", JavaScript treats `NaN` as … a number!

Arithmetic Operations

Most programming languages can carry out all the usual arithmetic operations—just like pocket calculator! Calculations can be set out as you'd expect, and they often use the operators and symbols we're familiar with. JavaScript is no exception. Let's try doing some math in the console.

Addition uses the `+` operator, as you'd expect:

```
5 + 4.3;
<< 9.3
```

This example shows that you can mix integers and floats in calculations.

Subtraction uses the `-` operator, and there are no problems dealing with negative numbers:

```
6 - 11;
>> -5
```

Multiplication uses the `*` operator:

```
6 * 7;
<< 42
```

Division uses the `/` operator, as if it was a fraction:

```
3/7;
<< 0.42857142857142855
```

This answer should actually be a recurring decimal, but JavaScript will truncate the answer (usually with a slight rounding error at the end).

Exponentiation can be carried out using the `**` operator. The following code will return 2 to the power of 3:

```
2**3;
<< 8
```

The same rules of precedence of arithmetic will be used in calculations, so exponents are calculated first, then multiplication and division are performed

before addition and subtraction. But you can use parentheses to change the order that operations are performed in. In the example below, the sum in the parentheses is completed first, then the division and finally the addition:

```
(8-5) + 6/3;
<< 5
```

You can also calculate the remainder of a division using the `%` operator. The following calculation will return the remainder when 23 is divided by 6:

```
23%6;
<< 5
```

The answer of 5 is returned because 6 divides into 23 three times, with a remainder of 5. The operator only returns the remainder.

Why would this ever be useful? Well, it comes in handy quite often in programming when anything repeats over and over again. For example, we can use it to find out what day it will be a million days from now using the following code (recall that `1e6` is a million in exponential notation):

```
1e6%7;
<< 1
```

This means that when 7 divides into a million there's a remainder of 1. What does that have to do with what day it is? Well, since there are 7 days in a week, and they keep repeating over and over every week, that means that even though the calculation hasn't told us how many full weeks are in a million days, it has told us that there will be a remainder of 1 day left over, so in a million days, it will be the same day as tomorrow!

Varying Variables

If a variable has been assigned a numerical value, it can be modified using different operators. For example, say we're making a game that keeps track of the number of points you've scored in a variable called `score`. First of all, we'd initialize the score to zero. Fire up a console and follow along with these examples:

```
let score = 0;
<< 0
```

One way of increasing the score would be to just add a value to it like so:

```
score = score + 10;
<< 10
```

This will increase the value held in the `score` variable by `10`.

The notation can seem strange at first, as the left-hand and right-hand sides are not equal, but remember that the `=` symbol is used for *assignment*, and we're assigning the `score` variable to its current value plus another 10.

There's a shorthand for doing this, called the **compound assignment operator**, `+=`:

```
score += 10;
<< 20
```

There are equivalent compound assignment operators for all the arithmetical operators that we saw in the previous section. For example, you can decrease the score by 5 like so:

```
score -= 5;
<< 15
```

The following code will multiply the value of `score` by 2—or, in other words, double it:

```
score *= 2;
<< 30
```

You can also divide the current value of `score` by a value. This code will divide it by 3:

```
score /= 3;
<< 10
```

You can also raise the value of `score` to a power. The following code will raise the value of score to the power of 2, which is the same as squaring it:

```
score **=2;
<< 100
```

You can also use the remainder operator. The following code will change the value of `score` to the remainder if its current value was divided by 7:

```
score %= 7;
<< 2
```

Increments

If you only want to increment a value by 1, you can use the increment operator, `++`. This goes either directly before the variable (prefix operator) or after it (postfix operator). Try entering the following code into the console to see how these work:

```
let points = 5;

points++;
<< 5

++points;
<< 7
```

So what's going on here? Both operators increase the value of `points` by 1, even though it doesn't immediately look like the first one has changed anything. The difference is when the increment takes place in relation to the value that's returned.

In the first operation, `points++` returns the original value of `score`, 5, *then* increases it to 6, whereas `++points` increases the value by 1, then returns the new value of 7.

There's also a `--` operator that works in the same way. The following example will return 7, then decrease the value of `points` by one to 6:

```
points--;
<< 7
```

And the following example will reduce the value of `points` by one first, then return that value:

```
--points;
<< 5
```

How Old?

Now that we've learned about different calculations and changing variables, it's time to do some coding. We're going to write a short program that will ask the user for their age and then convert this from years into in seconds.

Create a new Pen on CodePen and enter the following in the **JS** section (and since we're using prompt and alert boxes, don't forget to turn off **Auto-Updating Preview** in **Settings** > **Behavior**):

```
const ageInYears = prompt('How old are you (in years)?');
```

This will store the user's answer in a variable called `ageInYears`. Now let's convert that into seconds:

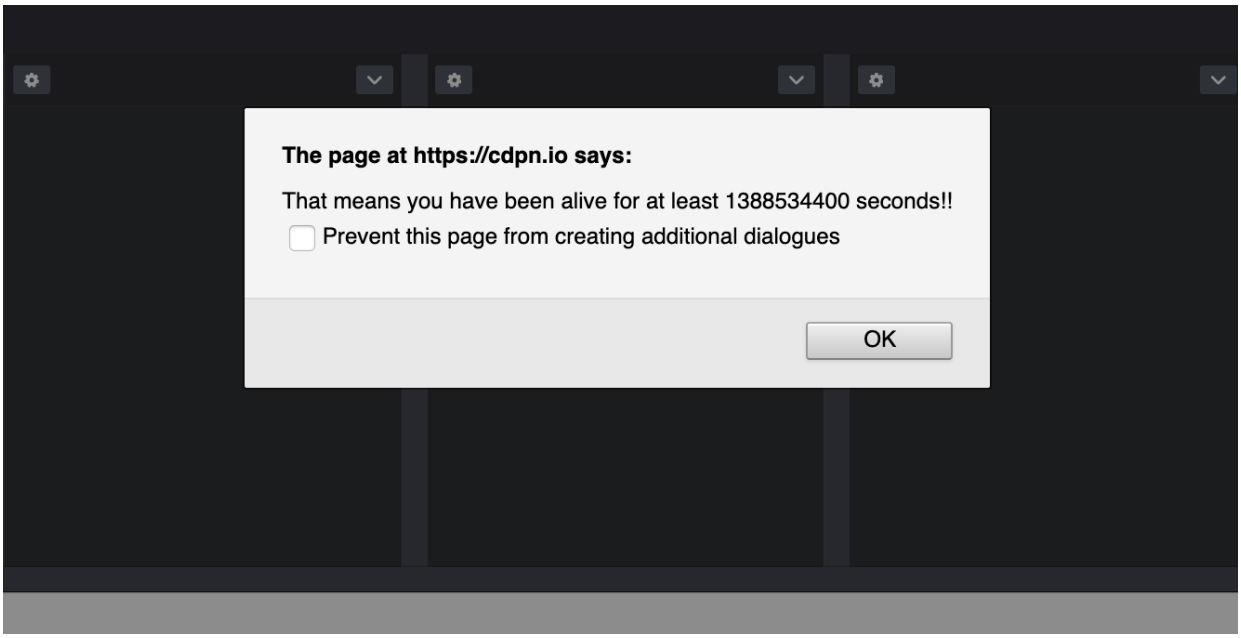
```
const ageInSeconds = ageInYears * 365.25 * 24 * 60 * 60;
```

There are approximately 31,557,600 seconds in a year, but we don't need to bother remembering that. All we need to do is multiply by 365.25, which is the approximate number of days in a year (the 0.25 is a slight overestimate and is the reason why we have leap years) to convert it into days, then multiply by 24 to convert into hours, then by 60 to convert into minutes and finally by 60 again to convert it into seconds.

Finally, all we need to do is use an alert box to inform the user of the result:

```
alert(`That means you have been alive for at least
${ageInSeconds} seconds!!`);
```

Click on **Run** to see it in action. It should look something like this:



You can see my code on CodePen.

Calculations with Numbers and Strings

Is it possible to add a number and string together? It seems a ridiculous question, but JavaScript will try its best to answer the question, as can be seen in the example below:

```
2 + 'two';
<< "2two"
```

What has happened here? The number `2` has been turned into the string `'2'`, and this has been concatenated with the string `'two'` like we saw in the last chapter. This process is called **type coercion**, and it occurs when the operands of an operator are of different types. JavaScript will attempt to convert one of the operands to an equivalent value of the other operand's type. For example, if you try to multiply a string and a number together, JavaScript will attempt to coerce the string into a number and then multiply them together:

```
'2' * 8;
<< 16
```

This may seem useful at first, but the process is not always logical or consistent and can cause a lot of confusion. For example, if you try to *add* a string and a number together, JavaScript will convert the number to a string and then concatenate the two strings:

```
'2' + 8;
<< '28'
```

JavaScript is trying to be useful, but in actual fact, type coercion makes it very difficult to spot errors in your code.

The best approach is to take control of the situation and be very explicit about the types of values you're working with and avoid using values of different types in operations. Let's take a look at how to do this in the next section.

Converting Between Strings and Numbers

We can convert numbers to strings using the `Number` constructor. This will convert the string form of a number into a number literal, as can be seen in the example below:

```
Number('23');
<< 23
```

If the string can't be converted into a number, `NaN` is returned:

```
Number('hello');
<<< NaN
```

To change a number literal into a string literal representation of that number, we use the `String` constructor, as can be seen in the example below:

```
String(3);
<< '3'
```

The most common use for these conversions comes from the fact that anything entered by the user in a prompt box or form is collected as a string, even if the value entered is a number. For this reason, use of the `Number()`

method is recommended to ensure any variables that you intend to be numbers are in fact represented by numbers.

For example, the code we used earlier to calculate the number of seconds the user had been alive relied on type coercion to make it work. This is because the `ageInYears` variable would be stored as a string. It was only when we multiplied it by 365.25 that JavaScript would have converted it to a number in the background, and we were lucky that, in this case, it worked out as we'd hoped.

It would be better if we forced the `ageInYears` variable to be stored as a number from the start by updating the first line of code to the following:

```
const ageInYears = Number(prompt('How old are you (in years)?'));
```

By wrapping `Number()` around the prompt, we ensure that the value entered is stored as a number and will behave as expected.

Random Numbers

Every programming language provides a way of generating random numbers. These are always useful when coding, particularly when it comes to adding some element of chance to games!

Random … ish

It's actually very difficult to produce a truly random number, so what programming languages produce are technically only *pseudo-random numbers*. Various factors are often used as a "seed" to generate the number —such as the time, or the position of the mouse pointer. Although these are deterministic (that is, not strictly random), when taken together they make it very difficult to replicate the same conditions, making the numbers effectively random.

JavaScript can generate random numbers using the `Math.random()` method. This will generate a number between 0 (inclusive) and 1 (exclusive). Try entering the following code in a console:

```
Math.random();
<< 0.7881970851344265
```

Hopefully you didn't get the same value as I did—because it's a random number!

But a random decimal value isn't always what we want. In fact, most of the time we will want a random integer.

We can fix this by first of all multiplying the result of `Math.random` by a value, in order to increase the upper limit. For example, if we multiply by 6, then it will return a random number between 0 and 6 (but not including 6), as can be seen in this example:

```
6 * Math.random();
<< 4.280981240354013
```

Now all we need to do is use the `Math.ceil()` method to get rid of the decimal part. This is similar to the `Math.floor()` method that we saw in the I Can Code a Rainbow program in Chapter 1. But `Math.ceil` rounds the value *up* to the next biggest integer, while also removing the decimal part of the number. For example, the following code will round up to 5:

```
Math.ceil(4.280981240354013);
<< 5
```

This means that the following code will generate a random number between 1 and 6:

```
Math.ceil(6 * Math.random());
<< 4
```

If this reminds you of rolling a dice, that's no coincidence. We're going to finish this chapter by writing some code to mimic the rolling of a dice.

Open up a new Pen on CodePen and enter the following code in the **JS** section:

```
const sides = prompt('How many sides does the dice have?');
```

This will store the number that the user enters in a variable called `sides`. Now let's add an alert message to get the user to "roll" the dice:

```
alert('Press Enter or click close to roll the dice...');
```

Next comes the code to create the random number:

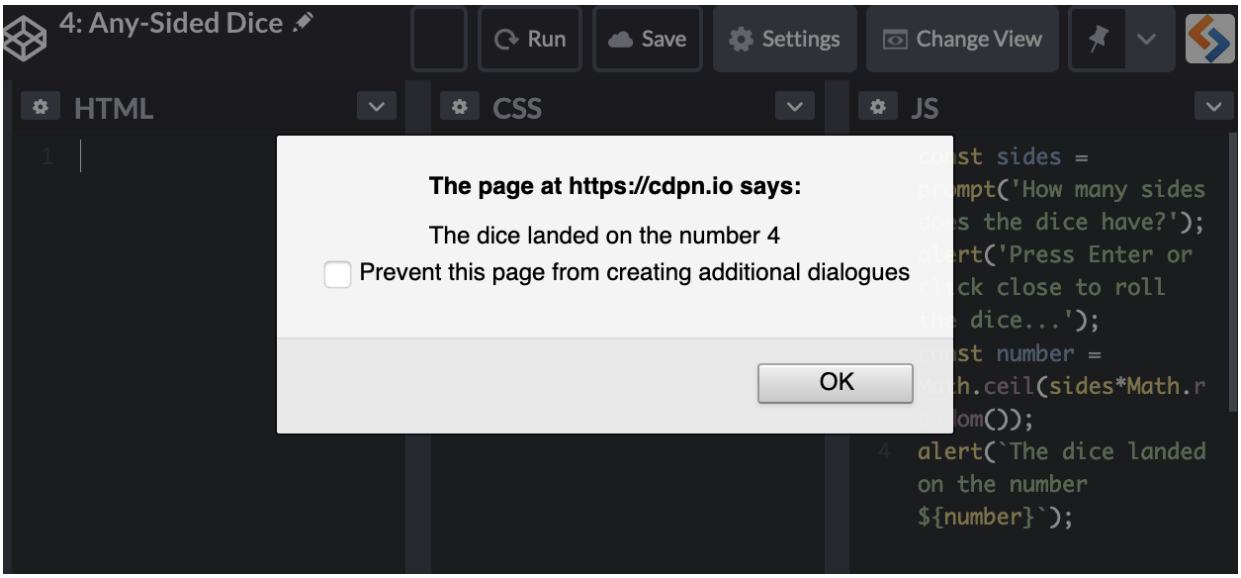
```
const number = Math.ceil(sides*Math.random());
```

This is similar to the code we used earlier, but we're multiplying `Math.random()` by the number of sides the user specified. This will produce a random decimal number between zero and one less than the number of sides. Applying `Math.ceil` will round that number up to the next integer, effectively making it a random number between 1 and the number of sides.

Now we just need to tell the user what the number was (again we're using a template literal to insert the number directly into a string):

```
alert(`The dice landed on the number ${number}`);
```

Click **Run** to test this. It should look something like this:



You can see my code on CodePen.

More Methods

There are loads more things that numbers can do. A full list of properties and methods can be found on the Mozilla Developer Network.

The `Math` object also has a large number of properties and methods that you might find useful. Again, you can see them all on the Mozilla Developer Network.

Challenges

1. Create a prompt box that asks the user for their age. Then tell them how old they'll be in 25 years. (Note that you might run into some problems with type coercion here, since the numbers entered in the prompt box are actually stored as strings.) You can see my code on CodePen.

2. Write some code that will use prompt boxes to ask the user for two numbers, then show an alert box that displays the result of multiplying those two numbers together. You can see my code on CodePen.

3. Can you write some code to split a bill for a meal? You'll need to use a prompt box to ask how much the meal was, and another to ask how many people it needs splitting between. Then an alert box needs to say how much each person has to pay. For this challenge, you'll need to think about the units and rounding. Bonus points if you can add an extra question that adds a tip as a percentage! You can see my code on CodePen.

4. Write some code that will produce a random number between two values. For example, can you write some code that will return a random number between 5 and 10 inclusive? You can see my code on CodePen.

Summary

- Numbers can be integers such as 2 or floats (decimals) such as 7.8, although JavaScript doesn't differentiate between them and just has a single primitive data type of `Number`.
- Numbers can be written in exponential notation such as `5e6` for five million.

- The arithmetic operations +, -, *, / and % can be applied to any two numbers.
- It's possible to change strings to numbers and vice versa.
- If you try to perform operations between a string and a number, JavaScript will attempt to convert the type to complete the operation. This can cause some inconsistent and unexpected results!
- A random number between 0 and 1 is returned by the `Math.random()` method.

Now that we've learned about strings and numbers, it's time to look at how JavaScript deals with collections of values.

Chapter 5: Collections

So far, we've met primitive data types such as strings, numbers and Booleans. But what if we want to store a whole collection of values in a single variable? For example, we might have a list of prices that we want store in the same place. **Composite data types** can be used to collect primitive values in a structured way.

Programming languages use a variety of different data structures to store values, but one of the most common is an array.

In this chapter, we'll be covering the following:

- creating arrays
- adding values to arrays
- removing values from arrays
- the spread operator
- multi-dimensional arrays

Arrays

An **array** is an ordered list of values. For example, consider the following shopping list:

- Apple
- Banana
- Cupcake

This could be represented as the following array:

```
['Apple', 'Banana', 'Cupcake']
```

Each value in the array has a numerical index that shows its position in the array. In the example above, `'Apple'` has an index of 0 (remember that computers start counting at zero!), `'Banana'` has an index of 1, and `'Cupcake'` has an index of 2.

Arrays can contain any type of value, such as numbers:

```
[2, 3, 5, 7, 11]
```

Or strings:

```
['Dog', 'Cat', 'Rabbit']
```

Or Booleans:

```
[ false, true, true, false, true ]
```

In most languages, including JavaScript, you're not restricted to using the same types of items inside arrays either. This array contains a variety different data types:

```
[ null, 1, 'two', true ]
```

Arrays in JavaScript

Now that we've been introduced to arrays, let's try coding some of them in JavaScript! To create an **array literal**, you simply write the values, separated by commas, inside a pair of square brackets. For example, we can create the shopping list example that we met earlier by entering the following in a console:

```
const shopping = ['Apple', 'Banana', 'Cupcake'];
```

To access a specific value in an array, we write its position in the array in square brackets. (This is known as its **index**.) For example, the following code will return the first item in the array:

```
shopping[0];
<< 'Apple'
```

(I hope you've remembered by now that the first item has an index of zero!)

If we wanted to see the value at index 2 of the array (the third item in the shopping list), we'd use the following code:

```
shopping[2];
<< 'Cake'
```

If an element in an array is empty, `undefined` is returned, as can be seen if we try to see what's at index 5:

```
shopping[5]; // there's nothing at position 5
<< undefined
```

Adding Values to Arrays

If we want to add a new item to an array that we've already created, we can simply assign the value to the position that we want it to go in, using index notation. For example, the following code will add `'Donut'` at position 3 in the array (which is the fourth item in the shopping list, remember):

```
shopping[3] = 'Donut';
```

We can take a look at the contents of the `shopping` array by simply typing its name into the console:

```
shopping;
<< ['Apple', 'Banana', 'Cake', 'Donut']
```

Each item in an array can be treated like a variable. You can change the value using the assignment operator `=`. For example, we can change the value at position 2 from `'Cake'` to `'Carrot'` using the following code:

```
shopping[2] = 'Carrot';
```

Let's check that this change has been made:

```
shopping;
<< ['Apple', 'Banana', 'Carrot', 'Donut']
```

You can use the index notation to add new items to any position in an array. For example, the following code will add `'Eggplant'` at position 5:

```
shopping[5] = 'Eggplant';
```

When we take a look at the `shopping` array in the console now, we can see that the sixth item (with an index of 5) has been filled with the string `'Eggplant'`. This has left a gap in the array at position 4, so this unused position in the array is filled by the value `undefined`:

```
shopping;
<< ['Apple', 'Banana', 'Carrot', 'Donut', undefined, 'Eggplant']
```

Removing Values from Arrays

The `delete` operator can be used to remove an item from an array. For example, if we decide that we really shouldn't be buying donuts, we could remove the value `'Donut'` from our `shopping` array using the following code:

```
delete shopping[3];
<< true
```

This removes the donut directly from its place in the list, leaving an empty slot in the array, as shown in the diagram below.

Now, if we take a look at the `shopping` array, we can see that the string `Donut` (with an index of 3), has indeed been removed … and a value of `undefined` has been left in its place:

```
shopping;
<< ['Apple', 'Banana', 'Carrot', undefined, undefined,
'Eggplant']
```

Watch out for this, as it can even trip up experienced programmers. The *value* that was in position 3 (`'Donut'`) has been deleted from the array, but the space that it occupied is *still there*, and it now contains a value of `undefined`. This means that the array still has the same number of elements, and the position can still be referenced as an index, but it will just return `undefined`:

```
shopping[3];
<< undefined
```

Finding the Length of an Array

Every array has a `length` property that tells us how many items it contains. For example, we can find the number of items in the `shopping` array using the following code:

```
shopping.length;
<< 6
```

Notice that the length is 6 because the positions that contain `undefined` are also counted when calculating the length of the array.

The value of the `length` property can be placed inside square brackets as part of the index to find the last item in an array. For example, the following code tells us the last item on our shopping list:

```
const length = shopping.length;
shopping[length - 1];
<< 'Eggplant'
```

This tells us that `'Eggplant'` is the last item in the array, but notice that we have to subtract 1 from the `length` property. This is because the index starts at 0, so the last item in the array will have an index of one less than the array's actual length.

The `length` property of an array isn't fixed, which means that you can dynamically change the length of an array by adding or removing items. In fact, the `length` property is mutable, which means you can change it directly, like so:

```
shopping.length = 8;
<< 8
```

This will make the length of our `shopping` array 8 instead of 6. If we take a look at the array, we can see that the extra slots are now empty:

```
shopping;
<< ['Apple', 'Banana', 'Carrot', undefined, undefined,
'Eggplant', undefined, undefined]
```

If we change the length of the array to a value shorter than its current length, all the extra elements will be removed completely. The following code will keep the first three items in the array and remove all the rest:

```
shopping.length = 3;
<< 3
```

Now if we check the contents of the array, we can see that only the first three items remain:

```
shopping;
<< ['Apple', 'Banana', 'Carrot']
```

No Second Chances

Once you make an array shorter by editing its length property, any items that are removed are lost for good. If you try to change the length back to 8, it will just add `undefined` into all the extra spaces rather than filling them with the values that were there before:

```
shopping.length = 8;
<< ['Apple', 'Banana', 'Carrot', undefined, undefined,
undefined, undefined, undefined,]
```

For this reason, be very careful not to lose any values you'll need later when changing the length of an array.

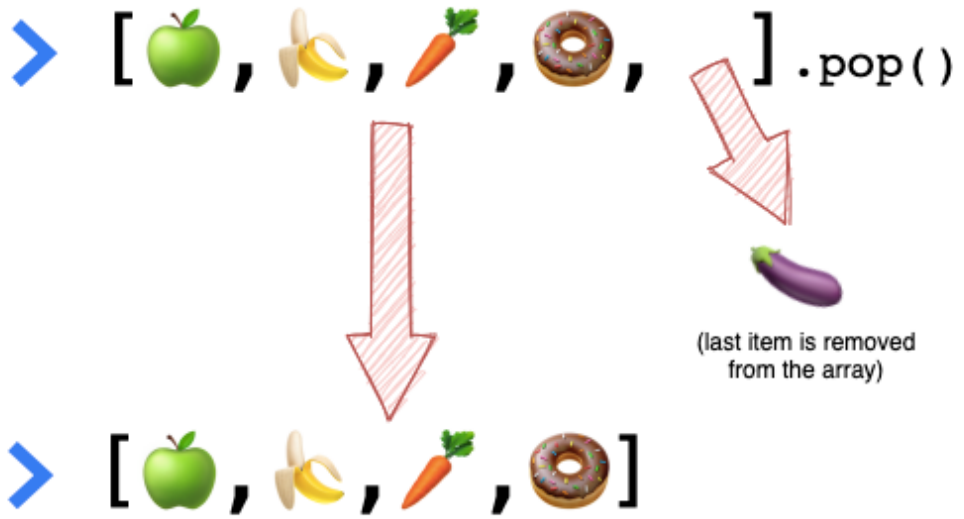
Popping and Pushing

Arrays have methods called `pop` and `push` that can be used to remove or add items.

The `pop()` method removes the last item from an array. To see it in action, let's create a new shopping list array:

```
const shopping = ['Apple', 'Banana', 'Carrot', 'Donut',
'Eggplant']
```

We can remove the last item in the array using the `pop()` method, which will "pop" the last item out of the array, as shown in the diagram below.



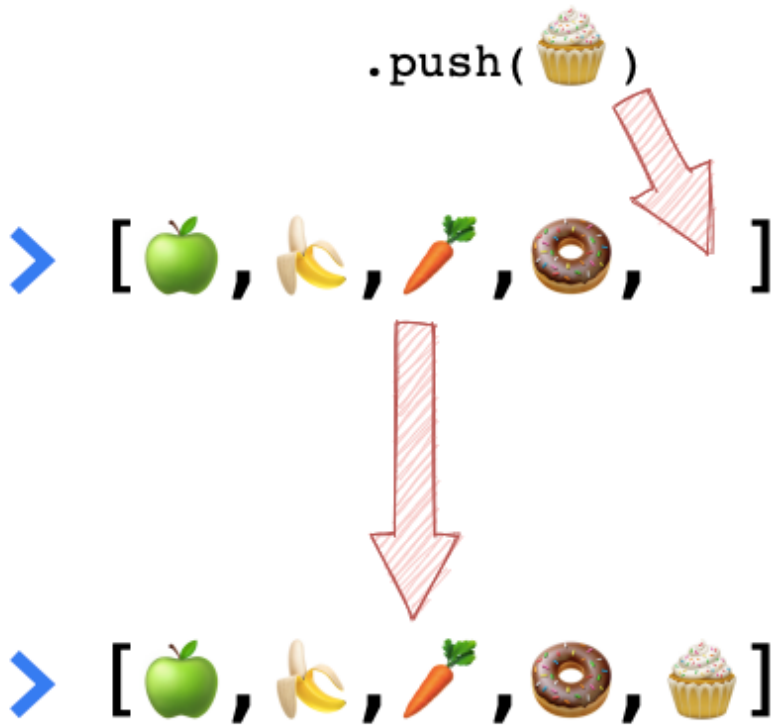
The code below shows how the method is applied to the `shopping` array:

```
shopping.pop();
<< 'Eggplant'
```

The method returns the last item of the array and also updates the array so that it no longer contains the item. If we take a look at the `shopping` array, we'll see that it no longer contains the string `'Eggplant'`:

```
shopping;
<< ['Apple', 'Banana', 'Carrot', 'Donut']
```

If we instead want to add a new value to the end of an array, we can use the `push()` method. For example, it can be used to add a cupcake to the end of the `shopping` array, as illustrated in the diagram below.



We would use the following code to do this in the console:

```
shopping.push('Cupcake');
<< 5
```

The method returns the new length of the array, which is 5, as the array now contains 5 items. We can check that `'Cupcake'` has been added by just typing `shopping` in the console:

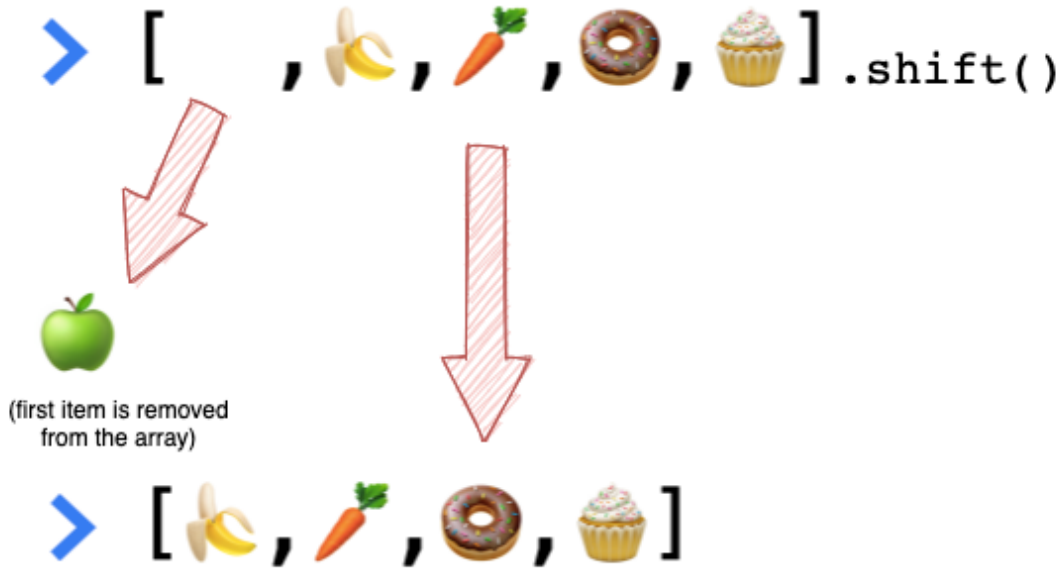
```
shopping;
<< ['Apple', 'Banana', 'Carrot', 'Donut', 'Cupcake']
```

Shifting and Unshifting

The `shift()` method is similar to `pop`, the difference being that it removes the *first* item in the array:

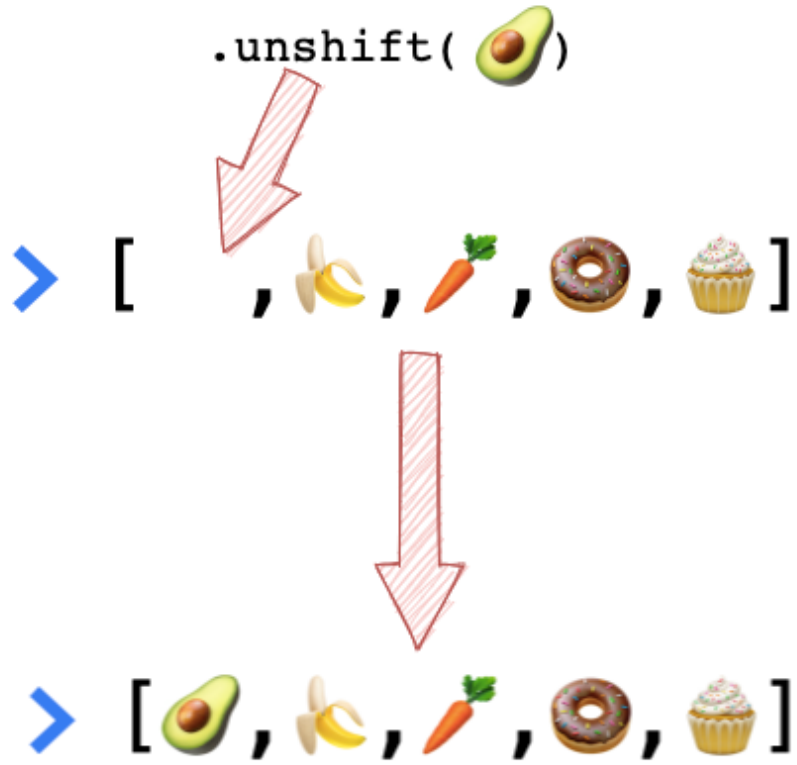
```
shopping.shift();
<< 'Apple'
```

Notice that it returns the value of the first item that has been removed from the array.



The `unshift()` method is similar to the `push()` method, but it adds a new item to the *beginning* of the array. For example, we could add `'Avocado'` to the beginning of our `shopping` array using the following code:

```
shopping.unshift('Avocado');
<< 5
```



The return value is the length of the array after the new item has been added. We can have a quick check to confirm that it has indeed been added:

```
shopping;
<< ['Avocado', 'Banana', 'Carrot', 'Donut', 'Cupcake']
```

The Spread Operator

The **spread operator** is an ellipsis of three dots (. . .) placed in front of an array that sits inside another array. It spreads out all the elements of the array into separate values.

If you think of an array as being like a box full of values, then the spread operator is the equivalent of emptying the contents of the box into another box.

To see the spread operator in action, try creating an array in the console:

```
const arrayA = [1,2,3];
```

This creates an array called `arrayA` that contains three elements.

Now, say we wanted to place all three elements from `arrayA` into a new array called `arrayB`. We might try doing something like this:

```
const arrayB = [arrayA];
```

Unfortunately, all we've done is place the whole of `arrayA` into `arrayB`. This is the equivalent of putting a box of values inside another box. This can be seen by the fact that `arrayB` only contains just *one* element, as we can see if we look at the length property:

```
arrayB.length;
<< 1
```

If we take a look at the contents of `arrayB`, we can see that the one item it contains is an array that matches `arrayA` (this is called a "nested array", which we'll discuss later in the chapter). This is because all the elements are still grouped together inside `arrayA`:

```
arrayB;
<< [[1,2,3]]
```

If we apply the spread operator to `arrayA` when it's placed inside another array, `arrayC`, it will unpack all the elements out and treat them as three separate values:

```
const arrayC = [...arrayA];
```

Now if we look at the length of `arrayC`, we can see that is does indeed contain three separate elements:

```
arrayC.length;
<< 3
```

And if we take a look at the contents of `arrayC`, we can see that it contains the same elements as `arrayA`:

```
arrayC;
<< [1,2,3]
```

The spread operator can be used to merge the elements of two or more arrays together in a new array. To see this in action, let's create three different arrays containing different types of food:

```
const fruit = ['Pineapple', 'Melon'];
const savory = ['Burger', 'Fries'];
const sweets = ['Cookie', 'Popcorn'];
```

We can now merge these three arrays into one array called `food` by placing them all inside a new array and applying the spread operator to each array so that all the elements are unpacked from their arrays into separate values:

```
const food = [...fruit, ...savory, ...sweets];
<< undefined
```

We can check this has worked by taking a look at the `food` variable:

```
food;
<< ['Pineapple', 'Melon', 'Burger', 'Fries', 'Cookie',
'Popcorn']
```

As you can see, all the items have been spread out of their original array containers and placed separately inside the new `food` array.

If the spread operator hadn't been used, the items in the new array would still have been enclosed inside their original arrays, as can be seen in the

code below:

```
const food = [fruit, savory, sweets];
<< [['Pineapple', 'Melon'], ['Burger', 'Fries'], ['Cookie',
'Popcorn']]
```

The spread operator can also be used to add new items to an array. For example, if we wanted to add `'Coffee'` to the end of the `food` array, instead of using the `push()` method, we could use the following code:

```
food = [...food, 'Coffee'];
```

In this example, the spread operator unpacks all the items out of the `food` array and places them inside the new array as separate items.

And instead of using the `unshift()` method to add the string `'Coconut'` to the beginning of the array, we could use the following code instead:

```
food = ['Coconut', ...food];
```

In fact, you could add `'Coconut'` to the front of the array and `'Coffee'` to the end in a single step using the following code:

```
food = ['Coconut', ...food, 'Coffee'];
```

We can see the result of this operation by taking a look at the contents of the array:

```
food;
<< ['Coconut', 'Pineapple', 'Melon', 'Burger', 'Fries',
'Cookie', 'Popcorn', 'Coffee']
```

Slicing and Splicing

The `slice()` method creates a subarray, effectively chopping out a slice of an original array starting at one position and finishing at another. For example, if we wanted to find the third and fourth items in our `food` array, we would use the following code:

```
food.slice(2,4);
<< ['Melon', 'Burger']
```

The first number in the parentheses tells us the index to start the slice at, and the second number tells us the index that the slice goes *up to*, without including that item. So in the example above, the slice will start at `'Melon'`, which has an index of 2, and then include all the items up to, but not including, the item with an index of 4, `'Fries'`.

This operation is non-destructive, so no items are actually removed from the array, as we can see if we take a look at the `food` array:

```
food;
<< ['Coconut', 'Pineapple', 'Melon', 'Burger', 'Fries',
'Cookie', 'Popcorn', 'Coffee']
```

The `splice()` method removes items from an array and then inserts new items in their place. For example, the following code removes the string `'Melon'` and replaces it with `'Mango'`:

```
food.splice(2, 1, 'Mango');
<< ['Melon']
```

The first number in the parentheses tells us the index at which to start the splice. In the example we started at index 2, which is the third item in the array (`'Melon'`).

The second number tells us how many items to remove from the array. In the example, this was just the one item.

The next value is then inserted into the array at the place where the items were removed from. In this case, the string `'Mango'` is inserted into the array at index 2.

Notice that the `splice()` method returns the items removed from the array as a new array, so in the example, it returned the array `['Melon']`.

Splicing permanently changes the value of the array, as we can see below:

```
food;
<< ['Coconut', 'Pineapple', 'Mango', 'Burger', 'Fries',
'Cookie', 'Popcorn', 'Coffee']
```

The `splice()` method is a particularly flexible method, as it can be used to insert or remove values from an array. Be careful, though: it's a **destructive** method, which means that it changes the array permanently.

To insert values into an array at a specific index without removing any items, we simply indicate that zero items are to be removed:

```
food.splice(4,0,'Pizza');
<< []
```

An empty array is returned (because nothing was removed), but the new value of `'Pizza'` has been inserted at index 4, which we can see if we look at the `food` array:

```
food;
<< ['Coconut', 'Pineapple', 'Mango', 'Burger', 'Pizza', 'Fries',
'Cookie', 'Popcorn', 'Coffee']
```

We saw earlier that we can use the `delete` operator to remove an item from an array. Unfortunately, this leaves a value of `undefined` in its place. If you want to remove a value completely, you can use the `splice()` method with a

length of `1` and without specifying any values to add. For example, if we want to remove `'Burger'` (at index 3) from our `food` array, we could use the following code:

```
food.splice(3,1);
<< ['Burger']
```

As you can see, the value that is removed will be returned as an array containing that value.

If we now look at the `food` array, we can see that the string `'Burger'` has been removed completely, without leaving any empty spaces:

```
food;
<< ['Coconut', 'Pineapple', 'Mango', 'Pizza', 'Fries', 'Cookie',
'Popcorn', 'Coffee']
```

Finding If a Value Is in an Array

We can find out if an array contains a particular value using the `indexOf()` method to find the first occurrence of a value in an array. If the item is in the array, it will return the index of the first occurrence of that item:

```
food.indexOf('Pizza');
<< 3
```

If the item isn't in the array, it will return `-1`:

```
food.indexOf('Burger');
<< -1
```

Arrays also have the `includes()` method. This returns a Boolean value depending on whether the array contains a particular element or not:

```
food.includes('Pizza');
<< true

food.includes('Burger');
<< false
```

You can also add an extra parameter to indicate which index to start the search from. For example, the following code starts searching for the string `'Coconut'` from index 1 (the second item in the array) onwards, so returns false, as the string appears before then:

```
food.includes('Coconut', 1);
<< false
```

Joining Array Items into a String

The `join()` method can be used to turn the array into a string that comprises all the items in the array, separated by commas:

```
food.join();
<< 'Pineapple,Melon,Fries,Bread,Cookie,Popcorn'
```

You can choose another separator instead of a comma by placing it inside the parentheses. Let's try using an ampersand with a space on either side:

```
food.join(' & ');
<< 'Pineapple & Melon & Fries & Bread & Cookie & Popcorn'
```

Reversing the Order of Array Items

We can reverse the order of an array using the `reverse()` method. For example, if we use it on the `food` array, we'll get the following:

```
food.reverse();
<< ['Coffee', 'Popcorn', 'Cookie', 'Fries', 'Pizza', 'Mango',
'Pineapple', 'Coconut']
```

Note that this changes the order of the array permanently.

Sorting Array Values

We can sort the items of an array into order using the `sort()` method. Calling this method on the `food` array will rearrange the array items into alphabetical order:

```
food.sort();
<< ['Coconut', 'Coffee', 'Cookie', 'Fries', 'Mango',
'Pineapple', 'Pizza', 'Popcorn']
```

Note that this also changes the order of the array permanently.

Alphabetical Numbers

Strings are sorted in alphabetical order by default, but so are numbers! This means that numbers are sorted by their first digit, rather than numerically. For example, `9` will come after `10`, so you'll get crazy results like those shown below:

```
[5, 9, 10].sort();
<< [10, 5, 9]
```

Don't worry, though. This can be fixed using something called a "callback", which we'll be covering later in the book.

More Methods

There are loads more things that arrays can do. A full list of properties and methods can be found on the Mozilla Developer Network. Take a look and see if you can find any interesting ones and try using them in the console.

Multi-dimensional Arrays

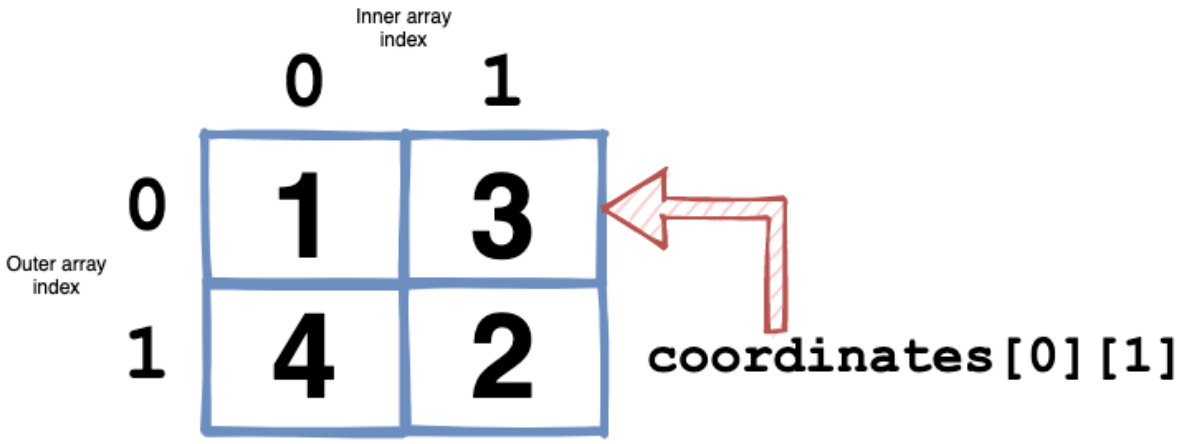
You can create an array of arrays, known as a **multi-dimensional array**, by placing multiple arrays inside a container array. This could be used to create a coordinate system, for example:

```
const coordinates = [[1,3],[4,2]];
<< [[1,3],[4,2]]
```

To access the values in a multidimensional array, we use two indices—one to refer to the item's place in the outer array, and one to refer to its place in the inner array:

```
coordinates[0][0]; // The first value of the first array
<< 1

coordinates[1][0]; // The first value of the second array
<< 4

coordinates[0][1]; // The second value of the first array
<< 3

coordinates[1][1]; // The second value of the second array
<< 2
```

The structure of the two-dimensional array can be seen in the image below.



The spread operator that we met earlier can be used to "flatten" multi-dimensional arrays. **Flattening** an array involves removing all nested arrays

by taking all the values out of their arrays and placing them on the same level in the parent array. You can see in the example below that just placing the `summer` and `winter` arrays inside a container array will create a nested array, but if we use the spread operator when placing the arrays inside the container array, we get a flat array that only contains their contents:

```
const summer = ['Jun', 'Jul', 'Aug'];
const winter = ['Dec', 'Jan', 'Feb'];
const nested = [ summer, winter ];
<< [ [ 'Jun', 'Jul', 'Aug' ], [ 'Dec', 'Jan', 'Feb' ] ]

const flat = [...summer, ...winter];
<< [ 'Jun', 'Jul', 'Aug', 'Dec', 'Jan', 'Feb' ]
```

JavaScript also has the `flat()` method, which will flatten an array without the need for the spread operator, as can be seen below:

```
[ [ 'Jun', 'Jul', 'Aug' ], [ 'Dec', 'Jan', 'Feb' ] ].flat();
<< [ 'Jun', 'Jul', 'Aug', 'Dec', 'Jan', 'Feb' ]
```

A summary of creating and manipulating arrays can be found in this post on SitePoint.

Challenges

1. Create a variable called `shoppingList` that starts as an empty array. Experiment using `pop`, `push`, `shift`, `unshift` and `splice`, along with the spread operator, to add and remove items from the list. You can see my code on CodePen.

2. Use three prompt boxes to ask the user for three different words and then place them in an array. Use an alert box to display the array. You can see my code on CodePen.

3. Use a prompt box to ask the user to enter a word. Then use the a combination of the `split`, `reverse()` and `join()` methods to write the word backwards. For example, “hello” would become “olleh”. (Hint: provide an empty string as the parameter to the `split()` and `join()` methods.) You can see my code on CodePen.

4. Write a snippet of code that asks the user for a list of comma-separated names in a prompt and that then creates an alert that displays the names in alphabetical order. You can see my code on CodePen.

Summary

- Arrays are an ordered list of values.
- An array literal is written using square brackets containing comma-separated values—such as `[2,3,5,7]`.
- Arrays can contain any type of value—even other arrays!
- The index is used to reference a specific item in an array. For example, `myArray[0]` refers to the first item in `myArray`.
- Multi-dimensional arrays are arrays that contain other arrays.
- Arrays have many methods that can be used to add, remove and manipulate items in the array.
- The spread operator is used by placing an ellipsis of three dots in front of an array. It has the effect of taking all the values out of the array and listing them as separate values inside a new array.

Now that we've learned about different types of data and collections of data, it's time to move on and learn how to control the flow of the program using logic, which we'll be covering in the next chapter.

Chapter 6: Logic

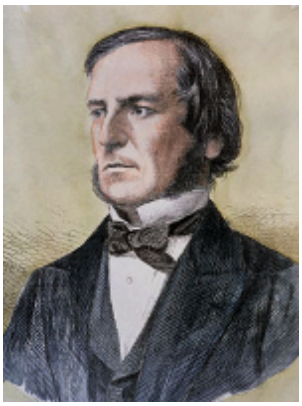
In previous chapters, we looked at the primitive values of strings and numbers. Now let's take a closer look at Booleans, another primitive data type, and find out how we can use them to control the flow of a program.

In this chapter, we'll be covering the following:

- Booleans
- logical operators
- comparison
- flow control
- if–else statements
- a favorite animal quiz
- switch statements
- a rock paper scissors game

Booleans

In Chapter 2, we saw that Booleans are one of the primitive data types used by programming languages. There are only two Boolean values, `true` and `false`. They're named after George Boole, an English mathematician who worked in the field of algebraic logic. Boolean values are fundamental in the logical statements that make up a computer program.



Every value in a programming language has an inherent Boolean value of either `true` or `false`. Most are `true`, and these are known as **truthy** values. A few of them are false, and these are known as **falsy** values.

In JavaScript, you can use the `Boolean` function to find out if a value is truthy or falsy:

```
Boolean('hello');
<< true

Boolean(42);
<< true

Boolean(0);
<< false
```

There are only nine falsy values in JavaScript, and these are listed below:

```
// double quoted empty string literal
""

// single quoted empty string literal
''

// empty template literal
``

// zero
0

// negative zero (considered different to 0 by JavaScript!)
-0

// Not a Number is falsy
NaN

// this one is obvious ...
false

// null is falsy
null

// undefined is also falsy
undefined
```

Other languages will have different falsy values (for example, in Python an empty array is falsy) and some have a lot fewer falsy values than others (Ruby has just just two, for example).

The fact that empty strings evaluate to false in JavaScript can cause a number of problems if you're not careful, so watch out for them.

