LEARN TO CODE

Learn HTML, CSS & JavaScript & build a website, app and game

in easy steps

Garry Owen

WDD TRW

LEARN TO CODE

Learn HTML, CSS & Javascript & build a website, app and game

Written by

Garry Owen

CONTENTS

INTRODUCTION

Hello web developer!

Sit back, strap yourself in and get ready for a fantastic learning experience. This book will take you through easy-to-follow, step-by-step lessons and give you all of the guidance you need to write your first program with some flair, make a useful website that will give you fast access to all of your favourite places online, make a quiz app that fits smartly onto your mobile phone and finally make a platform game. Top-notch!

This book is designed for you to be able to code everything and run it in a browser, and program it locally on your PC or Mac. In fact, by design HTML, CSS, and JavaScript can be run in any browser and coded in almost any text editor.

WHAT WILL YOU NEED?

Everything you need is available for free on Windows and Mac. Better than that, what you need is built-in, if that's the way you decide to go. However, I would recommend using the Google Chrome browser (this is not essential, you can use almost any browser), Visual Studio Code (…again not essential, you can use Notepad or similar if you wish) and lastly a hint of patience as you learn.

ABOUT THE AUTHOR

I have been coding in various languages for almost four decades. I work as an IT Director and have been employed in Director level roles for almost a decade.

One of the first lessons I ever learned as a programmer was 'Top-Down Design' and 'Step-wise Refinement'. In simple terms, this means breaking down a problem into ordered steps and then refining or optimising each step and where necessary breaking those steps down even further.

Let's take an everyday life example.

Make a cup of tea

1. Fill the kettle with water
2. Turn on the kettle
3. Add 2 tea bags to a tea pot
4. When the kettle has finished boiling pour over 1.5 litres of hot water
5. Stir the tea pot
6. Put the lid onto the tea pot and allow to brew for 3 minutes
7. Check the tea, if it is dark brown pour into a mug, filling to about ¾ full
8. Add milk until it appears an even caramel colour
9. Add 1 tea spoon of sugar and stir

Okay, so each step here can be further broken down, but that is only necessary if you need further clarification to understand the steps. That is the same with computer programming and indeed any other problem that you might face in your life.

For someone with prior knowledge, the instruction, 'make a cup of tea' is more than adequate. Shy of a please and thank you of course!

You only need to break down a problem far enough for you to be able to understand it. When you need to tackle a problem I highly recommend this philosophy.

Let's have some fun….

CONVENTION USED IN THIS BOOK

Each lesson is laid out in easy-to-follow steps. See the example below:-

LESSON OBJECTIVE:

Coding the 'Hello, World' program with a little flair.

WHAT IS THE 'HELLO, WORLD' PROGRAM?

Since the first "Hello, World!" program was written in 1972, it's become a tradition amongst computer science teachers and professors to introduce the topic of programming with this example. As a result, "Hello, World!" is often the first program most people write. However, we are not going to write any old 'Hello World' program. We're going to do it 'The Right Way!' and in style.

STEP ❶ - CONVENTIONS EXAMPLE

Open a suitable coding editor. This can be Notepad, Notepad++, Visual Studio Code (recommended), or similar….

Important information will be highlighted with this icon.

Tips will be highlighted with this icon.

Please note! To help you along the way, extra information is shown in information boxes, which relate to parts of the current lesson content. An example of this is shown here.

All URL's will be shown in blue, as below:-

https://google.com

All programming code will be shown with the text 'Syntax Highlighted', as shown in the example below.

URL: Uniform Resource Locator, otherwise known as a web address

The first part of the URL is called a protocol identifier (**HTTPS)** and it indicates what protocol to use, and the second part is called a resource name (**google.com)** and it specifies the IP address or the domain name where the resource is located.