To learn more about truthy and falsy values, see "Truthy and Falsy: When All is Not Equal in JavaScript".

Logical Operators

Logical operators can be used to combine two or more statements to produce a compound statement that returns a Boolean value.

Guess Who?

Guess Who? is a popular family game that involves trying to guess a person by asking "yes" or "no" questions about their appearance. We're going to consider a smaller version of the game to demonstrate some logical operators.

Our smaller game only has four characters: Alfie, Betty, Gemma and Del.

Each character has two particular characteristics: they either wear a hat or they don't, and they either wear glasses or they don't.

For example, if I asked “Who is wearing glasses?”, the answer would be “Betty and Del”.

Negation (Logical NOT)

Negation returns the opposite of a value’s Boolean value. So truthy values will return `false`, and falsy values will return `true`.

For example, if I was playing the Guess Who? game, the question “Who is NOT wearing glasses?” is a negation of the question “Who IS wearing glasses?” The word “NOT” is acting as the negation operator, and the answer is “Alfie and Gemma”.

If I asked “Who is NOT wearing a hat”, this would be the negation of all the people who *are* wearing a hat, and the answer would be “Alfie and Del”.

In JavaScript, negation is achieved by placing the logical NOT operator (`!`) in front of a value. This can be thought of in the same way as placing the word “NOT” in front of a statement. A truthy value will always negate to `false`, and a falsy value will always negate to `true`. We can see this by trying the following examples in the console:

```
!true; // negating true returns false
<< false

!0; // 0 is falsy, so negating it returns true
<< true

!'hello'; // all non-empty strings are truthy
<< false
```

Double negation (`!!`) is a shortcut that can be used to find out if a value is truthy or falsy, as it effectively “negates the negation”, changing a value back to its original Boolean value, as can be seen in the code below:

```
!!''; // empty strings are falsy
<< false

!!"hello"; // all non-empty strings are truthy
<< true

!!3; // all non-zero numbers are truthy
```

```
<< true

!!0; // zero is falsy
<< false
```

Double negation has the same effect as using the `Boolean()` function that we saw earlier in the chapter.

Logical AND

To see an example of the AND operator, let's play the Guess Who? game again. If I asked "Who is wearing glasses AND a hat?", the result would be "Betty".

In this case, the word "AND" is acting as the logical operator. The logical AND operator acts on two or more values and returns `true` if *all* the values are truthy and `false` if *any* of them are falsy.

In JavaScript, `&&` is used as the logical AND operator. The value that gets returned is the *last* truthy value if they are all truthy, or the *first* falsy value if some of them are falsy. This can be seen in the code example below:

```
'hello' && 42;
<< 42
```

Both the string `'hello'` and the number `42` are truthy values, so the last value, `42`, is returned.

The next statement returns `false` because it's the first falsy value in the statement:

```
true && false; // returns false because it is falsy
<< false
```

Logical OR

Let's go back to playing the Guess Who? game to demonstrate the logical OR operator.

If we asked "Who is wearing glasses OR a hat?", this would include all the people wearing glasses, a hat, or both, so the answer is "Betty, Gemma and Del".

In this example, the word "OR" is acting as the logical operator.

The logical OR operator acts on two or more values and returns true if *any* of the values are truthy and `false` if *all* the values are false.

In JavaScript, `||` is used as the logical OR operator. The value that's returned is the *first* truthy value if any of them are true, or the *last* falsy value if all of them are false.

You can see this in the code examples below. If both expressions are truthy, the first is returned:

```
'hello' || 'goodbye';
<< 'hello'
```

This returns `'hello'`, as it's the first truthy value in the compound statement in which both values are truthy.

If only one of the values is truthy, it will be this value that's returned, as can be seen in the next example:

```
'all' || 0; // it's all or nothing!
<< 'all'
```

In the last example, both values are falsy, so the *last* value is returned:

```
false || 0;
<< 0
```

Comparison

We often need to compare values when programming, and there's a number of ways to do this.

Equality

The **equality operator** can be used to check if two values are equal to each other. Most programming languages use the double-equals operator (`==`) to check for equality.

For example, if we wanted to know if the variable `answer` is 42, we could write:

```
answer == 42;
```

This would return `true` if the answer was 42 and false if it wasn't.

Why double-equals and not a single equals sign? Well, remember back in Chapter 2 we learned that the single equals sign was used for *assignment*. This means that the following code will assign the value of `42` to the variable `answer`, rather than check if they are equal:

```
answer = 42;
```

Assigning values instead of checking for equality is a common mistake that can often catch out rookie programmers.

Soft Equality

JavaScript does things a little differently from other languages. It has the double-equals operator (==), known as **soft equality**, or the triple-equals operator (===), known as **hard equality**.

What's the difference between hard and soft equality? Well, it's all to do with how strict JavaScript is when it comes to deciding whether or not two values are equal.

Consider the following example. Let's say we want to check if the variable `answer` is equal to the value of 42. We could check it using soft equality:

```
answer == 42;
<< true
```

This looks fine, because the value of the variable `answer` is indeed 42, but the following code highlights a problem with soft equality:

```
answer == '42';
<< true
```

As you can see, JavaScript is returning `true` when we're checking if the variable `answer` is equal to the *string* `'42'`, when its value is actually the *number* `42`.

This is an important difference: the string `'42'` is not the same as the number `42`, as they're completely different data types, but when soft equality is used, JavaScript doesn't take into account the data type and will attempt to coerce the two values to the same type when performing the comparison. This can lead to some very strange results. For example, it will say that a string containing whitespace is the same as the number zero, as you can see in the code example below:

```
"     " == 0;
<< true
```

The next example shows that the Boolean `false` is considered the same as the string `"0"` if soft equality is used:

```
false == "0";
<< true
```

As you can see, the soft equality operator gives some strange results (and these aren't the only examples).

Because of this, the soft equality should *never* be used to check if two values are actually equal.

Hard Equality

JavaScript uses a triple-equals (`===`) to test for hard equality. This will only return true if the two values are equal *and* are of the same data type. This can be seen in the example below, which checks if the variable `answer` is equal to the number `42` and string `'42'`:

```
answer === 42;
<< true

answer === '42';
<< false
```

As you can see, hard equality reports that the variable `answer` is equal to the *number* `42`, but not the string `'42'`.

You should always use hard equality when you want to test if two values are equal. This will avoid the problems caused by type coercion when using soft equality.

If you want to check whether a number represented by a string is equal to a number, you should convert it to a number yourself explicitly rather than relying on type coercion to do it in the background. For example, the following code could be used to convert the string `'42'` into the number `42`:

```
answer === Number('42');
<< true
```

This comes in handy when you're dealing with a value entered by a user that you expect to be a number, since most programming languages will treat user-entered data as a string.

Inequality

We can check if two values are *not* equal using the **inequality operator**. There's a soft inequality operator (`!=`) and a hard inequality operator (`!==`). These work in a similar way to the soft and hard equality operators:

```
16 != '16'; // type coercion makes these equal
<< false

16 !== '16';
<< true
```

As with equality, you should use the hard inequality operator, as this will give more reliable results unaffected by type coercion.

Greater Than and Less Than

We can check if a value is greater than another using the `>` operator. The following code can be used to check if the variable `answer` (that has a value of 42) is greater than 10:

```
answer > 10;
<< true
```

You can also use the "less than" operator (`<`) in a similar way to check if `answer` is less than 50:

```
answer < 50;
<< true
```

If you want to check if a value is greater than, less than *or equal* to another value, you can use the `>=` and `<=` operators. The following code will check if `answer` is less than or equal to 42:

```
answer <= 42;
<< true
```

These operators can also be used with strings, which will be alphabetically ordered to check if one string is “less than” the other:

```
'apples' < 'bananas';
>> true
```

Be careful, though, as the results are case-sensitive, and uppercase letters are considered to be “less than” lowercase letters:

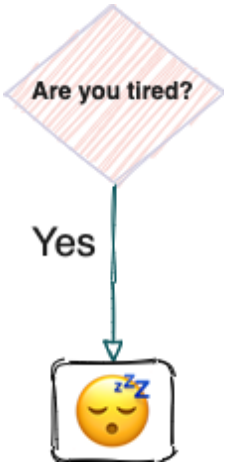
```
'apples' < 'Bananas';
>> false
```

Flow Control

In the previous section, we looked at how to compare statements, such as checking if two values are equal to each other. In this section, we’re going to control the flow of a program by running different blocks of code depending on whether a statement is true or false. This is a bit like reaching a fork in the road in a program, where you can decide which direction your code will go in. This will help to make our programs much more interesting, as they can start to have different results depending on what happens.

If Statements

An **if statement** can be used to run a block of code only if a certain condition returns true. For example, say that you wanted check if you’re tired. If you *are* tired, the plan is to go to sleep. This can be illustrated in the following diagram:



In pseudocode, this could be written as follows:

```
if you are tired then go to sleep.
```

In JavaScript, the code would look like this:

```
if (tired) {
  sleep();
}
```

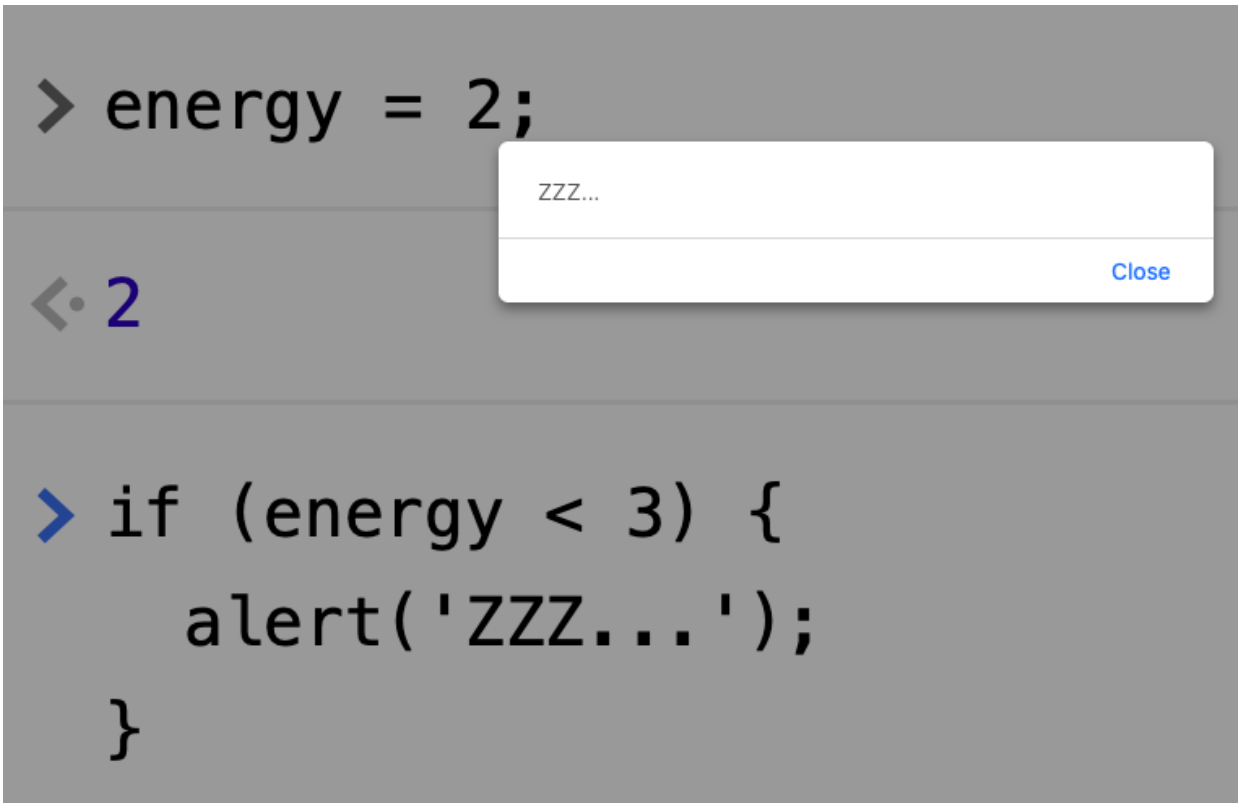
The code inside the block will only run if the condition in the parentheses is true. If the condition is not a Boolean value, it will be converted to a Boolean, depending on whether or not it's truthy or falsy.

Let's try writing some conditional code by writing the following in a console:

```
let energy = 10;
if (energy < 3) {
  alert('ZZZ...');
}
```

The alert box will only be displayed if `energy < 3` evaluates to `true`, so only if the value of the `energy` variable is less than 3. Since the `energy` variable was assigned the value of 10, this means that nothing will happen.

Try changing the value of the `energy` variable to a value below 3 in the console, and then enter the if block again. This time, the alert box will show like in the image below:

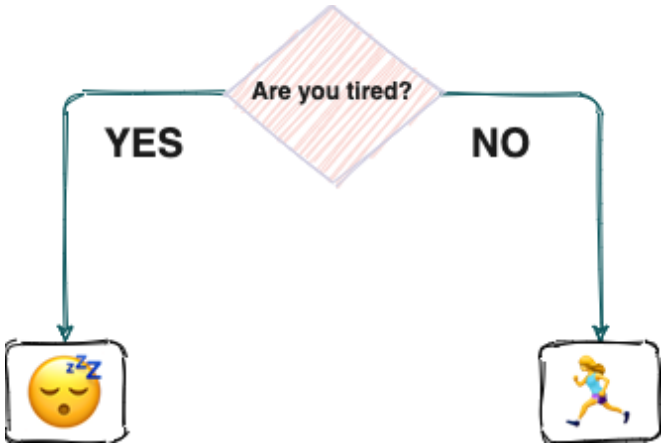


Obviously, in a real program we wouldn't be manually changing the variables like this. They would usually depend on user input or some other conditions in the program. But don't worry—we'll get to some more practical examples soon!

Else Statements

What if you want to do something else if the condition isn't true? That's just what an **else statement** is for.

For example, let's say you want to sleep if you're tired, but otherwise you want to go for a run. This can be illustrated in the diagram below:



In pseudocode, this could be written as:

```
if you are tired then go to sleep
else go for a run
```

In JavaScript, it would be written as:

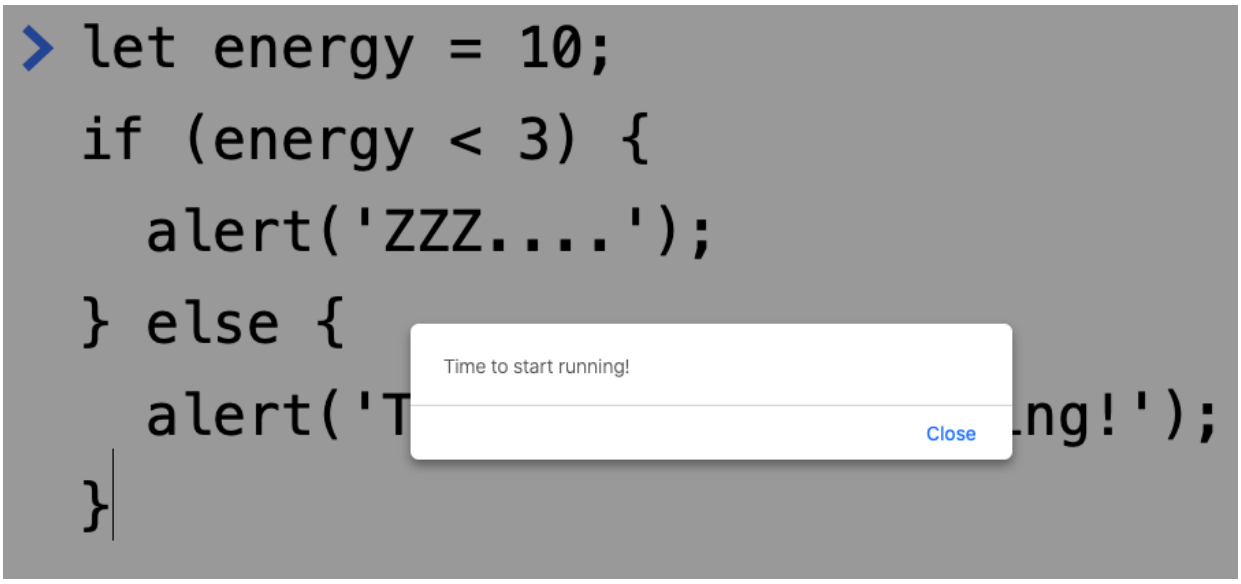
```
if(tired){
  sleep();
} else {
  run();
}
```

The `else` keyword adds an alternative block of code that is only executed if the original condition isn't true. This means only one of the blocks of code will run.

Let's write an if–else statement in the console:

```
let energy = 10;
if (energy < 3) {
  alert('ZZZ....');
} else {
  alert('Time to start running!');
}
```

This code should produce an alert similar to the one shown in the image below.



The Ternary Operator

The **ternary operator** (`?`) is a shorthand way of writing an if–else statement. It takes three operands, a condition and two blocks of code, in the following format:

```
condition ? code to run if condition is true : code to run if condition is false
```

The following code shows how to write the previous if–else statement using the ternary operator:

```
energy < 5 ? alert('ZZZ...') : alert('Time to start running!');
```

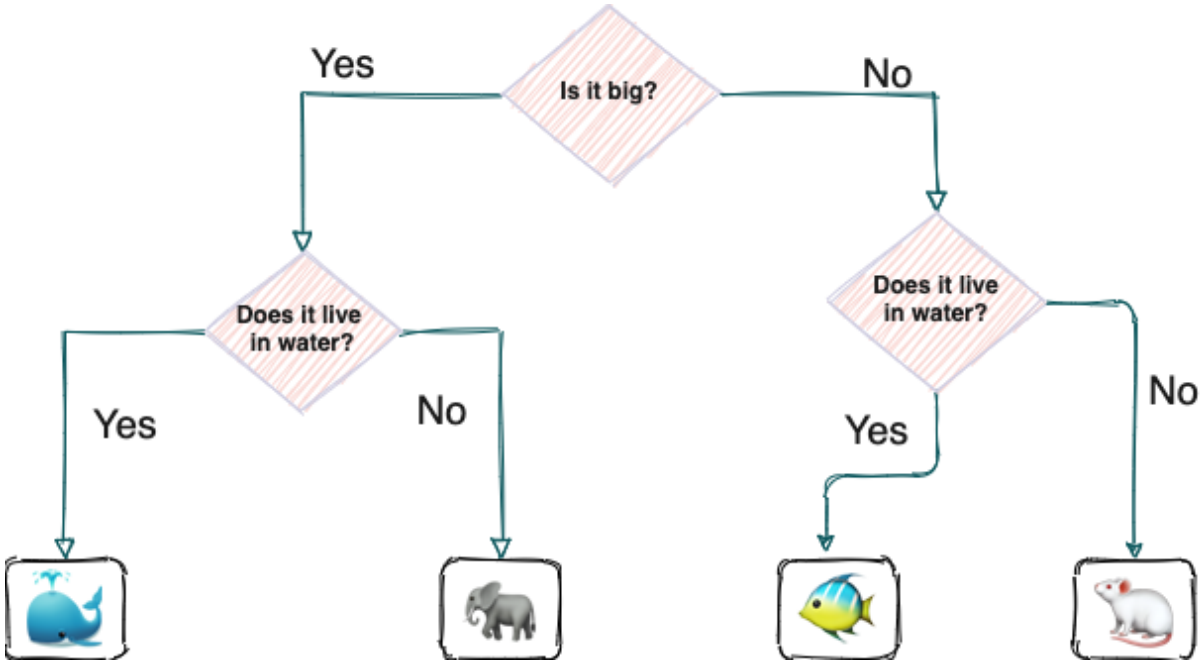
As you can see, the ternary operator can make your code more succinct (this code is now just a single line), but it can also make the code harder to read, so think carefully before using it.

What's Your Favorite Animal?

To demonstrate conditional statements, we're going to try coding an interactive quiz, not unlike one that you might find in a trashy magazine. You know the ones: they ask you a series of true or false questions and then

tell you who you're going to marry. Well, in our quiz, we're going to use some conditional logic to determine your favorite animal.

The diagram below shows how it might be presented in a magazine.



Luckily, we have a built-in way of asking true or false questions—the confirm box! We'll use confirm boxes to ask the question and return a value of true or false to each question. Open up a new Pen on CodePen and enter the following code:

```
const big = confirm('Is it a big animal?');
const livesInWater = confirm('Does it live in water?');
```

This will create two variables, `big` and `livesInWater`, which will contain `true` or `false` depending on what the user answers. We can process these using a combination of `if` and `else` statements and logical operators to find out the user's favorite animal. Add the following code to your Pen:

```
if(big && livesInWater){
  alert('Your favorite animal is a whale!');
}
```

This code will only run if the user answered yes to both questions, since we combined the variables using the AND operator (`&&`). We can now add some

`else` statements to cover the other options:

```
else if(big && !livesInWater){
  alert('Your favorite animal is an elephant!');
}
```

This corresponds to answering yes to the animal being big, but no to it living in water. We're using negation here to indicate that the variable `livesInWater` should be `false`. Basically, writing `!big` is the equivalent of saying "does not live in water".

We can do a similar thing to describe the alternative of answering no to being big, but yes to living in water:

```
else if(!big && livesInWater){
  alert('Your favorite animal is a fish!');
}
```

Finally, we need to describe the situation of choosing no to both options. This requires both variables to be negated:

```
else if(!big && !livesInWater){
  alert('Your favorite animal is a mouse!');
}
```

Note that because there was only one option left, we didn't actually need the last condition. We could have just written the following instead:

```
else {
  alert('Your favorite animal is a mouse!');
}
```

You can see my code on CodePen.

Switch Statements

The previous coding example showed that you can string multiple `if` and `else` statements together to make a logical decision tree.

Here's another example that could be used to produce a grade based on a test score:

```
const legs = Number(prompt('How many legs does your favorite
animal have?'));
if (legs === 0) {
  alert('Your favorite animal is a fish!');
} else if (legs === 2) {
  alert('Your favorite animal is a penguin!');
} else if(legs === 4){
  alert('Your favorite animal is an elephant!');
} else if(legs === 8){
  alert('Your favorite animal is an octopus!');
} else {
  alert(`I'm not sure what animal has that many legs!`);
}
```

The **switch operator** provides an alternative notation and can make your code easier to follow when there are lots of conditions to test, as in this example.

The example above can be rewritten using a `switch` statement like so:

```
const legs = Number(prompt('How many legs does your favorite
animal have?'));

switch (legs) {
case 0:
  alert('Your favorite animal is a fish!');
  break;
case 2:
  alert('Your favorite animal is a penguin!');
  break;
case 4:
  alert('Your favorite animal is an elephant!');
  break;
case 8:
  alert('Your favorite animal is an octopus!');
  break;
default:
  alert(`I'm not sure what animal has that many legs!`);
}
```

The value you're comparing goes in parentheses after the `switch` operator. A `case` keyword is then used for each possible value that can occur (0, 2, 4 and 8 in the example above). After each `case` statement is the code that should run if that case occurs.

The `default` keyword is used at the end for any code than needs to be run if none of the cases are true. In the example below, this gives the unfortunate message that you failed the test if you don't get a score of 10, 9 or 8.

You can see my code on CodePen.

Taking a `break`

It's important to finish each `case` block with the `break` keyword, as this stops any more of the case blocks being executed. Without a `break` statement, the program will "fall through" and continue to evaluate subsequent case blocks.

Rock Paper Scissors

Let's try another coding project on CodePen. This time, we'll try to create an interactive version of the classic game Rock Paper Scissors, where you play against the computer. This will give us a chance to use what we've learned in this chapter to determine who wins the game.

To get started, let's ask the player what their choice is. Open up a new Pen and add the following code to the **JS** section:

```
const player = prompt('Choose rock, paper or
scissors').toLowerCase().trim();
```

This uses a prompt box to ask the user to enter "rock", "paper" or "scissors".

We also "clean" their answer using the `toLowerCase()` and `trim()` methods. This makes sure their answer is all lowercase and removes any extra whitespace. This string is stored in the variable `player`.

Next, we need to program the computer's choice. The optimal strategy in rock paper scissors is to play at random, so we can base the computer's choice on a random number. Let's create that random number now. Add the following code:

```
const number = Math.ceil(3*Math.random());
```

This uses the `Math.ceil()` and `Math.random()` methods that we met in Chapter 4 to choose a random number between 1 and 3 and store it in the variable `number`. We now need to use this number to select the computer's choice. We're going to use a switch block to do this. Add the following code:

```
let computer;

switch(number){
  case 1:
    computer = 'rock'; break;
  case 2:
    computer = 'paper'; break;
  case 3:
    computer ='scissors';
}
```

First of all, this declares a variable called `computer` that will be used to store the computer's choice. Then we create a switch statement that takes the random number we chose earlier and then assigns the value of "rock" to the variable `computer` if it's 1, "paper" if it's 2, and "scissors" if it's 3.

We now have a variable called `player` and another called `computer`. Both of these contain a string of `'rock'`, `'paper'` or `'scissors'`, representing the player's and computer's choice respectively. Our next job is to decide who wins based on these values. The easiest result to test for is a draw, since this happens if the `player` and `computer` variables are the same. Add the following code to take care of this:

```
if (player === computer){
  alert(`It was a draw, we both chose ${player}`);
}
```

Notice how we're using a template literal and string interpolation to feed back the choice that they both made.

Next, we'll look at all the options that result in the player winning. There are three ways that this can happen:

- player chooses "rock" and computer chooses "scissors"
- player chooses "paper" and computer chooses "rock"

- player chooses “scissors” and computer chooses “paper”

All these options can be represented using logical AND and OR operators. This can be seen in the following code, which we need to add to our program:

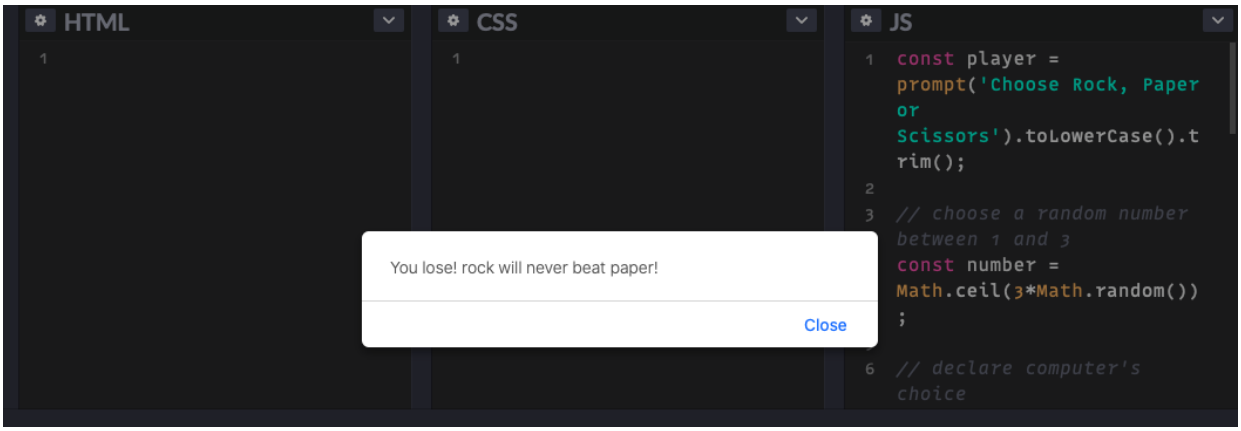
```
else if(player==='rock' && computer==='scissors'
          ||
          player==='paper' && computer==='rock'
          ||
          player==='scissors' && computer==='paper'){
  alert(`You win! ${player} always beats ${computer}!`)
```

Last of all, we need to account for the outcome of the computer winning. There are also three ways this can happen, but the good news is that, since this is a zero-sum game, these are the only outcomes left, so we can just finish with the following else statement:

```
else {
  alert(`You lose! ${player} will never beat ${computer}!`);
}
```

This just simply tells the player they lost and what the two choices were.

Try running the code and playing a few rounds against the computer. Hopefully you’ll do better than I did!



You can see my code on CodePen.

Challenges

1. Write some code that will ask a question and then check to see if the answer provided by the user is correct. It should then provide feedback to say if they were right or wrong in an alert box. You can see my code on CodePen.

2. Write a "higher or lower" game. The computer should pick a random number between 1 and 10, then ask the user if the next number will be higher using a confirm box. The computer should then choose another random number between 1 and 10 and tell the user if they were right or wrong. You can see my code on CodePen.

3. Write some code that picks two numbers at random and then asks the user to multiply them together. Use an alert box to tell the user if they've got the answer right or wrong. You can see my code on CodePen.

Summary

- Booleans are primitive values that can only be `true` or `false`.
- All values have an inherent Boolean value of `true` or `false`.
- Truthy values are values that have a Boolean value of `true`. Most values are truthy.
- Falsy values have a Boolean value of `false`. The values that are falsy vary in different languages. There are nine falsy values in JavaScript.
- Negation makes truthy values false and falsy values true. Double negation returns a value's Boolean value.
- Logical operators such as AND and OR can be used to combine multiple statements.
- Values can be compared to see if they're equal, greater than or less than other values.
- A conditional statement can be used to control the flow of a program based on a condition.
- The ternary operator is a concise way of writing an if–else statement in a single line of code.
- A switch statement can be used when there are multiple options.

In the next chapter, we'll be going loopy over loops.

Chapter 7: Going Loopy

Computers are great at carrying out repetitive tasks over and over again. They never get bored and will follow the instructions exactly every time. Loops are a way of getting a program to repeat a piece of code according to certain conditions.

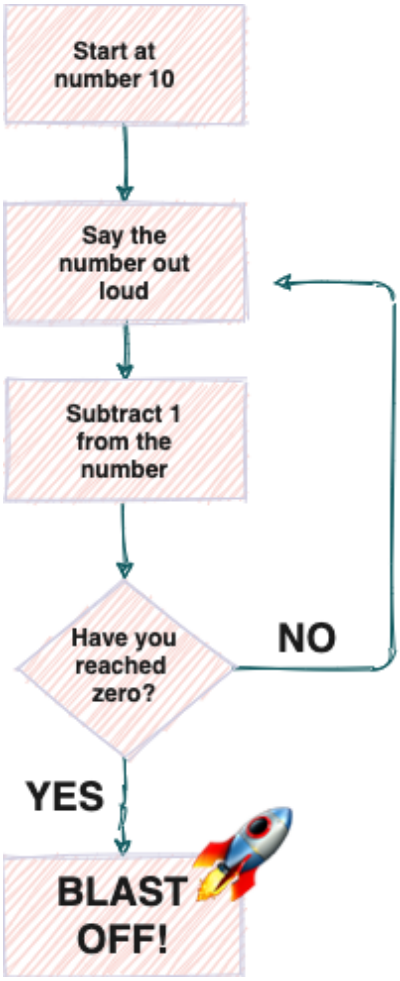
In this chapter, we'll cover:

- what a loop is
- infinite loops
- while loops
- do–while loops
- for loops
- nested loops

What's a Loop?

A **loop** is a piece of code that will continue to run until a certain condition is met.

To illustrate this, here's a diagram that shows the countdown to a rocket blastoff as a loop.



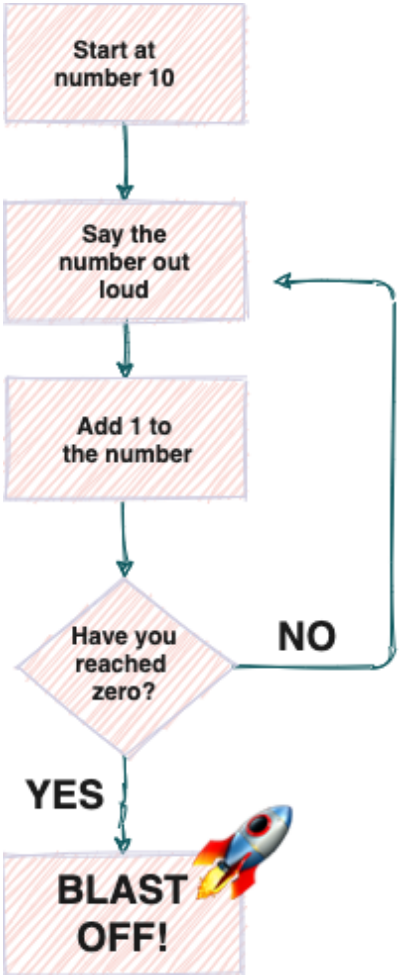
As you can see, we start at 10, say the number out loud, subtract one from the number and then check to see if we've reached zero. If we haven't reached zero, we loop back and repeat the process over again, so we continue to say the number out loud, then subtract 1 until we get to zero. Once this happens, we break out of the loop and the rocket can blast off!

We'll look at the various different ways to code loops in this chapter. As usual, we'll be demonstrating the code in JavaScript, although the principles are very similar in most other programming languages.

Infinite Loops

It's important that the condition to stop a loop is met at some point. Otherwise, your code will get stuck in an **infinite loop** that could possibly crash the program or make it "hang".

Consider the loop shown in the following diagram. It's very similar to the last example, but with one slight change. Can you spot it and explain why it means the loop will never end?



The problem in this example is that the number goes *up* by one every time it runs through the loop, so it will never reach zero and therefore will never break out of the loop and stop. Whenever you write a loop, you need to make sure there's a way for it to finish running.

While Loops

The first type of loop we'll look at is a **while loop**. This will repeatedly run a block of code while a certain condition is true.

The pseudocode for a while loop might look something like this:

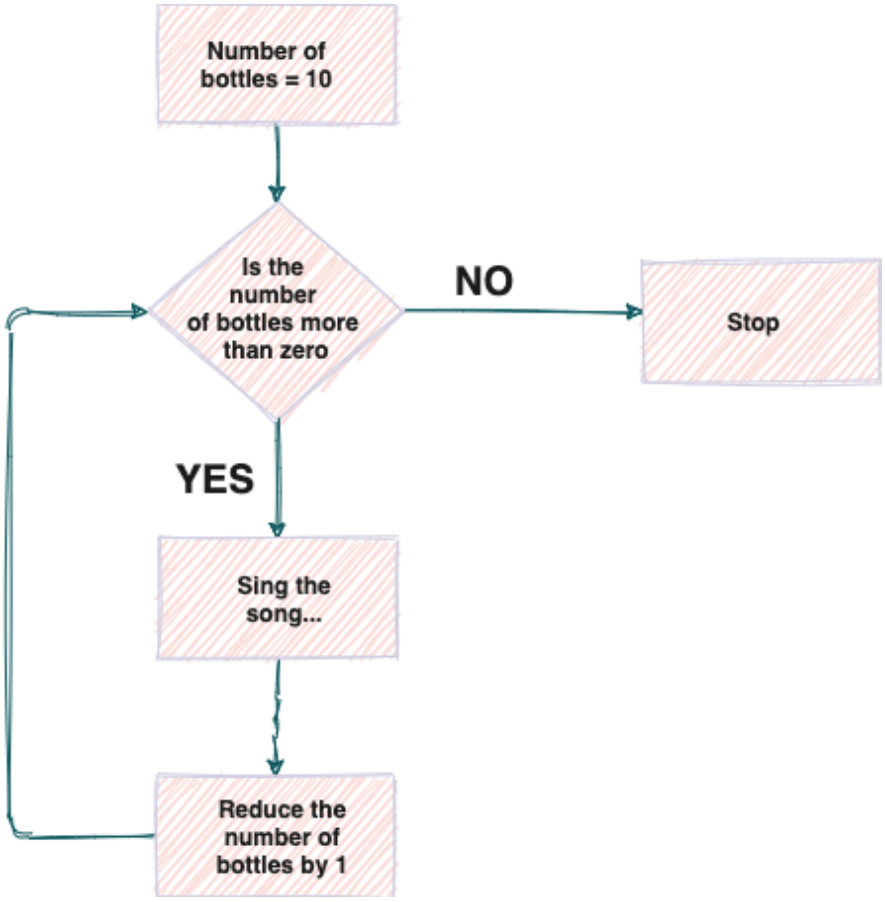
```
while (some condition is true) {
  do something
}
```

The program will continue to “do something” as long as the condition set in the parentheses remains true. Usually, the block of code will do something that has an effect on the condition being true, so eventually it won’t be true and the loop will end.

Ten Green Bottles

A classic demonstration of a loop is to get the computer to “sing” the lyrics to the popular Ten Green Bottles song.

The following diagram shows how this could be done using a while loop.



We start by initializing the number of bottles to be 10 before entering the main loop. There’s a check to see if there are more than zero bottles at the

start of every loop and if there are, the loop continues. Otherwise, it breaks out and stops.

Let's try writing this code in JavaScript. Open up a console and enter the following code:

```
let bottles = 10;
```

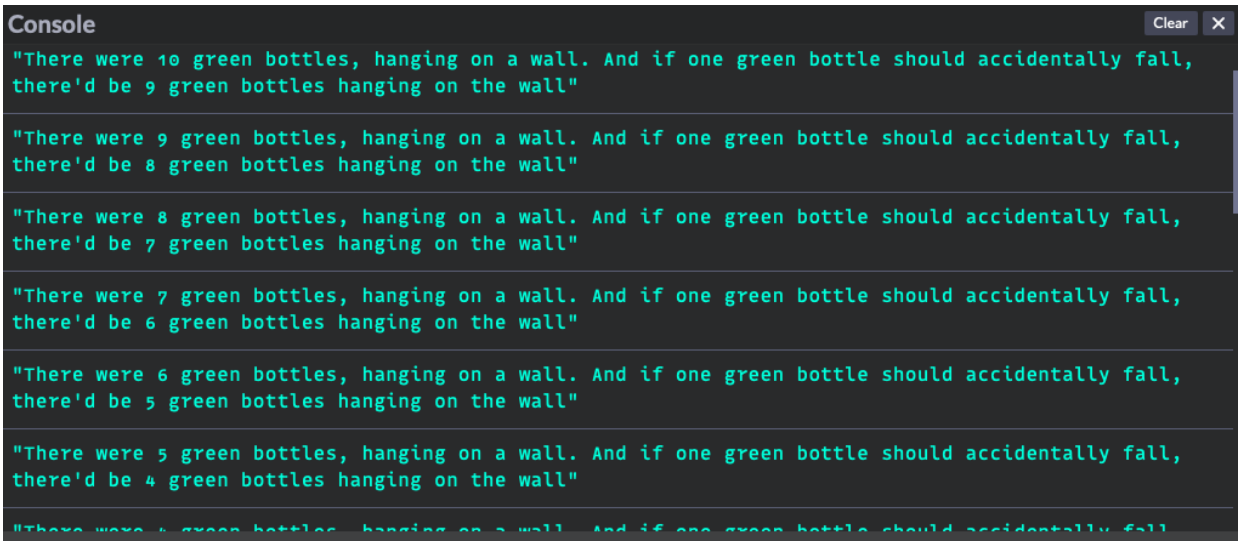
This will initialize a variable called `bottles` to 10. This represents the number of bottles remaining on the wall. Any variables that are used in a while loop must be declared and initialized before the loop is run. Otherwise, there'll be an error when they're referred to in the loop.

Now let's write the loop:

```
while (bottles > 0){
  console.log(`There were ${bottles} green bottles, hanging on a
wall. And if one green bottle should accidentally fall, there'd
be ${bottles-1} green bottles hanging on the wall.`);
  bottles--;
}
```

The loop starts with the `while` keyword, which is followed by the condition in parentheses that the `bottles` variable has to be greater than zero. The loop will continue to execute for as long as this condition is true. This basically means "keep repeating the block of code, as long as the number of bottles is greater than zero".

The block of code uses `console.log` to log a template literal that uses string interpolation to show the number of bottles remaining. The output will appear in the console. The decrement operator (`--`) is then used to decrease the `bottles` variable by one.



You can see my code on CodePen. (Click the **Console** tab on the bottom left to see the results in the CodePen console.)

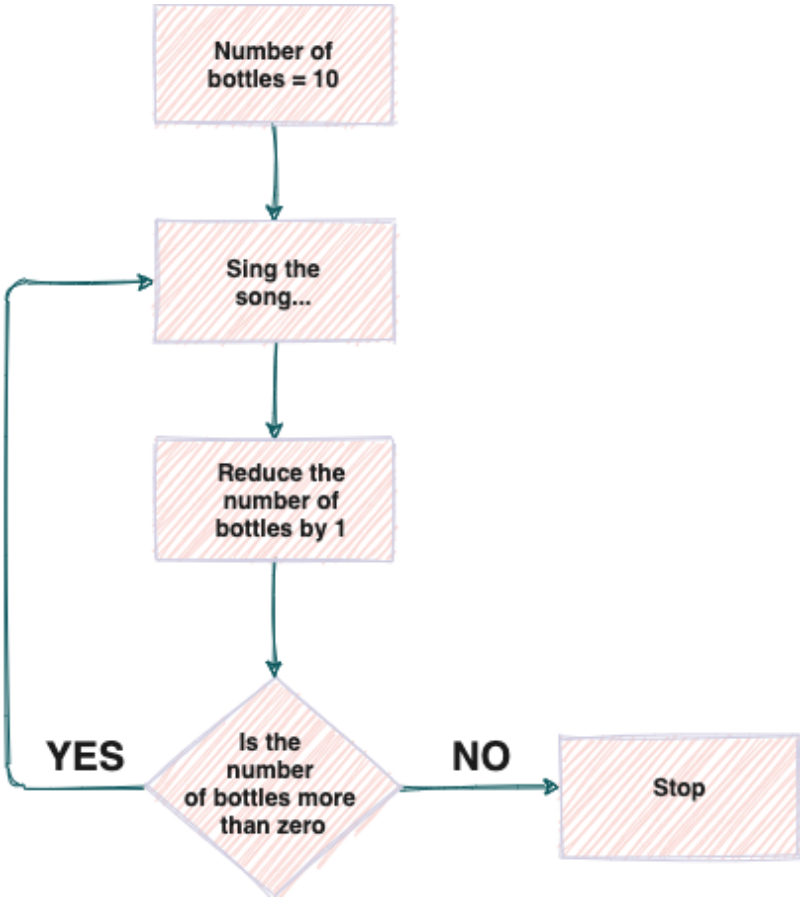
Do–while Loops

A **do–while loop** is similar to a while loop. The only difference is that the condition comes *after* the block of code. In pseudocode, it might look something like this:

```
do {
  // do something
} while(condition)
```

The key difference between a while loop and a do–while loop is that, in the do–while loop the do block of code comes first, meaning that it will always run at least once, regardless of whether the condition is true or not.

The following diagram shows how the Ten Green Bottles song can be implemented using a do–while loop.



As you can see, it's very similar to the diagram for the while loop, except that the condition has been moved to the end of the loop.

Let's try writing this in JavaScript. Enter the following code in a console:

```
let bottles = 10;
do {
  console.log(`There were ${bottles} green bottles, hanging on a
wall. And if one green bottle should accidentally fall, there'd
be ${bottles-1} green bottles hanging on the wall.`);
  bottles--;
} while (bottles > 0)
```

This code is very similar to the previous example and has exactly the same results.

You can see my code on CodePen.

For Loops

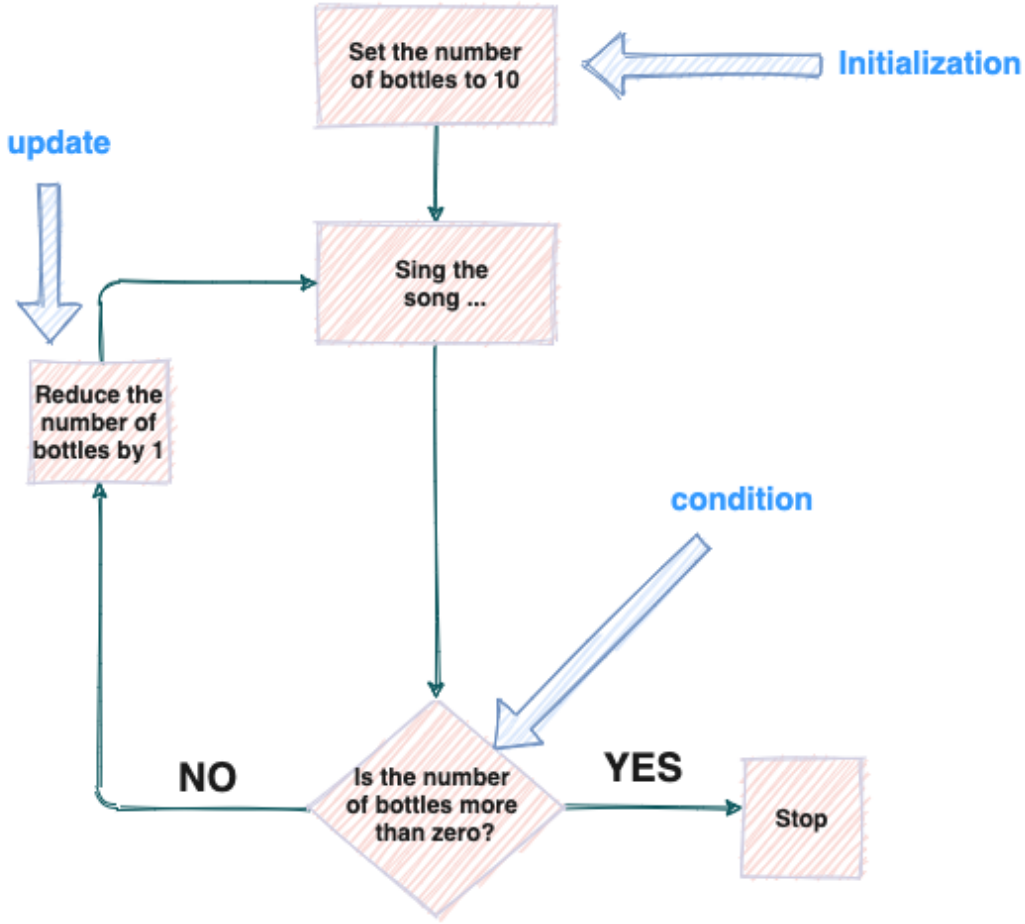
For loops are one of the most common type of loops and can be found in most programming languages. They take the following form:

```
for (initialization ; condition ; update) {
  // do something
}
```

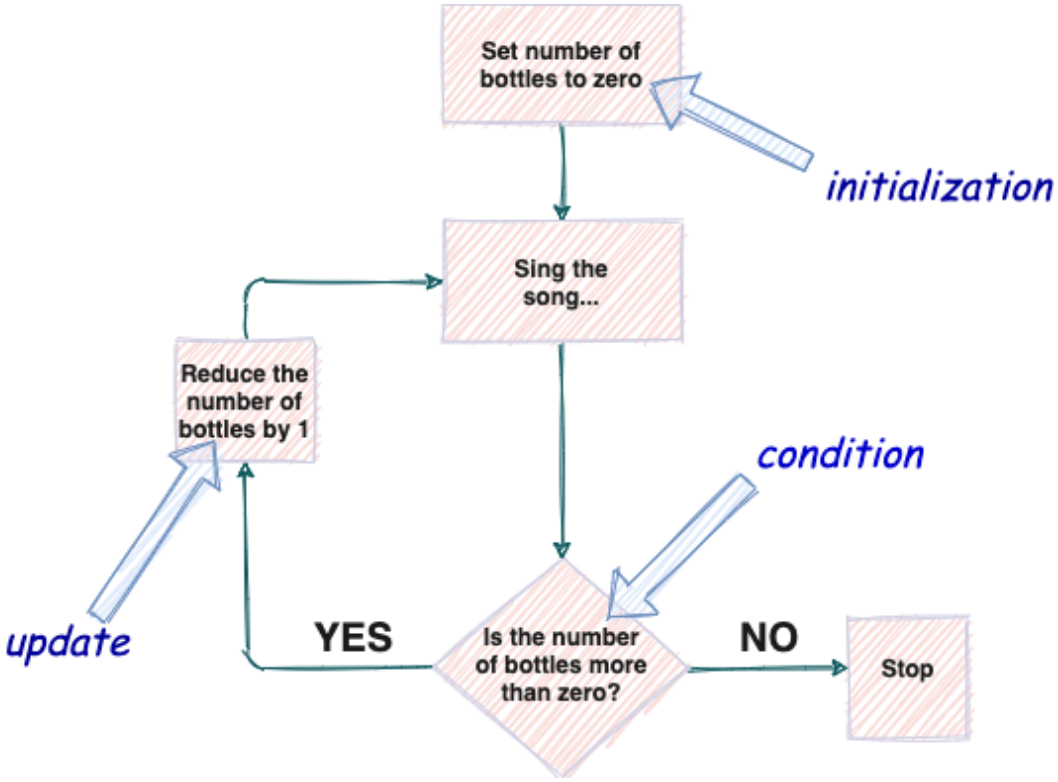
Three things get set inside the parentheses:

- The **initialization** code is run *before* the loop starts and is usually employed to initialize any variables that are used in the loop.
- The **condition** has to be satisfied for the loop to continue.
- The **update** code is what to do after *each iteration* of the loop, and it is typically used to update any values.

This process is illustrated in the following diagram.



For the Ten Green Bottles song, we would have to initialize the number of bottles to 10. The condition would be that this number has to be more than zero and the update code would be to reduce the number of bottles by 1. This is shown in the diagram below.



Let's try writing this in JavaScript. Enter the following code in the console:

```
for (let bottles = 10 ; bottles > 0 ; bottles--) {
  console.log(`There were ${bottles} green bottles, hanging on a
wall. And if one green bottle should accidentally fall, there'd
be ${bottles-1} green bottles hanging on the wall`);
}
```

This initializes the variable `bottles` to 10, then sets the condition to be `bottles > 0`, and uses the decrement operator `bottles--` to reduce the value of the `bottles` variable by one after every loop.

You can see my code on CodePen.

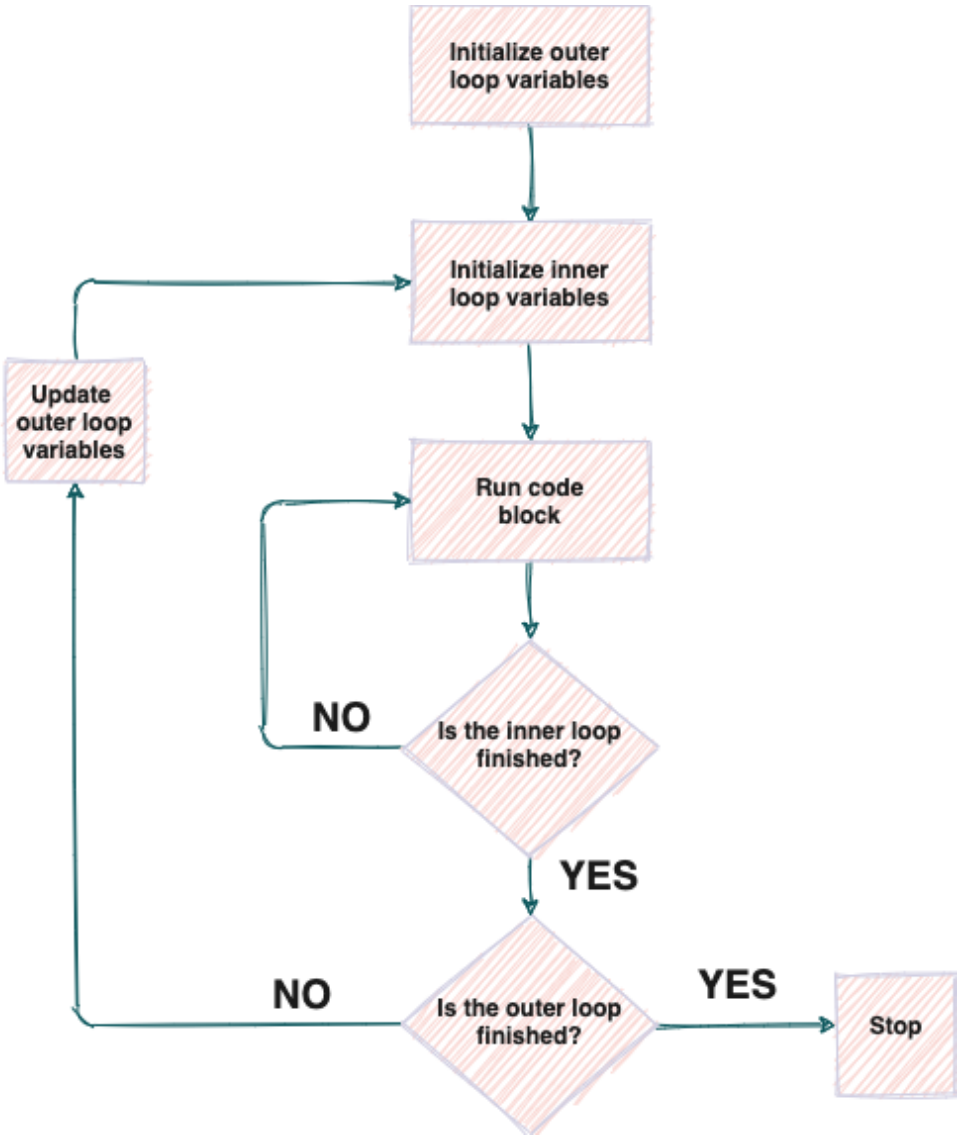
The results should be exactly the same as before. In fact, you may have noticed that it's possible to use a while loop, a do–while loop, or a for loop

to achieve exactly the same results.

A for loop is generally considered clearer (and is definitely more popular looking at code examples around the Web), probably because all the details of the loop (the initialization, condition, and update code) are all in one place and kept out of the actual code block.

Nested Loops

You can place a loop inside another loop to create a **nested loop**. These have an inner loop that runs all the way through before the next step of the outer loop occurs. This can be seen visually in the diagram below.



As you can see, the inner loop runs all the way through for every step of the outer loop. Once the inner loop has run all the way through, the variables for the outer loop are updated and it runs the inner loop all the way through again. This continues until the condition has been met for the outer loop to stop.

Let's write a program that creates a set of multiplication tables, going from a "one times" table up to a "twelve times" table. That is, 1 x 1 = 1, 1 x 2 = 2, 1 x 3 = 3 … all the way up to 12 x 12 = 144.

To achieve this, we want the outer loop to set the first number to 1 and then use the inner loop to multiply it by a second number that will go from 1 to

12 in each step. After we've multiplied 1 by every number, we'll increase the first number to 2 in the outer loop and multiply this by all the numbers from 1 to 12 in the inner loop, and so on, until we get to 12 multiplied by 12.

Enter the following code in the console:

```
// outer loop
for(let i=1 ; i<13 ; i++){
  // inner loop
  for(let j=1 ; j<13 ; j++){
    console.log(`${j} multiplied by ${i} is ${i*j}`);
  }
}
```

Loop Variables

We've created two variables in the code above, `i` and `j`. In loops, it's traditional to use a single letter for these "counter" variable names. Why? Because it's nice and neat! But you could use any variable name (as we did for our Ten Green Bottles for loop), of any length. But why write longer variable names than you need to?

It's important to note that you need to use a different letter for the variable in each nested loop, so that they don't get confused. In this example, we're using `i` in the outer loop and `j` in the inner loop.

The outer loop increases the variable `i` from 1 to 12. For every step of the outer loop, the inner loop increases the variable `j` from 1 to 12. So the first iteration starts with `i = 1` and `j = 1` and logs the following output to the console:

```
<< 1 multiplied by 1 is 1
```

In the next step, we're still inside the inner loop, so `i` remains as 1, but `j` increases to 2, giving this:

```
<< 1 multiplied by 2 is 2
```

`j` continues to increase until it reaches 12. After this, the program breaks out of the inner loop and returns to the outer loop, where `i` is updated from 1 to

2. It then re-enters the inner loop and `j` is initialized back to 1 and begins counting up to 12 again. This continues until the last iteration produces this line:

```
<< 12 multiplied by 12 is 144
```

Run this in the console. You should see an output in the console similar to that pictured below.

You can see my code on CodePen.

Challenges

1. Write a loop that counts from 1 to 100 and checks if the number is a multiple of 3 or 5. If it's a multiple of 3, it should log "Fizz" to the console. If it's a multiple of 5, it should log "Buzz" to the console. If it's a multiple of 3 and 5, it should log "FizzBuzz" to the console, and if it's a multiple of neither, it should just log the number to the console. The initial output should look something like this:

   ```
   << 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz,
   13, 14, FizzBuzz, ...
   ```

   Hint: you'll need to use the modulus operator (`%`) for this challenge. We discussed the modulus operator in Chapter 3. You can see my code on CodePen.

2. In the last chapter, we wrote some code for a higher or lower game, but it only gave the user one go at guessing higher or lower. Write some code that allows the user to keep guessing until they get it wrong. It should also tell them how many times they managed to guess correctly at the end of the game. You can see my code on CodePen.

3. In the last chapter, we wrote a times table question game (challenge 3). Modify the code so that it asks five questions and keeps score of how many the user gets right. You can see my code on CodePen.

Summary

- A loop is a block of code that runs over and over again until a certain condition is met.
- An infinite loop is a loop that never stops, as it's impossible to meet the condition for breaking out of the loop.
- A while loop will continue to run as long as a particular condition is true.
- A do–while loop will also run while a condition is true, but the condition comes at the end, rather than the beginning of the loop.
- A for loop sets the initial value, a condition for stopping, and increment at the start, and then runs a block of code until the condition is met. After every loop, the increment instruction is carried out.
- Nested loops can be formed by running a loop within another loop. The inner loop runs all the way through for *every* pass of the outer loop.

In the next chapter, we'll be learning all about functions—which are a fundamental part of any programming language!

Chapter 8: Functions

So far, we've learned quite a bit about coding, and if you've been completing the challenges, you've already put together some small bits of code. Wouldn't it be great if we could save those bits of code and run them at any time in a program? That's what we can use functions for!

In this chapter, we'll cover:

- defining functions
- calling a function
- return values
- parameters and arguments
- callbacks
- choosing the right type of function

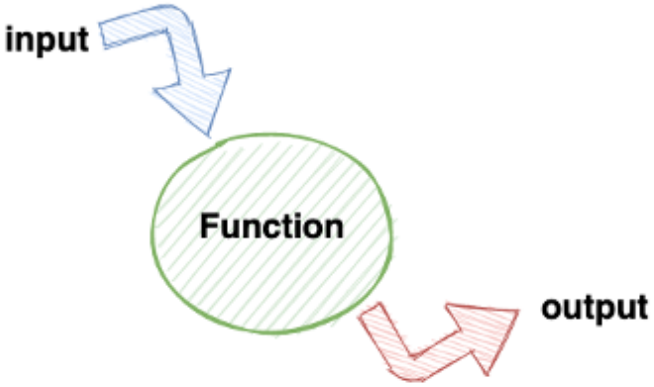
A **function** is a block of code that's almost like a small, self-contained mini program. They can help to reduce repetition and make code easier to follow.

Functions are a perfect application of the DRY principle ("don't repeat yourself"). We should always be trying to not repeat ourselves, thus cutting down on the amount of code we write. We do this all the time in real life. For example, we might say we're going to "bake a cake", which is far easier than saying we're going to "get all the ingredients, mix them together, put them in the oven for 20 minutes and then wait for it to cool …" That would take you a long time! By understanding what's meant by the phrase "bake a cake", we don't have to repeat the whole recipe and instructions for actually making it every time.

It works the same with programming. A block of code that we want to reuse can be wrapped into a function. Then, whenever we want to complete that same task, we just need to call on that function, instead of rewriting lots of lines of code every time.

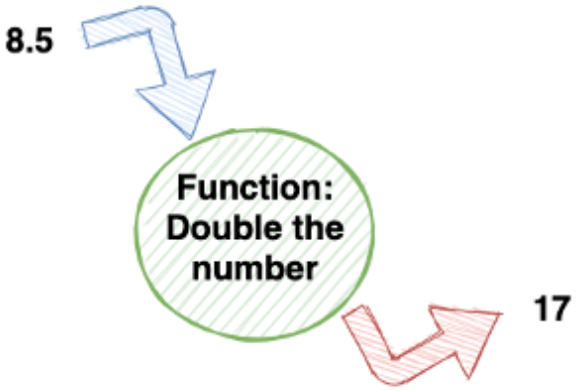
Another good thing about functions is that you can pass data to them, get them to do something with that data, and then have them output the results.

One way of thinking about this is to imagine the function as a machine that we put values into. The machine does something to those values and then outputs some new values.



In the cake example, the input is the ingredients, the process is the mixing and baking, and the output is the cake itself.

An example of a function might be one that doubles a number. The input would be a number, the process would be multiplying this number by 2, and the output would be the answer.



Subroutines, Procedures and Functions

A named block of code is known as a **subroutine**. If the code just performs a task, it should technically be called a **procedure**. It's only if the block of code outputs (or **returns**) a value that it's called a **function**.

Some languages, such as Ruby, differentiate between procedures and functions and will use a different syntax to create them. JavaScript doesn't do this, and only uses functions to create named blocks of code.

JavaScript functions will return a value of `undefined` if a return value is not explicitly specified.

Functions in JavaScript

Functions are an essential part of any programming language. They're the fundamental building blocks of most JavaScript programs, which makes understanding them an essential skill to master.

Defining a Function

The most common way to define a function in Javascript is to use a **function declaration** like the one shown below. Try writing the following in the console:

```
function hello(){
  alert('Hello, World!');
}
```

This creates a function called `hello`. The code block that describes what the function does is inside the curly braces. This code will create an alert box that displays the message “Hello, World!”

Defining the function doesn’t actually do anything, though. In order to actually run the code inside the function, you have to “call” it.

Calling Functions

Calling a function will run the block of code inside the function’s body. To call the `hello()` function that we defined above, just type the following in the console and press `Enter`:

```
hello();
```

This should produce an alert box like the one shown below.

The function can be called over and over again just by typing its name followed by parentheses (try it … until you get bored of having to keep closing those alert boxes!).

This is one of the advantages of using functions: there's no need to keep writing repetitive blocks of code. Another advantage is that all the functionality is kept in one place. So if you want to change part of it, you only need to update the one piece of code in the function declaration. For example, if the alert box is really starting to annoy us, we can just edit the `hello()` function to be this instead:

```
function hello(){
  console.log('Hello, World!');
}
```

Now the message will be logged to the console every time the function is called, rather than displayed in those annoying pop-up alert boxes. Imagine if this code had been used hundreds of times throughout our program: it would be such a pain having to change every instance of it. By using functions, we only have to change the code in one place, and *every* instance of it will be updated. This a perfect example of the DRY principle in action.

Function Expressions

Another way of defining a function is to create a **function expression**. This assigns an "anonymous function" to a variable, as you can see in the example below:

```
const goodbye = function(){
  alert('Goodbye, World!');
};
```

An **anonymous function** is created using the same function declaration we used earlier, but it doesn't have a name assigned to it. Instead, the function is assigned to a variable that's used to refer to the function. In the example above, the variable is called `goodbye`.

Assignment and Semicolons

Notice also that the example above ends with a semicolon. This finishes the assignment statement, whereas a normal function declaration ends in a block (semicolons are not placed at the end of a block).

A function expression is called in the same way as a function declaration—by writing the name of the variable it was assigned to, followed by parentheses:

```
goodbye();
```

Calling vs Referencing a Function

It's important to remember that you need parentheses to call a function. If you forget to add the parentheses, you're just referencing the actual function, rather than calling it.

Arrow Functions

Arrow functions were only added to JavaScript in 2015, and they look very different from the other ways of writing functions. They use a less verbose syntax, making them quicker to write, and they're identified by the "arrow" symbol (=>) that gives them their name.

Arrow functions are always anonymous, so if you want to refer to them, you must assign them to a variable, like we did with function expressions. The example below shows how to create an arrow function that's assigned to the variable `hola`:

```
const hola = () => alert('Hola, Mundo!');
```

The arrow function starts with the parentheses and arrow, `() =>`. Everything after that is the code that will run when the function is called. Notice that this code doesn't need to be put inside any curly braces (as long as it's only one line of code), which makes the function appear much more concise.

Arrow functions are called in exactly the same way as any other function: just write its name—and don't forget to add those parentheses at the end:

```
hola();
```

Return Values

Functions always **return** (or output) a value. This can can be specified in the body of the function using the `return` keyword.

For example, the following function will return the string `'Howdy, World!'`:

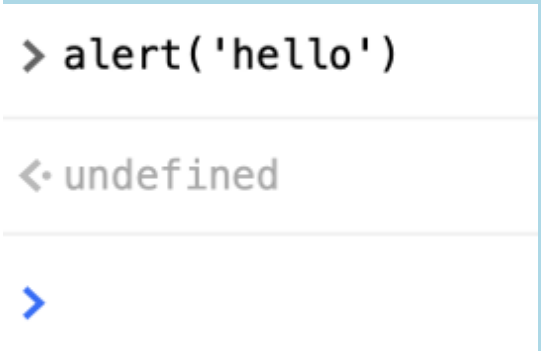
```
function howdy(){
  return 'Howdy, World!';
}
```

Return values can be any type of value—even functions! (This is covered in later chapters.)

Returning Undefined

If a return value isn't explicitly stated, it will return `undefined` by default. This is because a function must return a value in order to be considered a function. You'll often see `undefined` appear in the console after calling a

function that in essence is just a procedure and doesn't need to return a value. The `alert` function that we've been using is an example of this.



When using arrow functions, you don't even need to explicitly use the `return` keyword if the body of the function is just one line of code. If this is the case, the result of that code is the return value. For example, the `howdy` function above could be written using arrow notation like so:

```
const howdy = () => 'Howdy, World!';
```

Short, one-line functions like this are good candidates for using arrow functions.

Parameters and Arguments

Parameters and **arguments** are terms that are often used interchangeably to represent values that are provided for the function as an input. There's a subtle difference though: the parameters of a function are set when the function is *defined*, whereas the arguments of a function are provided when it is *called*.

To see an example of a function that uses parameters, let's create a function that squares numbers. In the example that follows, the `square` function accepts a single parameter, `n`, which is the number to be squared. In the body of the function, the name of the parameter acts just like a variable. We multiply this value by itself and return the result:

```
function square(n){
  return n*n;
}
```

When we call this function, we need to provide an argument, which takes the place of the parameter in the definition and is the number to be squared:

```
square(4.5);
<< 20.25
```

When defining arrow functions with a single parameter, the parameters come before the arrow and the main body of the function comes after. For example, the `square` function can be written like so:

```
const square = n => n*n;
```

Notice that arrow functions don't need parentheses around the parameter, making them even more succinct.

You can use as many parameters as you like when defining functions. For example, the following function finds the mean of any three numbers by adding them together and dividing the result by three:

```
function mean(a,b,c){
  return (a+b+c)/3;
}
```

Let's run that in the console:

```
mean(8, 3, 4);
<< 5
```

When using more than one parameter with arrow functions, you need to place them in parentheses, so the `mean` function would be written in arrow notation like this:

```
mean = (a,b,c) => (a+b+c)/3;
```

If a parameter isn't provided as an argument when the function is called, the function will still be called, but the missing argument will be given a value of `undefined`. For example, if we tried to call the `mean` function with only two arguments, it would return `NaN`, which is the result of trying to add a number to `undefined`. Try entering the example below in the console to see this:

```
mean(1,2);
<< NaN
```

```
> mean = (a,b,c) => (a+b+c)/3;
  mean(1,2);
```

```
<· NaN
```

If too many arguments are provided when a function is called, the function will work as normal and the extra arguments will just be ignored. The example below will return the mean of 1,2 and 3 and ignore the arguments of 4 and 5:

```
mean(1,2,3,4,5);
<< 2
```

Default Parameters

JavaScript provides a convenient way to specify default parameters for a function. These are values that will be used by the function if no arguments are provided when it's called. To specify a default parameter, you simply assign the default value to it in the function definition. For example, the following function accepts a parameter that's then added to the end of the string `'Hello '`. The parameter is given a default value of `'World'`:

```
function hello(name='World') {
  alert(`Hello, ${name}!`);
}
```

Now if we call this function without any arguments, it will default to using `'World'` as the `name` argument:

```
hello();
<< 'Hello, World!'
```

If you don't want to use `'World'` as the argument, you can just override this default value by specifying a different value as the argument. For example, how about saying hello to the universe?

```
hello('Universe');
<< 'Hello, Universe!'
```

Default parameters should always come after non-default parameters. Otherwise, you'd just have to enter default values when you call the function anyway. Consider the following function that calculates the discounted price in a store:

```
function discount(price, amount=10) {
  return price*(100-amount)/100;
}
```

This function takes two arguments: the price of an item, and the percentage discount to be applied. The store's most common discount is 10%, so this is provided as a default value. This means that the `amount` argument can be omitted in most cases and a 10% discount will still be applied. For example, you could calculate the sale price of an item that was $20 with a 10% discount by using the following code:

```
discount(20);
<< 18
```

Note that we didn't have to specify that the discount was 10%.

If a different discount needed to be applied, the `amount` argument can be provided. In the following example, an item that was $15 is discounted by 20%:

```
discount(15, 20);
<< 12
```

This will fail to work, however, if the parameters are defined in reverse order, like in the example below:

```
function discount(amount=10, price) {
  return price*(100-amount)/100;
}
```

Now if we try to use the function with just one argument, the function won't work, because `price` hasn't been set:

```
discount(20);
<< NaN
```

This set `amount` to 20, but there was no argument provided for `price`, so the calculation could not be completed and `NaN` was returned.

It will still work, if both values are entered, however:

```
discount(10,20);
<< 18
```

This somewhat defeats the object of having default parameters! The golden rule to remember here is to always put default parameters *after* all the other parameters.

Random Integers

We met the `Math.random()` method in the Numbers chapter (Chapter 4), which creates a random decimal between 0 and 1. We often need a random integer value in programs. In fact, we used the following code to choose a random number between 1 and 10 in the higher or lower game from the last chapter:

```
Math.ceil(Math.random()*10);
```

It's a useful exercise to abstract this code into a "helper" function that you can use whenever you want to generate a random integer.

In the code above, the value of 10 gives the upper bound of the range of numbers, so we could make this a parameter of the function. This would allow us to choose the upper bound when we call the function. The function might look like this, written as a function declaration:

```
randomInt(n){
  return Math.ceil(Math.random()*n);
}
```

This will return a random integer between 1 and the number provided as an argument. For example, we can generate a random number between 1 and 6 using the following code:

```
randomInt(6);
<< 3
```

It would also be useful if we could add an optional lower bound to this function. A lot of the time, when we require a random integer, the lower bound will be 1, so we'll set 1 as the default value:

```
randomInt(upper,lower=1){
  return Math.floor(Math.random()*(upper-lower+1)+lower);
}
```

This will now return a random integer between the lower and upper values provided as arguments. For example, the following code will return a random integer between 4 and 7:

```
randomInt(7,4);
<< 5
```

Because we've given the lower bound a default value of 1, we can use it as before with a single argument to generate a random number between 1 and 10:

```
randomInt(10);
<< 9
```

But it's slightly annoying that we have to provide the arguments as upper bound followed by lower bound. This is because any default parameters have to come last. However, there's a trick for getting around this!

If the second argument isn't provided, it's given a value of `undefined`. We can check for this and assign the values of the upper and lower bound accordingly:

```
function randomInt(lower,upper){
  if(upper===undefined){
    upper = lower;
    lower = 1;
  }
```

```
  return Math.floor(Math.random()*(upper-lower+1)+lower)
}
```

This code uses an if block to check if the second argument was provided, by checking if it has a value of `undefined`. If this is the case, we set the upper bound to equal the argument provided (stored in the variable `lower`) and then set `lower` to be our default value of 1.

Now we can provide the arguments in a more logical order:

```
randomInt(4,7);
<< 6
```

You can see my code on CodePen.

Assigning Return Values to Variables

An important concept in JavaScript is that functions are considered to be **first-class objects**. This means that they behave like all the other data types: they can be be assigned to variables and can even be used by other functions.

We can also assign the return value of a function call to a variable. To see this in action, try running the following code in the console:

```
const function hello(){
  return 'Hello, World!';
}

const message = hello();
<< 'Hello, World!'
```

The variable `message` now points to the return value of the `hello()` function, which is the string `'Hello, World!'`, as can be seen if we run the following code to display the value of the variable:

```
message;
<< 'Hello, World!'
```

This might all seem to be a lot of work and a bit pointless: why not just assign the string to the variable directly? The answer is that we can create more complex functions that have different return values depending on

certain conditions. This technique will then allow us to assign different values to variables depending on the outcome of a function.

For example, say we wanted to check a person was over 18. We could write a function that accepted their age as an argument and returned `true` if they were over 18, or `false` if they weren't. It might look something like this:

```
userIsChild = function(age){
  if(age<18){
    return true;
  } else {
    return false;
  }
}
```

Then, later in the program, we might want to restrict the access of some parts of the site to children using a variable called `restrictedAccess` that's true if the user enters an age under 18. We could use the following code to do this:

```
const restrictedAccess = userIsChild(age);
```

The variable `restrictedAccess` will now be true or false based on the return value of the `userIsChild` function.

Callbacks

A function that's passed as an argument to another function is known as a **callback**. The callback can then be called from within the body of the function that it's an argument of. Being able to call different functions from within functions makes functions even more flexible.

To see an example of this, let's look at the following function called `bake`. This is a JavaScript example of the baking analogy we used for functions at the start of the chapter. It accepts a string of ingredients as a parameter and then logs some messages to the console. Try entering the following in the console:

```
function bake(ingredients,callback){
  console.log(`Mix together ${ingredients}...`);
```

```
  console.log('Bake in the oven...');
}
```

Try calling the function with an argument like the one in the example below:

```
bake('flour, water & sugar');
<< "Mix together flour, water & sugar..."
<< "Bake in the oven...'
```

We can improve this function by adding a callback as a second parameter. This is a function that's called from within the `bake` function and allows us to add some extra information. Update the `bake` function by entering the code below into the console:

```
function bake(ingredients,callback){
  console.log(`Mix together ${ingredients}`);
  console.log('Bake in the oven');
  callback();
}
```

Overwriting a Function Declaration

You can overwrite a function declaration by declaring it again later in the code, but this won't work if you've assigned a function expression to a variable using `const`.

Now we can call the `bake` function with a callback that logs another message to the console. In the example below, the callback is an anonymous arrow function:

```
bake('flour, water & sugar', () => console.log('Add icing on
top...'));
<< "Mix together flour, water & sugar..."
<< "Bake in the oven...'
<< "Add icing on top..."
```

The callback doesn't have to be an anonymous function that's defined in the function call. We can define a named function and then provide the name as an argument. For example, enter the following function in the console:

```
function eat(){
  console.log('Eat every last crumb!');
```

```
}
```

Again, this just logs another message to the console, which isn't very exciting, but fine for demonstration purposes. Let's try providing this function as an argument to the `bake` function:

```
bake('flour,water & sugar', eat);
<< "Mix together flour, water & sugar..."
<< "Bake in the oven...'
<< "Eat every last crumb!"
```

Notice that the callback isn't followed by parentheses when it's provided as an argument. This is because it's called later, in the body of the function.

Callbacks are used extensively in many JavaScript functions, and we'll see much more of them later in the book. In fact, we'll finish the chapter with a practical example that solves a problem we encountered earlier in the book.

Sorting Arrays with a Callback

In the Arrays chapter (Chapter 5), we saw that arrays have a `sort()` method that sorts the items in the array into alphabetical order. This is fine for strings, but you might recall that it doesn't work so well for numbers:

```
[1,3,12,5,23,18,7].sort();
<< [1, 12, 18, 23, 3, 5, 7]
```

The reason for this is that the numbers are converted into strings and then placed in alphabetical order, so `'12'` comes before `'3'` because it starts with a `'1'`.

So how do you sort an array of numerical values? The answer involves providing a callback as an argument to the `sort()` method. This callback tells the method how to compare any two values in the array. Let's call them `a` and `b`. The callback function should return the following:

- a negative value if `a` comes before `b`
- `0` if `a` and `b` are equal
- a positive value if `a` comes after `b`

An easy way to sort values numerically is to use subtraction: `a-b`. This will return a negative value if `b` is bigger than `a`, zero if `a` and `b` are equal, or positive if `a` is bigger than `b`. We can do this by adding an anonymous arrow callback function as the second argument, like so:

```
[1,3,12,5,23,18,7].sort((a,b) => a-b);
<< [1, 3, 5, 7, 12, 18, 23]
```

The results returned by the callback function help the `sort()` method understand whether an item is bigger or smaller than another, so that it can then order the items numerically.

Sorted!

Sorting Through `sort()`

In programming, you'll often come across tricky situations that require sophisticated solutions. Thankfully, others have walked this path before you, and there are ready-made solutions to almost any problem. The solution used above to sort numbers is a small example: it's a snippet of code that's been worked out by others needing to do the same thing. You don't actually need to understand how it works. Still, it never hurts to dig more deeply into the workings of JavaScript, so if you want a more in-depth explanation of how the `sort()` method works, check out these two articles:

- "Sophisticated Sorting in JavaScript"
- "How to Sort an Array of Objects in JavaScript"

Choosing the Right Type of Function

In this chapter, we've covered three different ways of creating functions. There are some subtle differences between them that will be covered in more detail later in this book. Function declarations are very similar to how functions are declared in other languages, whereas function expressions behave just like any other value assigned to a variable, rather than being a special feature of the language.

Arrow functions look very different from function declarations and expressions, and they can also produce different results in some situations, meaning they're not always suitable as a like-for-like replacement. They can be quicker to write, since they don't need curly braces around the body of the function and the `return` keyword isn't needed if it's only a single line of code. These advantages make arrow functions particularly popular for quick, one-line functions, and you'll see quite a few examples later in the book. Many of the advantages of arrow functions tend to disappear, however, once the body of the function becomes longer than one line.

Which method you choose can often be as much about personal taste as about which is best suited to the situation at hand. By the time you've reached the end of this book, you'll hopefully be well equipped to choose the best type of function to use in any particular situation.

Challenges

Now that you've learned how to define and call functions, it's time to have a go at creating some functions. Here are some challenges to help develop your programming skills.

1. Write a function called `lastChar` that accepts a string as an argument and returns the last letter of the string. For example, `lastChar('JavaScript')` should return `'t'`. You can see my code on CodePen.

2. Write a function called `reverse` that accepts a string and returns the string written backwards. For example, `reverse('JavaScript')` should return `'tpircSavaJ'`. Applying the function twice should return the original value, so `reverse(reverse('JavaScript'))` should return `'JavaScript'`. You can see my code on CodePen.

3. Write some functions that will add and remove some items from a list. The list should be stored as an array and `add('apples')` should add the string `'apples'` to the array and `remove('apples')` should remove `'apples'` from the array. You can see my code on CodePen.

Summary

- Functions can be defined using the `function` declaration, or by creating a *function expression* by assigning an anonymous function to a variable.
- Arrow functions are a shorthand notation that can used for writing short, one-line, anonymous functions.
- All functions return a value. If this isn't explicitly stated, the function will return `undefined`.
- A parameter is a value that's written in the parentheses of a function declaration and can be used like a variable inside the function's body.
- An argument is a value that's provided to a function when it's called.
- The return value of a function can be assigned to a variable.
- A callback is a function that's provided as an argument to another function and can be called from within the body of that function.

Functions are a really important part of any programming language. But so are "objects"—and we'll be learning about them in the next chapter!

Chapter 9: Objects

Objects are abstract data types used in programming. They can be used to represent real-world objects such as people, animals or places, as well as more abstract concepts such as bank account details, dates or file structures.

In some languages, such as Ruby, practically everything is an object. There are no primitive values, just objects. Even the number zero is an object. A large number of languages, such as Java, are **class-based** languages. These involve creating classes that act as a blueprint for creating all the objects in that class. For example, a `Car` class might specify that all `Car` objects will have an engine, four wheels and a choice of color.

JavaScript is *not* a class-based language (although you can define classes if you want to, as we'll see later in the book). It allows you to create objects quickly and easily without the need for defining a class first.

In this chapter, we'll be covering the following topics:

- properties and methods
- creating object literals
- object properties
- object methods
- nested objects
- `this`

Properties and Methods

Objects can have *properties* and *methods*. **Properties** are information about the object and **methods** are actions that the object can perform.

For example, we could represent an octopus using the following object model.

Octopus

Properties
- Name: “Octavius”
- Legs: 8

Methods
- Swim
- Walk
- Regrow leg *(yes, they can really do that!)*

Objects are often used to keep any related information and functionality together in the same place. For example, if you were writing some code involving a square, you might create a “square” object with the following properties and methods.

Square

Properties
    Length: 5
Methods
    Perimeter = 4 * Length
    Area = Length * Length

Creating Objects in JavaScript

Object literals are a distinguishing feature of the JavaScript language, as they allow objects to be created quickly without the need for defining a class, which is common in many other languages. An **object literal** is a self-contained set of related values. Properties can be of almost any data type, such as numbers, strings, Booleans, arrays, or even other objects. If a property is a function, then it's known as a **method**.

To create an object literal in JavaScript, simply enter a pair of curly braces. The following example creates an empty object that's assigned to the variable `myObject`:

```
const myObject = {};
```

You don't have to start with an empty object. You can place the properties and methods inside the curly braces. The following code could be used to represent information about a square:

```
const square = {
    sides: 4,
    length: 5,
```

```
    perimeter: 20,
    area: 25
}
```

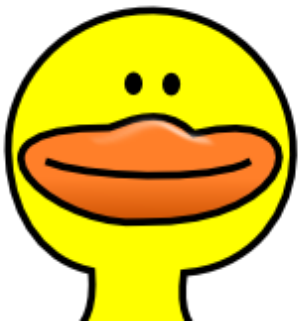
Each property has a **key** and an associated **value**. In the example above, the first property has a key of `sides` and its value is 4. Each key–value pair is separated by a comma.

Trailing Commas

Notice above how the last key–value pair doesn't end with a comma. These days, this "trailing comma" is optional, but before ECMAScript 5, a trailing comma would have thrown an error. As long as your code doesn't need to work in older browsers (which is an important consideration), including a trailing comma can be useful if you're likely to add further key–value pairs to your object later on, because it means you don't have to remember to add the comma later. As a general rule, I won't be using trailing commas in this book.

Methods

Methods look similar to the other properties, but are defined as a function. The following code shows how we would create an object literal to represent a duck with a `quack()` method:



```
const duck = {
  name: 'Quacky',
  legs: 2,

  quack: function() {
    alert('QUACK! QUACK!');
```

```
  }
};
```

There's also an alternative notation for describing methods that omits the `function` keyword and just places parentheses after the method name:

```
const duck = {
  name: 'Quacky',
  legs: 2,

  quack() {
    alert('QUACK! QUACK!');
  }
};
```

This way of writing methods was only introduced into JavaScript quite recently, but has become popular due to it requiring less typing.

Guess Again

Remember the four characters from our mini game of Guess Who? in Chapter 6?

We could create objects to represent each of them. Each object would contain properties describing their name and Boolean values to indicate if they're wearing glasses and a hat or not. For example, the following object would represent Alfie:

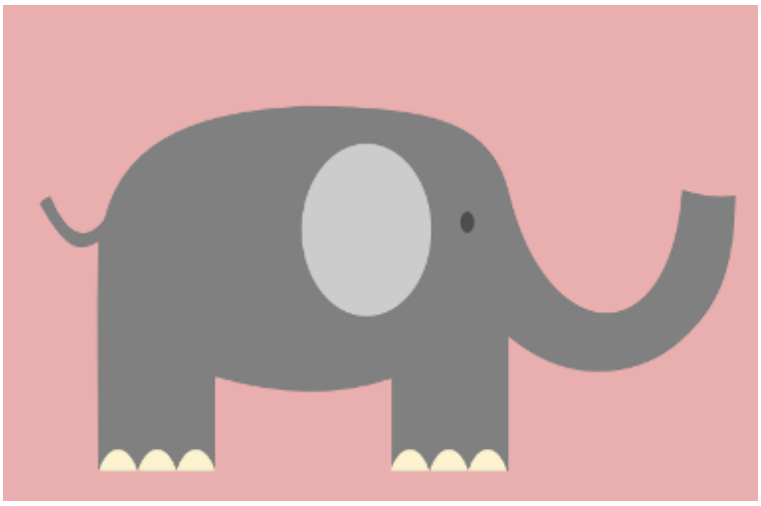
```
const alfie = {
  name: 'Alfie',
```

```
  picture: '👨‍🦲',
  glasses: no,
  hat: no
}
```

Creating Objects from Variables

We can also create objects using variables that already exist, as can be seen in the example below:

```
const name = 'Dumbo';
const legs = 4;
const fly = () => console.log('Fly, fly away!');

const elephant = { name, legs, fly };
```



Notice that if the variables have already been declared and assigned to values or functions, you only have to place the variable names inside the object literal and it will assign the properties and methods to match the variables.

This is an example of **object property shorthand syntax** that makes it easier to create an object with property and method names that are the same as variables and functions that already exist. The long-hand version would involve some pointless repetition when creating the `elephant` object:

```
const elephant = {
  name: name,
  legs: legs,
```

```
  fly: fly
}
```

Properties and Methods

You can access the properties of an object using the dot notation that we've already seen in previous chapters. The example below shows how we can access the `name` property of the `elephant` object:

```
elephant.name;
<< "Dumbo"
```

You can also access an object's properties using bracket notation. The property is represented by a string inside square brackets, so it needs to be placed inside single or double quotation marks:

```
elephant['name'];
<< "Dumbo"
```

The dot notation for accessing properties is by far the most common, but bracket notation can come in handy if the name of the property is stored as a string. For example, if you had a variable called `info` that was equal to the string `'name'`, the following code would be equivalent to writing `elephant.name`:

```
const info = 'name';
elephant[info]
<< "Dumbo"
```

This wouldn't work if you used `elephant.info` because JavaScript would try to find a property called `info`.

In most other cases, though, you should stick to using the dot notation.

If you try to access a property that doesn't exist, `undefined` will be returned:

```
elephant.arms;
<< undefined
```

The `in` operator can be used to check whether an object has a particular property. So, for example, we can check if the `elephant` object has

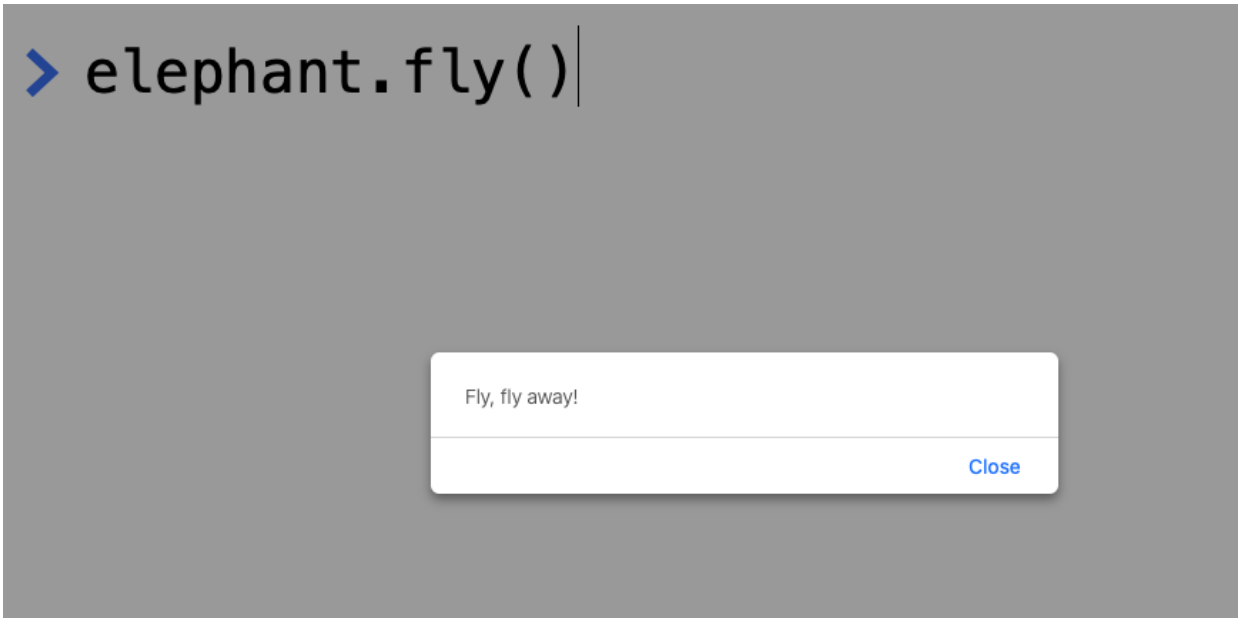
properties called `arms` or `legs` using the following code:

```
'arms' in elephant;
<< false

'legs' in elephant;
<< true
```

Calling Methods

Methods are just like functions, so we call them in the same way. We can refer to the object's method using the dot or bracket notation, just like with properties, but with parentheses at the end, so we would use the following code to call the `fly` method:

```
elephant.fly()
<< "Fly, fly away!"
```



You can also use the square bracket notation to call methods:

```
elephant['fly']();
<< "Fly, fly away!"
```

Adding More Properties and Methods

Objects are **mutable** by default, which means that their properties and methods can be changed or removed and new properties and methods can be added, even if they were declared using `const`.

Frozen Objects

It's possible to *freeze* an object and make it immutable. We won't be going into that here, but you can can read more about it on the Mozilla Developer Network site.

New properties and methods can be added to objects at any time in a program. This is done by simply assigning a value to the new property. For example, if we wanted to add a new `ears` property to our `elephant` object, we would do it like so:

```
elephant.ears = 2;
<< 2
```

Now if we take a look at the `elephant` object, we can see that it has the `ears` property:

```
elephant;
<< { name: 'Dumbo',
    legs: 4,
    ears: 2,
    fly: [Function: fly]  }
```

Changing Properties

You can change the value of an object's properties at any time using assignment. For example, we can change the value of the `name` property like this:

```
elephant.name = 'Elmer';
<< 'Elmer'
```

We can check the update has taken place by taking a look at the object:

```
elephant;
<< { name: 'Elmer',
    legs: 4,
```

```
    ears: 2,
    fly: [Function: fly]  }
```

Removing Properties

Any property can be removed from an object using the `delete` operator. For example, if we wanted to remove the `fly` method from the `elephant` object (because Elmer can't fly, of course!), we would enter the following:

```
delete elephant.fly;
<< true
```

Now if we take a look at the `elephant` object, we can see that Elmer can no longer fly:

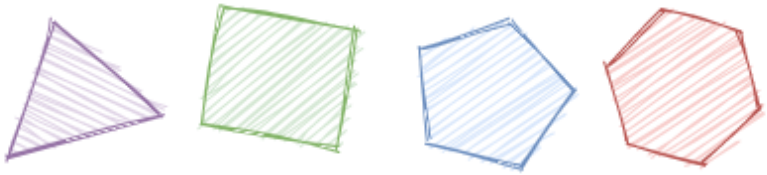
```
elephant;
<< { name: 'Elmer',
    legs: 4,
    ears: 2  }
```

Nested Objects

It's even possible for an object to contain other objects. These are known as **nested objects**.

Here's an example of an object that contains a group of nested objects. Each nested object represents a different shape. The container object has been assigned to the variable `shapes`:

```
const shapes = {
  triangle: { sides: 3 },
  square: { sides: 4 },
  pentagon: { sides: 5 },
  hexagon: { sides: 6 },
  octagon: { sides: 8 },
  megagon: { sides: 10e6 },
}
```



The values in nested objects can be accessed by referencing each property name in order using either dot or bracket notation:

```
shapes.triangle.sides;
<< 3

shapes['megagon']['sides'];
<< 1000000
```

You can even mix the different notations (but please don't do this if you can help it!):

```
shapes.hexagon['sides'];
<< 6
```

this

The keyword `this` can be used inside an object to refer to the object itself. It's often used in methods to gain access to the object's properties.

To see an example of when we might use `this`, consider the example of the square object we saw earlier:

```
const square = {
    sides: 4,
    length: 5,
    perimeter: 20,
    area: 25
}
```

The problem with this object is that all the information about the square is hard-coded as properties. This isn't necessary, since we can use methods to calculate the perimeter and area based on the values of the `sides` and `length` properties. We can use `this` to gain access to those properties. Try entering the following code in the console:

```
const square = {
    sides: 4,
    length: 5,
    perimeter() { return this.sides * this.length },
    area() { return this.length * this.length }
}
```

The `perimeter` and `area` properties have now been changed to methods that return the desired value. Inside the body of these methods there are references to other properties of the object: `this.sides` is the equivalent to writing `square.sides`, and is the number 4.

If we call these methods, they should give us the same results as when the corresponding properties were hard-coded earlier:

```
square.perimeter();
<< 20

square.area();
<< 25
```

Using methods in this way makes the object much more flexible. If we change the value of the `length` property, the perimeter and area will also change to reflect this:

```
square.length = 8;
<< 8

square.perimeter();
<< 32

square.area();
<< 64
```

Roll the Dice

Let's finish the chapter with a coding project. We'll create a dice object that has a `sides` property and a `roll()` method that returns a number between 1 and the number of sides.

Here's the code to create our dice object:

```
const dice = {
  sides: 6,
  roll() {
     return Math.ceil(Math.random()*this.sides);
  }
}
```

This object has a `sides` property that tells us that the dice has six sides. It also has a `roll()` method that will return a random number between 1 and 6.

Inside the `roll()` method, we use `this.sides` to refer to the value of the object's `sides` property instead of simply hard-coding the value 6. This will make the object more flexible: if we later decide to change the number of sides, we'll only have to change the `sides` property and won't need to update the roll function.

We also use the `random()` and `ceil()` methods of the `Math` object, which we met in Chapter 4, to return a number between 1 and the number of sides.

Let's take it for a spin:

```
dice.roll();
<< 5

dice.roll();
<< 3
```

Now let's try changing the number of sides by modifying the `sides` property:

```
dice.sides = 20;
<< 20
```

Now the `roll()` method will return a random number between 1 and 20 instead, without us having to modify it:

```
dice.roll();
<< 12

dice.roll();
<< 18
```

Challenges

1. Create objects that describe some of your favorite things. For example, you could create some objects that model your favorite sports stars and provide properties such as `speed`,`power` and `agility`. Or you could create some objects that represent products for sale on an ecommerce site, including properties such as `description`, `price` and `rating`. Have a play around in the console and practice reading and updating the properties and calling the methods. Top Trump cards could be a good inspiration for this challenge, as each card contains a large amount of information about their subject.