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- Bootstrap CDN CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstra
p/3.4.1/css/bootstrap.min.css">
```

Syntax highlighting is a feature of text editors that are used for programming, scripting, or mark-up languages. The feature displays text, especially source code, in different colours and fonts according to the category of terms.

Word Wrapped:

A line of text that requires more space is wrapped around and displayed on additional lines below.

Due to page width constraints, the code is also word-wrapped.

Please note all code for each lesson is available via the support website, in easy to download files. Direct links are available at the end of each lesson.

HELLO, WORLD!

LESSON OBJECTIVE:

Coding the ‘Hello, World’ program with flair.

WHAT IS THE ‘HELLO, WORLD’ PROGRAM?

Since the first “Hello, World!” program was written in 1972, it's become a tradition amongst computer science teachers and professors to introduce the topic of programming with this example. As a result, “Hello, World!” is often the first program most people write. However, we are not going to write any old ‘Hello World’ program. We’re going to do it ‘The Right Way!’ and in style.

STEP ❶ - CHOOSE A SUITABLE CODING EDITOR

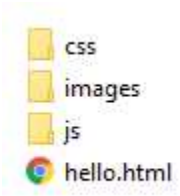
Open a suitable coding editor. This can be Notepad, Notepad++, Visual Studio Code (recommended), or similar.

STEP ❷ - SET-UP YOUR FILE STRUCTURE

Make a new folder on your desktop and call it ‘hello world’.

Within it, make another folder named ‘css’, a folder named ‘js’, and finally, a folder named ‘images’. Next, open your chosen coding editor, make a new file and save your empty file as ‘hello.html’, within your ‘hello world’ folder.

If you have followed these steps correctly you should have the following file structure - a folder called 'hello world' and one file (hello.html) and three folders:



You're doing great!

STEP❸ - DEFINE YOUR HTML STRUCTURE

In the most simple of descriptions, an HTML layout can be recognised as shown below. Enter the code exactly as you see it into your 'hello.html' file and hit Save in the File menu or press CTRL and S.

To save more easily, use shortcut keys -

CTRL and S, for Windows or Command and S, for Mac.

```
<html>
    <head>
    </head>
    <body>
    </body>
</html>
```

To tell the browser we want to display an HTML page, we begin with the <html> opening mark-up.

Next, we have the <head>. Within the opening and closing <head> and </head> tags we put links to external files, such as CSS and JavaScript, as

well as this we can include meta data and incorporate page styles if we desire, plus much more.

Lastly, we have the <body>. Within the <body> and </body> tags we put the main content of the webpage.

Next, update your 'hello.html' file, as below:

```
<html>

<head>
	<title>Hello</title>
	<link rel="stylesheet" href="css/app.css">
</head>

	<body>
		<h1>Hello, World!</h1>
	</body>
</html>
```

OK, you'll already recognise the HTML file structure. So, what else do we have?

```
<title>Hello</title>
```

This sets the page title, which can be viewed on the page tab of your browser.