2. Write objects that represent Betty, Gemma and Del in the Guess Who? game. You can see my code on CodePen.

3. Create a circle object that has a radius property and calculates the circumference and area. Hint: you might need to use the `Math.PI` constant for this challenge. You can see my code on CodePen.

4. Create a list object that has an array property called `items` and an `add` and `remove()` method that can be used to add items to the list. For example, `list.add('Apples')` should add `Apples` to the `list.items` array. Hint: you could use the functions you created in challenge 3 in the last chapter as methods. You can see my code on CodePen.

Summary

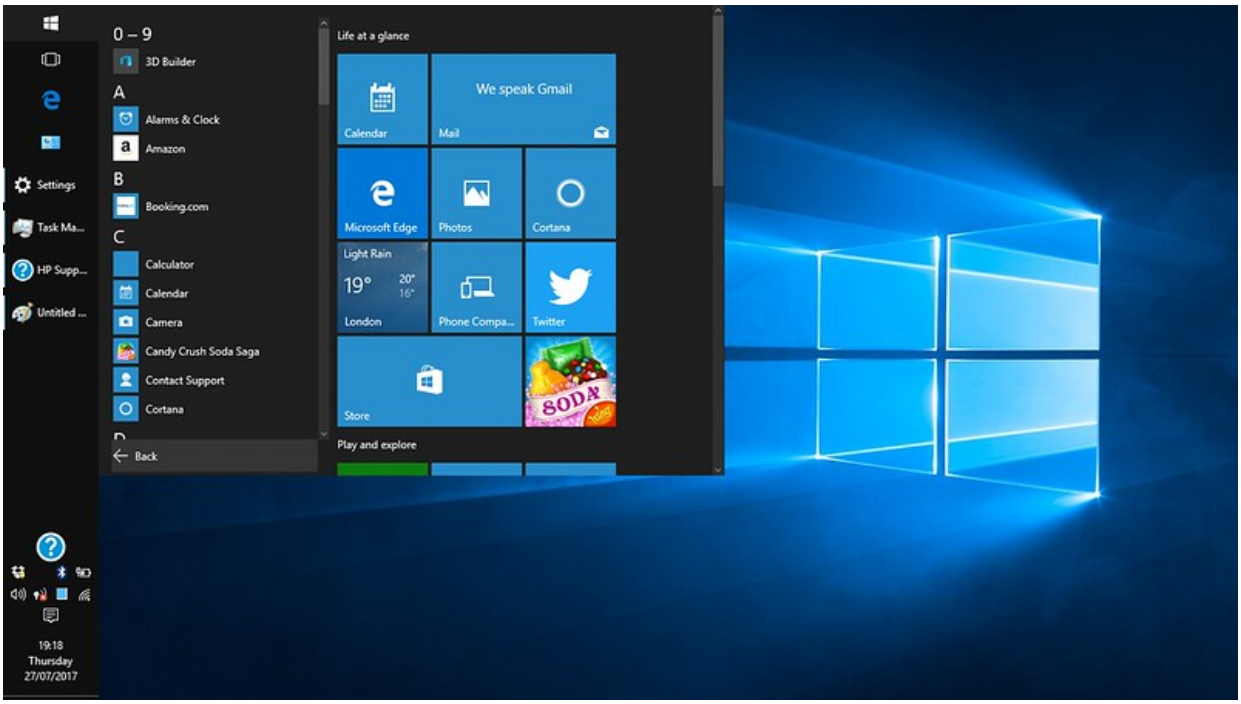
- Objects have properties that contain information about the object, and methods that are actions the object can perform.
- Object literals in JavaScript are a collection of key–value pairs placed inside curly braces (`{}`) and separated by commas.
- Properties can be strings, numbers, Booleans or arrays.
- Methods are stored as functions.
- An object's properties and methods can be accessed using either dot notation or square bracket notation.
- Objects are mutable, which means that their properties and methods can be changed or removed (even if they're created using `const`).

- Nested objects can be created by placing objects inside objects.
- The keyword `this` can be used in properties and methods to refer to the object itself.

Now that we've learned about all the basics of programming, it's time to look at how our code can interact with the browser environment.

Chapter 10: The Document Object Model

So far, we've mainly been interacting with our code in the console or by using pop-up boxes. Since the advent of the computer monitor, computer programs have been able to provide some kind of graphical user interface (GUI) to allow users to interact with programs. For example, operating systems such as Windows or iOS provide GUIs that allow you to interact with your computer or device.



Most programming languages will have a library that allows you to create a GUI. In a browser, we can use HTML and CSS to create a GUI for the user to interact with.

The Document Object Model, or DOM for short, allows us to access the elements of a web page from within our code. It provides tools for adding and removing elements, and for updating the content and style of a page.

The DOM is **language agnostic**, which means that it can be used in any programming language, although JavaScript is the language it's most associated with. We'll be focusing on using JavaScript to access the DOM in this chapter.

We'll be covering the following topics:

- an introduction to the DOM
- getting elements
- navigating the DOM
- getting and setting an element's attributes
- updating the DOM by creating dynamic markup
- changing the CSS of an element

The Document Object Model

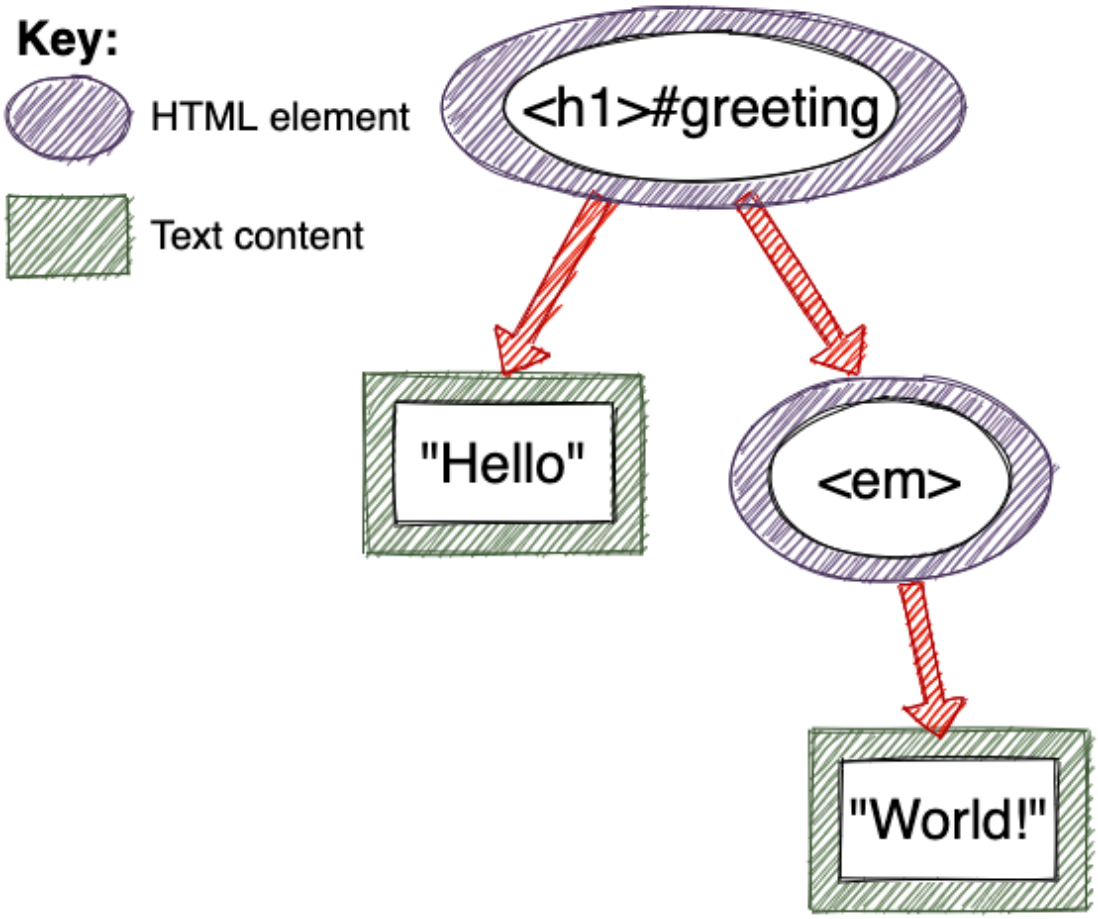
The **Document Object Model** represents an HTML document as a network of connected **nodes** that form a tree-like structure.

The DOM treats everything on a web page as a node. It represents HTML tags as **element nodes** and any text inside the tags as **text nodes**. All these nodes are connected to make a node-tree that describes the overall structure of the web page.

To demonstrate this, let's take a look at the following short snippet of HTML code:

```
<h1 id='greeting'>Hello, <em>World!</em></h1>
```

This can be represented as the node-tree diagram shown below.



The `<h1>` tag contains everything, so this appears as an element node (represented by purple ovals in the diagram) at the top of the node tree. The word “Hello” is text, so this is a child text node (represented by green rectangles in the diagram). The `<em>` element is inside the `<h1>` tag, so it’s a child element node. This makes the `<h1>` element node a parent node of these child nodes. The text inside the `<em>` tags is a text child node of the `<em>` element node.

In the browser, it would look like this:

Hello, *World!*

Being able to visualize the HTML markup as a node tree will make navigating the DOM a much easier experience.

HTML Document vs the DOM

When you visit a web page, your browser first downloads the page document with all its HTML, text, images and so on. The browser then creates a representation (or mental model, if you like) of that document, which is the Document Object Model. Then the browser uses that DOM to display the web page on your device. JavaScript allows changes to be made to the web page, but those changes are made to the DOM, rather than to the original, hard-coded HTML. You'll see evidence of this later in the chapter.

Inspecting the DOM

Browsers actually let you view the Document Object Model of a web page, and you can even play around with it in all sorts of ways. (It's a bit like looking under the hood of a car.) If you right-click anywhere on a web page, you'll see an option to "inspect" the page. That will display the DOM, along with all sorts of other developer tools. The DOM will look similar to the original HTML, but not exactly the same—especially if JavaScript has modified it.

Getting an Element

The DOM provides a useful method called `getElementById` that returns a reference to the element with a particular ID attribute.

For example, we can get a reference to the `<h1>` heading element in the previous example using its `id` of `greeting`.

Let's try this out on CodePen. Open a new Pen and add the following code into the **HTML** section:

```
<h1 id='greeting'>Hello, <em>World!</em></h1>
```

We can now get a reference to the `<h1>` element by adding the following code to the **JS** section:

```
const hello = document.getElementById('greeting');
```

If no element exists with the ID provided, `null` is returned. We can see what gets returned by typing the name of the variable we assigned the element to (`hello`) into the console:

```
hello;
<< "<h1 id='greeting'>Hello, <em>World!</em></h1>"
```

As you can see, the variable `hello` now points to the HTML element with the ID of `greeting`. Now that we have a reference to that element in our code, it means we can do things with it!

Updating the HTML

The easiest way to update the HTML on a page is to use the `innerHTML` property. This will return all the HTML that's enclosed inside that element's tags as a string. We can see this by entering the following code into the console:

```
hello.innerHTML;
<< "Hello, <em>World!</em>"
```

The great thing about the `innerHTML` property is that it's also writable, so it gives us a convenient way to insert a chunk of HTML inside an element.

To demonstrate this, let's add some JavaScript code into the **JS** section on CodePen to give a more personalized greeting. First of all, let's make sure we have a reference to the heading:

```
const hello = document.getElementById('greeting');
```

Next, we'll use a prompt box to ask the user for their name and store it in the variable `name`:

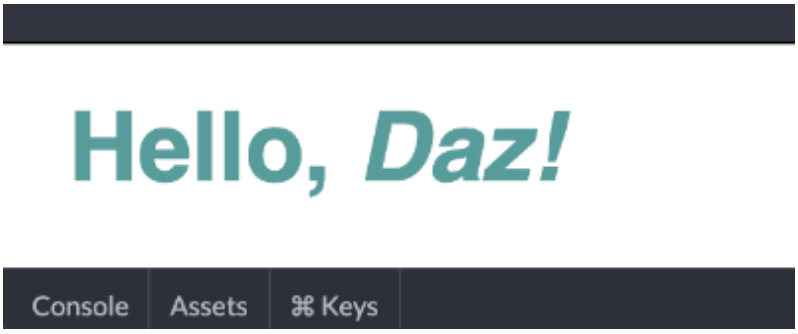
```
const name = prompt('What is your name?');
```

Last of all, we'll replace the `innerHTML` property with our own personalized greeting, which uses the `name` variable that we've just collected from the user:

```
hello.innerHTML = `Hello, <em>${name}!</em>`;
```

Notice that we've used a template literal to produce the HTML. These are incredibly useful when creating chunks of HTML to dynamically insert into a web page, as they allow variables to be inserted directly into them.

If everything went to plan, you should see something like this (but hopefully with your name!):



You can see my code on CodePen.

Getting Multiple Elements

Let's have a look at how to select more than one element at once. Create a new Pen on CodePen and add the following code into the **HTML** section:

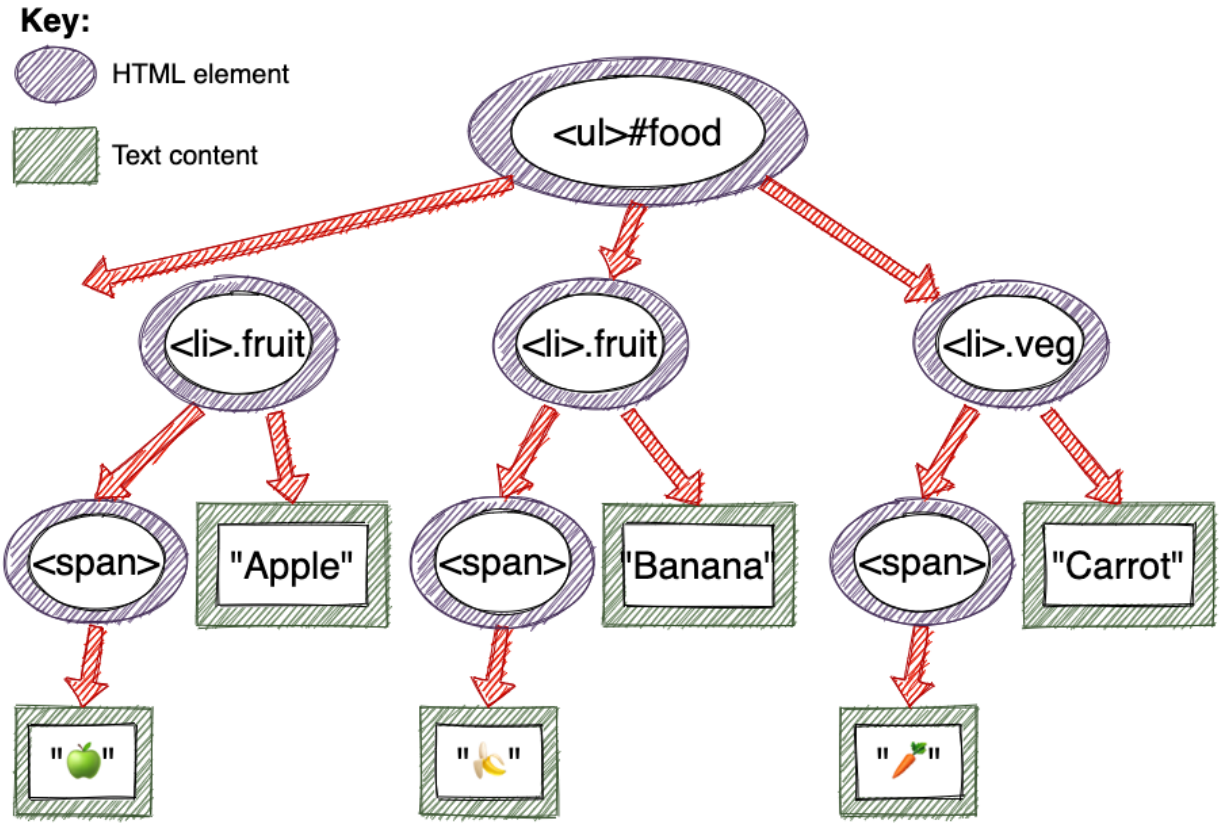
```
<ul id='food'>
  <li class='fruit'><span>🍎</span>Apple</li>
  <li class='fruit'><span>🍌</span>Banana</li>
  <li class='veg'><span>🥕</span>Carrot</li>
</ul>
```

This should create a list of items that looks something like this:

- 🍎Apple
- 🍌Banana
- 🥕Carrot

Console | Assets | Comments | ⌘

The DOM tree for this snippet of HTML is shown in the diagram below:



As we've already seen, we can gain access to the `<ul>` element using the following code:

```
const food = document.getElementById('food');
```

But how do we access a group of elements, such as the list items? Luckily, the DOM also provides a few methods that allow us to access groups of elements.

Getting Elements by Tag Name

We can use `getElementsByTagName()` to return a collection of all the elements with the tag name provided as an argument. For example, we can get all the list items (HTML tag of `<li>`) in the document using this code:

```
const items = document.getElementsByTagName('li');
```

The variable `items` now contains a collection of all the list-item elements. You can access each item in the collection using the square bracket index notation that we've previously seen used with strings and arrays. For example, entering the following code into the console will return the first item in the collection:

```
items[0];
<< "<li class='fruit'><span>🍏</span>Apple</li>"
```

We can also find out how many elements are in the collection using the `length` property:

```
items.length;
<< 3
```

Getting Elements by Their Class Name

We can also use the `getElementsByClassName()` method to return a collection of all elements that have a particular class name. For example, the following code will return a collection of all elements with the class of `fruit`:

```
const fruit = document.getElementsByClassName('fruit');
```

The variable `fruit` contains a collection of the two elements that have a class of `fruit`, as we can see by entering the following into the console:

```
fruit.length;
<< 2

fruit[0];
<< "<li class ='fruit'><span></span>Apples</li>"

fruit[1];
<< "<li class='fruit'><span></span>Banana</li>"
```

Note that, if there are no elements with the given class, a collection will still be returned, but it will contain no items and have a length of 0.

Get Element vs Get Elements

Did you notice that you can only get *one* element by ID (`document.getElementById`) but multiple elements by *class name* (`document.getElementsByClassName`)? An easy way to remember this is that you're only allowed to use a particular ID *once* per HTML document, while you can use a particular class name multiple times in the same document.

Query Selectors

Another way to get elements in the DOM it to use **query selectors**. The nice thing about these is that they allow you to use CSS notation to target specific elements.

The `document.querySelector()` method allows you to use CSS notation to find the *first* element in the document that matches the CSS selector provided.

For example, instead of using `document.getElementById` to get a reference to the element with an ID of `food`, we could use the following code:

```
const food = document.querySelector('#food');
```

The `document.querySelectorAll()` method also uses CSS notation, but returns a list of *all* the elements in the document that match the CSS query selector.

For example, the following two statements are identical and return the same node list:

```
document.getElementsByClassName('fruit')`;
document.querySelectorAll('.fruit');
```

Query selectors are powerful methods that can emulate all the previous methods, as well as allowing more fine-grained control over which element nodes are returned, as they allow you to specify precise items on a page. For example, CSS pseudo-selectors can be used to pinpoint a particular element.

The following code, for example, will return only the last list item in the `food` list:

```
const carrot = document.querySelector('ul#food li:last-child');
<< "<li class='veg'><span>🥕</span>Carrot</li>"
```

CSS Selectors

You do have to know a bit about CSS selectors to be able to use this method! If you're not quite up to speed with them, or just need a refresher, you might want to check out SitePoint's "CSS Selectors" guide, or have a read of *CSS Master*, by Tiffany Brown.

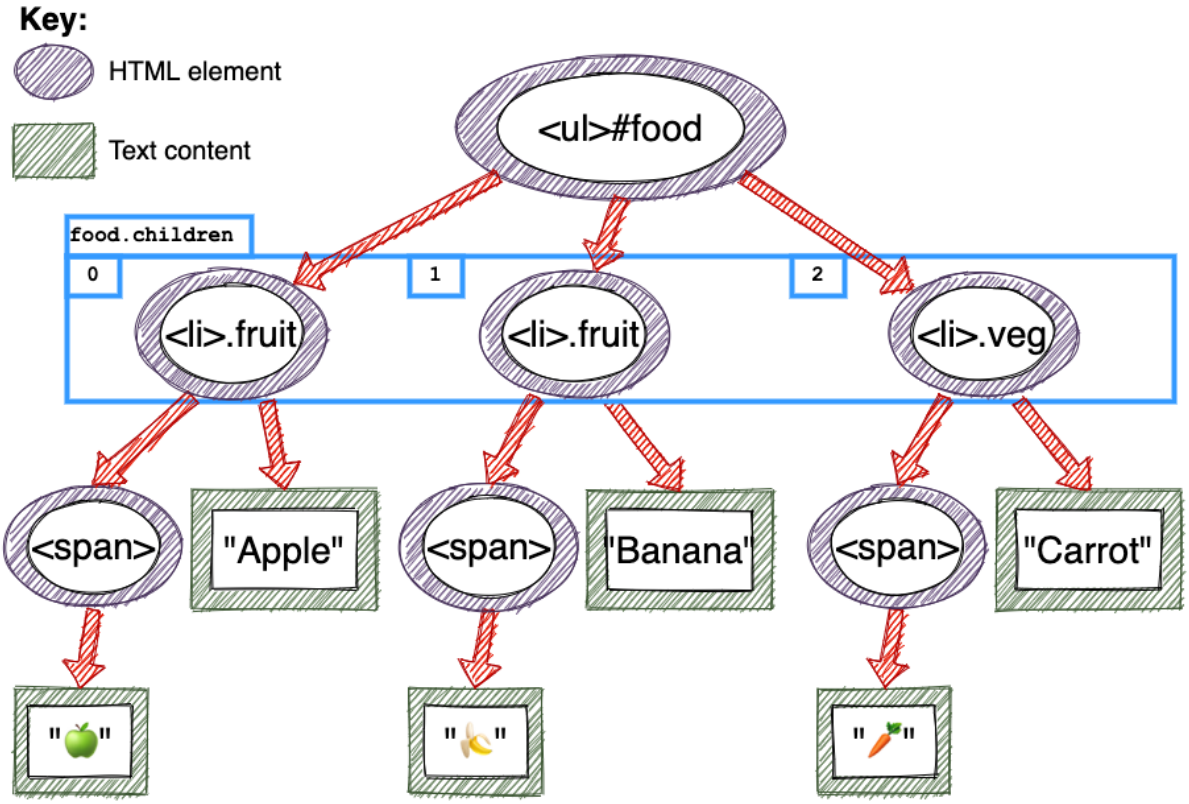
Navigating the DOM Tree

DOM nodes have a number of properties and methods for navigating around the document tree. Once you have a reference to an element, you can walk along the document tree to find other nodes.

Let's have a look at a couple of the more useful ones.

Child Nodes

You can get a collection of all the child elements of an element using the `children` property. In our food list example, `food.children` will return a node list of all the child elements of the `<ul>` element that has an ID of `food`. These are highlighted in the diagram below.



The code below shows how we can get access to a node list containing the three child `<li>` elements:

```
food.children.length;
<< 3

food.children[0];
<< "<li class='fruit'><span>🍏</span>Apple</li>"
```

Before we go on, let's create some references to these child nodes in our program. Add the following code to the **JS** section of your Pen:

```
const apple = food.children[0];
const banana = food.children[1];
const carrot = food.children[2];
```

(Here's what your Pen should look like now, if you're following along.)

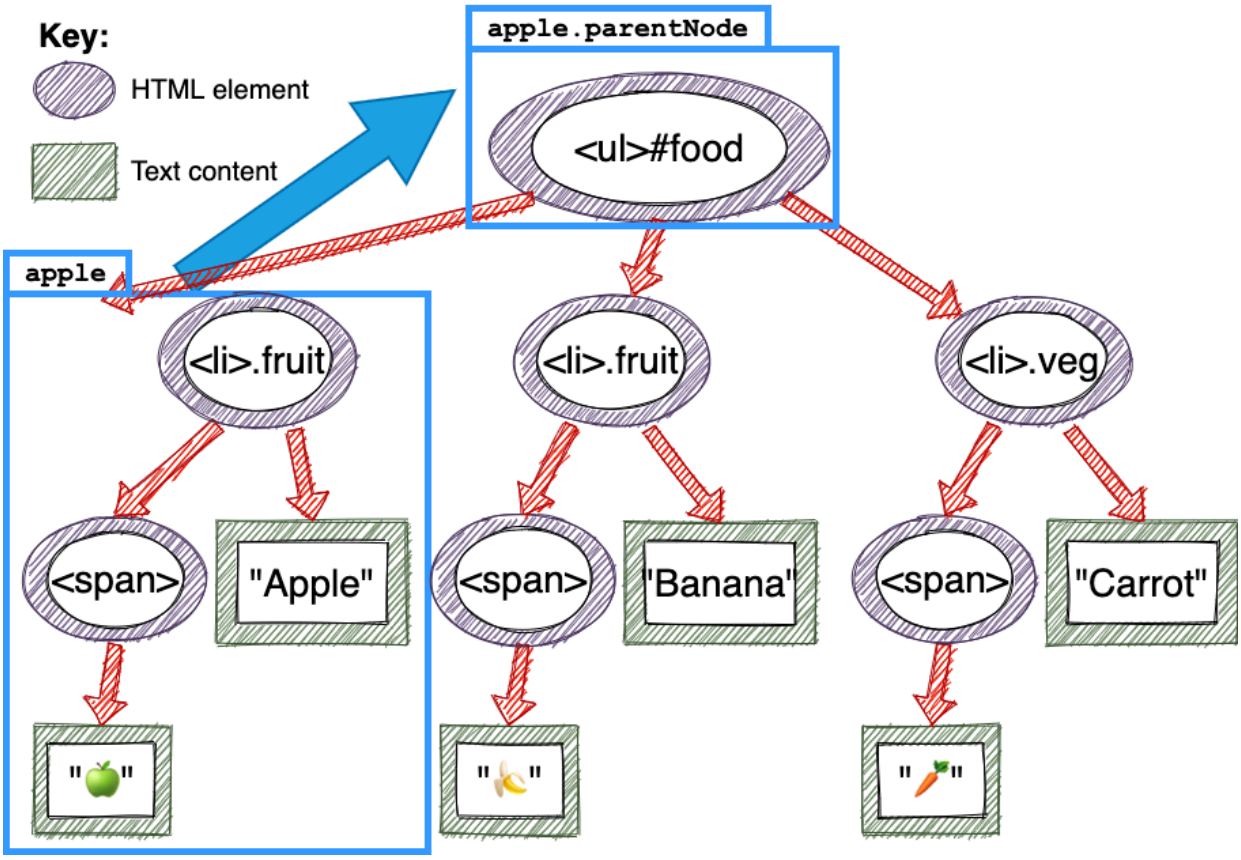
Parent Node

The `parentNode` property returns the parent node of an element.

The following code returns the `food` node because it's the parent of the `apple` node:

```
apple.parentNode;
<< "<ul id='food'>
<li class='fruit'><span>🍎</span>Apple</li>
<li class='fruit'><span>🍌</span>Banana</li>
<li class='veg'><span>🥕</span>Carrot</li>
</ul>"
```

This relationship can be seen in the diagram below:



Creating Dynamic Markup

So far, we've looked at how to gain access to different elements of a web page and find out information about them. But the real power of the DOM is

its ability to dynamically update the markup, like we did earlier in the chapter with the `innerHTML` property.

In this section, we're going to learn how to create new elements and add them to the page, update elements that already exist, and remove any unwanted elements from the page.

Creating an Element

The document object has a `createElement()` method that takes a tag name as a parameter and returns that element. For example, we could create a new item for our food list as a DOM fragment in memory by writing the following in the console:

```
const melon = document.createElement('li');
```

At the moment, this element is empty. To add some content, we'll use the `innerHTML` property to add the HTML content:

```
melon.innerHTML = `<span>🍈</span>Melon`;
```

Add the two lines above to your Pen.

Adding Text with `textContent`

There's also a `textContent` property that can be used to add text to an element, but you can't use any HTML elements in it. For example, this would be fine:

```
melon.textContent = '🍈Melon';
```

But if you tried to add the `<span>` tags around the emoji, it wouldn't parse the HTML:

```
// this code won't parse the <span> tags ...
melon.textContent = '<span>🍈</span>Melon';
```

As you can see in the image below, the tags just get displayed as text on the web page, which usually isn't what we want.

Adding Elements to the Page

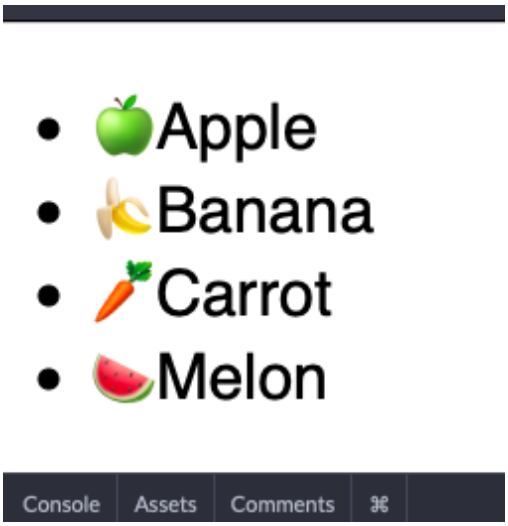
If you've updated your Pen, you'll see that nothing has changed yet. Now that we've learned how to create HTML elements, we need to learn how to put them on the page. There are a number of methods to allow you to insert nodes into the DOM.

Let's firstly look at `appendChild()`. Every DOM node has an `appendChild()` method that will add another node (given as an argument) as a child node. The following example will add the `melon` element we created above to the end of the `food` list:

```
food.appendChild(melon);
```

The following diagram shows where the new element we created will be inserted into the list.

And the next image shows that the melon is now part of the list.

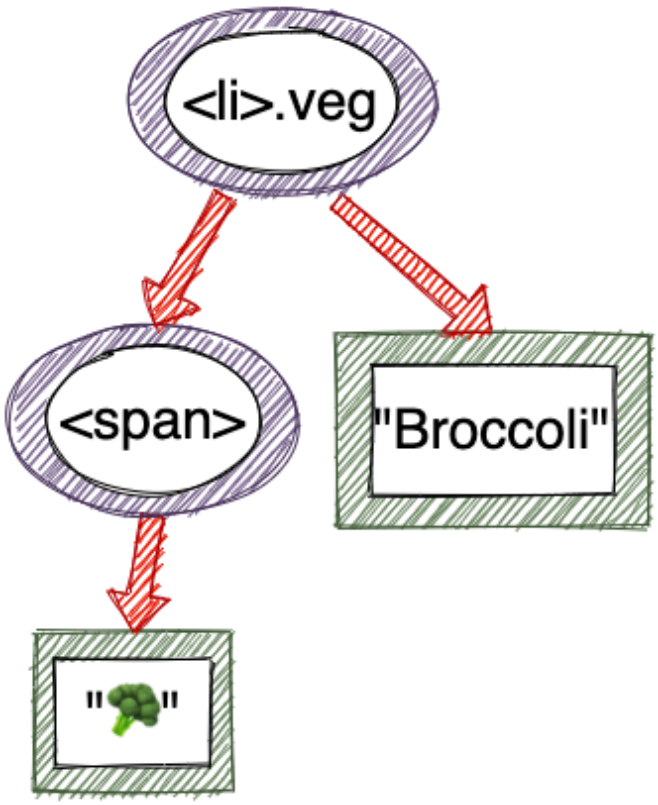


(Here's the live Pen of what's pictured above.)

Building Elements Node by Node

An alternative to using `innerHTML` to populate the content of an element is to build each node individually and then use the `appendChild()` method to put them all together.

To demonstrate this, let's create a new list element that matches the node tree shown below.



We'll need to create a node for each node in the diagram, then put them all together. We've already seen the `document.createElement()` method that's used to create element nodes (the purple ovals in the diagram). There's also a `document.createTextNode()` method that we can use to create the text nodes (the green rectangles in the diagram).

Let's create the nodes now. Add the following to the **JS** section of your Pen:

```
const broccoli = document.createElement('li');
const text = document.createTextNode('Broccoli');
const span = document.createElement('span');
const emoji = document.createTextNode('🥦');
```

This will create all the nodes and assign variables to them. Now we can put them together to form the list item:

```
span.appendChild(emoji);
broccoli.appendChild(span)
broccoli.appendChild(text);
```

Next, we need to insert this new list item into the HTML.

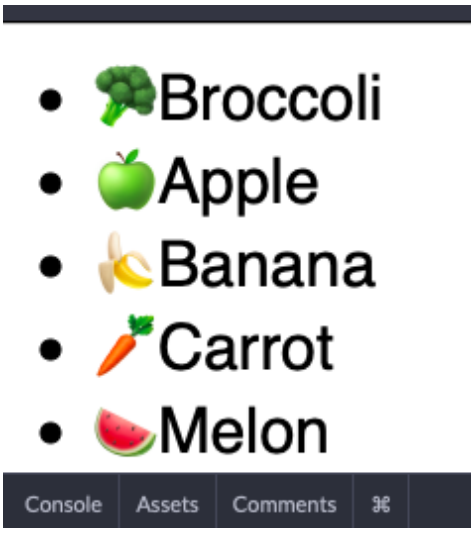
Insert Before

The `appendChild()` method is useful, as you'll often want to add a new element to the bottom of a list. But what if you want to place a new element in between two existing elements?

The `insertBefore()` method will place a new element before another element in the markup. It's important to note that this method is called on the *parent node*. It takes two parameters: the first is the new node to be added, and the second is the node that you want it to go before. (It's helpful to remember that the order of the parameters is the order they'll appear in the markup.) For example, we can place our new `broccoli` element before the `apple` element with the following line of code:

```
food.insertBefore(broccoli,apple);
```

As you can see, this will place the broccoli element at the top of the list, before the apple.



And here's our updated Pen.

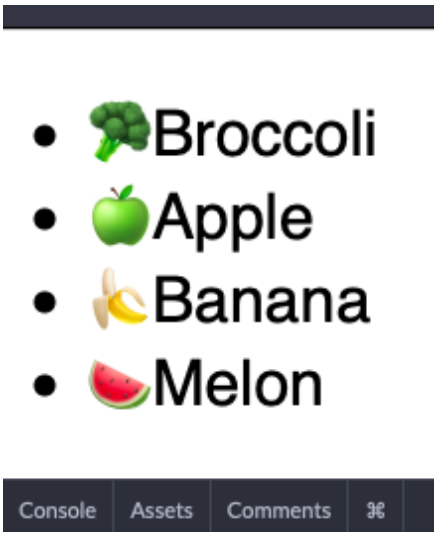
Somewhat annoyingly, there's no `insertAfter()` method, so you need to ensure that you have access to the correct elements to place an element exactly where you want it.

Removing Elements from a Page

An element can be removed from a page using the `remove()` method. This method is called on the node to be removed. It returns a reference to the removed node. For example, if we wanted to remove the `carrot` element, we would use the following code:

```
carrot.remove();
<< "<li class='veg'><span>🥕</span>Carrot</li>"
```

As you can see in the diagram below, the carrot is no longer there:



Gone but Not Gone

Let's remind ourselves here about the difference between the original HTML and the DOM. This is what your updated Pen should look like now. The carrot item has been removed from the rendered view (that is, what you see in the browser). But notice that the original carrot code is still there in the HTML. JavaScript has removed the carrot list item from the Document Object Model, but not from the actual HTML.

Replacing Elements on a Page

The `replaceChild()` method can be used to replace one node with another. It's called on the parent node and has two parameters: the new node, and the node that's to be replaced. For example, if we wanted to change the content of the third list item (you can eat too many bananas), we could replace the text node with a new one, like so:

```
const lemon = document.createElement('li');
lemon.innerHTML = `<span>🍋</span>Lemon`;
food.replaceChild(lemon,banana);
```

As you can see in the image below, the lemon has replaced the banana in our list.

Here's our updated Pen.

Getting and Setting Attributes

All HTML elements have a large number of possible attributes, such as `class`, `id`, `src`, and `href`. The DOM has a number of methods that can be used to get or set these attributes.

Getting an Element's Attributes

The `getAttribute()` method returns the value of the attribute provided as an argument. For example, we can find out the class of the `apple` element by entering the following code into the console:

```
apple.getAttribute('class');
<< "fruit"
```

If an element doesn't have the given attribute, it returns `null`. For example, if we enter the following code into the console, we can see that the `broccoli` element doesn't have a `src` attribute (we didn't add one when we created the element earlier):

```
broccoli.getAttribute('src');
<< null
```

Setting an Element's Attributes

The `setAttribute()` method can change the value of an element's attributes. It takes two arguments: the attribute that you wish to change, and the new value of that attribute.

For example, if we want to add the class of `veg` to the `broccoli` element, we can use the following code in the console:

```
broccoli.setAttribute('class', 'veg');
```

Multiple Classes and `setAttribute`

Using the `setAttribute()` method will overwrite the current value. When used to update the `class` attribute, this will overwrite *all* the classes that an element has. It's usually much better to use the `classList` property to update the class attribute, which we'll cover later.

The `className` Property

As we've seen, we can modify the class name of an element using the `setAttribute()` method. There's also a `className` property that allows the class of an element to be set directly. In addition, it can be used to find out the value of the class attribute:

```
apple.className;
<< "fruit"
```

We can also use it to set the class attribute of an element. For example, the `melon` element that we created earlier also doesn't have a class attribute, so let's fix that now:

```
melon.className = 'fruit';
<< "fruit"
```

Multiple Classes and `className`

As with `setAttribute` above, changing the `className` property of an element by assignment will also overwrite all other classes that have already

been set on the element. This problem can be avoided by using the `classList` property instead, which we'll cover next.

The `classList` Property

The `classList` property is a list of all the classes an element has. It also has a number of methods that can be used to modify the class of an element.

The `add()` method can be used to add a class to an element without overwriting any classes that already exist. For example, we could add a class of `fruit` to the `lemon` element that we created earlier with the following code:

```
lemon.classList.add('fruit');
```

The `remove()` method will remove a specific class from an element. For example, we could remove the class of `fruit` that we just added with the following code:

```
lemon.classList.remove('fruit');
```

The `contains()` method will check to see if an element has a particular class:

```
apple.classList.contains('fruit');
<< true

apple.classList.contains('veg');
<< false
```

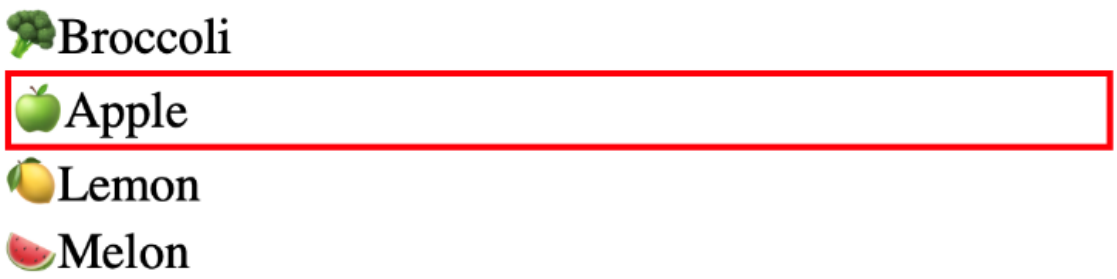
Doing It with Style

Every element node has a `style` property. This can be used to dynamically modify the presentation of any element on a web page.

To see an example of this, try adding the following code to the **JS** section of your Pen:

```
apple.style.border = "red 2px solid";
<< "red 2px solid"
```

Here's the updated Pen. A red border has been added around the `apple` list element.



Hyphens in CSS Property Names

Any CSS property names that are separated by hyphens must be written in camelCase notation when referenced in JavaScript. So, the hyphen is removed and the next letter is capitalized. This is because JavaScript doesn't allow hyphens in property names.

For example, the CSS property `background-color` becomes `backgroundColor`, and `font-size` becomes `fontSize`.

Being Classy

While it can be useful to edit the styles of elements on the fly, it can get messy if you want to change a large number of styles all at once.

Usually, a better alternative is to dynamically change the class of an element and have different styles for each class in the CSS.

For example, if we wanted to add a red border around the `apple` element (to highlight it for some reason), we could either do it in the way we saw earlier, or we could add a class of `highlighted` to the `apple` element:

```
apple.classList.add('highlighted');
```

Then add the following to the **CSS** section:

```
.highlighted {
  border: red 2px solid;
}
```

Here's our Pen updated with that code.

Another advantage of using this method is that it gives us more flexibility if we decide later on to change how we highlight elements. We might want to use a blue background and bold text instead of a red border, for example. All we'd need to do is change the code for the `highlighted` class in the CSS, rather than having to dig around in the JavaScript code to change the style property of every single element that needed highlighting. Using classes to update the style of elements is a much more DRY approach.

A Simple To-do List

We're going to finish this chapter by writing some code that will create a to-do list and allow us to add items to it and cross them off when they're done. All of this will be done using dynamic HTML, created with JavaScript. Open up a new Pen on CodePen to get started.

Let's start off by creating the list element, adding this code to the **JS** section:

```
const list = document.createElement('ul');
document.body.appendChild(list);
```

This creates an unordered list (`<ul>`) and appends it to the `<body>` tag of the document. You won't see anything rendered yet, because the list is empty—but it is there! (If you inspect the DOM, you'll see the empty `<ul></ul>` code there now.)

Next, we'll write a function for adding tasks. Add the following code to the **JS** section:

```
function add(item){
  const li = document.createElement('li');
  li.innerHTML = item;
  list.appendChild(li);
}
```

This function will create a list item (`<li>`) tag and then append it as a child to the `<ul>` element we just created. The content of the list item will be whatever is provided to the function as an argument, so `add('Read a book')` will append the following HTML to the list element:

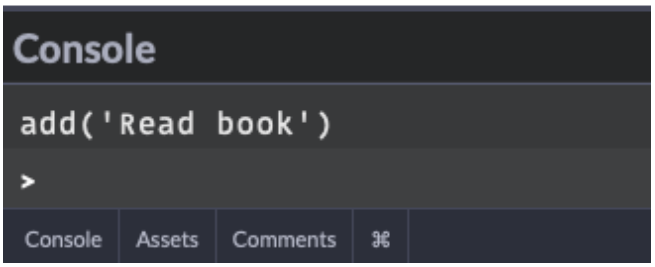
```
<li>Read a book</li>
```

Open up the console in CodePen and add a few tasks to the list:

```
add('Bake cake');
add('Sing song');
add('Read book');
```

As you press `Enter`, you should see each item appear on the screen.

- Bake cake
- Sing song
- Read book



Now let's write a function that will toggle a class of `complete` to an item. If the item doesn't have a class of `complete`, it will be added, and if it already has the class, it will be removed:

```
function toggle(item){
  item.classList.toggle('complete');
}
```

We'll also need to add some code to the **CSS** section to style the items that have a class of `complete` so that they appear to have been crossed out:

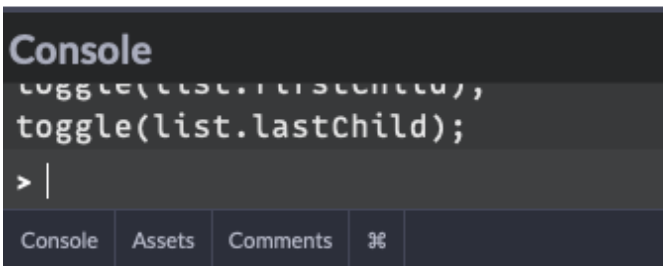
```
.complete{
  text-decoration: line-through;
}
```

Open up the console again and let's see if this works. The function accepts an argument of the actual element we want to cross out, so we can try crossing out the first and last items in the list with the following code:

```
toggle(list.firstChild);
toggle(list.lastChild);
```

Each time you press `Enter`, you should see the relevant item get crossed out on the page.

- ~~Bake cake~~
- Sing song
- ~~Read book~~



You can see my code on CodePen.

Obviously it's frustrating having to interact with the list through the console, but at least we've shown that it's possible to add and edit content dynamically using JavaScript. In the next chapter, we'll look at making this more interactive without the need for the console.

Challenges

1. Create some dynamic HTML in a new Pen, including at least a heading and paragraph of text, using just JavaScript. There should be nothing at all in the **HTML** section on CodePen. You can see my code on CodePen.

2. Add a function called `destroy` to the to-do list that we just wrote. It should remove a task from the list that's provided as an argument. So, for example, `destroy(list.lastChild)` should remove the last item in the list. You can see my code on CodePen.

Summary

- The Document Object Model is a way of representing a page of HTML as a tree of nodes.
- The DOM has a number of methods that can be used to access elements on a page.
- The DOM provides a number of methods that can be used to navigate around the DOM tree.
- An element's attributes can be accessed using the `getAttribute()` method and updated using the `setAttribute()` method.
- The `innerHTML` property can be used to quickly insert raw HTML into an element.
- The `createElement()` and `createTextNode()` methods can be used to create dynamic markup.
- Markup can be added to the page using the `appendChild()` and `insertBefore()` methods.
- Elements can be replaced using the `replaceChild()` method and removed using the `remove()` method.
- The CSS properties of an element can be changed by updating the `style` property.

Now that we've learned how to navigate and dynamically update the markup of a web page, it's time to start interacting with it. In the next chapter, we'll be learning how to handle events.

Chapter 11: The Main Event

In the last chapter, we were introduced to the DOM and how it allows us to interact with the elements of a web page. In this chapter, we'll be looking at how events provide the link between user interactions and our program. That's right—this chapter is where your code starts to become fully interactive and the fun really begins!

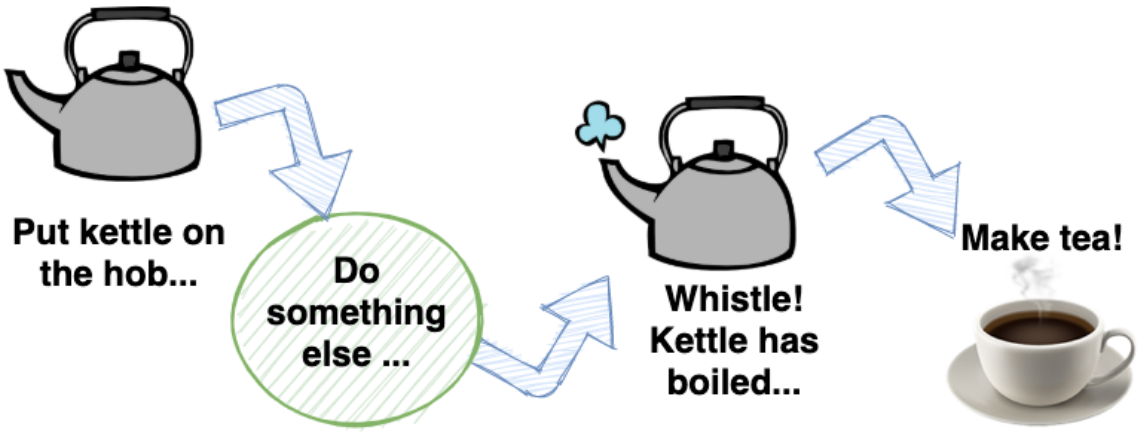
In this chapter, we'll cover the following topics:

- event-based programming
- event listeners
- click events
- the event object
- submitting forms
- mouse events
- keyboard events
- removing event listeners
- event delegation

Event-based Programming

Imagine boiling some water in a pan. The only way to tell if the water has boiled is to keep checking at regular intervals. This is annoying, since you have to keep stopping what you're doing to check, and there's also the chance that you might miss the point when the water boils and it will boil over, making a mess.

The better approach is to use a kettle that whistles when the water boils. This means you can go off and do something else, safe in the knowledge that you'll be alerted as soon as the water has boiled.



Event-based programming is a style of coding that reacts to events in a similar way to how you react to the whistle on the kettle. The events in a program are usually user actions, such as pressing a key, moving the mouse or tapping the screen, but they can also be other things, such as a timer, a notification, or a change in the state of the application.

Event Listeners

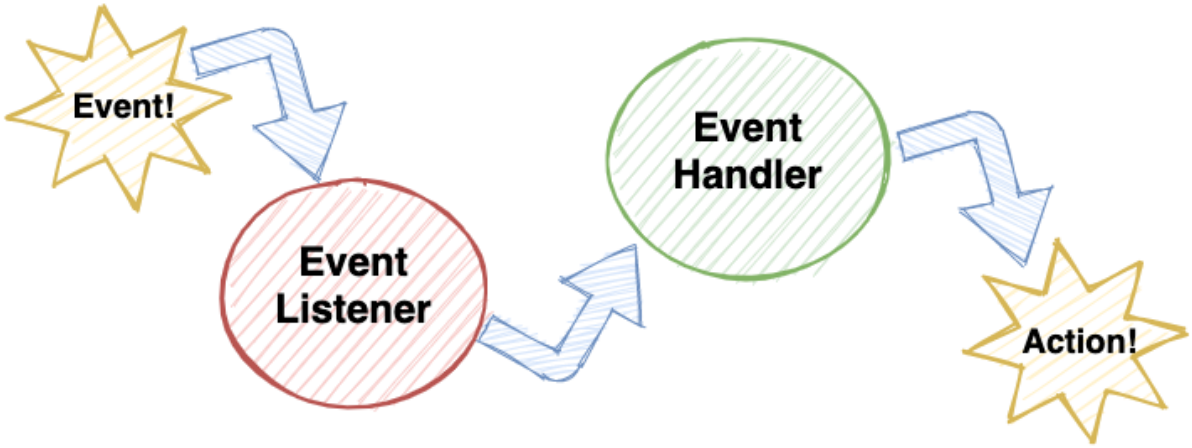
An **event listener** is triggered when an event happens. Event listeners work in a similar way to the whistle on the kettle. Instead of the program having to constantly check to see if an event has occurred, the event listener sits in the background until the event happens and then lets the program know so it can respond immediately. This allows the program to continue with other tasks while waiting for the event to happen.

Event Handlers

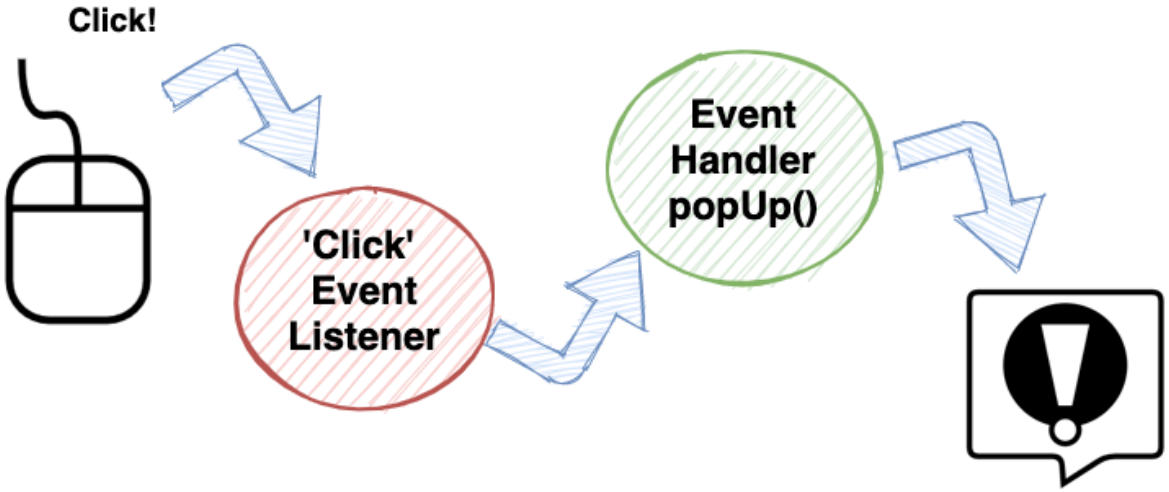
An **event handler** is a set of instructions on what to do when an event happens. In the kettle example above, this would be the instructions on what to do when the kettle has boiled—namely, to take it off the hob and make some tea!

The diagram below shows this process, starting with an event that triggers the event listener, which then informs the event handler and results in an

action.



In programming, an event handler usually takes the form of a callback function that's called when the event listener is triggered. For example, you might want a pop-up notification to appear when a user clicks on a button. This would be achieved by setting an event listener that's triggered when the user clicks the button. The event handler would then call function that displays the notification. This function would be called every time the event listener is triggered, which means that the message will appear every time the button is pressed. This sequence is shown in the diagram below.



JavaScript is an event-based language, and we'll be looking at at some examples of common browser events in the rest of the chapter.

Click Events

In JavaScript, we can attach an event listener to an element of the page by calling the `addEventListener()` method on the element. To demonstrate this, we'll attach an event listener to the document object that represents the whole page. This will fire and call the event handler function when the user clicks on the page. The `click` event occurs when a user clicks with the mouse, presses the `Enter` key or spacebar, or taps the screen, making it an all-round event covering many types of interaction. Open up a new Pen on CodePen and enter the following code in the **JS** section:

```
document.addEventListener('click',bang);
```

This event listener accepts two arguments: the first is the event to listen for (`click` in this example), and the second is the event handler (which is a callback function named `bang` in this example). So, this event listener will listen out for a `click` event and call the `bang` function whenever a click occurs.

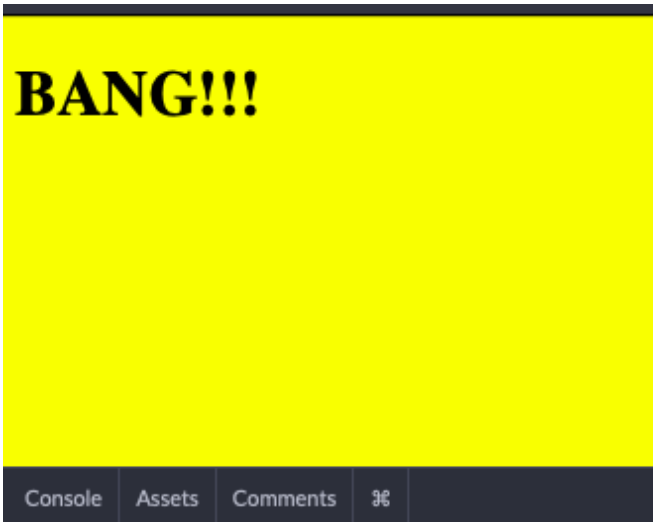
Note that parentheses are not placed after the `bang` function when it's used as the argument to an event listener. Otherwise, the function would actually be called when the event listener is set, instead of when the event happens!

Before we can test this out, we need to define the `bang` function. Add the following code beneath the event listener:

```
function bang(){
  document.body.style.background = 'yellow';
  document.body.innerHTML = `<h1>BANG!!!</h1>`;
}
```

This function uses some of the DOM methods we saw in the last chapter. First of all, it changes the background of the body to yellow using the `style` property. Then it updates the inner HTML with a heading element that says "BANG!!!"

But this function won't get called until a click event is fired. Give it a go: as soon as you click anywhere on the page, you'll see the bang message.



You can see my code on CodePen.

Clicking Elements

The event listener in the last example was a **global event listener** that was listening out for clicks on the whole page. We can also attach event listeners to specific elements on the page. These will only fire when the event occurs to those elements.

To see an example of this, we'll create a button that changes the page's background color when it's clicked. Open up a new Pen on CodePen and add the following code in the **HTML** section:

```
<button id='red'>Red</button>
```

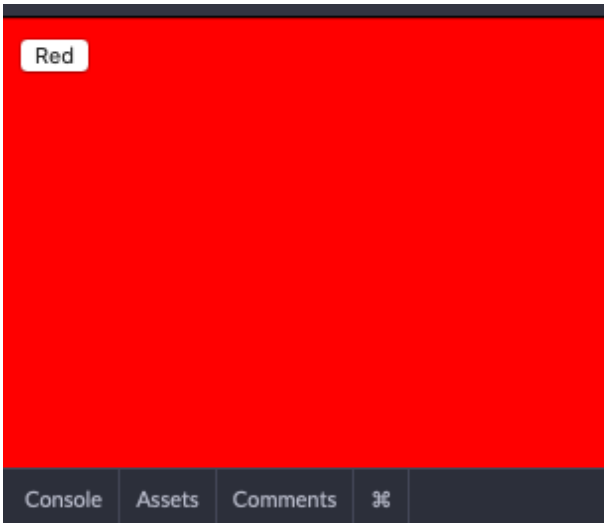
This adds a button to the page. The following code will give us a reference to that button in the **JS** section:

```
const redButton = document.getElementById('red');
```

Now that we have a reference to the button, we can attach an event listener to it:

```
redButton.addEventListener('click', e =>
document.body.style.background = 'red');
```

This is listening for a `click` event, just like in the last example, but the second parameter, instead of being the name of a function, is an anonymous function that's defined directly inside the event listener. This is useful for short functions, and arrow functions are a good choice to use for them, since it's just a one-liner that's used to change the style of the body background to red.



You can see my code on CodePen.

The eagle-eyed among you might be thinking, "But what's that letter `e` doing there?" If you spotted that, well done. This is a parameter to the function and is a reference to the event object, which we should probably look at in more detail.

The Event Object

Whenever an event happens, the function that's called is automatically passed an **event object** as a parameter. The event object contains a large amount of information about the event, stored as properties. For example, the `target` property points to the element that triggered the event.

Let's add another button to our example so we can take a closer look at the event object. Add the following line of code to the **HTML** section:

```
<button id='green'>Green</button>
```

This gives us another button to play around with. Now in the **JS** section, add the following code to give us a reference to that button:

```
const greenButton = document.getElementById('green');
```

Let's add an event handler for this button in the **JS** section:

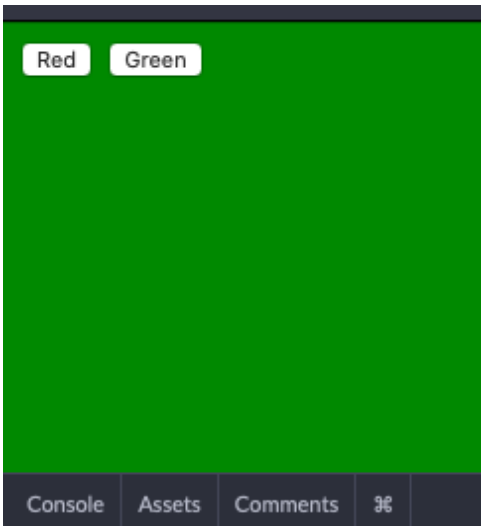
```
greenButton.addEventListener('click', e =>
document.body.style.background = e.target.textContent);
```

To `e` or Not to `e`?

The function parameter doesn't have to be `e`. It just makes sense to use `e` since it's short for "event".

This is very similar to the event listener we attached to the red button, except instead of hard-coding the color that we want to set the background to, we've set it to be `e.target.textContent`. The `target` property points to the element that triggered the event when it was clicked, so the `greenButton` element in this case. The `textContent` property refers to the text inside this element, which is "Green".

If you try clicking on the button, it will change the CSS `background-color` property of the page to whatever text is inside the button element. Try changing the text to another color and you'll see that the background color updates to this color. This is another example of DRY code, since we only need to specify the color in one place.



You can see my code on CodePen.

The event object has a lot more properties and methods that are summarized on the Mozilla Developer Network site.

Forms

Forms are a very common method of interacting with a web page and can be found on many sites.

Traditionally, when a form is submitted, it's sent to a server where the information is processed using a "back-end" language such as Python or Ruby. However, it's possible to stop the form being sent to the server and to instead use JavaScript on the "front end" to process the information.

Submitting a Form

Let's start with a simple form that contains one input field and a button for submitting the form. Open up a new Pen on CodePen and add the following code in the **HTML** section:

```
<p>Enter your name in the box below:</p>
<form name='myForm'>
  <input type='text' name='myName'>
  <button type='submit'>Submit</button>
```

```
</form>
<div id='hello'></div>
```

This should produce an input box with a **Submit** button next to it. Note that the button has a type attribute of `submit`: this will trigger a `submit` event when it's pressed. We've also added an empty `<div>` element at the bottom of the code with an ID of `hello`, which can't be seen on the screen because it's currently empty. We'll be using this as a container to hold the output after the form has been submitted.

First of all, we need some references to these elements. Add the following code to the **JS** section:

```
const form = document.forms.myForm;
```

The DOM provides a convenient way of accessing any forms on a web page using `document.forms`, which will return a collection of all the forms on the page. If you add the name of the form, it will return the form with that name (note that our form has a name attribute of `myForm` in the HTML code).

Next, add the following JavaScript code:

```
const hello = document.getElementById('hello');
```

This gives us a reference to that empty `<div>` element with the ID of `hello` in the HTML.

Next, we'll add an event listener to the form:

```
form.addEventListener('submit',sayHello);
```

This will call the `sayHello` function when the form is submitted. So let's write that function next:

```
function sayHello(e){
 hello.textContent = `Hello, ${form.myName.value}!`;
}
```

This function updates the `textContent` property of the `hello` div with a template literal. Notice that, inside the template literal, there's a reference to `form.myName.value`. This returns a string that matches what's inside the `<input>` field, because the `<input>` tag has a name attribute of `myName`. So, if you write "JavaScript" inside the input box, the `hello` div will contain the following message: "Hello, JavaScript!"

Referencing a Form Element

Any elements inside a form can be referenced as a property of the form element using their `name` attribute. In the example above, `form.myName` is a reference to the `<input>` element inside the form, because it has a name attribute of `myName`.

Try entering your name in the input field and either pressing `Enter` or clicking the **Submit** button to submit the form.

Unfortunately, you'll probably get an error message something like this:

This is because forms have a default behavior that means they get submitted to a server. In this case, it has tried to submit to `/boomboom/v2/index.html`, which is a page on CodePen's server that we don't have access to. Hence, the error message. Don't worry, though, as this is easy to fix.

A Form's `action` Attribute

Forms have an `action` attribute that can be used to specify the page you want to submit them to on the server. If this isn't specified, the form will try to submit itself to the page it's on by default. This is what happened in this case (since we didn't specify an `action` attribute), but CodePen won't let us submit forms to their server!

Preventing Default Behavior

Some elements on a web page have default behaviors that are built into the browser. As we've just seen, forms get sent to the server by default when they're submitted.

Luckily, the event object has a method called `preventDefault` that will stop any default behavior from happening when an event occurs. We can use it in our `sayHello` function to stop the form from being submitted to the server. Update the function so that it looks like the following:

```
function sayHello(e){
  e.preventDefault();
  hello.textContent = `Hello ${form.myName.value}!`;
}
```

Now try submitting the form again and it should look something like this:

You can see my code on CodePen.

Keyboard Events

There are two main events that relate to a user pressing a key. These are `keydown`, when a key is pressed, and `keyup`, when it's released.

The event object for these events contains two properties that can tell us which key was pressed:

- The `code` property returns a string that relates to the actual physical key that was pressed.
- The `key` property returns the actual output that will appear on the page. This can be influenced by special keys being held, such as `shift` or `alt`, the keyboard layout settings, and locale settings (such as currency keys).

For example, if you press the spacebar, the `code` property will be `Space`, but the `key` property will be (an empty space, which is what it produces).

You can see the codes that each key produces by adding the following code in the **JS** section of CodePen:

```
document.addEventListener('keydown',e => document.body.innerHTML
= `<div>e.code: ${e.code}</div><div>
e.key: ${e.key}</div>`);
```

This will fire every time a key is pressed anywhere in the document and update the `innerHTML` property of the page body to show the value of `e.code` and `e.key`. Try pressing some keys, such as the arrow keys, the `alt` key, or a letter key while holding down `shift`, and notice what gets returned.

Here's my code on CodePen.

Deciding whether to use the `key` or `code` property depends on whether you're interested in knowing the actual key that was pressed, or the output of pressing the key, so you need to think about this when writing your code.

Live Input

To see an example of how we can use keyboard events, let's build a live input bar that updates the output in real time. Open up a new Pen on CodePen and add the following to the **HTML** section:

```
<input id='input'>
<div id='output'></div>
```

This creates an input element to type into and an empty `<div>` to display the output. Both of these have `id` attributes so that we can access them from within our code. Let's add this now to the **JS** section:

```
const input = document.getElementById('input');
const output = document.getElementById('output');
```

Finally, all we need to do is attach an event listener to the input element:

```
input.addEventListener('keyup', e => output.textContent =
input.value);
```

This contains an anonymous function that will be called when the `keyup` event is fired. The reason for using `keyup` instead of `keydown` is because the character on the key appears inside the input field between these two events, so when the `keydown` event fires, the character isn't there, but it will be by the time the `keyup` event fires. This is important, because the function inserts whatever's in the input field into the `output` div by overwriting the `textContent` property.

Try typing something into the input field. You should see the text appear underneath every time you press a key.

You can see [my code on CodePen](#).

Mouse Events

There's a number of events that relate to the mouse pointer and its interactions with elements on the page.

Mouse Move

Every time the mouse pointer moves, the `mousemove` event fires. To see an example of this, we'll write some code that tells us the coordinates of the mouse pointer whenever it moves.

Open up a new Pen on CodePen and start by adding the following event listener in the **JS** section:

```
document.addEventListener('mousemove',showCoords);
```

This will call the `showCoords` function whenever the mouse pointer moves anywhere on the page. Let's write that function now:

```
function showCoords(event){
  document.body.textContent = `(${event.x},${event.y})`;
}
```

This function updates the `textContent` property of the document body to contain the coordinates of the mouse pointer. The `x` and `y` properties of the event object tell us the horizontal and vertical position of the mouse pointer respectively. We can use a template literal to insert these values into the text content of the document's body every time the mouse moves. Try moving your mouse pointer around on the page and you should see its exact coordinates displayed in the top left corner.

You can see my code on CodePen.

Mouse Over

The `mouseover` event is fired when the mouse pointer moves over an element. To see this in action, let's create a little game called Find the Bomb. Open up a new Pen on CodePen and enter the following code in the **JS** section:

```
const bomb = document.createElement('div');
  bomb.textContent = '💣';
  bomb.style.position = 'absolute';
  bomb.style.top = Math.floor(200*Math.random()) + 'px';
  bomb.style.left = Math.floor(200*Math.random())+'px';
  bomb.style.fontSize = '64px';
  document.body.appendChild(bomb);
```

This uses a lot of what we learned in the last chapter to create a `<div>` element that contains a bomb emoji as its text content. It then sets the

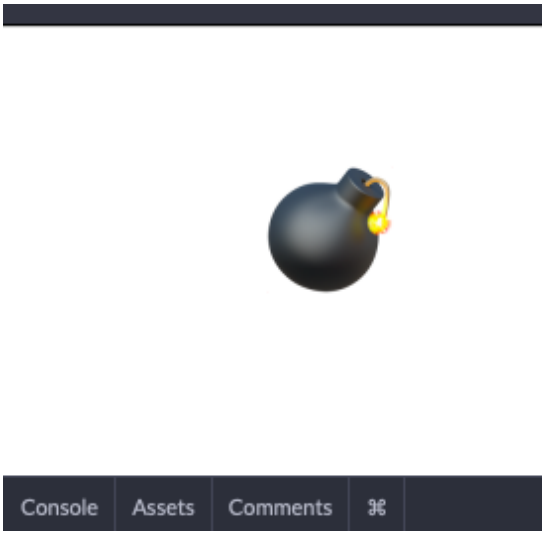
position property to `absolute`, which allows us to set the position of the bomb relative to the left side and top of the page using the `left` and `top` properties respectively. We then set these properties to a random number, using what we learned in Chapter 4. Last of all, we use the `appendChild()` method to place the bomb on the page. It should appear at a random position every time the code is run.

Next, we need to add the event listener:

```
bomb.addEventListener('mouseover', e => {
 document.body.style.background = 'red';
 document.body.innerHTML = '<h1>BOOOOOOM!!!</h1>';
});
```

This listens out for the `mouseover` event. The second parameter is an anonymous function that will be called when the mouse pointer goes over the bomb.

Have a go and see what happens!



You can see my code on CodePen.

Mouse Up and Down

The `mousedown` event fires when the left mouse button is pressed down and the `mouseup` event fires when it's released. This event can be attached to a

particular element, so the event will only fire when the mouse pointer is over the element.

We can combine these events with the `mousemove` event to implement a very simple drag-and-drop interface. Open up a new Pen on CodePen and enter the following code in the **HTML** section:

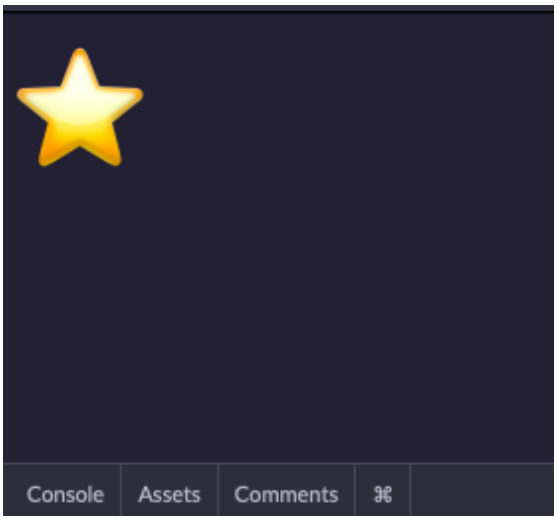
```
<div id='star'>⭐</div>
```

This is simply a `<div>` element containing a star emoji (you can choose any emoji you like!), which is the element that we'll be moving around the page.

Add the following code to the **CSS** section next:

```
#star{
  font-size: 64px;
  position: absolute;
}
```

This makes the element a bit bigger and sets the position property to `absolute`, which means that its position can be changed using the `left` and `top` properties. *This is essential for the drag and drop to work.*



Next, we'll move on to the JavaScript code. Add the following to the **JS** section:

```
const star = document.querySelector('#star');
```

Our first line gets a reference to the star element, using a query selector. Next we need to add a couple of event handlers to this element:

```
star.addEventListener( 'mousedown', start);
star.addEventListener( 'mouseup', stop );
```

This will call the function `start` when the mouse button is pressed down on the star and will call the `stop` function when the mouse button is released. Let's write the `start` function first:

```
function start(e) {
  document.addEventListener( 'mousemove', move);
}
```

This just adds an event listener to the whole page. This is listening for the `mousemove` event and it calls the `move` function. Let's write that next:

```
function move(e){
  star.style.left = `${e.x}px`;
  star.style.top = `${e.y}px`;
}
```

The `move` function is what makes the dragging possible. It simply sets the position of the star (`star.style.left` and `star.style.top`) to equal the position of the mouse pointer that's stored in the event object properties of `x` and `y`. (The event object is assigned to the parameter `e` in the code above.) Note that a template literal is used so that `px` can be added to the end (the units are measured in pixels).

This will result in the star following the mouse pointer around the screen. We want this behavior to stop when the mouse button is released and the `stop` function is called. To make it stop, we need to remove the `mousemove` event listener.

Removing Event Listeners

An event listener can be removed using the `removeEventListener()` method. This will remove an event listener that matches the same parameters.

Naming `addEventListener` Functions

You shouldn't use anonymous functions as an argument to `addEventListener` if you want to remove the event listener later in the program. This is because there needs to be a reference to the same function name in the arguments of `removeEventListener`.

We want to remove the following event listener that we added earlier:

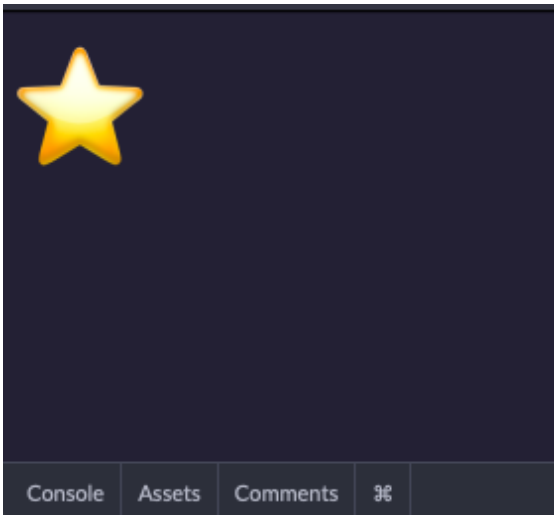
```
document.addEventListener( 'mousemove', move);
```

This means that the `stop` function should be the following:

```
function stop(){
  document.removeEventListener( 'mousemove', move );
}
```

This will remove the event listener, so the position of the star will stop matching the position of the mouse and so stop moving. If the user clicks on the star again, the event listener will be added once more and the dragging will start again.

Try dragging and dropping the star around the page. It should look something like this:



You can see my code on CodePen.

Just a Basic Example

This is a *very* basic implementation of drag-and-drop behavior, so it might be a bit jumpy when you try using it, depending on which browser you use. A more robust version is provided in the Challenges section at the end of the chapter.

Simple To-do List

We're going to finish the chapter by putting a few of the things we've learned together to build a slightly more sophisticated to-do list app than we built previously. It will allow you to add items to the list and strike them out by clicking on them—this time without having to mess with the console.

To get started, open up a new Pen on CodePen and enter the following into the **HTML** section:

```
<form name='addTask'>
  <input type='text' name='newTask'>
  <button type='submit'>ADD</button>
</form>

<ul id='list'></ul>
```

This is almost identical to the HTML code we used earlier in the chapter for our form input example. It creates a form with a single input box and submit button. There's also an empty unordered list element (`<ul>`) That we'll dynamically place the items into when the program is running.

Now for the code to get it working! First of all, we need to get some JavaScript references to the HTML elements:

```
const list = document.getElementById('list');
const form = document.forms.addTask;
```

This gives us a reference to the `form` element and that empty `<ul>` element.

Now let's focus on adding the items to the list. To do that, we'll need to attach an event listener to the form:

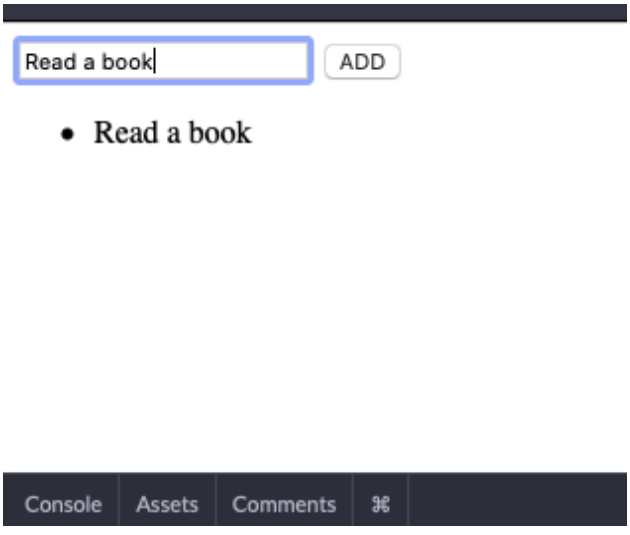
```
form.addEventListener('submit',addTask);
```

This will call the `addTask` function when the form is submitted. All we need to do now is write the `add` function:

```
function addTask(e){
  e.preventDefault();
  const task= document.createElement('li');
  task.textContent = form.newTask.value;
  list.appendChild(task);
}
```

The first thing this function does is stop the form from submitting to the server using the `preventDefault()` method. After this, we use some of the DOM methods we learned in the last chapter to create a new list item (`<li>`) element that's assigned to the variable `task`. Next, we make the `textContent` property equal the text that was entered into the form field, which is stored in `form.newTask.value`. Last of all, we use the `appendChild()` method to add this new element to the bottom of the `list` element.

Try adding a task in the input field, and if everything has gone to plan, it should look something like this:



Now all we need to do is make it possible to strike the items out when they're clicked on. Let's add an event listener to deal with clicking on the items:

```
list.addEventListener('click',strikeTask);
```

You might have noticed that I've added the event listener to the list element rather than the individual list items.

If we were to attach `click` event listeners to all the `<li>` tags, we'd need a separate event listener for each element. We'd also have to keep adding new event listeners on the fly every time a new item was added. This could lead to a huge number of event listeners all containing identical code and basically doing the same thing—which is not very DRY at all!

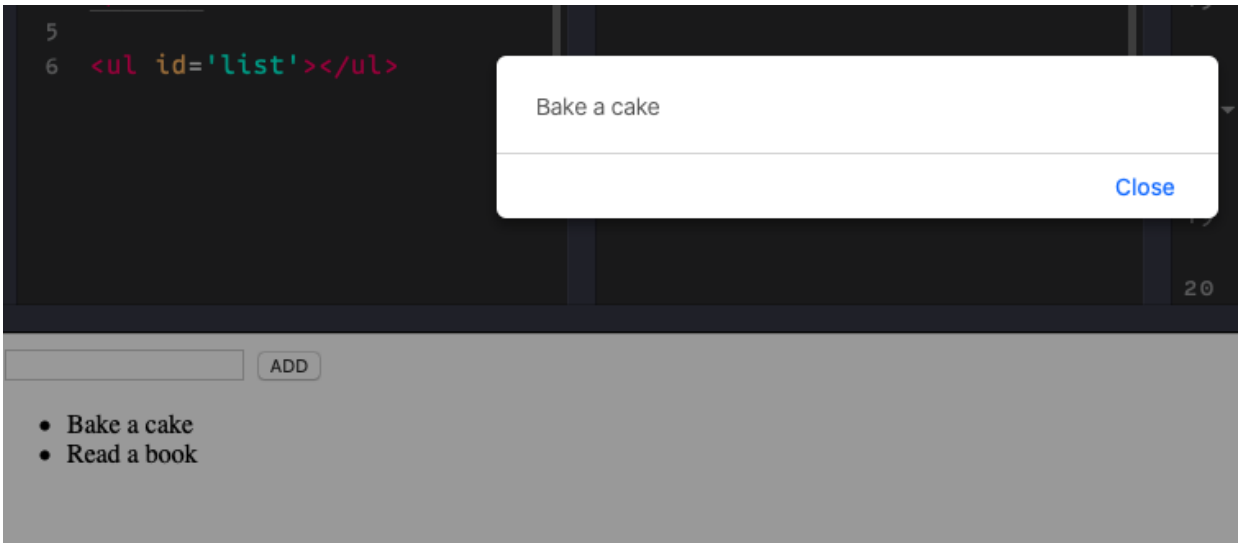
Event Delegation

The DRY way to deal with this is to use **event delegation**. This attaches the event listener to the parent `<ul>` element instead of each individual list item. The event will still fire when one of the child elements is clicked on, and we can then use the `target` property of the event object to identify the actual element that was clicked on.

Let's see this in action. Add the following code to the bottom of the **JS** section:

```
function strikeTask(e){
  alert(e.target.textContent);
}
```

This is the `strikeTask` function that fires when the list is clicked on. For now, it includes an alert to show the text content of the element that was clicked on and identified using `event.target`. Try adding some items and then click on one of them. You should see something like this:



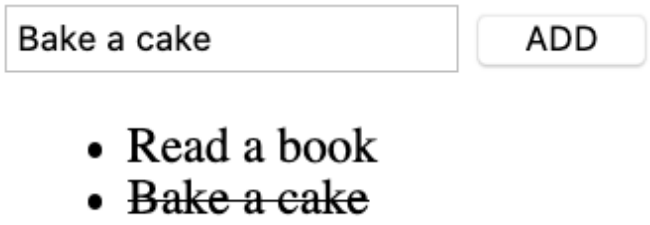
Now that we're sure it's working, let's update the code so that it crosses out the item instead. Update the `strike` function like so:

```
function strikeTask(e){
  e.target.classList.toggle('complete');
}
```

This uses a method of the `classList` property that every element has, called `toggle`. This will add a class if the element doesn't have it, and remove the class if it already has it. This means that clicking on an item will either add a class of `complete` or remove it if it's already there. This on its own doesn't do anything, but now we can add some CSS to style any elements that have a class of `complete` with the following code:

```
.complete {
  text-decoration: line-through;
}
```

Now if you try adding some items and clicking on them, you should be able to strike them out to your heart's content, as you can see below.



You can see my code on CodePen.

Challenges

1. The drag-and-drop example we created in this chapter is quite a simple implementation. A more robust version can be seen here. Take a look at the code and see if you can understand what's going on. It also uses touch events.

2. Can you update the simple to-do list code so that the `add` function clears the input box and gives it focus after an item has been added, instead of the text just remaining there? (Hint: you might want to look up the `focus()` method that form elements have.) You can see my solution here.

3. Instead of crossing out the items in the simple to-do list, can you change the functionality so that it removes the items when they're clicked on? (Hint: you could use the `remove()` method that we covered in the last chapter.) You can see my solution here.

Summary

- Events occur when a user interacts with a web page.
- An event listener is attached to an element and then invokes a callback function when the event occurs.
- The event object is passed to the callback function as an argument, and contains a number of properties and methods relating to the event.

- There are many types of events, including click events, mouse events and keyboard events.
- Forms also have a `submit` event that can be used to intercept a form before it has been submitted.
- You can remove an event using the `removeEventListener()` method.
- The default behavior of elements can be prevented using the `preventDefault()` function.
- Event delegation is when an event listener is added to a parent element to capture events that happen to multiple child elements in order to avoid adding an event listener for every single child element.

Now that we've covered events, we can make our code much more interactive. In the next few chapters, we're going to be taking our coding skills to the next level with some more advanced skills.

Chapter 12: Going Loopy Over Arrays

You're now a good three quarters of the way through this book and well on your way to learning how to code. In fact, you already know enough to be able to write some short programs.

Now that you've learned the basics, it's time to dive a bit deeper into some more advanced topics in the last quarter of the book. The last few chapters will focus on some of the more specific parts of JavaScript—although a lot of the overarching concepts will still apply in other languages.

In this chapter, we're going to learn about how to iterate over collections such as arrays and objects. We'll be covering the following:

- spreading strings
- array iteration
- JavaScript array iteration methods: `forEach`, `map`, `reduce`, `filter`, `find`, `every`, `some`
- iterating over objects

Spreading Strings

We met the spread operator (. . .) way back in Chapter 5. It can be used in JavaScript to spread out the items of an array when the array is placed inside another array.

The spread operator can also be used to spread out all the characters of a string inside an array. Each character of the string becomes an individual item in the array:

```
[..."Hello"];
<< ["H", "e", "l", "l", "o"]
```

This can be a very useful technique, as it allows us to use array methods on a string. For example, we could reverse a string with the following code:

```
[..."Hello"].reverse().join('');
<< "olleH"
```

The code above uses **chaining**, which is the process of applying one method after another in a chain of method calls. Each subsequent method is called on the return value of the previous method. In the example above, the spread operator returns an array, then we immediately call the `reverse()` method on this array that returns another array, and then we immediately call the `join()` method on this array, which returns the final string. The following code snippets break down what's happening in each step of the chain:

1. The string starts as this:

   ```
   "Hello";
   ```

2. Then the spread operator is applied and returns the following array:

   ```
   [..."Hello"];
   << ["H", "e", "l", "l", "o"]
   ```

3. Then the `reverse()` method is called to return the following array:

   ```
   [..."Hello"].reverse();
   << ["o","l","l","e","H"]
   ```

4. Finally, the `join()` method is called on the array to change it back to a string:

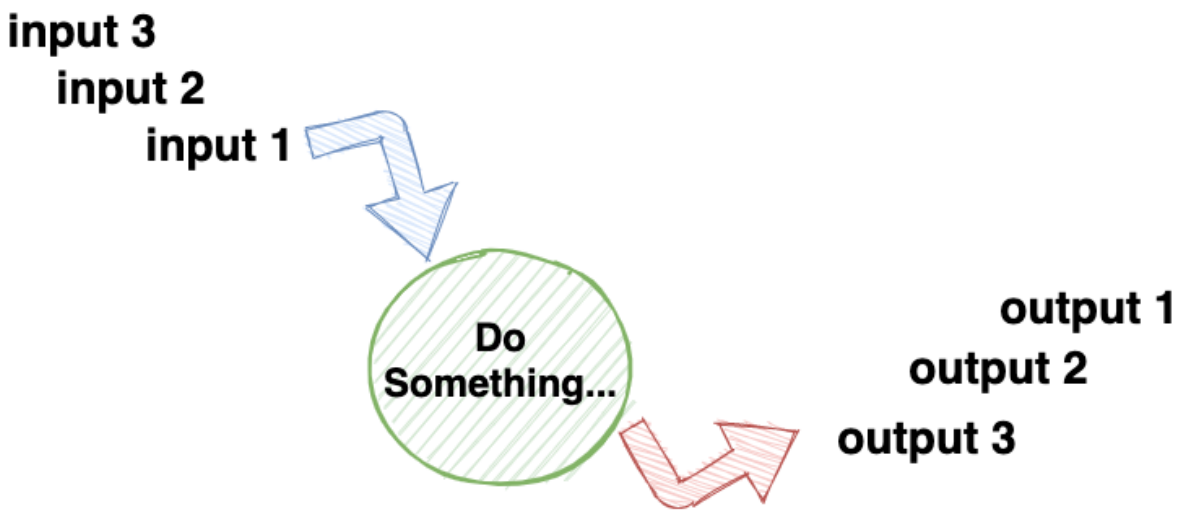
   ```
   [..."Hello"].reverse().join('');
   "olleH"
   ```

Each of those steps happens one after the other without returning any of the intermediate values, so all we see is the final return value.

One particular benefit of using the spread operator on strings is that we can use iteration methods on them. So, what are iterators?

Array Iteration Methods

Array iteration methods are methods that allow you to loop over an array and apply an operation to every item in the array. Every application of the operation to an item in the array is called an **iteration**.



JavaScript includes a number of array iteration methods that use callbacks to provide the instructions about what to do in each iteration. These methods always leave the original array unchanged, but often return a new array or value.

Efficient Arrow Functions

You'll notice that arrow functions are frequently used to declare the callbacks in the following examples. They're good candidates here, because they're short, can be written in a single line, and have an implicit return value.

forEach

The `forEach()` method iterates over every item in an array and calls a callback function on every iteration. The callback function takes three parameters:

- the value of the current item in the array

- the index of the current item in the array
- a reference to the array itself

If the method doesn't use the index or array references, they don't need to be specified. (Often only the value of the current item is provided as an argument.)

The following code shows an example that logs some information about every item in the array to the console. It iterates over each item in the array and logs a statement to the console for each item in the array. Try entering the following code (available in this Pen) into the console:

```
['🍏','🥢','🥕'].forEach(
  (item,index,array) => console.log(`Item at position ${index}
is ${item} (there are ${array.length} items in total).`)
);
```

On the first iteration, the value of `item` is "🍏", the value of `index` is 0 and the value of `array` is ['🍏','🥢','🥕']. This logs the following string to the console:

```
"Item at position 0 is 🍏 (there are 3 items in total)."
```

On the second iteration, the value of `item` is "🥢", the value of `index` is 1, and the value of `array` is ['🍏','🥢','🥕']. This logs the following string to the console:

```
"Item at position 1 is 🥢 (there are 3 items in total)."
```

On the final iteration, the value of `item` is "🥕", the value of `index` is 1, and the value of `array` is ['🍏','🥢','🥕']. This logs the following string to the console:

```
"Item at position 2 is 🥕 (there are 3 items in total)."
```

This is summarized in the table below:

Iteration	`item`	`index`	`array`	`console.log`

Iteration	item	index	array	console.log
1	"🍏"	0	['🍏','🍐','🥕']	"Item at position 0 is 🍏 (there are 3 items in total)."
2	"🍐"	1	['🍏','🍐','🥕']	"Item at position 1 is 🍐 (there are 3 items in total)."
3	"🥕"	2	['🍏','🍐','🥕']	"Item at position 2 is 🥕 (there are 3 items in total)."

An Array of Cards

We can use the `forEach()` method to quickly create an array that represents a deck of cards. We'll do it by iterating over an array that represents the suits, and another that represents the value of each card in a suit. (You can copy the code for the following example from CodePen.)

Open up a console and declare the following variables:

```
const suits = ['♠','♦','♣','❤'];
const values = ['Ace',2,3,4,5,6,7,8,9,10,'Jack','Queen','King'];
const deck = [];
```

The first variable `suits` is an array that contains string representations of each of the suits. The variable `values` is an array that contains a string representation of each of the card values. The variable `deck` is an empty array that will eventually hold the full deck of cards.

Now that we have these arrays set up, we can iterate over the `suits` array and the `values` array to combine the elements into a single card that can be placed into the `deck` array using the `push()` method:

```
suits.forEach(suit =>
    values.forEach(value =>
        deck.push(`${value} of ${suit}`);
    )
);
```

The first `forEach` iterates over the `suits` array. For every iteration, the anonymous callback function is called, with the parameter `suit` representing

the current item in the `suits` array. The second `forEach` then iterates over the `values` array and uses the parameter `value` to represent each item in that array. We then use a template literal to interpolate the suit and value of the card to create a string that's pushed into the `deck` array.

For example, on the first iteration, the value of `suit` is the first item in the `suits` array, ♠, and the value of `value` is the first item in the `values` array, `Ace`. This results in the string `Ace of ♠` being pushed into the `deck` array. This iterates over every value of the `values` array before moving on to the next value in the `suits` array. Some of the steps are shown in the table below:

Iteration	**`suit`**	**`value`**	**returned string**
1	"♠"	"Ace"	"Ace of ♠"
2	"♠"	2	"2 of ♠"
3	"♠"	3	"3 of ♠"
...			
13	"♠"	"King"	"King of ♠"
14	"♦"	"Ace"	"Ace of ♦"
...			
52	"❤"	"King"	"King of ❤"

The final result is that the `deck` array contains all these strings:

```
deck;
<< ["Ace of ♠", "2 of ♠", "3 of ♠", "4 of ♠", "5 of ♠", "6 of
♠", "7 of ♠", "8 of ♠", "9 of ♠", "10 of ♠", "Jack of ♠", "Queen
of ♠", "King of ♠", "Ace of ♦", "2 of ♦", "3 of ♦", "4 of ♦", "5
of ♦", "6 of ♦", "7 of ♦", "8 of ♦", "9 of ♦", "10 of ♦", "Jack
of ♦", "Queen of ♦", "King of ♦", "Ace of ♣", "2 of ♣", "3 of
♣", "4 of ♣", "5 of ♣", "6 of ♣", "7 of ♣", "8 of ♣", "9 of ♣",
"10 of ♣", "Jack of ♣", "Queen of ♣", "King of ♣", "Ace of ❤",
"2 of ❤", "3 of ❤", "4 of ❤", "5 of ❤", "6 of ❤", "7 of ❤",
"8 of ❤", "9 of ❤", "10 of ❤", "Jack of ❤", "Queen of ❤",
"King of ❤"]
```

You can see my code on CodePen.

Map

The `map()` method also iterates over an array and uses a callback function as a parameter that's called on each item in the array. In contrast to `forEach`, `map` returns a *new array* that replaces each value with the return value of the callback function.

The callback to the `map()` method has the same three parameters as the `forEach()` method:

- the value of the current item in the array
- the index of the current item in the array
- a reference to the array itself

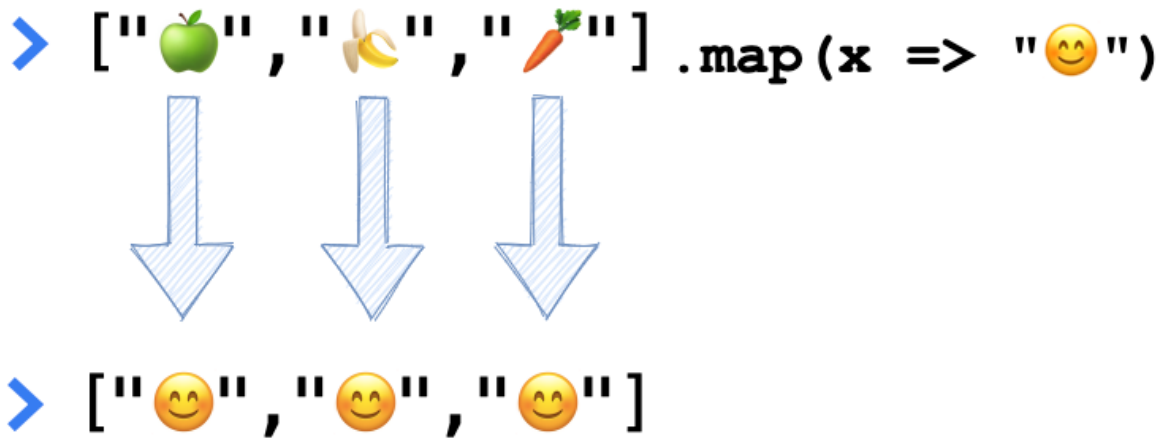
To demonstrate this, try the following code in the console:

```
["🍏","🥕","🥕"].map((value,index,array) => '😊');
<< ["😊","😊","😊"]
```

The callback function is an anonymous arrow function with parameters of `value`,`index` and `array`. The return value is just the string `'😊'`, so every item in the array is mapped to a smiley face!

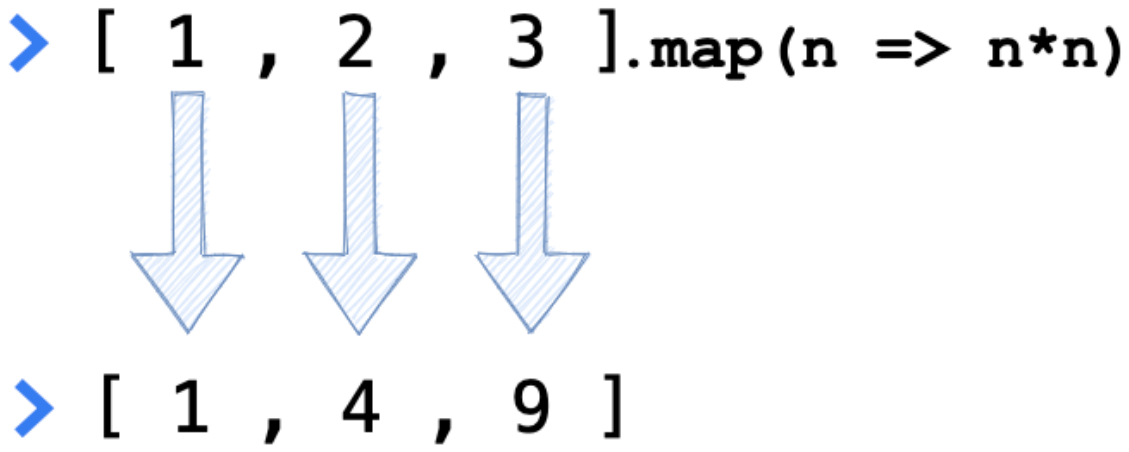
In this case, the callback doesn't need to use the `item`, `index` or `array` parameters, so they don't actually need to be there. The following callback also works with a single parameter of `x`:

```
[1,2,3].map(x => '😊');
<< [ 😊,😊,😊 ]
```



The return value can be based on the arguments provided to the callback. In the following example, every number in the array is mapped to the square of itself in a new array:

```
[1,2,3].map(n => n*n);
<< [1, 4, 9]
```



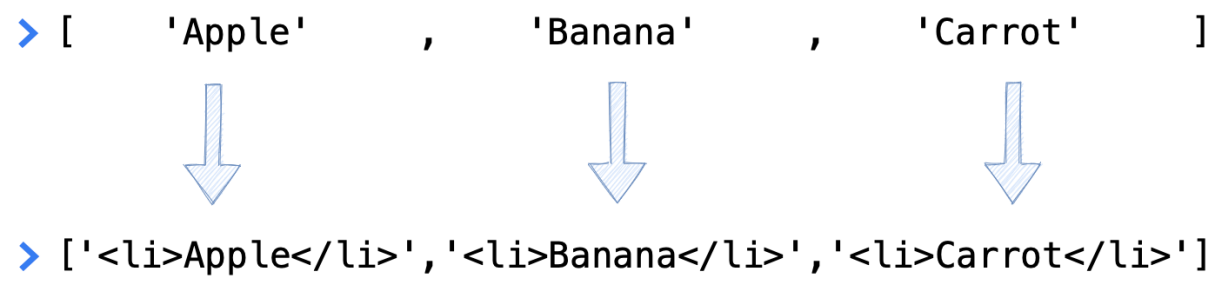
You can use any name for the parameters. In the example above, we use `n` to represent each item in the array. Since the callback function only relies on the value of each item, we only needed to provide this parameter.

A particularly good use of the `map()` method is to add HTML tags to blocks of text. For example, the following code takes an array of items and then

returns each item inside `<li>` tags:

```
['Apple','Banana','Carrot'].map(item =>
`<li>${item}</li>`);
<< ["<li>Apple</li>", "<li>Banana</li>", "<li>Carrot</li>"]
```

The parameter `item` represents each string in the array on each iteration. The return value of the callback is a template literal that uses string interpolation to insert the string inside `<li>` tags.



We can then chain the `join()` method with an empty string argument to the end to concatenate all the items into a single string of HTML:

```
['Apple','Banana','Carrot'].map(item =>
`<li>${item}</li>`).join('');
<< "<li>Apple</li><li>Banana</li><li>Carrot</li>"
```

This can then be inserted into a `<ul>` element to make an unordered list based on the contents of the array. Let's try doing this on CodePen. Start with an empty `<ul>` element in the **HTML** section:

```
<ul id='list'></ul>
```

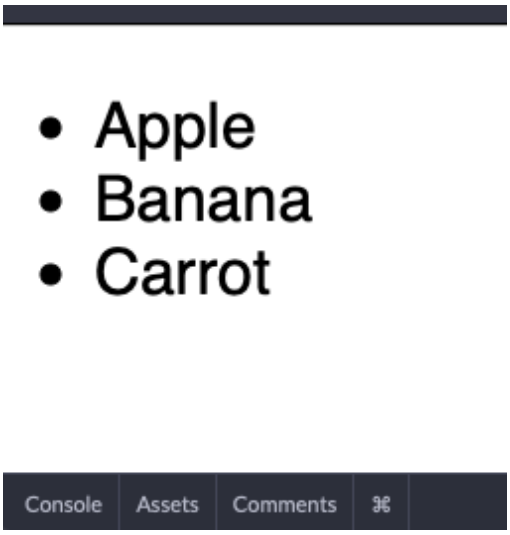
Now we'll get a reference to that element in the **JS** section:

```
const list = document.getElementById('list');
```

Now all we need to do is update the `innerHTML` property of this element with our string of HTML:

```
list.innerHTML = ['Apple','Banana','Carrot'].map(item =>
`<li>${item}</li>`).join('');
```

This renders a list of items in the document, based on the items in the array:



You can see my code on CodePen.

Reduce

The `reduce()` method is another method that iterates over each value of an array, calling a callback function each time. The main difference is that, instead of returning an array, it combines each result from the callback into a single value.

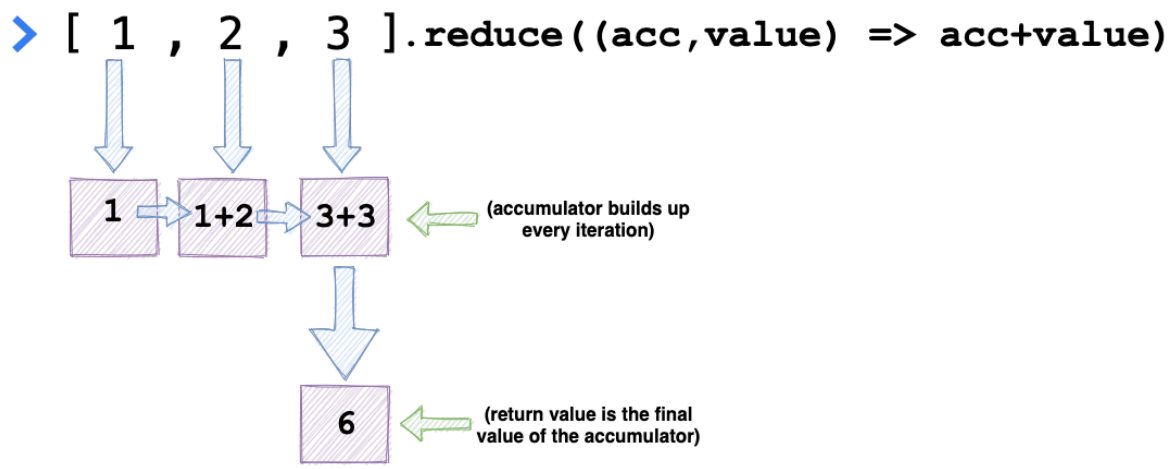
The callback to the `reduce()` method describes how to combine each value of the array to produce a cumulative running total and has four parameters:

- An accumulator that stores the running total. The final value of this will be returned by the method.
- The value of the current item in the array.
- The index of the current item in the array.
- A reference to the array itself.

Most of the time, only the first two parameters are used, and any that aren't used don't need to be referenced.

The following example adds up all the numbers in the array and returns the total:

```
[1,2,3].reduce(
  (acc,value) => acc + value
);
<< 6
```



In the example above, the value of `acc` starts as 1 (the first item in the array). After every iteration, the value of the item in the array is added to the value of `acc`. So in the next iteration, 2 is added to `acc` to make it 3. Then in the last iteration, 3 is added to `acc`. Once every item in the array has been added together, the final value of `acc`, which is 6, is returned.

By making a small change to the calculation in the callback function, we can multiply all the numbers together instead:

```
[1,2,3].reduce(
  (acc,value) => acc * value
);
<< 6
```

Adding vs Multiplying

It's only a coincidence that the answer is the same when we added and multiplied the numbers together. (Try using an array with different numbers to see.)

The `reduce()` method also accepts an optional second argument that comes after the callback and allows us to set the initial value of the accumulator.

For example, the following method adds all of the numbers in the array together (getting 6), then adds that result to 10 before returning 16:

```
[1,2,3].reduce(
  (acc,value) => acc + value
,10);
<< 16
```

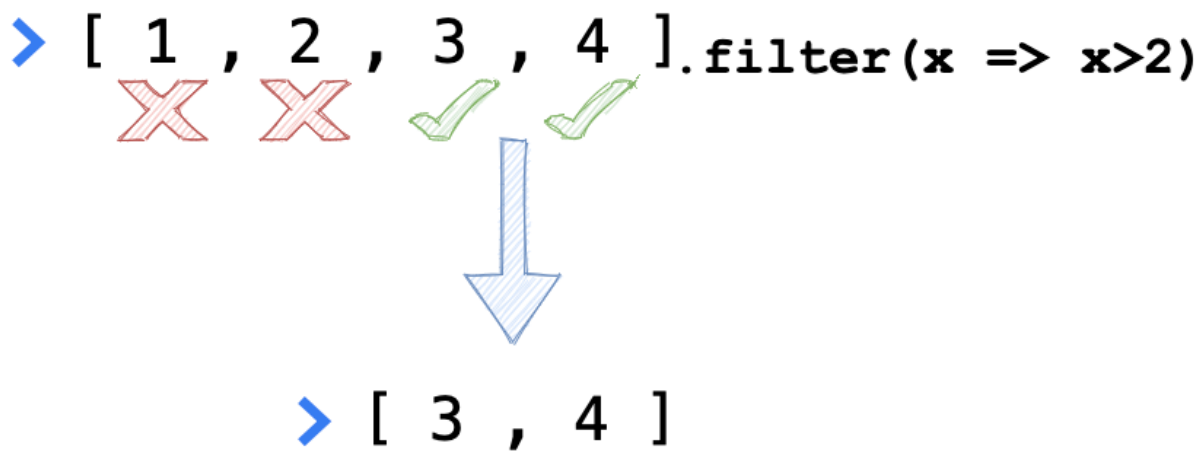
Filter

The `filter()` method tests each item in an array to see if they match certain conditions defined in a callback function. Any items that return a truthy value in the callback are then returned as a *new array*.

For example, we can filter an array of numbers to leave just the numbers bigger than 2 using the following code:

```
[1, 2, 3, 4].filter(x => x > 2 );
<< [3,4]
```

In each iteration, the callback checks to see if the item in the array, represented by the parameter `x`, is greater than 2. If it is, the callback returns `true` and the value is left in the new array that's returned. This process can be seen in the diagram below.



The `filter()` method provides a useful way of finding all the truthy values from an array:

```
[ 0, 1, '0', false, true, 'hello' ].filter(Boolean);
<< [ 1, '0', true, 'hello' ]
```

This uses the fact that the `Boolean` function will return the Boolean representation of a value, so only truthy values will return `true` and be returned by the `filter()` method.

To find all the falsy values, the following filter can be used:

```
[ 0, 1, '0', false, true, 'hello' ].filter(x => !x);
<< [ 0, false ]
```

This uses the not operator (`!`) to return the complement of a value's Boolean representation. This means that any falsy values will return `true` and be returned by the filter.

Guess Who Filter

Way back in Chapter 6, we played the Guess Who? game with these four characters:

In Chapter 9, we created objects to represent each of them:

```
const alfie = {
  name: 'Alfie',
  glasses: false,
  hat: false
};
const betty = {
```

```
  name: 'Betty',
  glasses: true,
  hat: true
};
const gemma = {
  name: 'Gemma',
  glasses: false,
  hat: true
};
const del = {
  name: 'Del',
  glasses: true,
  hat: false
};
```

We can place these objects into an array called `people` like so:

```
const people = [alfie,betty,gemma,del];
```

Now we can have some fun with this array by using the `filter()` method to play Guess Who?:

```
people.filter(person => person.glasses && person.hat);
<< [{ name: 'Betty', glasses: true, hat: true}]
```

The filter above returns the object representing Betty because she wears glasses and a hat. If we want to find the person *not* wearing a hat and *not* wearing glasses, we could use the negation operator inside the callback instead:

```
people.filter(person => !person.glasses && !person.hat);
<< [{ name: 'Alfie', glasses: false, hat: false}]
```

By changing the conditions in the callback, you can filter the array in different ways. Try changing these conditions to see if you can return each person.

You can see my code on CodePen.

Find

The `find()` method works in a similar way to the `filter()` method, but it returns the *first* value that matches the criteria defined in the callback. For example, the following code returns the first number that's greater than 2:

```
[1, 2, 3, 4].find(x => x > 2 );
<< 3
```

The following code finds the first programming language that begins with the letter "J":

```
['C','C++','Ruby','Python','JavaScript','Swift','Java'].find(wor
d => word.startsWith('J'));
<< "JavaScript"
```

We can use this to find people in our `people` array from the Guess Who? game that match the given criteria. The following example finds the first person who wears glasses, but not a hat:

```
people.find(person => person.glasses && !person.hat).name;
<< "Del"
```

Notice that, because the `find()` method returns the first matching element—which in this case is an object—we can chain the property of `name` on the end so that only the value of this property is returned, rather than the whole object.

Every

The `every()` method iterates over each item in an array and returns `true` if *every* item in the array matches the criteria defined in the callback. For example, the following code checks if all the words in the array are longer than a single character:

```
['C','C++','Ruby','Python','JavaScript','Swift','Java'].every(wo
rd => word.length > 1);
<< false
```

Some

The `some()` method is very similar to the `every()` method. It also iterates over each item in an array until a given condition defined in the callback returns `true`. Once this happens, the iteration stops and the method returns `true`. If it reaches the end of the array without any of the items returning `true`, it returns `false`. It's a useful way to find out if at least one item in an array fits certain criteria.

For example, the following code checks if any of the words are longer than seven characters:

```
['C','C++','Ruby','Python','JavaScript','Swift','Java'].some(wor
d => word.length > 7);
<< true
```

A useful summary of array iteration methods can be found in "The Array Iterators Cheatsheet for JavaScript".

There are some good examples of chaining iteration methods together in "Filtering and Chaining in Functional JavaScript".

Iterating over Objects

Objects are collections, just like arrays, and it's also possible to iterate over an object's properties.

For example, consider the following `rectangle` object that's similar to the `square` object we created back in Chapter 9:

```
const rectangle = {
  height: 4,
  length: 5,
  perimeter() { return 2 * (this.height + this.length); },
  area() { return this.length * this.height; }
}
```

We can loop through all the properties of an object using a for–in loop that iterates over every key in the object. Try entering the following code into a console (you'll need to define the rectangle object first):

```
for(const prop in rectangle) {
  console.log(`${prop}: ${rectangle[prop]}`);
}
<< height: 4
<< length: 5
<< perimeter: function (){ return 2 * (this.height +
this.length) };
<< area: function { return this.length * this.height }
```

In this example, the variable `prop` is used to reference each property name in each step of the iteration. We can then use `rectangle[prop]` to look up the value of that property. In the example, we've simply used `console.log` to log the property and its value to the console using a template literal.

Keys and Values

JavaScript also provides a set of methods that return an array containing all the properties and values of an object. This means that we can then use any array methods, such as the array iteration methods we've learned about in this chapter.

The `Object.keys()` method returns an array of all the keys of the object that's provided as an argument:

```
Object.keys(rectangle);
<< ["height","length","perimeter","area"]
```

As you can see, the key of each property is listed as a string in the array that's returned.

The `Object.values()` method works in the same way, but returns an array of the values of each property instead:

```
Object.values(rectangle);
<< [4,5, perimeter(), area()]
```

Each value in the array is the same as its original data type—so numbers and functions in this case.

The `Object.entries()` method returns an array of key–value pairs. These key–value pairs are returned as sub-arrays inside the array:

```
Object.entries(rectangle);
<< [["height",4],["length",5],["perimeter",function()],
["area",function()]]
```

Because these methods return an array, we can use chaining to immediately call one of the iteration methods we covered earlier in the chapter. For example, we could use the `forEach()` method to log the properties and their values to the console:

```
Object.entries(rectangle).forEach(subArray
=>console.log(`${subArray[0]}: ${subArray[1]}`));
<< "height: 4"
   "length: 5"
   "perimeter: function perimeter() {return 2 * (this.height +
this.length);}"
   "area: function area() {return this.length * this.height;}"
```

This uses a parameter called `subArray` to refer to each sub-array in the `Object.entries` array. We then use index notation to refer to each item in the sub-array, so `subArray[0]` refers to the property name and `subArray[1]` refers to the property value. This is fine, but using index notation like this makes the code difficult to follow.

We can make it easier to read by explicitly naming each item in the sub-array when we define the callback, like so:

```
Object.entries(rectangle).forEach(([prop,value])=>console.log(`$
{prop}: ${value}`));
```

This allows us to refer to the first item in the sub-array as `prop` and the second item as `value`, which makes the code much more readable. This process is known as **destructuring**.

To-do List Project

In the last chapter, we used events to produce a working to-do list. The problem with this was that the to-do items were added directly to the web page and never stored anywhere in the program. This can be an issue if the program wants to know the state of the to-do list at any point. This concept is known as **state management**. One way to manage the state of the to-do

list is to store the tasks in an array. This means that, at any point in the program, we can use the array to find any of the tasks, or calculate how many tasks there are needing to be completed. And the good news is that, if you've been following along with all the challenges, you'll already have some `add` and `remove` functions for arrays from back in Chapter 8.

We're going to update the to-do list code on CodePen so that it includes some state management.

Forking Pens

CodePen has a **Fork** option (bottom right) that allows you to make a copy of a Pen and make changes without affecting the original code. This is useful when you want to make some major changes to your code.

Our first job is to create an array to store the tasks in. Add the following code to the top of the **JS** section:

```
const tasks = [];
```

Next, we need to change the function that adds a task. This is the `addTask` function that's called when the form is submitted:

```
function addTask(e){
  e.preventDefault();
  tasks.push(form.item.value);
}
```

This code now adds tasks to the `tasks` array when the form is submitted, instead of writing them onto the page.

When a user clicks on a task, it's removed from the page using the `removeTask` function. We need to update this function so that it removes the task from the array instead:

```
function removeTask(e){
  const index = tasks.indexOf(e.target.textContent);
  if(index>-1){
    tasks.splice(index,1);
  }
}
```

This code should now work, but if you try to add a task, nothing will appear. However, if you take a look at the `tasks` variable in CodePen's built-in console (click on the **Console** button in the bottom-left corner), you should see that the array has some items in it:

```
tasks;
<< ["Bake Cake","Read Book"]
```

Now all we need to do is take the tasks from this array and display them in the browser. We're going to create a function called `renderList` that will take the array of tasks and render them in the document. First of all, we'll use `map` to wrap each task inside `<li>` tags:

```
tasks.map(task => `<li>${task}</li>`);
```

This will return the following array:

```
["<li>Bake Cake</li>","<li>Read Book</li>"]
```

If we chain `join('')` to the end, we can concatenate all the HTML into a single string:

```
tasks.map(task => `<li>${task}</li>`)).join('');
```

This will return the following string of HTML:

```
"<li>Bake Cake</li><li>Read Book</li>"
```

This can now be inserted into the DOM—by updating the `innerHTML` property of the list element (an empty `<ul></ul>` element that's already in the HTML):

```
list.innerHTML = tasks.map(task => `<li>${task}</li>`).join('');
```

This code now needs placing inside a `render` function that wraps each item in the `task` array in `<li>` tags using `map`, joins them together as a single string of HTML, and then inserts this into the list element. We'll also add a couple of lines to clear the text in the input field and give it focus:

```
function render(){
  list.innerHTML = tasks.map(task => `<li>${task}
```

```
</li>`).join('');
  form.item.value = '';
  form.item.focus();
}
```

Finally, we need to call the `render` function whenever the array is updated—which is after the `addTask` or `removeTask` functions are called. Update these functions so they call `renderList` after updating the array:

```
function addTask(e){
  e.preventDefault();
  tasks.push(form.item.value);
  render();
}

function removeTask(e){
  const index = tasks.indexOf(e.target.textContent);
  if(index>-1){
    tasks.splice(i,1);
  }
  render();
}
```

Now if you try to add tasks, they should appear in a list, and clicking on a task will remove it. This is no different from what it did before, but what's changed is all in the background: we can now keep track of all tasks that have been created from within the program by accessing the `tasks` array. This means we can keep track of how many tasks still need completing by looking at the return value of the `tasks.length()` method. Let's add some code that will display this underneath the list of tasks. First of all, add an empty `<div>` element to the bottom of the **HTML** section:

```
<div id='count'></div>
```

We'll also need to add a reference to this in the **JS** section:

```
const count = document.getElementById('count');
```

Next, add the following line of code to the `render` function so that it includes a line that updates the `count` div:

```
count.textContent = `${tasks.length}
task${tasks.length==1?'':'s'} left to complete.`
```

This uses a template literal to update the `textContent` property with the number of tasks and also uses a ternary operator to use “task” instead of “tasks” if there’s only one task in the list.

Try adding and removing some tasks. You should see the count update as they appear in the list.

You can see my code on CodePen.

Challenges

1. Write a function that accepts an array of strings as an argument and uses `map` to return a new array of the same words written in uppercase. Can you get it to return an array with all the words written backwards? You can see my code on CodePen.

2. Write a `spanner` function that accepts a string and returns a string, with each individual character wrapped in a `<span>` tag using the `map()` and `join()` methods. For example, `spanner('Hello')` should return the string `"<span>H</span><span>e</span><span>l</span><span>l</span><span>o</span>"`. You can see my code on CodePen.

3. Extend the `spanner` function you created in challenge 2 to create a `coloredLetters` function that colors each letter in a different color. It should accept a string and array of colors as parameters and then add a `style` attribute to each `<span>` tag. You can see my code on CodePen.

Summary

- The spread operator can be applied to strings to spread them into an array containing each character as a separate item.
- Iteration methods loop through every value in a collection and apply an operation to each value.
- JavaScript has a number array methods that apply a callback to every item in the array.
- The `forEach()` method applies the code in the callback for every item in the array.

- The `map()` method returns a new array by applying the code in the callback to every item in the array.
- The `reduce()` method returns a single value by applying an accumulator function defined in the callback to every value in the array —for example, adding up all the values in the array.
- The `filter()` method returns a new array containing only the values from the original array that match the criteria defined in the callback.
- The `find()` method returns the first item in an array that matches the criteria given in the callback.
- The `every()` method returns `true` if every item in the array matches the criteria given in the callback. Otherwise, it returns `false`.
- The `some()` method returns `true` if any item in the array matches the criteria given in the callback. It returns `false` if none of the items in the array match the criteria.
- You can iterate over objects using a for–in loop that will give you access to each property in the object.
- The `Object.keys()`, `Object.values()` and `Object.entries()` methods will return an array of an object's keys, values, or key–value pairs respectively.

In the next chapter, we'll be getting functional with functions.

Chapter 13: Let's Get Functional

We covered functions back in Chapter 8, but now it's time to dig a bit deeper and look at some more advanced topics specific to JavaScript.

In this chapter, we'll be covering the following:

- named parameters
- the rest operator
- hoisting
- scope
- recursive functions
- closures
- functional programming
- pure functions

Named Parameters

When a function has quite a few parameters, it can be difficult to remember what order to write the arguments in when you call the function. For example, consider this function that returns a styled `<div>` element:

```
function heading(text,color,size,bgcolor){
  return `<h1 style='color:${color};background-
color:${bgcolor};font-size:${size}>${text}</h1>`
}
```

This uses **positional parameters**, which means that when the function is called, the position the argument is placed in determines which parameter it's assigned to in the function. So, if you wanted a heading of "Hello, World!", sized to 48px, colored red on a blue background, you'd write the following:

```
heading('Hello, World!','red','48px','blue');
```

It might get difficult to remember the order those parameters are listed in when you come to call the function, which could lead to the colors getting mixed up or even worse, trying to assign a font size of `red`!

A number of languages use **named parameters** to get round this problem. With these, you can assign the value of each argument by name, instead of relying on its position. This means the arguments can be listed in any order, like so:

```
heading(color = 'red',bgcolor = 'blue',text ='Hello,
World!',size = '48px'});
```

Unfortunately, JavaScript doesn't support named parameters—strictly speaking—so the example above wouldn't work. The good news is that it's possible to mimic them by using an object as a parameter, so that the object's properties act like named parameters.

The following example shows how this can be done with the example above. The function needs rewriting, with the parameters listed as the properties of an object literal:

```
function heading({text,color,size,bgcolor}){
  return `<h1 style='color:${color};background-
color:${bgcolor};font-size:${size}>${text}</h1>`
}
```

Now the function can be called, providing an object as argument, with values provided for the named properties:

```
heading({color: 'red',bgcolor: 'blue',text:'Hello, World!',size:
'48px'});
<< "<h1 style='color:red;background-color:blue;font-
size:48px''>Hello, World!</h1>"
```

Notice that the order of the properties in the argument object isn't the same as their order in the object provided as a parameter to the function. The advantage of this method is that you don't have to remember the order to write the arguments. Another advantage is that it makes your code easier to follow, since each argument has a descriptive label that explains what it represents.

This technique is useful when a function has a large amount of parameters, particularly if some of them are optional.

You can see this example on CodePen.

The Rest Parameter

There are times when we don't know how many arguments will be provided to a function. The easiest way to deal with a varying number of arguments is to use the **rest parameter**. This consists of three dots placed in front of the last parameter in a function declaration. It will collect all the arguments together in an array.

Try entering the following function into the console. This function accepts any number of arguments and uses the rest parameter to collect them all together in an array called `numbers`. It then applies the `reduce()` method that we saw in the last chapter to this array and returns the sum of all the numbers that were entered as an argument:

```
function add(...numbers){
  return numbers.reduce((acc,n) => acc + n);
}
```

Now try calling the function for different numbers of arguments, like in the examples below:

```
add(2,3);
<< 5

add(1,2,3,4,5);
<< 15
```

Recursive Functions

A **recursive function** is one that calls itself! This might sound a bit crazy, but it's perfectly possible to place a self-referential function call inside the body of the function. The function calls itself until a certain condition is met. It's a useful tool when iterative processes are involved.

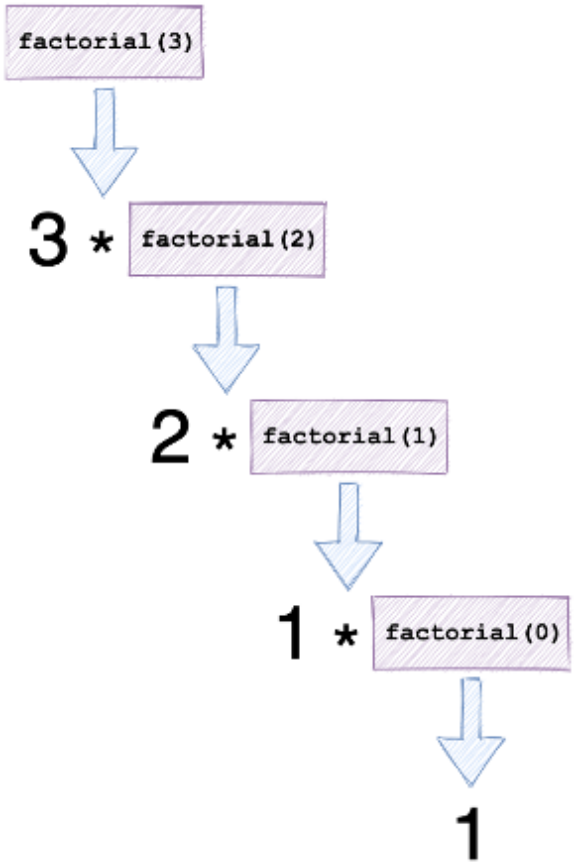
A common example is a function that calculates the factorial of a number:

```
function factorial(n) {
  if (n === 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

This function will return 1 if 0 is provided as an argument (0 factorial is 1), otherwise it will multiply the argument by the result of invoking itself with an argument of one less. The function will continue to call itself until finally the argument is 0 and 1 is returned. The best way to see what's happening is with an example:

```
factorial(3);
<< 6
```

This will multiply 3 by the return value of `factorial(2)`, which will multiply 2 by the return value of `factorial(1)`, which will multiply 1 by the return value of `factorial(0)`, which is 1. Working backwards gives `1 * 1 * 2 * 3`. This can be seen in the diagram below:



Recursive functions can be used for operations similar to those performed by the loops we saw in Chapter 7. They are particularly useful when implementing divide and conquer algorithms. You can read more about recursive functions in "Recursion in Functional JavaScript".

Scope

The concept of a variable's "scope" is important in any programming language. The **scope** of a variable or function refers to the parts of the program where they can be accessed. Most languages have some form of **lexical scope**, which means the scope of a variable is based on where it appears in the code. Often placing a variable inside a block will restrict its scope to that block. A few languages, such as Perl and other Lisp-style languages, use **dynamic scope**, which means the scope of a variable can change while the program is running.

Global scope covers the entire the program. Any variable or function that can be accessed anywhere in the program is said to have global scope.

Local scope refers to a function or variable that's only available inside a particular code block. Any function or variable defined inside a block can only be accessed inside that particular block, when they're "in scope".

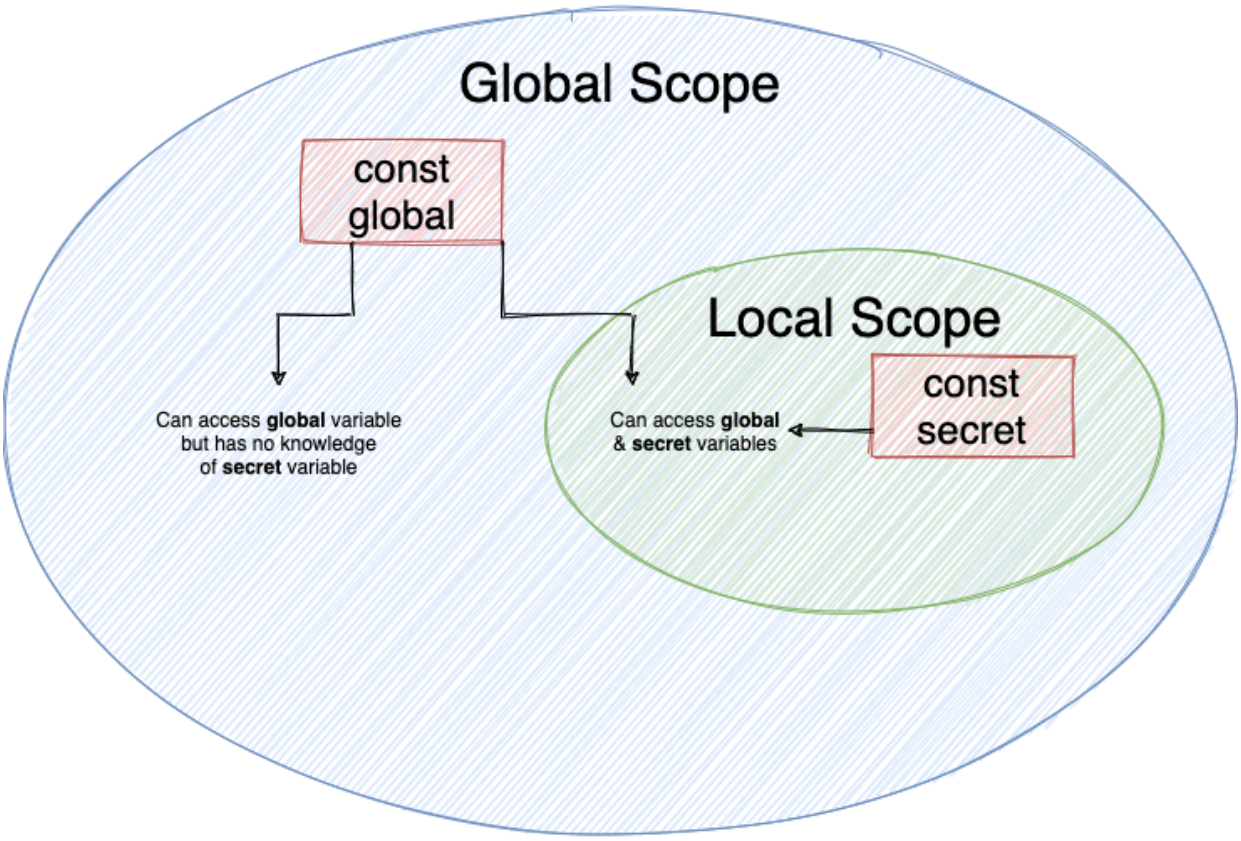
To demonstrate this, let's try defining some variables in the console. First of all, define a variable in the global scope by simply defining it in the main body of the program:

```
const global = 'Hello, Global Scope!';
```

Now, we can create a local scope by creating a function called `local`. Inside the body of this function, we'll declare a variable called `secret`:

```
function local(){
  const secret = 'Hello from the local scope.';
}
```

The diagram below shows the scope of each variable. `global` was declared in the global scope and can be accessed anywhere inside that scope (including from within the local scope). `secret` was defined inside the local scope of the `local` function and can only be accessed from within that scope.



If we try to log the value of the variable `global` to the console, we can see we have access to it:

```
console.log(global);
<< Hello, Global Scope!
```

However, if we try to access the variable `secret`, we get an error message, saying it can't be found:

```
console.log(secret);
<< ReferenceError: "Can't find variable secret"
```

This is because the variable `secret` is only available from inside the body of the `local` function. The last command was made from the global scope, where we don't have access to the local scope of the function.

To see that we can get access to the variable `secret` from within the scope of the `local` function, let's redefine the function to log the `secret` variable to the console:

```
function local(){
  const secret = 'Hello from the local scope.';
  console.log(secret);
}
```

Now if we call the `local` function, we can see that it does indeed have access to the `secret` variable:

```
local();
<< Hello from the local scope.
```

However, we only have access to this variable while the code inside the function is running. Once the function has been called, any variables defined inside its scope can't be accessed.

Although it would seem a good idea to keep everything in the global scope, it's actually considered bad practice. In fact, it's good practice to keep all variable declarations *out* of the global scope. This is to avoid **naming collisions**, where variables or functions have the same name and get mixed up with each other. This might not seem like a problem at the moment, but it becomes more of an issue if you're working with someone else's code (either as part of a team or using an external code library), since you won't know which variable names are already being used. By keeping variables and functions in their own scope, we make it much easier to keep track of what they do.

Block Scope

All variables in JavaScript have local scope inside functions, but variables only have **block scope** if they're declared using `const` and `let`. If a variable is declared using `var`, it can be accessed outside the scope of the block it was defined in.

If you try entering the following code into the console, it won't work, because the variable `message` only has scope inside the block, so it can't be accessed outside the curly braces:

```
if(name==="JavaScript"){
  const message = "Hello ${name}!";
}
```

```
console.log(message);
<< ReferenceError: "Can't find variable: message"
```

If `var` is used instead of `const`, this code will run without an error. Alternatively, `console.log(message)` could be placed inside the block.

Hoisting

Functions that are defined using a function declaration are automatically **hoisted** to the top of a program's scope. This means that they can be called before they've been defined. For example, in the following code, the function `hoist()` can be called before it's actually defined:

```
// function is called at the start of the code
hoist();

// ...
// ... lots more code here
// ...

// function definition is at the end of the code
function hoist(){
  console.log('Hoist Me!');

}
```

This can be quite useful, as it means that all function definitions can be placed together, possibly at the end of a program, rather than every function having to be defined before it's used.

An error will be thrown if you attempt to refer to a variable before it has been declared using `const` and `let`. For this reason, you should try to declare any variables at the beginning of a block so that hoisting isn't necessary.

This means that a function expression (where an anonymous function is assigned to a variable) can't be called before it has been declared, unlike a function declaration.

This is probably the biggest difference between function declarations and function expressions, and it may influence your decision regarding which one to use. Some people like the fact that using function expressions means

you're required to define all functions and assign them to variables prior to using them, while others prefer to have the option to keep all their functions in one place as function declarations. You can read more about the differences in "Quick Tip: Function Expressions vs Function Declarations".

Functions That Return Functions

We've already seen that functions can accept another function as an argument (a callback), but they can also *return* another function.

The example below shows a function called `returnHello()` that returns a "Hello, World!" function:

```
function returnHello() {
  return function() {
    console.log('Hello, World!');
  }

}
```

When the `returnHello()` function is called, all it does is return another function:

```
returnHello()

<< function returnHello() {
     return function() {
       console.log('Hello World!');
    }

   }
```

Alternatively, arrow functions can be used to make the declaration look neater (although it's harder to see what's happening):

```
returnHello = () => () => console.log('Hello, World!');
```

The expression after the first arrow is its return value, which is another arrow function: `() => console.log('Hello, World!')`. So when `returnHello` is called, it returns this function.

To make use of the function that's returned, we need to assign it to a variable:

```
const hello = returnHello();
```

Now we can call the function that was returned by placing parentheses after the variable that it was assigned to:

```
hello();
<< Hello, World!
```

This might seem a bit pointless, but let's now take it a step further and use this technique to create a generic function that returns a function for creating a fragment of HTML using a particular tag that's provided as an argument:

```
const html = tag => text => `<${tag}>${text}</${tag}>`
```

We now use this function to create a function that creates an `<h1>` element:

```
const h1El = html('h1');
<< text => `<${tag}>${text}</${tag}>`
```

Now this function can be used to return some strings of headings:

```
h1El('Hello, World!');
<< "<h1>Hello, World!</h1>"
```

If we also want a function for creating paragraphs, we can use the generic `element` function to return one for us:

```
paraEl = element('p');
<< text => `<${tag}>${text}</${tag}>`
```

Then we use this function to create a paragraph fragment:

```
paraEl('The quick, brown fox jumped over the lazy dog.');
<< "<p>The quick, brown fox jumped over the lazy dog.</p>"
```

We can also chain the two function calls together to create a function and use it in a single call:

```
html('h2')('Hello, World!');
<< "<h2>Hello, World!</h2>"
```

Closures

Closures are one of JavaScript's most powerful features, and they rely on the concept of scope and functions returning other functions.

To demonstrate the concept, let's go back to this function that we saw earlier in the chapter when discussing the concept of scope:

```
 function local(){
  const secret = 'Top Secret!';
  console.log(secret);
  }
```

This function contains a variable called `secret` which, as we saw, only exists within the scope of the function.

A **closure** is formed when a function can access variables that are declared outside its scope. In our example, we can add an anonymous function that references the `secret` variable:

```
function local() {
const secret = 'Top Secret!';
function () {
    console.log(secret);
  }

}
```

The anonymous function is said to form a "closure" over the `secret` variable because, although the variable isn't defined inside the body of the function, the function can still access it. This on its own isn't so special, since we could already access the `secret` variable from within the `local` function.

However, if we *return* the anonymous function, we'll *keep* access to the `secret` variable outside the scope of the `local` function:

```
 function local() {

```

```
const secret = 'Top Secret!';
return function () {
    console.log(secret);
  }

}
```

The function that's returned will retain access to the `secret` variable even after the `local` function has been called. This forms a *closure* over the `secret` variable.

To make use of the closure, we need to assign a variable to the return value of the `local()` function:

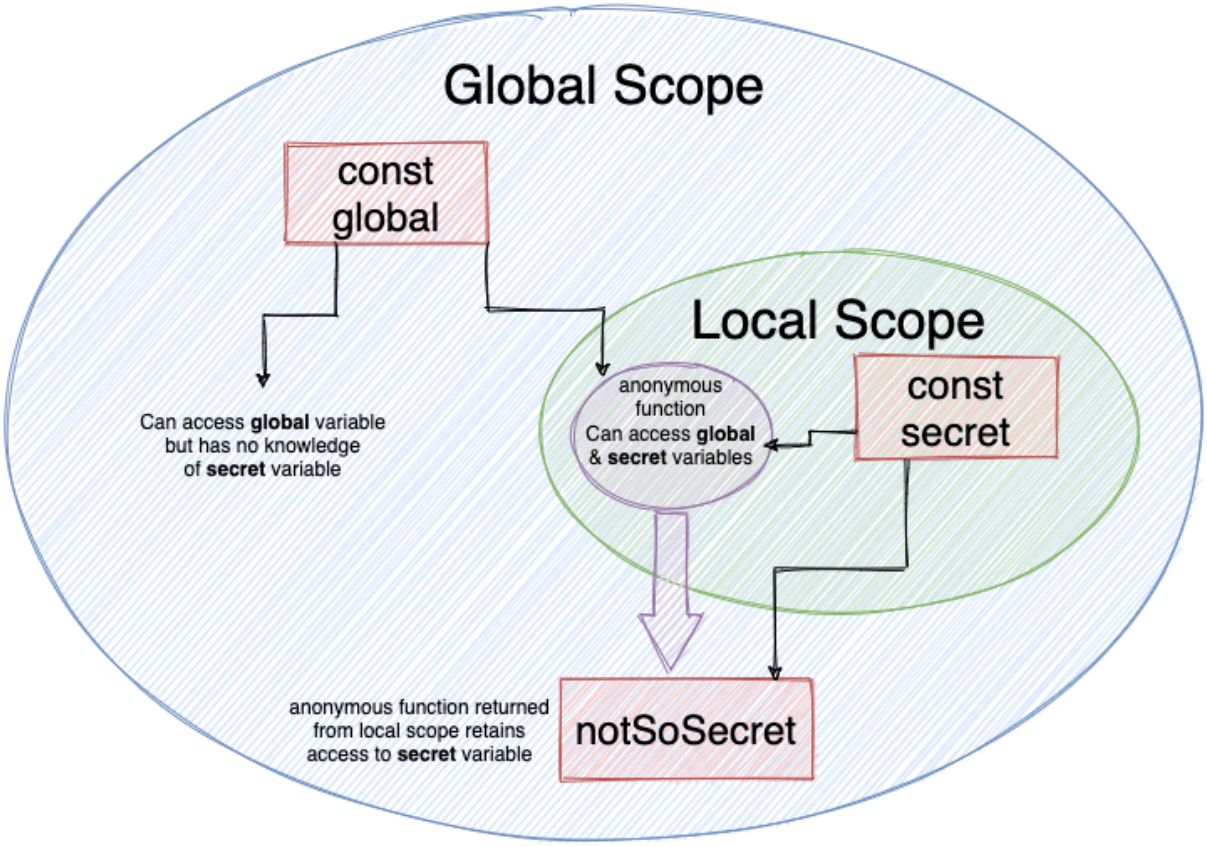
```
const notSoSecret = local();
```

The variable `notSoSecret` now points to the anonymous function that's returned by the `local()` function. This function now has access to the `secret` variable that was declared in the `local` function outside the scope of that function. We can see this is the case by calling the `notSoSecret` function:

```
notSoSecret();
<< "Top Secret!"
```

Thanks to the closure, we now have access to the `secret` variable in the global scope!

The diagram below shows how the anonymous function forms a closure over the `secret` variable, returning it into the global scope.



Closure Countdown!

Closures not only have *access* to variables declared in a parent function's scope, but they can also change the value of these variables. This allows us to do things like create a countdown function that decreases a variable every time it's called. Enter the following code into the console:

```
function countdown(start){
  let i = start;
  return function() {
    return i--;
  }

}
```

This function declares a variable `i` and assigns it the value of the number provided as an argument. It then returns a function that forms a closure around the variable `i`, which means it can access the value of `i` and can also

change the value of `i`. It does this using the `--` operator to decrease the value of `i` by 1 every time it's called.

We can create a counter by assigning the return value of the `counter()` function to a variable:

```
const count = countdown(3);
```

The variable `count` now points to a function that has full access to the variable `i` that was created in the scope of the `counter()` function. Every time we call the `count()` function, it will return the value of `i` and then decrease it by 1:

```
count();

<< 3

count();
<< 2


count();
<< 1
```

Functional Programming

Functional programming is a style of programming that has become very popular in recent years, partly due to purely functional languages such as Clojure, Scala, and Erlang. JavaScript has always supported functional-style programming, because functions are first-class objects. The techniques we've covered in this chapter—such as using anonymous functions as arguments, returning values to other functions, and creating closures—are all fundamental elements of functional programming.

Pure Functions

A key aspect of functional programming is its use of pure functions. A **pure function** is one that adheres to the following rules:

- The return value of a pure function should only depend on the values provided as arguments; it doesn't rely on values from somewhere else in the program.
- There are no side effects: a pure function doesn't change any values or data elsewhere in the program. It only makes non-destructive data transformations and returns new values, rather than altering any of the underlying data.
- A pure function has referential transparency. Given the same arguments, it will always return the same result.

In order to follow these rules, any pure function must have:

- At least one argument: otherwise, the return value must depend on something other than the arguments of the function, breaking the first rule.
- A return value: otherwise, there's no point in the function (unless it has changed something else in the program, in which case, it has broken the "no side effects" rule).

Pure functions help to make functional programming code more concise and predictable than in other programming styles. Referential transparency makes pure functions easy to test, as they can be relied on to return the same values when the same arguments are provided. Another benefit is that any return values can be saved in memory, since they're always the same, making the function more efficient. The absence of any side effects tends to reduce the amount of bugs that can creep into your code, because there are no surprise dependencies. Pure functions only rely on any values provided as arguments.

Let's take a look at how *not* to write a pure function. The next example shows an impure function that returns the value of adding two values together:

```
let number = 42;

let result = 0;

function impureAdd(x) {
  result = number + x;
}
```

```
impureAdd(10);
result;

<< 52
```

The function `impureAdd()` is an impure function, as it breaks the rules outlined above. It requires the value `number`, which is defined outside the function. It has the side effect of changing the value of `result` (also defined outside the program), and it will return a different value if the value of the variable `number` is different.

Here's an example of a pure function that achieves the same result:

```
const number = 42;


function pureAdd(x,y) {
  return x + y;
}

result = pureAdd(number,10);

<< 52
```

The `pureAdd` function requires two arguments to add together, so the variable `number` has to be passed to it as an argument. This is an example of a non-destructive data transformation, as the value stored in the variable, `number`, remains the same after it has been passed through the function as an argument. There are no side effects to this function; it simply returns the result of adding the two numbers together. This return value is then assigned to the variable `result`, instead of the function updating the value of the variable directly. This function will also always return the same value given the same inputs.

A Benefit of `const`

Using `const` to declare variables will help to avoid destructive data transformations, since any primitive values can't be changed (although variables that are assigned to arrays or objects using `const` can still be mutated, so it's not a complete solution).

Functional programming uses pure functions as the building blocks of a program. The functions perform a series of operations without changing any underlying data in the program. Each function forms an abstraction that should perform a single task, while encapsulating the details of its implementation inside the body of the function. This means that a program becomes a sequence of expressions based on the return values of pure functions. The emphasis is placed on using **function composition** to combine pure functions together to complete more complex tasks.

You can read more about function composition in "Function Composition: Building Blocks for Maintainable Code".

Pure Array Updates

In previous chapters of this book, we used the `push()` method to add values to an array. For example, our to-do list app contains the following `addTask` function:

```
function addTask(item){
  list.push(item);
}
```

A couple of things make this an impure function. It relies on the variable `list`, which exists outside the function body, and the `push()` method is a destructive transformation, meaning that the value of the array is changed.

We can purify this function by adding the array that the item is to be added to as a parameter. That way, the function isn't accessing any variables outside its scope. We can also return a new array that includes all the items from the old array, using the spread operator, and the new item added on the end, like so:

```
function addTask(task,list){
   return [...list,task];
}
```

The `removeTask` function is also impure:

```
function removeTask(task){
  const index = list.indexOf(task);
```

```
  if(index>-1){
    list.splice(i,1);
  }
   return list;

}
```

This is because our code, once again, is referencing the `list` variable from outside the function, and the `splice()` method is mutating the `list` array (that is, it's making a permanent change to it). We can fix this by adding `list` as a parameter and using the `filter()` method instead to remove the item:

```
function removeTask(task,list){
  return list.filter(x => x !== task);
}
```

This will return an array containing all the tasks that aren't equal to the item provided as an argument, effectively removing the task from the array.

To test these out, let's create a short list of tasks in the console:

```
let tasks = ['Bake cake','Read book','Sing song'];
```

Make sure you've defined the new pure functions in the console or in the **JS** section of CodePen, then try entering the following code in the console:

```
function addTask(task,list){
   return [...list,task];
}

function removeTask(task,list){
  return list.filter(x => x !== task);

}
```

Now let's test out the `addTask` function by adding the string `'Learn to code'` to the `tasks` array:

```
addTask('Learn to code',tasks);
<< ["Bake cake","Read book","Sing song","Learn to code"]
```

As you can see, the function returns a new array with the string `'Learn to code'` added to the end. But the function *hasn't* changed the value of the `tasks` variable, as we can see if we check its value:

```
tasks;
<< ["Bake cake","Read book","Sing song"]
```

Now let's check that the `removeTask` function works:

```
removeTask('Read book',tasks);
<< ["Bake cake","Sing song"]
```

This has returned a new array that doesn't contain the string `'Read book'`. Once again, the original `tasks` variable hasn't been changed:

```
tasks;
<< ["Bake cake","Read book","Sing song"]
```

Updating an Array Value with Pure Functions

If you actually did want to update the value of the `tasks` array, you could still use pure functions to do this, by assigning the `tasks` variable to the return value of the function call.

For example, if you wanted to update the `tasks` array with the string `'Sing song'` removed, you could use the following code:

```
tasks = removeTask('Sing song',tasks);
<< ["Bake cake","Read book"]
```

We could then confirm that this has indeed updated the `tasks` array:

```
tasks;
<< ["Bake cake","Read book"]
```

Higher-order Functions

Higher-order functions are functions that accept another function as an argument, or return another function as a result, or both.

Closures are used extensively in higher-order functions, as they allow us to create a generic function that can be used to then return more specific functions based on its arguments. This is done by creating a closure around a function's arguments that keeps them "alive" in a return function. For example, consider the following `multiplier` function:

```
function multiplier(x){
  return function(y){
    return x*y;
  }
}
```

The `multiplier` function returns another function that traps the value of the argument `x` in a closure. This is then available to be used by the returned function.

We can now use this generic `multiplier` function to create more specific functions, as can be seen in the example below:

```
doubler = multiplier(2);
```

This creates a new function called `doubler` that multiplies an argument by two:

```
doubler(10);
<< 20
```

The `multiplier()` function is an example of a higher-order function. This means that we can use it to build other, more specific functions by using different arguments. For example, an argument of 3 can be used to create a `tripler()` function that multiplies its arguments by 3:

```
tripler = multiplier(3);

tripler(10);
<< 30
```

This is one of the core tenets of functional programming: it allows generic, higher-order functions to be used to return more specific functions based on specific parameters.

A neat trick to use with higher-order functions is to chain arguments together. The following example will multiply 3 and 5 together:

```
multiplier(3)(5);
<< 15
```

This works because `multiplier(3)` returns an anonymous function and we immediately call it with an argument of `5` by adding the parentheses on the end.

Challenges

1. Write a recursive function that sings the Ten Green Bottles song that we programmed using various loops in Chapter 7. It should take the number of bottles as a parameter, sing one verse, then call itself to sing it again. For example, `greenBottles(10)` would sing the song, starting at 10 green bottles. You can see my code on CodePen.

2. Create a higher-order function called `exponent` that returns another function that will calculate numbers to the power of the argument provided. For example, `exponentBase2 = exponent(2)` should result in a function called `exponentBase2` that will return 2 to the power of the argument provided, so `exponentBase2(3)` will return 8. You should also be able to write `exponent(2)(3)` and also get 8. You can see my code on CodePen.

3. Update the to-do list app so that the `add` and `remove` functions are pure functions. You can see my code on CodePen.

Summary

- Named parameters can be reproduced in JavaScript using an object as the parameter. This allows the arguments to be provided as properties of the object in any order.
- The rest parameter allows any number of values to be provided to a function as an array.

- Recursive functions are functions that continually call themselves until a certain condition is reached.
- The scope of a variable determines where and how that variable can be accessed.
- Functions declared using the `function` keyword are *hoisted* to the top of the scope, meaning they can be called even before they've been defined.
- Variables should be declared before they're referred to.
- A closure is formed when a function accesses a variable that has been declared in the surrounding scope of the function. By returning the function, we can maintain access to the variable outside the scope it was declared in.
- Functional programming is a style of programming that combines pure functions that perform a single task in order to perform more complex tasks.
- Pure functions shouldn't reference anything outside the scope of the function that has not been supplied as an argument, and they shouldn't cause side effects by changing anything outside the scope of the function. They should have *referential transparency*, which means that, given the same arguments, they should return the same value.
- Higher-order functions accept functions as arguments and return functions. They can be chained together to complete more complex operations.

In the next chapter, we'll be taking a deeper look into objects and object-oriented programming.

Chapter 14: Getting Classy

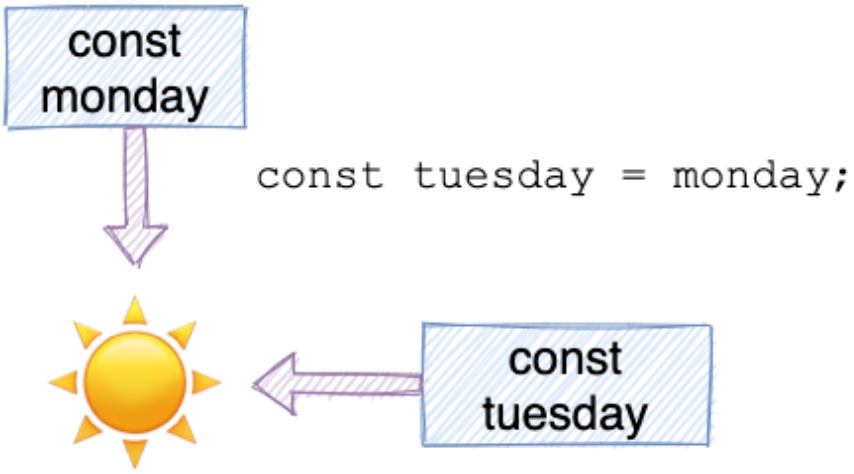
We first encountered objects in Chapter 9. In this chapter, we're going to take a more in-depth look at object-oriented programming and how you can use JavaScript to create classes. We'll be covering the following topics:

- copying objects in JavaScript
- object-oriented programming
- encapsulation
- polymorphism
- inheritance
- classes in JavaScript
- extends
- a Pet Unicorn project

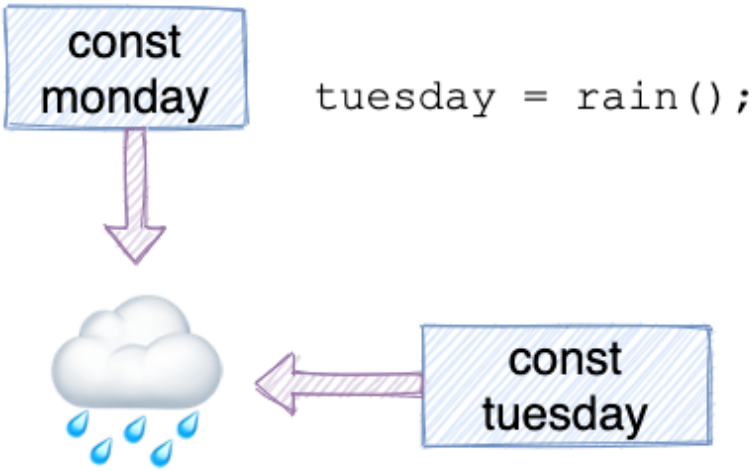
Copying Objects in JavaScript

An important concept in JavaScript is that objects are assigned by **reference**. This means that, if two variables are assigned to the same object, both variables will point to the same object in memory.

For example, imagine you're creating a weather app and the variable `monday` is pointing to a sunny image object. If you assign the variable `tuesday` to the same image object using the code `const tuesday = monday`, both variables will be pointing to the same image object, as can be seen in the diagram below.



If you then update the image object that `tuesday` is pointing to, it will also change the image object that `monday` is pointing to, as can be seen in the diagram below, where `tuesday` is changed to a rainy image object.



This means that any changes you make to *either* variable will affect the other, making it impossible to change `monday` without changing `tuesday`.

To demonstrate the problems this can cause, let's create some objects to model the three bears from the Goldilocks story.

Console Issues

At the time of writing, jsonsole.com doesn't support some of the code that's used in this example. For this reason, I'd recommend using the console on CodePen instead. (It's located at the bottom left of the CodePen interface.)

First of all, let's create an object that describes Papa Bear and assign it to the variable `papa` in the console:

```
const papa = {
  name: 'Papa Bear',
  type: 'bear',
  color: 'brown',
  food: 'porridge',
  size: 'large'
};
```

Now, if we wanted to create another object to represent Mama Bear, we could start with a copy of the `papa` object, since a number of the properties are the same. We can then update the properties that need changing.

A common mistake is to think that this copy of the `papa` object can be made like so:

```
const mama = papa;
```

The variable `mama` will have all of the same properties as the `papa` object. The problem is that we haven't actually copied the `papa` object; the variables `mama` and `papa` both reference exactly the same object!

We can see this if we try to make a change to the `name` property of `mama`:

```
mama.name = 'Mama Bear';
```

Now if we check the value of the name property of the `papa` object, we'll discover a problem:

```
papa.name;
<< 'Mama Bear'
```

Changing the `name` property of `mama` has resulted in the `name` property of `papa` changing as well. This happens because the variables `mama` and `papa`

both point to the same object in memory. Any changes made to either variable will affect the other.

The solution to this problem is to apply the spread operator to the object we want to copy. You'll need to restart the console and define the `papa` object again before entering the following code to make a copy of the `papa` object:

```
const mama = {...papa}
```

This creates a brand new object literal and spreads the properties of the `papa` object out inside it. This means that the variable `mama` is pointing to a brand new object in memory. Any changes made to it will not affect the `papa` object, as we can see if we update the `name` and `size` properties:

```
mama.name = 'Mama Bear';
<< "Mama Bear"

mama.size = 'medium';
<< "medium"
```

Now if we take a look at the two objects, we can see that the `mama` object has been updated, but the `papa` object hasn't changed:

```
papa;
<< {
  name: 'Papa Bear',
  type: 'bear',
  color: 'brown',
  food: 'porridge',
  size: 'large'
}

mama;
<<
{
  name: 'Mama Bear'
  type: 'bear',
  color: 'brown',
  food: 'porridge',
  size: 'medium'
}
```

Shallow and Deep Copies

The code above makes a **shallow** copy of the `papa` object. This means that only the first level of properties is copied. If any of the properties contained nested objects, these objects would still be copied by reference. A **deep** copy involves making a copy of *every* property, including nested objects.

Making a deep copy of an object can get tricky, and I'd recommend using a well-tested solution, such as the `_.cloneDeep` method used by the Lodash library.

Copying an object and updating some properties can be accomplished in one step by adding the new property at the end of the object literal. Let's do this by making another copy of the `papa` object to represent Baby Bear:

```
const baby = {...papa, name: 'Baby Bear', size: 'small'};
```

In this example, we've copied the `papa` object, then updated the `name` and `size` properties inside the same object. This works because any properties with the same name will overwrite any previously defined properties.

We can check that this has worked by taking a look at the `baby` object:

```
baby;
<< {
  name: 'Baby Bear'
  type: 'bear',
  color: 'brown',
  food: 'porridge',
  size: 'small'
}
```

Just right!

Copying Arrays and Functions

In JavaScript, arrays are a special type of object, so the same problems occur when using arrays: they're copied by reference, and using the spread operator only makes a shallow copy.

Functions are also only copied by reference, so any changes made will affect all references to the function.

Object-oriented Programming

Object-oriented programming (OOP for short) is a style of programming that encapsulates related pieces of code in objects that maintain state throughout the life of the program. The objects can then be reused or easily modified as required. Many modern languages such as Java, C++, Ruby and Python are object oriented.

Three key concepts in OOP are:

- encapsulation
- polymorphism
- inheritance

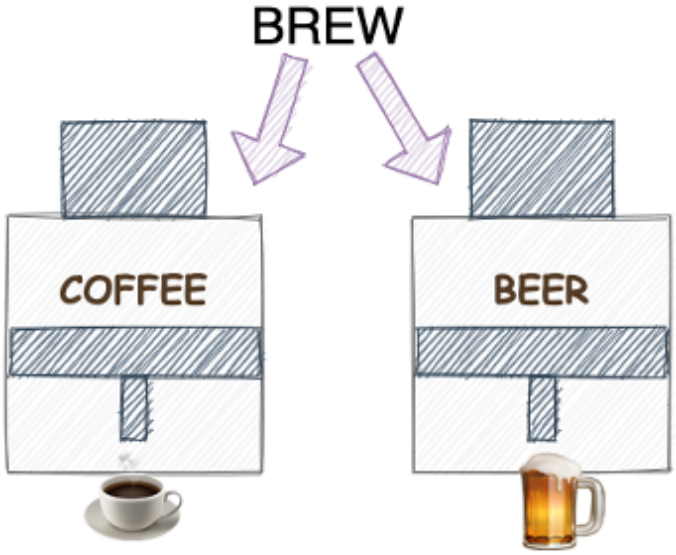
I'm going to use the example of a coffee machine to illustrate how each of these concepts can be applied in a programming environment. In many ways, a coffee machine can be thought of as an object, since it has properties such as speed, strength, and capacity, and it also has methods or actions it can perform, such as brewing, switching on, and switching off.

Encapsulation

When the coffee machine is turned on, it makes some noises and then, a few minutes later, it produces a steaming cup of hot coffee. You don't need to know how the machine works in order to make the perfect cup of coffee; you just press **on**. This demonstrates the concept of **encapsulation**: the inner workings are kept hidden inside the object and only the essential functionalities are exposed to the end user, such as the **on** button. In OOP, this involves keeping all the programming logic inside an object and making methods available to implement the functionality, without the outside world needing to know *how* it's done.

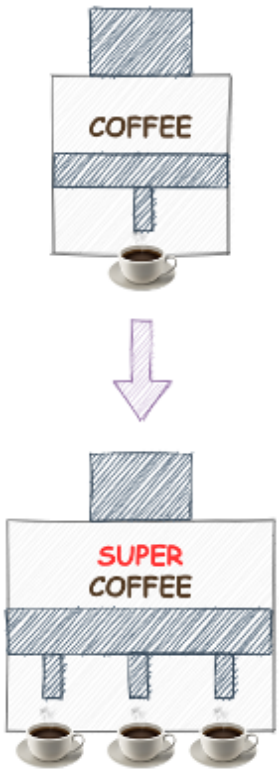
Polymorphism

The **on** button starts a process of brewing in the coffee machine. If there was such thing as a beer machine, it would also have an **on** button that started the process of brewing. Even though both machines have a process called "brewing", the results of that process are very different: one produces coffee and the other beer. This demonstrates the concept of **polymorphism**, where the same process produces different results in different objects. In OOP, this means that various objects can share the same method, but implement them in different ways.



Inheritance

Imagine a super coffee machine that's a new, improved model capable of making three cups of coffee at once. Even though it has some extra features, it still uses many of the same parts as the original coffee machine. This demonstrates the concept of **inheritance**, where the features of one object are taken and then new features are added. In OOP, this means that we can take an object that already exists and inherit all its properties and methods. We can then improve on its functionality by adding new properties and methods.



Classes

Most object-oriented languages, such as Java and Ruby, use classes to define a blueprint for an object. Objects are then created as an instance of that class and inherit all the properties and methods of the class. In the coffee machine example, the `CoffeeMachine` class would represent the design, and each machine that's made on the production line, as well as any other models of coffee machine, would be instances of that class.

Classes vs Prototypes

JavaScript didn't originally have classes built into the language. But it's always allowed the use of existing object literals, rather than classes, as the blueprint for creating similar objects. It's therefore known as a **prototype-based** language. In the coffee machine example, this might involve building an actual prototype machine and then using this prototype as the basis for making all the other machines. JavaScript does now support classes, as we'll see in the next section, but it still uses the same prototypal inheritance model in the background.

Classes in JavaScript

Back in the Objects chapter (Chapter 9), we created the `dice` object shown below:

```
const dice = {
  sides: 6,
  roll() {
    return Math.ceil(Math.random()*this.sides)
  }
}
```

Imagine you're creating an online role-playing game, not unlike Dungeons and Dragons. It will require lots of dice with different numbers of sides to be used. Instead of creating a separate object for each dice, we can create a `Dice` class that can be used to create many copies of this object.

To create this class, enter the following code into the console:

```
class Dice {
  constructor(sides=6) {
    this.sides = sides;
  }

  roll() {
    return Math.ceil(Math.random()*this.sides);
  }
}
```

Class Naming Convention

By convention, the names of class declarations are usually capitalized in class-based programming languages.

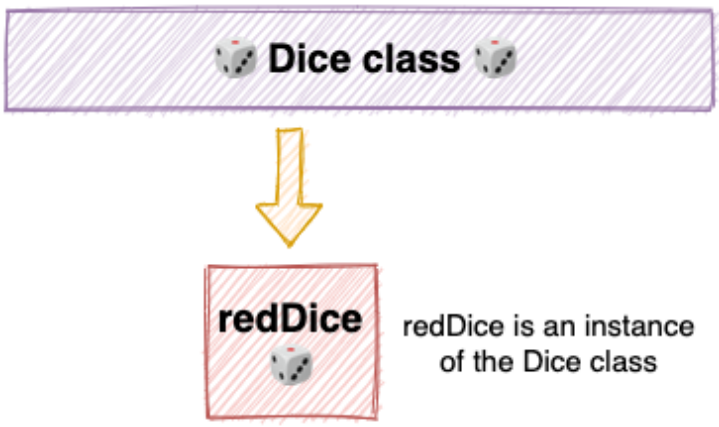
All JavaScript classes have a function called `constructor`. (If you don't explicitly write it, it will be created in the background.) This function is called every time a new instance of the class is created.

The keyword `this` represents the object that will be returned by the class. In the `Dice` class above, we use it to make the `sides` property equal the argument that's provided to the constructor function, or 6, if no argument is provided. It also adds a method called `roll()` that returns a random number from 1 up to the number of sides the dice has.

We can now create an *instance* of the Dice class using the `new` operator:

```
redDice = new Dice();
<< Dice {sides: 4}
```

This calls the `constructor` function defined in the `Dice` class and returns an object that was assigned to the variable `redDice`, which is said to be an *instance* of the `Dice` class. This is an object with a `sides` property and `roll()` method.



We can confirm this using the `instanceof` operator:

```
redDice instanceof Dice;
<< true
```

Each new object that's created in this way will have a `sides` property and `roll()` method from the class definition, and they're known as instances of the `Dice` class. For example, we can create a four-sided blue dice using the following code:

```
blueDice = new Dice(4);
<< Dice {sides: 4}
```

Parentheses

The parentheses aren't required when instantiating a new object using the `new` operator. The following code would also achieve the same result:

```
greenDice = new Dice;
<< Dice {sides: 6}
```

The parentheses are required, however, if any arguments are provided. For example, if we want to create another `Dice` object with 20 sides, we would have to add 20 as an argument, like so:

```
whiteDice = new Dice(20);
<< Dice {sides: 20}
```

You can read more about ES6 Classes in "Object-oriented JavaScript: A Deep Dive into ES6 Classes".

Classy Components

One use of classes is to create reusable HTML components. This allows you to define a block of HTML in a single class and then reuse it multiple times in a program.

For example, let's create a class called `Notice` and use it to create a `<div>` element that displays a message on a web page.

Open up a new Pen on CodePen and enter the following code in the **JS** section:

```
class Notice {
  constructor(message='Hello, World!') {
    this.element = document.createElement('div');
     this.element.textContent = message;
     this.css = 'background:silver;border:3px gray
solid;color:gray;font:18px sans-serif;padding:8px;margin:10px';
     this.element.style.cssText = this.css;
  }

  render(element) {
     element.appendChild(this.element);
  }
}
```

The `constructor` function of this class creates a `<div>` element and then sets the `textContent` property to be the same as `message`, which is provided as an argument when creating a new instance. It also sets the CSS properties using the `style.cssText` property that allows you to set multiple CSS properties in a single string.

Each instance of the `Notice` class also has a `render()` method that's used to render the element on the page. This uses the `appendChild()` method that we saw in the DOM chapter (Chapter 10) to append the element to an element that's provided as an argument to the method.

Let's try testing out this class to create a `Notice` component. Add the following line of code to the bottom of the **JS** section:

```
welcome = new Notice;
```

This creates a reference to the `<div>` element that's returned by the `constructor` function. We can now insert this into the page using its render method. Add the following line of code to the bottom of the **JS** section:

```
welcome.render(document.body);
```

This will append the element to the body of the document, and should look something like this:

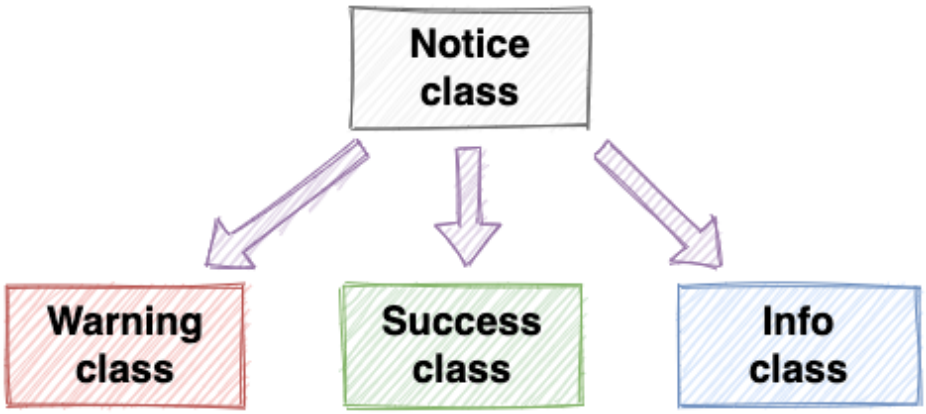
Hello, World!

You can see my code on CodePen.

Inheritance in JavaScript

Now that we have our class for creating notice components, we can create some different types of notices based on this class, but more specific. For example, we could create `Warning`, `Success` and `Info` notices that are all based on the generic notice element we created in the previous section.

We can use the concept of inheritance to ensure that our new notices inherit all the properties and methods of the `Notice` class. This means that the `Warning`, `Success` and `Notice` classes will inherit all their properties and methods from the parent `Notice` class, as can be seen in the diagram below.



In JavaScript, a class inherits from another class using the `extends` keyword. Add the following code to the bottom of **JS** section in your Pen to see how this works:

```
class Warning extends Notice {
 constructor(message='WARNING!') {
    super(message);
  }
}
```

This creates a new class called `Warning` that inherits all the properties and methods of the `Notice` class. The `super` keyword references the parent class (`Notice` in this example) and is used to access the properties and methods of the parent class.

We don't want the `Warning` class to be exactly the same as the `Notice` class, though. (What would be the point in that?) We need to update some of the style properties to change how it looks. Update the class definition to include the following code:

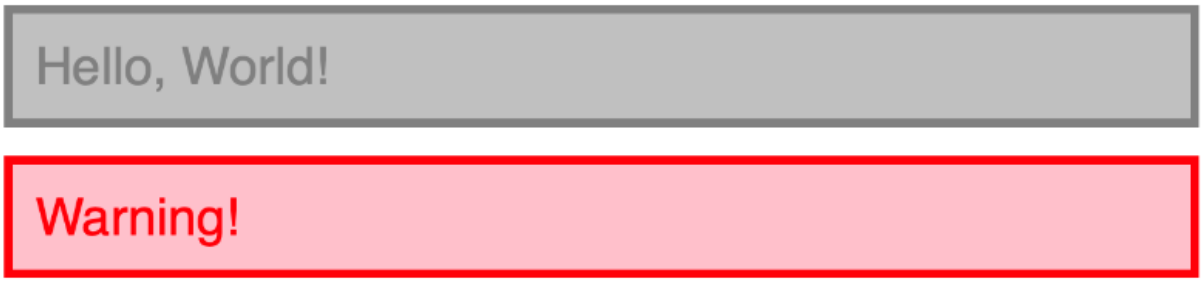
```
class Warning extends Notice {
 constructor(message='WARNING!') {
    super(message);
    this.element.style.background = 'pink';
    this.element.style.color = 'red';
    this.element.style.borderColor = 'red';
  }
}
```

These extra lines of code update the style properties to change the colors of the background, border and text.

To check that this has worked, let's create an instance of the `Warning` class and insert it into the document:

```
const warning = new Warning('Warning!');
warning.render(document.body);
```

This should append the warning notice to the body of the page, placing it underneath the notice that's already there, as can be seen in the image below.



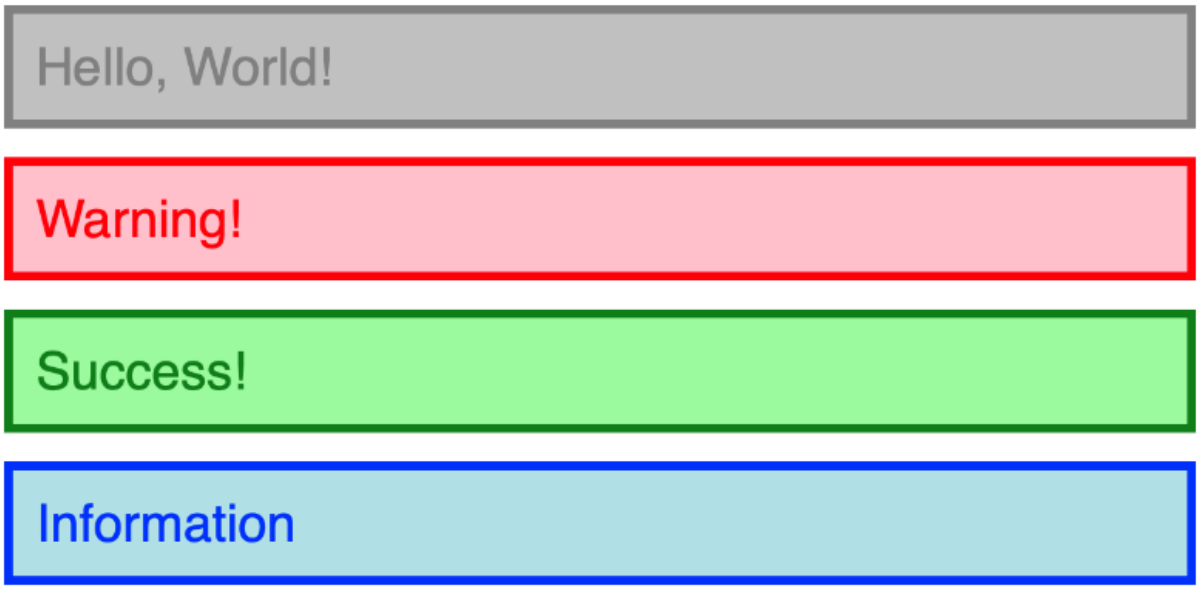
We can use a similar process to create two more classes for success and info notices by adding the following code:

```
class Success extends Notice {
 constructor(message='Success!') {
    super(message);
    this.element.style.background = 'palegreen';
    this.element.style.color = 'green';
    this.element.style.borderColor = 'green';
  }
}

class Info extends Notice {
 constructor(message='Information') {
    super(message);
    this.element.style.background = 'powderblue';
    this.element.style.color = 'blue';
    this.element.style.borderColor = 'blue';
  }
}
```

These classes extend the `Notice` class in the same way that the `Warning` class did, but they use different colors and default messages. We can test these out and add them to the page using the following code:

```
success = new Success;
success.render(document.body);

info = new Info;
info.render(document.body);
```

The page should now look similar to the screenshot below.



You can see my code on CodePen.

Custom HTML Elements

It's possible to go a step further than we did in this example and create your own custom HTML elements.

The Pet Unicorn Game

We're going to finish this chapter by creating a virtual pet game, based on the classic Tamagotchi toys. This game will involve the player looking after a pet unicorn. The player gets to name their pet unicorn and then needs to decide when to feed it, play with it, and put it to bed.

The game will use a `Unicorn` class to instantiate an object that represents a pet unicorn. The object will have properties that keep track of how the unicorn is feeling, and methods that allow the player to interact with their pet unicorn. This game will run entirely in the console by entering the methods manually.

The `Unicorn` class will need the following properties:

- `name`: for referring to the unicorn

- `food`: for keeping track of how much food the unicorn has in its belly
- `fun`: for recording how much fun the unicorn is having
- `energy`: for keeping track of how much energy the unicorn has

The `Unicorn` class will also need the following methods:

- `eat`: this will increase the value of `food`, since eating will put food in the unicorn's belly, but it will decrease the value of `fun`, since standing around eating can be a bit dull.
- `sleep`: this will increase the value of `energy`, since sleeping provides the unicorn with some rest, but the value of `food` will decrease as the food is digested. It will also decrease the value of `fun`, since sleeping can be a bit boring.
- `play`: this will increase the value of `fun`, because playing is fun, but it will decrease the value of `energy`, because it can be hard work!

Let's create the constructor function that will be used to instantiate a new instance of the `Unicorn` class. It should define the properties outlined above. Open a new Pen on CodePen and add the following class definition to the **JS** section:

```
class Unicorn {
  constructor(name='Sparkles'){
    this.name = name;
    this.food = 3;
    this.fun = 3;
    this.energy = 5;
    console.log(`Your new pet unicorn, ${this.name}, is born!`);
  }
}
```

This defines the constructor function that will be called when a new `Unicorn` object is created. The function has a `name` parameter that has a default value of `'Sparkles'`. This is used to set the `name` property. We also set the `food`, `fun` and `energy` properties to their initial values of 3, 3 and 5 respectively. A message is also logged to the console to inform the player that their pet unicorn has been born.

Next, we need to define the `eat`, `play()` and `sleep()` methods. These will need to update the relevant properties and also provide some feedback in the

console to let the player know that something has happened. We won't give details about how the actual values of the properties have changed, though, as these are encapsulated inside the object (and we don't want to give away the internal workings of the game too much!).

Let's start with the `eat()` method. This should be placed right after the `constructor` function, but still inside the class block:

```
eat(){
   console.log(`${this.name} gobbles up some glitter.`);
    this.food += 3;
    this.fun -= 1;
    this.timeGoesBy();
  }
```

This logs a message to the console to tell the user that their pet has eaten something. It then increases the `food` property by 3 to indicate that some food has been eaten and decreases the `fun` property by 1 because any time spent eating means less time playing. At the end of the method, we call another method called `timeGoesBy`. This method will simulate time passing in the game and check to see if any of the unicorn's property values have dropped too low. We'll write this method soon, but first we have to add the `play()` and `sleep()` methods. Add these underneath the `eat()` method:

```
play(){
    console.log(`${this.name} frolics in the meadow.`);
    this.fun +=2;
    this.energy -=2;
    this.timeGoesBy();
  }

sleep(){
    console.log(`${this.name} falls asleep, dreaming of stars &
rainbows.`);
    this.energy += 5;
    this.fun -= 2;
    this.food -= 3;
    this.timeGoesBy();
  }
```

These both follow a similar pattern to the `eat()` method: log a message to the console, update the properties, then call the `timeGoesBy()` method. Let's add that method now:

```
timeGoesBy(){
    if(this.energy < 0){
     this.dies('exhaustion');
    }
     if(this.food < 0){
        this.dies('starvation');
     }
    if(this.fun < 0){
      this.dies('boredom');
    }
  }
```

This method uses an `if` statement to check if any of the unicorn's properties of `energy`, `food` and `fun` have dropped below zero. If this is the case, the unicorn will unfortunately die. When this happens, we call the `dies()` method, passing an argument that describes the reason why the unicorn died. We need to write this method next:

```
dies(reason){
  console.log(`${this.name} died of ${reason}.`);
}
```

This method accepts an argument that's a string describing the reason the unicorn died. It then logs a message to the console to tell the user that their pet unicorn has died, along with the reason why.

Try creating a new pet unicorn in the console on CodePen. Try calling the different methods and see how long you can keep your unicorn alive. The console log below shows the output of my feeble attempt:

```
pet = new Unicorn;
<< "Your new pet unicorn, Sparkles, is born!"

pet.play();
<< "Sparkles frolics in the meadow."

pet.play();
<< "Sparkles frolics in the meadow."

pet.eat();
<< "Sparkles gobbles up some glitter."

pet.play();
```

```
<< "Sparkles frolics in the meadow."
"Sparkles died of exhaustion."
```

The moral of the story is to remember to give your pet some rest!

You can see my code on CodePen.

Challenges

1. Add a random element to the Pet Unicorn game that changes the properties by slightly different amounts every time a method is called. For example, instead of the `sleep()` method increasing energy by 5, it could increase it by 4, 5 or 6. You can see my code on CodePen.

2. Add a graphical interface to the Pet Unicorn game. This could involve the following features: an input box that allows the user to enter the name of a unicorn to create it; a picture of a unicorn with its name that appears in the document when the object is instantiated updates about how the pet is feeling in the document, instead of using `console.log`; and buttons that are pressed to perform each action of `eat`, `sleep` and `play`.

   You can see my code on CodePen.

Summary

- Objects are copied by reference in JavaScript. To make a hard copy of an object, you need to use the spread operator.
- Object-oriented programming (OOP) is a programming concept that involves keeping code in objects.
- Encapsulation means that the inner workings of an object are kept away from the users.
- Polymorphism means that objects can share the same methods, but implement them in different ways.
- Inheritance means that a class can inherit all the properties and methods from a parent class and then add their own, more specific properties and

methods, or change some of the properties and methods from those of the parent class.
- You can create classes in JavaScript using the `class` keyword. The class describes the properties and methods that each instance of that class will have.
- The `constructor` function defined in the class is called whenever a new instance of a class is created using the `new` keyword.

In the next chapter, we'll be learning about how to code with time and dates.

Chapter 15: It's About Time

Working with times and dates can be tricky in programming languages. For this reason, most languages have some sort of built-in `Date` object that helps to make the process easier. JavaScript is no exception, and we'll be taking a look at how that works in this chapter. We'll be covering the following:

- the UNIX epoch
- times and dates
- timing functions
- asynchronous programming
- animation
- a Cookie Grabber game

The UNIX Epoch

The UNIX epoch is an arbitrary date of January 1, 1970, which is used in programming as a reference point in time from which to measure dates. This allows dates to be expressed as an integer that represents the number of seconds since the epoch. As you can imagine, this produces some very large numbers, and there's a potential problem looming in 2038 when the number of seconds since the epoch will be greater than 2,147,483,647, which is the maximum value that some programming languages can deal with. Fortunately, JavaScript will be fine, as it can handle bigger values than this.

Times and Dates

Date objects contain information about dates and times. Each object represents a single moment in time.

In JavaScript, we can use a constructor function to create a new date object using the `new` operator. Try entering the following code in the console:

```
const today = new Date();
```

The variable `today` now points to a `Date` object. To see what the date is, we can use the `toString()` method that all objects have:

```
today.toString();
<< 'Mon Dec 07 2020 17:40:51 GMT+0000 (GMT)'
```

If an argument isn't supplied, the date will default to the current date and time based on the system clock and time zone setting. This means that it can be manipulated and shouldn't be relied on as an accurate representation of the actual date.

It's possible to create `Date` objects for any date by supplying it as an argument to the constructor function. This can be written as a string in a variety of forms, as can be seen in the examples of different holiday dates below:

```
const christmas = new Date('2021-12-25');
// date format is YYYY-MM-DD
christmas.toString();
<< "Sat Dec 25 2021 00:00:00 GMT+0000 (GMT)"

const chanukah = new Date('28 November 2021');
// First day of Chanukah
chanukah.toString();
<< "Sun Nov 28 2021 00:00:00 GMT+0000 (GMT)"

const eid = new Date('Wednesday, May 12, 2021');
// Eid-al-Fitr
eid.toString();
<< "Wed May 12 2021 00:00:00 GMT+0100 (BST)"
```

As you can see, the string passed to the `Date` constructor can be in a variety of formats. However, in order to be more consistent, I'd recommend that you provide each part of the date as a separate argument. The parameters that can be provided are as follows:

```
new Date(year,month,day,hour,minutes,seconds,milliseconds);
```

Here's an example:

```
const halloween = new Date(2021, 9, 31);
halloween.toString();
<< "Sun Oct 31 2021 00:00:00 GMT+0100 (BST)"
```

Counting the Months

Slightly confusingly, the numerical value for months starts at zero, so January is 0, February is 1, and so on up to December, which is 11.

An alternative is to use a **timestamp**, which is a single integer argument that represents the number of milliseconds since the epoch (January 1, 1970):

```
const diwali = new Date(1635984000000);
diwali.toString();
<< "Thu Nov 04 2021 00:00:00 GMT+0000 (GMT)"
```

Getter Methods

The properties of date objects aren't able to be viewed or changed directly by assignment. Instead, they have a number of methods, known as **getter methods**, that return information about the date object, such as the month and year.

Once you've created a date object, it will have access to all the getter methods. There are two versions of most methods: one that returns the information in local time, and the other that uses Coordinated Universal Time (UTC). The `getTime()`, `getTimezoneOffset()` and `getYear()` methods don't have UTC equivalents.

UTC vs GMT

UTC is the primary time standard by which the world regulates clocks. It was formalized in 1960 and is much the same as Greenwich Mean Time (GMT). The main difference is that UTC is a standard that's defined by the scientific community, unlike GMT. (You can read about why it's called "UCT" and not "CUT" on Wikipedia.)

The `getDay()` and `getUTCDay()` methods are used to find the day of the week that the date falls on. They return a number, starting at 0 for Sunday, up to 6 for Saturday. The code below shows us that Diwali is on a Thursday (day 4) in 2021:

```
diwali.getDay();
<< 4
```

The `getDate()` and `getUTCDate()` methods return the day of the month for the date object. (Note that these values start counting from 1, not 0, so they return the actual day of the month.) The code below shows us that Diwali is on the 4th in 2021:

```
diwali.getDate();
<< 4
```

The `getMonth()` and `getUTCMonth()` methods can be used to find the month of the date object. It returns an integer, but remember to count from 0! The code below shows us that Diwali is in November (month 10) in 2021:

```
diwali.getMonth();
<< 10
```

The `getFullYear()` and `getUTCFullYear()` methods return the year of the date object. There's also a `getYear()` method, but it isn't Y2K compliant, so shouldn't be used, as it returns nonsensical results for dates after the year 2000, as can be seen in the code below:

```
diwali.getYear();
<< 121
```

This can be fixed if we use the `getFullYear()` method instead:

```
diwali.getFullYear();
<< 2021
```

There are also `getHours()`, `getUTCHours()`, `getMinutes()`, `getUTCMinutes()`, `getSeconds()`, `getUTCSeconds`, `getMilliseconds()`, and `getUTCMilliseconds()` methods that will return the hours, minutes, seconds and milliseconds since midnight.

The `getTime()` method returns a timestamp representing the number of milliseconds since the epoch:

```
diwali.getTime();
<< 1635984000000
```

This can be useful for incrementing dates by a set amount of time. For example, a day can be represented by `1000 * 60 * 60 * 24` milliseconds. The following example uses this to subtract a day from the date of Christmas in order to find the date of Christmas Eve:

```
const christmasEve = new Date(christmas.getTime() - 1000 * 60 *
60 * 24);
christmasEve.toString();
<< "Fri Dec 24 2021 00:00:00 GMT+0000 (GMT)"
```

The `getTimezoneOffset()` method returns the difference, in minutes, between the local time on the computer and UTC. For example, my time zone is currently the same as UTC, so it returns 0:

```
new Date().getTimezoneOffset();
<< 0
```

You can see more examples of `Date` get methods on the W3Schools site.

Setter Methods

Most of the getter methods covered in the previous section have equivalent setter methods. These are methods that can be used to change the value of the date held in a `Date` object. Each of the methods takes an argument representing the value to which you update the date. The methods return the timestamp of the updated date object.

As an example, we can change the value of the date stored in the `diwali` variable so that it contains the date of Diwali in 2022, which is on Monday, October 24, 2022:

```
diwali.setDate(24);
<< 1637712000000

diwali.setMonth(9);
<< 1635030000000

diwali.setFullYear(2022);
<< 1666566000000
```

Note that the values returned by these functions is the timestamp representing the number of milliseconds since the epoch. To see the actual date, we need to use the `toString()` method:

```
diwali.toString();
<< "Mon Oct 24 2022 00:00:00 GMT+0100 (BST)"
```

There are also `setHours()`, `setUTCHours()`, `setMinutes()`, `setUTCMinutes()`, `setSeconds()`, `setUTCSeconds`, `setMilliseconds()` and `setUTCMilliseconds()` methods that can be used to edit the time portion of a `Date` object.

Alternatively, if you know the date as a timestamp, you can use the `setTime()` method:

```
diwali.setTime(1666566000000);
<< 1666566000000
```

Time Zone Milliseconds

The number of milliseconds required in the timestamp might be slightly different for you, depending on your time zone.

You can see more examples of `Date` set methods on the W3Schools site.

Date and Time Support

Working with dates and time zones can be tricky. But there will soon be a new `Temporal` object that aims to fix some of the issues with the `Date` object. (You can learn more about it in "An Introduction to the JavaScript Temporal API".)

If you're writing code that uses lots of times and dates, it might be worth using an external library such as date-fns, which provides a number of methods to make it easier to work with dates. You can read more about using it in "Learn date-fns: A Lightweight JavaScript Date Library".

What Day Will It Be?

Now that we've learned all about how to work with the `Date` object, let's build a small application to tell you what day it will be in a set number of days.

Open up a new Pen on CodePen and add the following HTML:

```
<form name='myForm'>
  <input type='number' name='number' value='1'>
  <button type='submit'>Submit</button>
</form>
<div id='output'></div>
```

This is a form that asks users to enter a number of days. We'll tell them what day it will be after that number of days. Note that, even though the `<input>` element has a `type` attribute of `number` (which is good practice), it will still be submitted as a string.

Now let's add some JavaScript to work out what day it will be. First of all, we'll declare some variables that refer to the `<form>` element and the `<div>` that will contain the output. Add the following code to the **JS** section:

```
const form = document.forms.myForm;
const output = document.getElementById('output');
```

We also need to add an array that contains the names of each day of the week. This will match the name of a day to the corresponding number returned by the `getDay()` method, so `dayNames[0]` will be `'Sunday'`, `dayNames[1]` will be `'Monday'`, and so on. Add the following line of code to the **JS** section:

```
const dayNames =
['Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','Sa
turday'];
```

Next, we need to write a function that will calculate and display the day of the week. Add the following function to the end of the **JS** section:

```
function nameThatDay(e){
  e.preventDefault();
  const numberOfDays = Number(form.number.value);
  const milliseconds = numberOfDays * 24 * 60 * 60 * 1000;
  const timestamp = Date.now() + milliseconds;
```

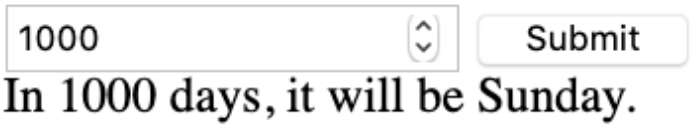
```
  const day = new Date(timestamp).getDay();
  output.textContent = `In ${numberOfDays} days, it will be
${dayNames[day]}.`
}
```

This function starts by preventing the default behavior of the form, so it won't be submitted to a server. It then takes the value that was entered in the form and converts it to a number, stored in the variable `numberOfDays`. This value is converted into milliseconds by multiplying it by the number of milliseconds in a single day. We then add this to the number of milliseconds since the epoch to produce a timestamp that can be used as an argument to the `new Date()` constructor function to create a new `Date` object. We can then call the `getDay()` method to return which day of the week this date is on. We then use the `dayNames` array as a lookup table to find the corresponding day name string using this value as an index.

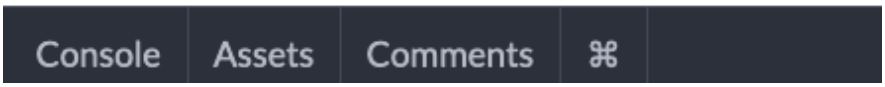
Finally, we need to add an event listener that will call the `nameThatDay` function when the form is submitted. Add the following line of code to the end of the **JS** section:

```
form.addEventListener('submit',nameThatDay);
```

Now that everything is in place, try using the form to see what day it will be in a thousand days (it should be the same day as it was yesterday).



In 1000 days, it will be Sunday.



You can see my code on CodePen.

Timing Functions

JavaScript provides a couple of useful methods that enable us to schedule when a function is called, or to call a function at regular intervals. These can be useful for causing a delay before something happens, or displaying a timer on the screen.

setTimeout

The `setTimeout()` method accepts a callback function as its first parameter and a number of milliseconds as its second parameter. The callback function that's provided as the first argument will be called after the time given as the second argument.

To see this in action, try entering the following code into a console. It should show an alert box after seven seconds. This is because the first argument is an anonymous arrow function that displays the alert box, and the second argument is 7000 milliseconds, or 7 seconds:

```
setTimeout( () => alert("Time's Up!"), 7000);
<< 1
```

Notice that the method returns an integer. This is an ID used to reference that particular timeout. It can also cancel the timeout using the `clearTimeout()` method. Try calling the code again, making a note of the number that's returned:

```
setTimeout( () => alert("Time's Up!"), 7000);
<< 2
```

Now quickly enter the following code before the alert pops up, making sure you enter the number that was returned previously (it might not be 2 in your case!):

```
clearTimeout(2);
<< undefined
```

If you're quick enough, and use the correct ID, the callback will be canceled and the alert will never appear.

Instead of trying to remember the value that's returned when the timeout is set, it's easier to assign a variable to the return value, like in the code

example below:

```
const timer = setTimeout( () => alert("Time's Up!"), 7000);
<< 3
```

This variable can then be used to clear the timeout later:

```
clearTimeout(timer);
<< undefined
```

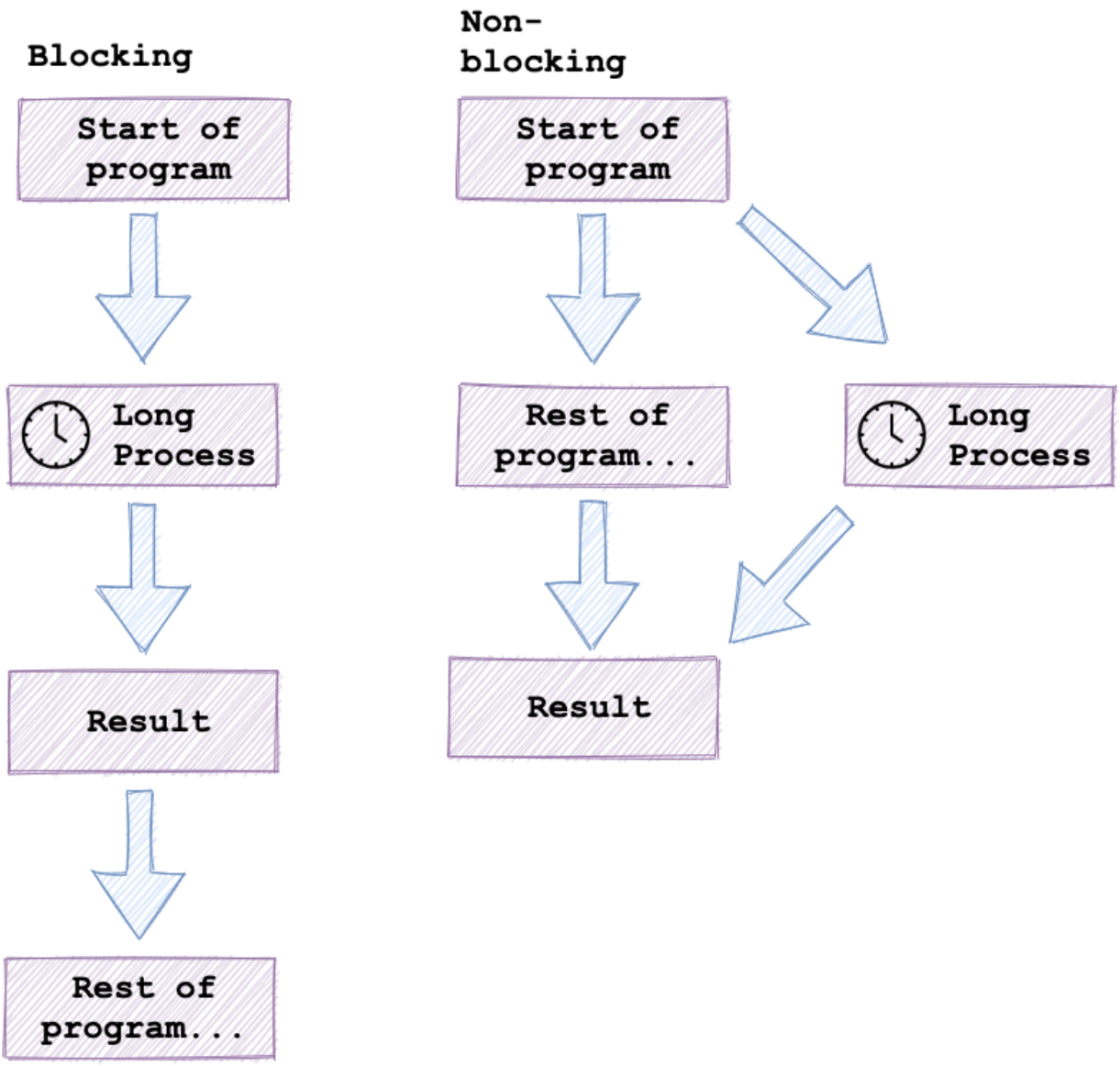
Asynchronous Programming

A **synchronous** program is one that processes each line of code in the order it appears. The problem with this approach is that, if a particular part of the program takes a long time to complete, it will hold up the rest of the program. This means that time-consuming events such as the completion of a file download, getting data from a database, or loading a game will block anything else from happening.

One solution is to start a new **thread** to run different parts of the program. This means that if a process is taking a long time in one thread, other threads can be used for other tasks. The problem with multi-threaded programs is that it can be difficult to keep track of what's happening in each thread, especially if the results of one thread are needed in another.

Another solution is to write **asynchronous** code, which runs out of order, or *asynchronously*. Instead of waiting for an operation to finish, a callback (or **promise**) is created, and the rest of the program continues to run. This ensures that waiting for a process to complete doesn't hold up the execution of other parts of the program. Once the task is complete, the promise is said to be resolved and the callback returns the result back to the program.

The diagram below shows how a long process blocks the rest of the code from running when run synchronously, compared to a non-blocking, asynchronous approach.



JavaScript runs in a single-threaded environment, so it can only process one piece of code at a time, so *asynchronous programming* is an essential tool to ensure there aren't any processes blocking the the rest of the program from running.

We can demonstrate the asynchronous nature of JavaScript by using the `setTimeout()` method to fake a time-consuming process. The following code places a three-second timeout between two messages that get logged to the console. The timeout is taking the place of a slow operation—such as downloading a file. Try entering the code into the console, but try guessing what the output will be before you press return:

```
console.log('Hello');
setTimeout(() => { console.log('File Downloaded!'); }, 3000);
console.log('World');
```

Did you guess correctly?

If the program ran synchronously, the following would happen:

- “Hello” would be logged to the console
- there’d be a three-second delay
- “File Downloaded!” would be logged to the console
- “World” would be logged to the console

This is a **blocking** approach, since the three-second wait is blocking the execution of the rest of the program until the callback has been resolved.

JavaScript is a **non-blocking** language, so the callback doesn’t block the rest of the program happening. This is what actually happens:

- “Hello” is logged to the console
- a three-second timeout starts, but the program continues
- “World” is logged to the console
- after three seconds, “File Downloaded!” is logged to the console

It’s important to keep in mind that JavaScript is single-threaded, so only one task can be processed at a time. This means that, even if a task only takes a small amount of time to complete, the callback still has to wait for the rest of the program to finish running before it can run. In fact, even if we change the delay to zero, the order in which the strings are logged to the console won’t change. Try the following code in the console to see:

```
console.log('Hello');
setTimeout(() => { console.log('File Downloaded!'); }, 0);
console.log('World');
```

Notice the result of the callback is still logged to the console last, despite the timeout being set to zero milliseconds (the equivalent of an instant file download). You might think this means the callback will happen immediately, but a callback always has to wait for the any other functions to be resolved before it can be processed.

Delay and Sleep

Most programming languages have a `delay` or `sleep` function that will stop the flow of the program. For example, the following Ruby code uses the built-in `sleep()` method to pause the execution of the code for three seconds:

```
print 'Hello'
sleep 3
print 'World!'
```

This will show “Hello”, then pause for three seconds before displaying “World!”

Even though this looks similar to the way `setTimeout` works, it’s not exactly the same, as the asynchronous nature of JavaScript means that the program doesn’t stop running.

“Delay, Sleep, Pause, & Wait in JavaScript” explains how to create a sleep-like function in JavaScript.

Intervals

The `setInterval()` method works in a similar way to `setTimeout`, except that it will repeatedly call the callback function at regular intervals, based on the number of milliseconds provided as the second argument.

The previous example used an anonymous function, but it’s also possible to use a named function like so:

```
function hello(){ console.log('Hello, World!'); }
```

Now we can set up the interval and assign it to a variable:

```
const interval = setInterval(hello,1000);
<< 1
```

This should show the message “Hello, World!” in the console every second (1,000 milliseconds).

To stop this, we can use the `clearInterval()` method and the variable `interval` as an argument (this is because the `setInterval()` method returns its ID, so this will be assigned to the variable `interval`):

```
clearInterval(interval);
```

The `setInterval()` method is particularly useful for coding games that require a "game loop" to run at a set interval. (In fact, we'll be using it to code a game at the end of this chapter!)

`requestAnimationFrame`

There's also another method called `requestAnimationFrame` that can be used for animations and to run a game loop. It doesn't allow you to specify the time interval, but calls a callback approximately 60 times per second. You can read more about it and programming games in JavaScript in *HTML5 Games: Novice to Ninja*, an excellent book by Earle Castledine.

Stopwatch

We can use the `setInterval()` method to implement a simple stopwatch in the browser. Open up a new Pen on CodePen and add the following HTML:

```
<div id='stopwatch'>00:00</div>
<button id='start'>Start</button>
<button id='stop' disabled>Stop</button>
<button id='reset'>Reset</button>
```

This includes a `<div>` element that's used to display the starting time of `'00:00'`, as well as three buttons that will be used to control the stopwatch. (Note that the **Stop** button starts with an attribute of `disabled` to prevent it from being clicked.)

A little bit of CSS will help to make the watch look more authentic. Add the following code to the **CSS** section:

```
#stopwatch{
  border: black 3px solid;
  width: 200px;
  text-align: center;
```

```
  color: black;
  font: 64px monospace;
  padding: 30px;
  margin: 0;
}
```

Now we need to actually make the stopwatch work. First of all, we need to get some references to the elements in the HTML. Add the following code to the **JS** section:

```
const stopwatch = document.getElementById('stopwatch');
const startButton = document.getElementById('start');
const stopButton = document.getElementById('stop');
const resetButton = document.getElementById('reset');

let time = 0;
```

The first four variables correspond to the four elements in the **HTML** section. The last variable, `time`, will be used to keep track of the running time on the stopwatch. Since it will constantly be changing, it needs declaring using the `let` keyword.

Now we need to write a function to start the stopwatch called `start`. Add the following function definition to the **JS** section:

```
function start(){
 timer = setInterval(function(){
  time++;
  // hundredths of second
  let hundredths = time%100;
  if(hundredths<10){
    hundredths = '0'+hundredths;
  }

  // seconds
  let seconds = (time - hundredths)/100;
  if(seconds<10){
    seconds = '0'+seconds;
  }
  // display time
  stopwatch.textContent = `${seconds}:${hundredths}`;
},10);
 startButton.disabled = true;
 stopButton.disabled = false;
}
```

This is a long function, but at its heart is the `setInterval()` method that calls an anonymous function every ten milliseconds, or every one hundredth of a second, indicated by the second argument of 10.

This anonymous function increments the value of the `time` variable by 1 using the `++` operator. After this, the function parses this value to separate the time into seconds and hundredths of a second. It does this by using the `%` operator to find the remainder when `time` is divided by 100, which is how many hundredths of a second are left over. There's also an `if` statement that checks to see if either value is less than 9. If it is, it adds a zero to the front, so that the watch always displays two digits.

To get an idea of what's happening here, it might be best to show an example. Say that the value of `time` is 543, which means that 543 hundredths of a second have elapsed. The value of `hundredths` will be `543%100`, which is 43 (the remainder when it's divided by 100, or the last two digits). To find the value of `seconds`, we subtract the value of `hundredths` from time to get 500, then divide by 100 to convert from hundredths of a second to seconds and get 5. Because this value is lower than 9, we'll append a zero to the front, to give it a value of `'05'`. Note that this changes the type of `seconds` from a number to a string, but since we're eventually going to use string interpolation to display these values on the page, it doesn't matter if they're strings or numbers.

Finally, we update the `textContent` of the `stopwatch` element with the formatted time.

Right at the end of the function, there are some lines that disable the **Start** button and enable the **Stop** button, so the user can stop the timer running. Let's add a function to make that happen. Add the following function to the **JS** section:

```
function stop(){
  clearInterval(timer);
  stopButton.disabled = true;
  startButton.disabled = false;
}
```

The `stop` function simply clears the interval called `timer` that was set in the `start` function. This will stop everything happening: the value of `time` will stop increasing and the `textContent` property of the `stopwatch` element won't get updated. We also disable the **Stop** button and enable the **Start** button.

Last of all, we need a function to reset the stopwatch back to zero. Add the following code to the end of the **JS** section:

```
function reset(){
  time = 0;
  stopwatch.textContent = '00:00';
}
```

This resets the `time` variable back to zero and resets the display on the stopwatch back to `'00:00'` by updating the `textContent` property.

The last thing we need to do is hook each of these functions up to their respective buttons by creating an event listener for each of them. To do this, add the following code to the end of the **JS** section:

```
startButton.addEventListener('click',start);
stopButton.addEventListener('click',stop);
resetButton.addEventListener('click',reset);
```

Now the buttons should start, stop and reset the stopwatch.

You can see my code on CodePen.

Animation

CSS provides a plethora of ways to animate elements using various transformations. You can read more about them in these recommended books:

- *CSS Master, 2nd Edition*
- *CSS Animation 101*
- *CSS Animation: De-Animating the Undead Horde*

JavaScript can be used to make these animations more interactive. Let's take a look at a simple example to see how it works.

Jumping Frog

We're going to use CSS animation to create a jumping frog effect. To start with, we need an element for our frog to go in the **HTML** section:

```
<div id='frog'>🐸</div>
```

Now let's add some CSS to get the frog into position:

```
#frog{
  font-size: 64px;
  position: absolute;
  top: 100px;
}
```

This just makes the frog a bit bigger and gives it an absolute position on the page. To use CSS animations, we need to create a `@keyframes` rule that describes each frame of the animation. Add the following to the **CSS** section:

```
@keyframes jump {
  to{transform: translateY(-100px)}
}
```

This `@keyframes` rule only describes the end state of the animation, which is a vertical translation of `100px` upwards (negative values move up in the Y direction). To make it come back down again, we'll use the `animation-direction` property with a value of `alternate`. This plays the animation forwards, then backwards, producing a jumping effect. Add the following lines of code to the `#frog` selector's styles:

```
animation-name: jump;
animation-duration: 700ms;
animation-iteration-count: infinite;
animation-direction: alternate;
```

You should see the frog jumping up and down.

(Here's a Pen showing what we've done so far.)

We can use JavaScript to add some interactivity to this animation by adding an event handler to the frog element that will run the animation when a `click` occurs instead of just repeatedly running it. First of all, remove the animation properties from the **CSS** section and add the following code to the **JS** section:

```
const frog = document.getElementById('frog');
document.addEventListener('click',e =>  frog.style.animation =
'jump 700ms 2 alternate')
```

This event listener will fire when the user clicks anywhere on the page, and it will call an anonymous arrow function to update the `style.animation` property of the `frog` element. We've also used the animation shorthand property to write all the properties in a single line. Now the frog will only jump if you click on the page. And it will only jump once: any further clicks won't make the frog budge at all. The reason for this is that, once the `style.animation` property is set, it will only run the animation once. Updating the style property again with the same value won't make it run again. To make it work again, we need to remove those styles once the animation has finished. Luckily, there's an event listener for that exact situation: `animationend`. Add the following code to the end of the **JS** section:

```
frog.addEventListener('animationend',e => frog.style.animation =
'none');
```

This removes the animation styles once the animation has finished. Any further clicks will add the styles back in and the frog will jump again.

There are two other event listeners for CSS animations that you might find useful:

- `animationstart`: this will fire when the animation starts
- `animationiteration`: this will fire at the start of every iteration of an animation, except for the first

You can see my code on CodePen.

Cookie Grabber Game

Our final project is a game based on the fairground classic Whack-A-Mole, although this will be a more animal-friendly version that involves "grabbing" cookies that randomly appear on the screen by clicking on them.

We'll use the `setInterval()` method to make a countdown timer that puts the player under pressure to grab as many cookies as possible.

To start with, let's set up a new Pen with the following HTML:

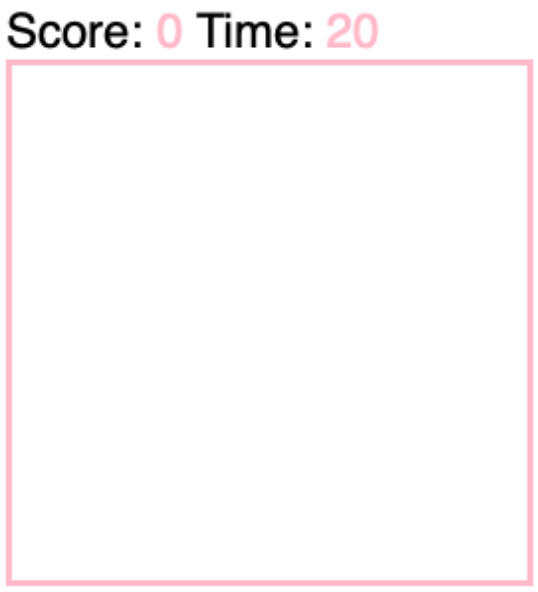
```
<div id='info'>Score: <span id='score'>0</span> Time: <span
id='clock'>20</span></div>
<div id='game'></div>
```

This is just two `<div>` elements: one for placing information about the score and time remaining, and the other for the main game.

Next, let's move on to the **CSS** section and add some styles:

```
#game {
 padding: 30px;
 border: 3px solid pink;
 position: relative;
 width: 200px;
 height: 200px;
}

.cookie{
  font-size: 32px;
  position: absolute;
  cursor: grab;
}

#info{
  font: 24px sans-serif;
}
#score,#clock{
  color: pink;
}
```

Most of this is setting the size and colors of the the various elements of the game. You should now be able to see the main game area on the screen.



Finally, let's move on to the **JS** section and code the actual game mechanics. First of all, we need to get a reference to the elements in the **HTML** section. They all have IDs, so we can use `getElementById`:

```
const game = document.getElementById('game');
const clock = document.getElementById('clock');
const scoreboard = document.getElementById('score');
```

Next, we want to declare and initialize the variables to track the time and score:

```
let time = 20;
let score = 0;
```

In the next step, we start the main part of the game by using the `setInterval()` method to call a `gameLoop` function every 500 milliseconds:

```
timer = setInterval(gameLoop,500);
```

The `gameLoop` function contains the main game logic. Every time it's called, it needs to add a cookie to the game area, update the `time` variable, and also check to see if the time has run out. Let's create that function now:

```
function gameLoop(){
  addCookie();
  time--;
```

```
  clock.textContent = time;
  // check to see if time has run out
  if(time <= 0){
    clearInterval(timer);
    game.removeEventListener('click',grab);
    game.innerHTML = '';
  }
}
```

We're using another function called `addCookie` to add a cookie to the game area, and we'll write that next. We reduce the `time` variable using the decrement operator (`--`), and then update the `textContent` property of the element that displays the time, represented by the `clock` variable. Finally, we check to see if time has reached, or gone below, zero. If it has, we stop the timer using the `clearInterval()` method with an argument of `timer`, which will point to whatever number was returned when the interval was created. We also remove the event listener that's used for "grabbing" cookies. (We haven't created this yet, so don't worry if you don't recognize it!) Last of all, we remove any cookies that haven't been grabbed by setting the `innerHTML` property of `game` to be an empty string, which has the effect of clearing the game area.

Our next job is to write the `addcookie` function. This needs to create an element with a class of `cookie` and place it at a random position in the game area. Add the following code to the end of the **JS** section:

```
function addCookie(){
  const cookie = document.createElement('div');
  cookie.textContent = '🍪';
  cookie.className = 'cookie';
  cookie.style.top = `${randomInt(200)}px`;
  cookie.style.left = `${randomInt(200)}px`;
  game.appendChild(cookie);
}
```

This creates an empty `<div>` element and assigns it to the variable `cookie`. We then update the `textContent` property to be a cookie emoji. We then set the `className` property to be `cookie`. This is so that it picks up the styles we set in the **CSS** section, and we'll also be using it later to identify that a cookie has been clicked on. The code to place the cookie in a random position uses the `style.top` and `style.left` properties that act like

coordinates from the top left corner of the `game` div. We then use the `randomInt` helper function that we created in Chapter 8 to choose a random integer between 1 and 200 (the width of the game area) to set both these properties. Finally, we use the `appendChild` DOM method to add this element as a child of the `game` element. This means that any cookies created will appear at a random position inside the game area.

Next, we need to add the player interaction. When the player clicks on a cookie, we want them to "grab" it. We could attach an event listener to each cookie we create to listen for `click` events, but it's much easier to use event delegation and attach the event listener to the `game` element instead. Add the following code to the end of the **JS** section:

```
game.addEventListener('click',grab);
```

This will fire the `grab` function every time the player clicks anywhere inside the game area. This means that we need to write the `grab` function and also check to see if a cookie has been clicked on. Add the following function declaration to the bottom of the **JS** section:

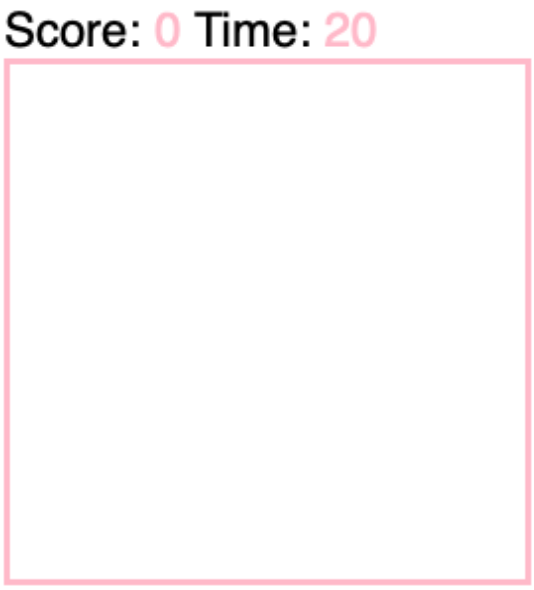
```
function grab(event){
  if(event.target.className === 'cookie'){
    score ++;
    scoreboard.textContent = score;
    event.target.remove();
  }
}
```

This looks at the `target` property of the event object that's passed to any event handler. This property returns the element that was actually clicked on. This means we can then check to see if the object has a class of `cookie`. If it does, we increase the score by 1 using the increment operator (`++`), update the `textContent` of the `scoreboard` element so that it shows the new score, and then remove the element that was clicked on from the DOM. This means that, if the player clicks on a cookie, their score will increase by 1 and the cookie will "disappear"!

Last of all, we need to include the `randomInt` helper function that we used to create the random integers. Because function declarations are hoisted, we can place this out of the way, at the bottom of our code:

```
function randomInt(lower,upper){
  if(upper===undefined){
    upper = lower;
    lower = 1;
  }
  return Math.floor(Math.random()*(upper-lower+1)+lower)
}
```

And that's it! Have a go running the code and see how many cookies you can grab. It should look something like this:



You can see my code on CodePen.

Challenges

1. Design a form that allows a user to enter their date of birth and find out the day on which they were born. For example, entering "1/1/2000" (January 1, 2000) should return "Saturday". You can see my code on CodePen.

2. Create a Tabata exercise timer. This should count down in alternating rounds of 20 seconds of work and ten seconds of rest for eight rounds. The timer should indicate whether it's currently work or rest. For bonus points, add **Start**, **Stop** and **Reset** buttons. You can see my code on CodePen.

3. Improve the Cookie Grabber game by adding some extra features. A few suggestions might include: adding a button for users to press to start the game; making the cookies appear at random frequencies; making the cookies a random size when they appear, with the number of points they're worth inversely proportional to their size (the smaller the cookie, the harder it is to grab, so the more points you get!)

   Can you think of any more? You can see my code on CodePen.

Summary

- The UNIX epoch is an arbitrary date of January 1, 1970, that computer languages use as a reference point to measure how much time has elapsed.
- Most programming languages have a `Date` object that allows you to store dates and times in various formats.
- The `setTimeout()` method calls a callback after a number of milliseconds provided as an argument.
- Asynchronous programming means that code can be executed out of order. It stops long processes blocking the flow of code while the program is waiting for them to complete.
- The `setInterval()` method repeatedly calls a callback every time a given time interval has passed.
- JavaScript can be used to add interactivity to CSS animations.

We've now reached the end of the last practical chapter in this book. But there's still one more chapter to go—one that will hopefully inspire you on to the next phase of your coding journey.

Chapter 16: End Of Line

Just because you've nearly finished this book doesn't mean your coding journey is coming to an end. On the contrary, it's only just beginning. As the saying goes: as one code block closes, another opens …

The aim of this final chapter is to give you some ideas and inspiration about where you go next. We'll be taking a look at the following topics:

- coding best practice
- going further with JavaScript
- learning a new language
- always learning
- things to build

Coding Best Practice

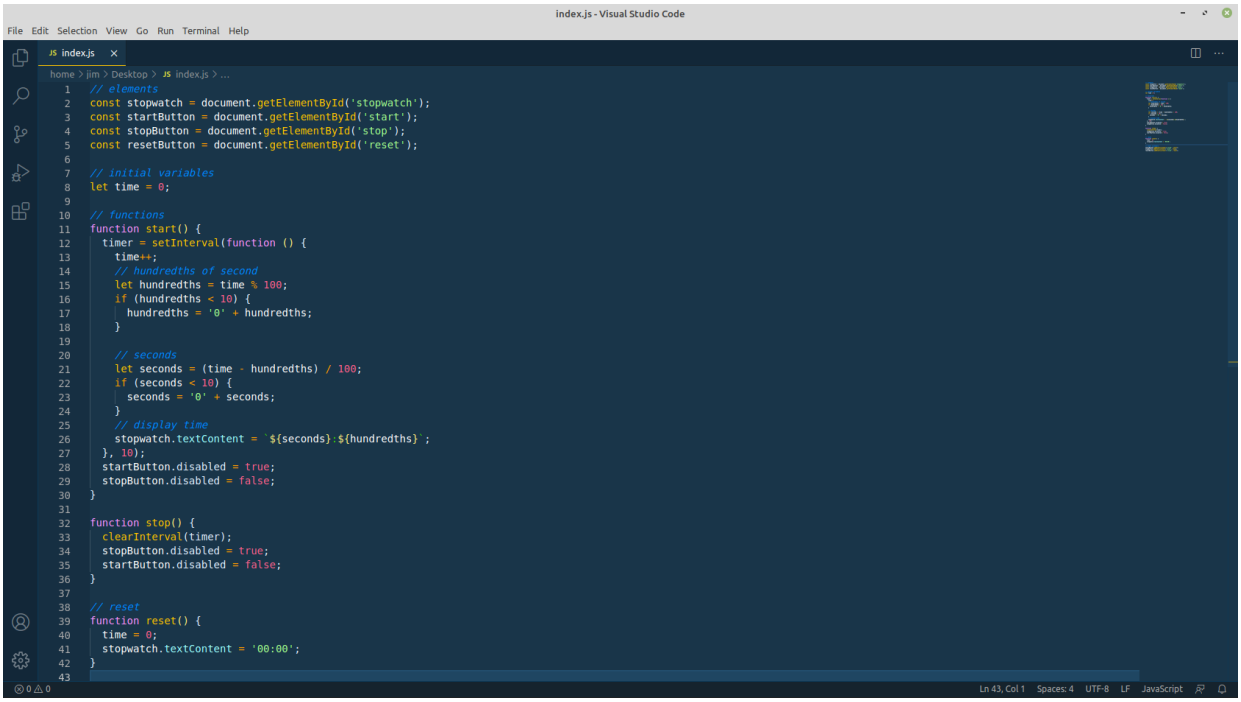
Now that you can code with confidence, it's time to set up your coding environment and learn some best practices that can help you take your coding to the next level.

Coding Tools

CodePen has been useful for writing the code examples in this book and instantly seeing the results, but now it's time for you to set up your own coding environment. Your first job is find an editor to write your code in.

There are a large number of text editors available, but you won't go far wrong with VS Code, Sublime Text or Atom. All of these are full-featured text editors, including syntax highlighting, tabs, file browsing and code completion. Atom was even built using HTML, CSS and JavaScript!

You can find out more about setting up VS Code by reading *Visual Studio Code: End-to-End Editing and Debugging Tools for Web Developers*, by Bruce Johnson.



If you use the Chrome web browser or ChromeOS, there's also the option to use Text and Caret, both of which are very good options.

Style Guides

Another good practice is to follow a coding style guide. These are usually written by teams of developers to ensure they agree on how they write code. The style guides used by Google and Airbnb are both publicly available. Following the standards outlined in a style guide will help improve the quality of the code you write and help to keep your coding style consistent, as shown in "Why I Use a JavaScript Style Guide and Why You Should Too".

You can also use **linting** tools that will highlight any style errors in your code and help ensure that any code you write follows the style guidelines you've specified. Try pasting some of your code into the JSLint online tool for a report on its quality.

Version Control

Version control software allows you to track all the changes that are made to your code, because every version of your code is kept and can be recalled at any time.

One of the most popular tools for source control management is Git, written by Linus Torvalds, the creator of Linux. Git enables you to roll back to a previous version of your code. You can also branch your code to test new features without changing the current stable codebase. Git is a **distributed** source control system, which means that many people can fork a piece of code, develop it independently, then merge any of their changes back into the main codebase. It also means that if you make a mistake, you can roll back to the last working version of the code.

Git uses the command line to issue commands, but there's a large number of GUI front ends that can be installed to give a visual representation of the code, including the excellent GitHub Desktop.

There are also various online services that can host Git repositories, including GitHub, GitLab and Bitbucket.

As you write more complex code, your life will be made much easier by integrating Git into your everyday workflow. You can find out more about Git in *Jump Start Git*, by Shaumik Daityari.

Testing

As you start to write more code, you should try to get into the habit of testing it. And by that, I don't just mean trying it out a few times to see if it works. **Testing** is an integral part of the development process and involves writing code that tests your code as you write it. Errors and bugs are a part of coding, and testing helps to identify and deal with them quickly.

Test-driven development (TDD) is the process of writing tests *before* you even write any code. The idea is that you decide what you want your code to do, write a test for it, and then write the code to make it pass the test. It might seem strange at first, but it helps to ensure that your code does what it's supposed to do.

A good place to get started with JavaScript testing is with "A Beginner's Guide to Testing Functional JavaScript". Popular test libraries include Jasmine, Mocha and Jest.

Going Further with JavaScript

This book has focused on using JavaScript to teach the fundamentals of programming. If you've enjoyed what we've covered so far, the good news is that there's still lots more to learn.

Advanced JavaScript

We've covered the basics of JavaScript in this book, but there's much more to learn. A good place to start would be to learn about web **APIs** (application programming interfaces). Some highlights include: the Canvas API, which can be used to create 2– and 3–D drawings; the Fetch API, for making HTTP requests and retrieving data from other servers; the Web Storage API and cookies, which both allow you to store data on the user's computer; and the WebSocket API, which allows multiple users to access the same site simultaneously.

There's also more to learn about asynchronous programming and JavaScript's event loop, including promises and the `async` and `await` keywords. JavaScript also has maps and sets, which are different types of data structures.

JavaScript **modules** involve separating your code into separate files that can be reused in different projects. You can also import external libraries written by other people to add extra functionality to your code. Have a read of "Understanding ES6 Modules" to find out more.

One area that we didn't touch upon in this book, but which is very useful to learn, is **regular expressions** (or **regex**, for short). These are patterns that describe the structure of a string, and they're implemented in almost every programming language. They can be used to search strings and perform "find and replace" type operations. Regular expressions can look a little strange at first, and they're something of a dark art that could easily fill a

whole book! They're certainly useful when it comes to manipulating large portions of text. These are some useful resources to help you learn more about regular expressions:

- Regular-Expressions.info
- regex101
- *Mastering Regular Expressions* (a comprehensive book)

JavaScript is a language that's constantly evolving, with new features added to the language every year. If you're keen to find out about its newest features, you should read *JavaScript: The New Toys*, by T.J. Crowder.

The talk "JavaScript: Who, What, Where, Why and Next", by Laurie Voss, details the current state of JavaScript and where trends are heading in the future.

Libraries

A JavaScript **library** is a piece of code providing methods that make it easier to achieve common tasks. JavaScript is an extremely flexible language that can accomplish most programming tasks. That said, not all undertakings are as easy to carry out as they should be. A library will abstract functionality into easier-to-use functions and methods. These can then be used to complete common tasks without having to use lots of repetitive code. A good example of this is date-fns (a library we met in the previous chapter), which makes working with dates much more manageable.

jQuery is perhaps the most famous of all JavaScript libraries. It provides numerous methods for making DOM manipulation, event handling and API calls easier to implement. It's not as popular as it once was, as "vanilla" JavaScript can now do many of the things that jQuery was often used for, but it's still used by a large number of websites. You can't go wrong with *jQuery: Novice to Ninja: New Kicks And Tricks*, by Earle Castledine, if you want to learn anything about using jQuery.

Lodash is a popular library of utility functions that provide additional functionality to the language. Lodash provides access to a number of well-

tested utility functions that will save you writing your own implementation and effectively reinventing the wheel, so it's often worth considering.

Having said that, writing your own library of helper functions is also a useful exercise that can help to improve your coding skills. This involves creating a set of functions that you regularly use in projects. One example would be the `randomInt` function that we used in a number of the coding challenges towards the end of this book.

Node.js

JavaScript was originally thought of as a front-end programming language for the browser. Node.js fundamentally changed the JavaScript landscape when it was released in 2009, as it meant that JavaScript could be run on a computer or server without the need for a browser. This means that JavaScript can now be used to write server-side code and interact with the file system.

If you install Node.js, you'll be able to run JavaScript without using your browser and have access to the Node REPL (read-eval-print loop), which is a command-line JavaScript console on your own computer. It also gives you access to a large repository of Node packages that provide a wide range of functionality using the Node Package Manager. Many of these packages are tools that help to automate your workflow and compile your JavaScript code before deploying it.

Node.js can also be used to build server-side applications and dynamic websites that interact with back-end databases. The asynchronous nature of JavaScript makes it suited for real-time update applications with lots of concurrent users, as it's able to quickly deal with requests in a non-blocking way.

If you want to learn more about Node.js, a good place to start is "What Is Node and When Should I Use It?", and *Your First Week With Node.js*.

Deno

The creator of Node.js, Ryan Dahl, has created a new project called Deno, which he claims fixes a number of problems that Node.js has.

"Node.js vs Deno: What You Need to Know", by Nilson Jacques, provides a useful comparison of Node.js and Deno, and Craig Buckler's TechExeter talk "A First Look at Deno" provides a useful introduction.

Learn Another Language

Now that you've learned the basics of programming using JavaScript, you should find it much easier to pick up another language. There's a vast choice of languages out there for you to try.

If you enjoyed learning about object-oriented programming, you might want to try learning Ruby or Python. You can get started by reading *The Python Apprentice*, and then progress to *The Python Journeyman* and *The Python Master*.

You might want to try learning a classical language that needs compiling, such as Java, C# or Swift.

If you want to learn more about functional programming, you could try learning Haskell, Elixir, Erlang, Scala, or Clojure.

There are also other scripting languages such as Lua or Bash. Go is a modern, dynamic language that has many similarities to JavaScript, and TypeScript is a superset of JavaScript that adds a number of features. In particular, it's a strongly typed language (meaning that you have to declare the type of variables, as we discussed in Chapter 2).

A relatively new language that has proven to be very popular is Rust. It's a low-level language, similar to C++, and it can be used to write system-level, high-performance code. "Rust Tutorial: An Introduction to Rust for JavaScript Devs" is a good introduction for people who know how JavaScript works.

And then there are the truly wacky languages, such as Ook that only has three commands: `Ook.`, `Ook?` and `Ook!`, ArnoldC, a language made up

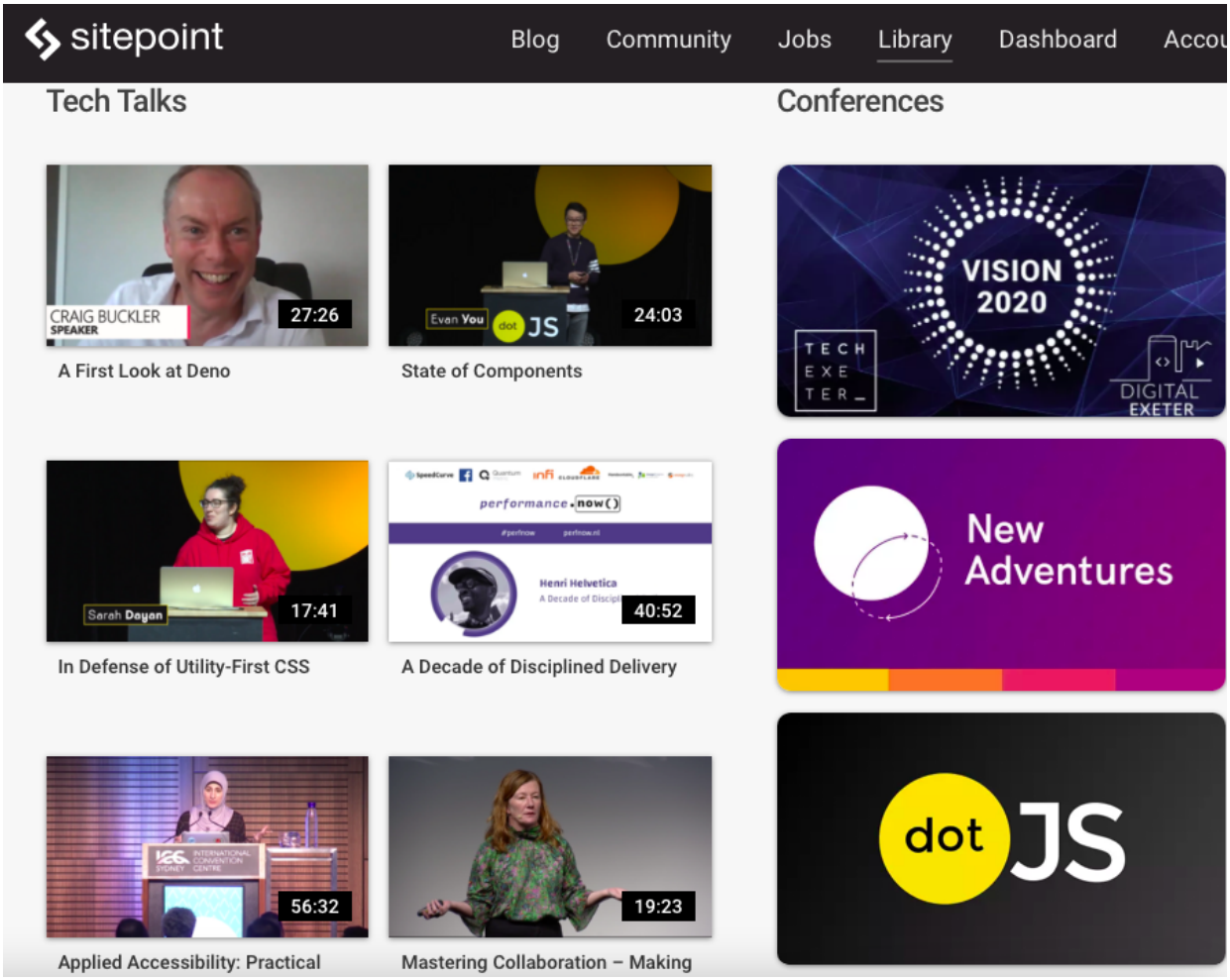
entirely of Arnold Schwarzenegger quotes, and Tabloid, with code that reads like a tabloid newspaper!

One thing's for sure: you're probably not going to run out of languages to learn, and they'll all help to increase your understanding of coding concepts and give you an appreciation of the various programming paradigms.

Always Learning

The world of coding is fast-moving, and it's getting faster every year. You need to ensure you keep up to date with recent developments and best practices. Here are some suggestions of how you can keep your knowledge current:

- read the many books and articles on SitePoint
- watch video tutorials
- follow programmers on social media
- read blog posts and listen to podcasts such as Syntax and JavaScript Jabber
- join a local or online user group
- attend conferences or local meetups
- watch online tech talks (there are loads available on SitePoint



Carry On Coding

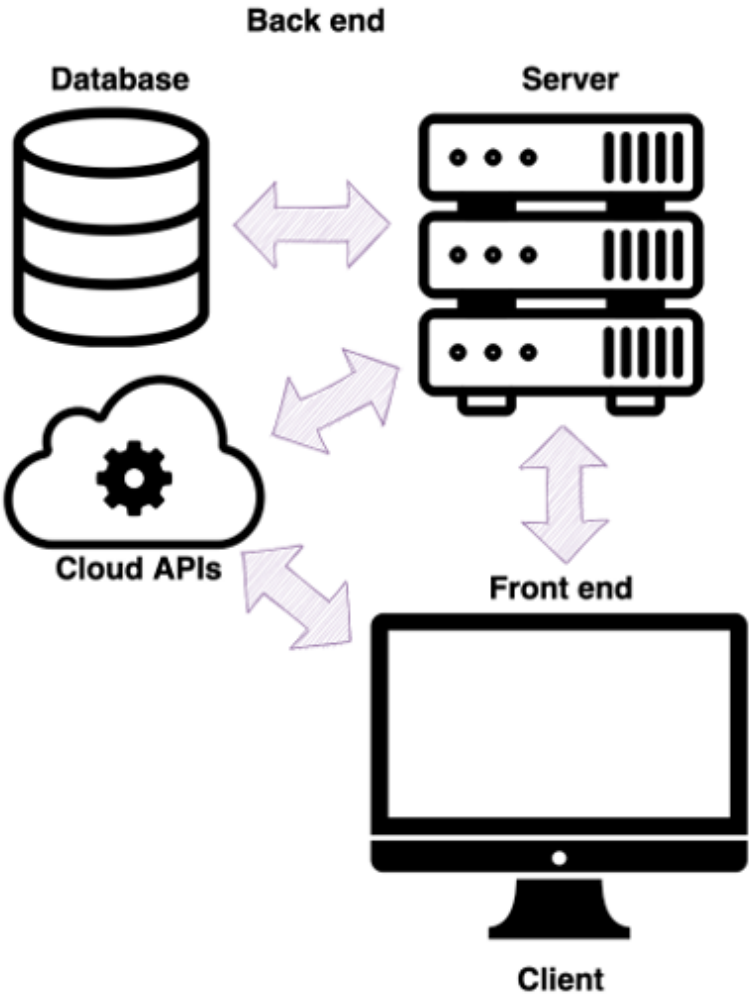
You can learn all the theory you want, but the only way you'll actually get better is by actually coding. By putting ideas into practice and solving real problems, you'll really start to get a feel for coding. Here are some ideas of things you could build using your JavaScript coding skills.

The following ideas are intended to get your creative juices flowing and, I trust, spark an idea for a project. It's by no means a complete list of what you can do with JavaScript, as the possibilities are endless and only limited by your imagination.

Web Applications

JavaScript was originally created to help make websites interactive. But since those humble beginnings it has developed into a powerful language that's capable of producing some of the most sophisticated web applications such as Gmail, Netflix and Uber. The combination of HTML, CSS and JavaScript can be used to create an interactive website or online application that can then be hosted online with little effort.

Basic websites can be built using just HTML and CSS, though they'll often use JavaScript to add some extra interaction and DOM updates. Sophisticated web applications, on the other hand, will usually involve a back-end server that hosts a database along with a front end that includes the HTML, CSS and JavaScript. They'll usually also host their own API that other sites can use to interact with the data stored in their database. Each of these layers makes up what's known as the **development stack**. **Full-stack development** involves working on each part of the stack.



JavaScript can now be used to write code for both the front end of an application and the back end that runs on the server. It can even be used to interact with databases. This makes JavaScript a useful one-stop-shop language for writing full-stack applications. If you take your JavaScript skills to the next level, you'll be fully equipped to develop serious, high-performance web applications.

Progressive web apps (PWAs) are web applications that use local storage to make them appear more like a native application. They load quicker and appear snappier to use, without the need for a constant network connection. In his talk "Planning Your Progressive Web App", Jason Grigsby goes through some of the key things to think about when building a progressive web app.

Many modern websites and apps interact with third-party applications using online APIs to share and interact with data. This is often stored as JSON (JavaScript Object Notation), which is a string representation of the object literals we met in Chapter 9. It's a lightweight data format that allows data to be serialized in a human-readable form. Although it was derived from JavaScript objects, most modern programming languages include methods for generating and parsing data in JSON format.

One method of production that's becoming increasingly popular is the **Jamstack**. This is a stack that's made up of JavaScript, APIs and Markup (JAM). The idea is to build websites with HTML, CSS and JavaScript and to use APIs to fetch data from web services. Phil Hawksworth did a talk at Pixel Pioneers explaining how it works.

In his "JavaScript Saves the World" talk, Asim Hussain provides some useful tips on how to make web applications more efficient, which results in a better experience for users and is both cost effective and good for the planet.

Many modern web applications need to keep track of data—whether it's data they fetch from external web services or data generated by the user. This means that the web page needs to be able to keep track of any changes to the data and update the user interface (UI) accordingly. There are several

frameworks that aim to make it easier to produce interactive UIs and help with *state management*.

The big players are React and Vue. Both of these take their own unique approach to the same goal of creating components that update themselves as the user interacts with them, or as the data changes. There's no shortage of online tutorials for both React and Vue, and both have good introductory tutorials on their websites, so it's worth having a play around with both of them to see which one you prefer using.

There's also Svelte, which is a lighter option that takes a slightly different approach from React and Vue and might be more suitable for smaller projects. The tutorial on their website is an excellent place to get started.

A good place to start would be the to-do list app that we created in this book. You could try to develop it into a full-featured web application.

Why not try building the next online sensation?

Game Development

Most online games used to be written using Flash, but JavaScript is the go-to language for building modern browser-based games. The development of WebGL and browser GPUs means that fast, rendered 3D games in the browser are a realistic possibility. Modern online classics such as HexGL, World's Biggest PAC-MAN and Swooop show what can be run in the browser. These are great examples of what can be done, but games don't have to be overly complex: the success of Flappy Bird shows that a good idea that's well implemented can be incredibly popular.

If you're interested in writing an HTML5 game, then *HTML5 Games: Novice To Ninja*, by Earle Castledine, is the perfect place to start.

Why not try writing the next blockbuster game?

Mobile App Development

Android and iOS don't use JavaScript as their native programming language. However, it's still possible to build an application using HTML, CSS and JavaScript and then use tools such as Ionic, Cordova or PhoneGap to convert your code into native code that can be run on the Android and iOS platforms.

Why not try writing the next big mobile app?

Desktop App Development

Electron is an open-source library that allows you to build desktop applications using HTML, CSS and JavaScript. It uses Chromium (the open-

source version of Google Chrome) and Node.js to create applications that can run on Windows, macOS and Linux.

Electron was developed by GitHub when they built their Atom text editor. Since then, Electron has become a popular option for developers who want to create a desktop version of a web app. It has been used to create desktop applications such as WhatsApp, Microsoft Teams and Slack.

Why not try writing the next big desktop app? You can make a start with Electron's Quick Start Guide.

Internet of Things

JavaScript is becoming the language of choice for communicating with the "Internet of Things". This includes a range of devices, from watches and virtual-reality headgear to home assistants and even drones! JavaScript is often the language that's used to write code for this ever-growing list of electronic devices. Why not try writing some custom code for your latest wearable or home device?

Challenges

This chapter has provided a lot of ideas about what you could do next, but the huge number of suggestions above may well be causing you some choice paralysis! So the aim of the following challenges is to focus you on the key things to do next. If you complete each challenge, you'll be well on the way to becoming a rounded developer:

1. Install a text editor. There are many videos that show how to set up an editor like VS Code. (You can really go to town with these setups, as Wes Bos demonstrates in his video on "ESLint + Prettier + VS Code — The Perfect Setup".)
2. Install Node.js.
3. Use npm to install React.
4. Build a web app using React. (There are lots of tutorials for doing this on SitePoint.)
5. Install Git and use it for version control.

6. Push your app to GitHub.
7. Build another React app that fetches and displays data from another website.
8. Repeat …

Summary

- Set up your coding environment by installing a text editor to write your code in.
- A style guide will help to ensure that your code is written in a consistent style and is easy to read.
- Tests will help to keep your code free from bugs and errors.
- Version control such as Git can help to manage any changes to your code.
- There's still more JavaScript for you to learn, including web APIs and modules.
- JavaScript libraries such as jQuery, Lodash and date-fns provide methods for making development easier.
- Node.js lets you run JavaScript without a browser and develop server-side applications.
- There are lots of other programming languages that you can learn.
- JavaScript frameworks such as React, Vue and Svelte help to create user interfaces that update in response to changes in the underlying data.
- JavaScript can now be used to program every part of the web development stack, making it the perfect language for full-stack development.
- You can use JavaScript to build websites, web applications, games, phone apps or desktop apps.
- Your programming skills can be used to control just about anything around the home, thanks to the Internet of Things.
- Keep your knowledge up to date by reading books and articles, listening to podcasts, watching videos and following developers on social media.
- Keep writing lots of code and building things!

That brings us to the end of the chapter and the book. I hope you've enjoyed learning how to code, and I hope this chapter has inspired you to take your coding to the next level. Whatever you decide to do, the most important thing to remember is to keep coding. The more you code, the better you'll get. Coding is about solving problems and creating things. So what are you waiting for? Get coding! The only limit is your imagination!