```
<link rel="stylesheet" href="css/app.css">
```

This connects to an external style sheet, which can be found in the 'css' folder. The external style sheet provides a list of styling rules for your HTML page.

NB! Your code will NOT be syntax highlighted if you are using Notepad. Instead, it will be displayed as plain black text.

You can now double click on your 'hello.html' file to open it in your default browser. You'll get a white page with the <h1> title 'Hello, World!' displayed in your browser's default font. So far, it's plain and frankly quite dull, but we're far from done yet though.

STEP 4 - EXPLORING THE USE OF CASCADING STYLE SHEETS

Next, we'll explore the use of Cascading Style Sheets with our HTML mark-up.

WHAT ARE CASCADING STYLE SHEETS?

Cascading Style Sheets are used to format web pages. For example, CSS can be used to define colour, width, height, margins, opacity, padding, etc. There are three ways that you can implement CSS: internal, external, and inline styles. All of these will be explained in detail through the book.

How does CSS (Cascading Style Sheets) work with our HTML mark-up?

As we said above, CSS provides a set of rules for formatting the layout and styling of an HTML page. These rules are applied using either id, class, or keyword selector. A CSS selector is the first part of a CSS Rule. It is a

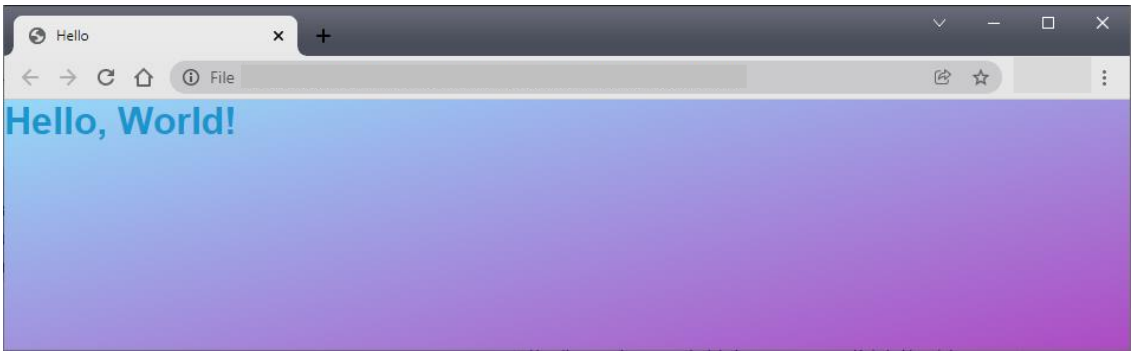
pattern of elements and other terms that tell the browser which HTML elements should be selected to have the CSS property values inside the rule applied to them. If that sounds like gobbledygook, don't worry too much right now. It will all become clear.

Now make a new file called app.css and save it in the 'css' folder. Then add the following code:

```
html, body {
    margin: 0px;
    background-image: linear-gradient(to bottom right, #95d9f9, #ae4bc3);
}
h1 {
    color: #1b95cd;
    font-family: tahoma, sans-serif;
}
```

Save the file. (CTRL + S)

At the moment, there are only two elements on our webpage. That is the h1 heading and the html / body of the page. If you look at the web page now and refresh it you will see that these two elements have had styling rules applied to them.



The background and font have been coloured and the font has been changed. Let's take a closer look.

Firstly, we can see that the margin is set to 0 pixels. This command means all margins are set to zero. Margins can be set separately using margin-left, margin-right, margin-top, and margin-bottom or as a combined margin command. When used in this way, you can still set the independent values, like so:

```
margin: 0px 0px 0px 0px;
```

Where the first 0px is for the top, the next for the right, the next for the bottom, and the last one is for the left. (TRBL).

In other words, the frame will be rendered into the user's view tight into the corners of the screen.

Next, we style the body element with a linear gradient. It is set to the bottom right, which means the gradient will render from the top left of the page to the bottom right. After the comma, we have `#95d9f9,` which is a hexadecimal code for a shade of light blue and finally, we have `#ae4bc3`, which is a hexadecimal code for a shade of purple.

Next, we styled the h1 tag. This means anything on the page between an <h1> and </h1> will be styled with the rules set.

Firstly we set the font colour. Notice color is spelled in the American way in code. The colour is set using a deeper blue with the hex code: `#1b95cd;`

Colours can be set in one of three ways. One as a hexadecimal code, as demonstrated, two as an RGB code, and third, some colours can be set by name. For instance, red, blue, green. We'll be using hex codes and RGB and RGBA codes throughout this book because they give greater control over the output.

Secondly, we changed the font, from the default to Tahoma, sans-serif. Most people will be familiar with this font from usage in word processing applications etc. The sans-serif simply means no serif. A serif is the decorative edges some fonts have. The slight projection finishes off the stroke of a letter in certain typefaces. Like Times New Roman, as an example.

OK, let's take this up a few notches.

STEP ❺ - ADD MULTI-LANGUAGE SALUTATIONS

As a website is intended to be available the world over, let's say 'Hello' in a few languages to make it more cultured and diverse.

Edit your 'hello.html' file and add the following code.

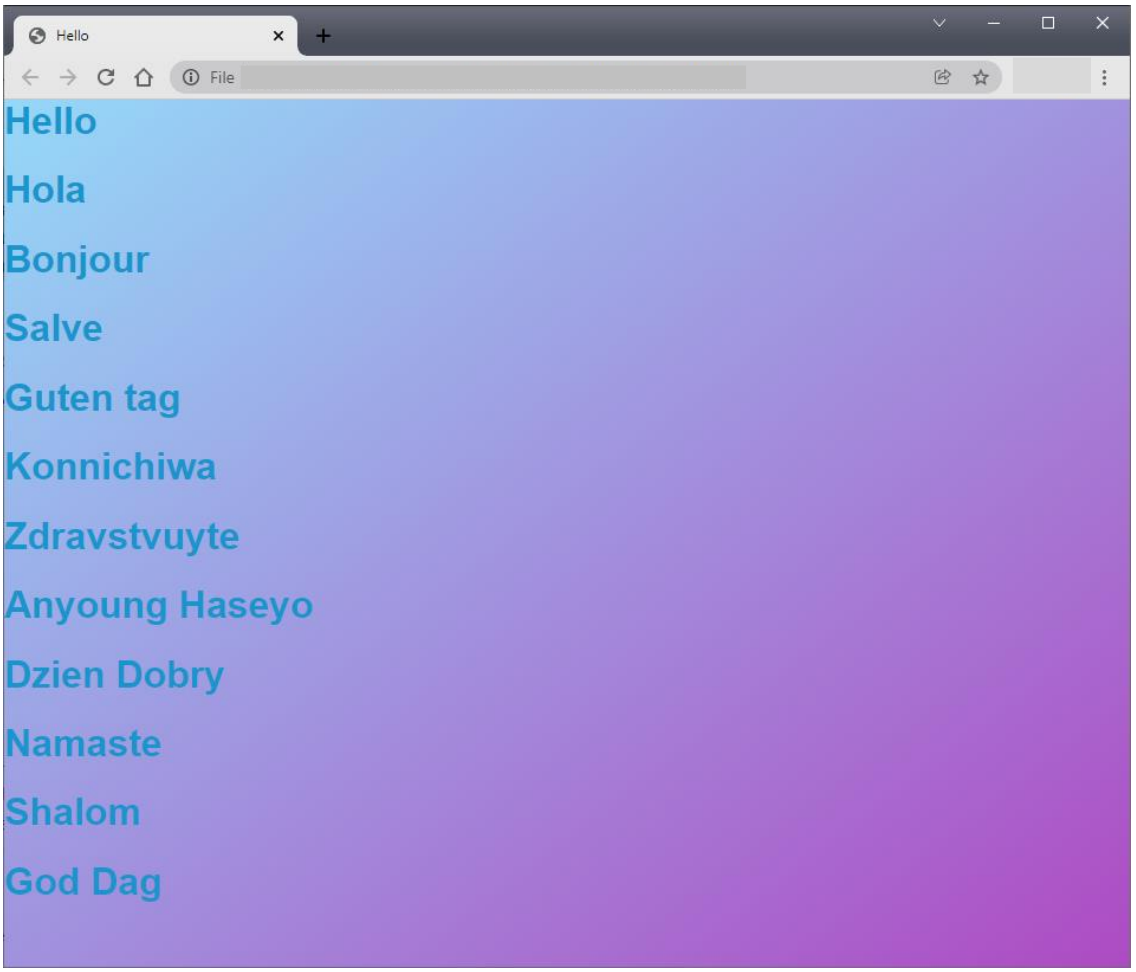
```
<h1 class="GB"> Hello </h1>
<h1 class="ES"> Hola </h1>
<h1 class="FR"> Bonjour </h1>
<h1 class="IT"> Salve </h1>
<h1 class="GE"> Guten tag </h1>
<h1 class="JP"> Konnichiwa </h1>
<h1 class="RU"> Zdravstvuyte </h1>
<h1 class="KO"> Anyoung Haseyo </h1>
<h1 class="PO"> Dzien Dobry </h1>
<h1 class="HI"> Namaste </h1>
<h1 class="HE"> Shalom </h1>
<h1 class="SW"> God Dag </h1>
```

Replacing the:

```
<h1>Hello World!</h1>
```

Line of code.

So far, if you have followed the instructions correctly, refreshing your browser should render the following result:



There is only one real difference here. That is the addition of the classes inside each of the <h1> tags. Presently they have no effect and each of the <h1> tags is simply following the existing rules we made for the <h1>. Next, we are going to spice things up a bit more by adding some JavaScript. JavaScript can utilise classes, keywords, and ids in much the same way as CSS.

STEP ❻ - JAVASCRIPT FUNCTIONS

Using JavaScript we can update our CSS script. To do so we are going to make a JavaScript function.

WHAT IS A JAVASCRIPT FUNCTION?

A JavaScript function is a block of code designed to perform a particular task. A JavaScript function is executed when it is invoked (called).

Make a new file called app.js and save it in the 'js' folder. Then enter the following code.

```
//Generate salutations
//t - top, l - left, c - colour, s - font-size, z - z-index, id - class

function sals(t, l, c, s, z, id){
    var sal = document.getElementsByClassName(id);
    sal[0].style.top = t;
    sal[0].style.left = l;
    sal[0].style.color = c;
    sal[0].style.fontSize = s;
    sal[0].style.zindex = z;
}

sals('5%', '50%', '#1b95cd', '250px', '20', "GB");
sals('1%', '10%', 'red', '150px', '19', "ES");
sals('15%', '30%', 'blue', '100px', '18', "FR");
sals('35%', '60%', 'green', '170px', '17', "IT");
sals('54%', '20%', 'orange', '170px', '16', "GE");
```

```
sals('54%', '35%', 'yellow', '100px', '15', "JP");
sals('10%', '55%', 'grey', '80px', '14', "RU");
sals('27%', '21%', 'lightgrey', '100px', '13', "KO");
sals('37%', '31%', '#333', '65px', '12', "PO");
sals('67%', '51%', 'pink', '125px', '11', "HI");
sals('47%', '11%', 'lightblue', '115px', '10', "HE");
sals('27%', '31%', 'lightgreen', '185px', '9', "SW");
```

We'll need to connect this script to our template in the 'hello.html' file. Edit it as follows:

Just inside the </body> tag add the following line of code:

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

So let's talk a little about what the JavaScript code is doing.

```
//Generate salutations
//t - top, l - left, c - colour, s - font-size, z - z-index, id – class
```

This part of the code is comments to help you to remember the purpose of the code if you ever revisit it. You can use the // double forward slashes in front of any text in JS to comment out a single line, as well as /* as the opening and */ as the ending to comment over many lines, like this:

```
/* Generate salutations
t - top, l - left, c - colour, s - font-size, z - z-index, id - class */
```

This second style of commenting can also be used in CSS for commenting your code. This is a highly recommended practice. You should always

comment your code. It is particularly useful if you are collaborating on a project with other developers.

I have noted here what each of the function parameters represents. We can pass data to each of the parameters to use inside our function.

Next, we have the sals (short for salutations) function itself.

```
function sals(t, l, c, s, z, id){
    var sal = document.getElementsByClassName(id);
    sal[0].style.top = t;
    sal[0].style.left = l;
    sal[0].style.color = c;
    sal[0].style.fontSize = s;
    sal[0].style.zindex = z;
}
```

Okay, so the first line here opens the function which we've named sals and passed the parameters t, l, c, s, z, and id. That's all done with the one line of code.

```
function sals(t, l, c, s, z, id){
```

And at the end of the function, we close it with the curly brace.

```
}
```

Inside the function, our first line of code makes a variable called sal. Our variable holds an id tag which is associated with our class name given in our 'hello.html' file.

Variable:

A variable is something that can be changed. In computer programming we use variables to store information that might change and can be used later in our program.

```
<h1 class="GB"> Hello </h1>
```

Here you can see the first of the class name associations, whereby the class="GB" part of the code is used to attribute our CSS and JS code to our HTML mark-up.

So essentially this line of code is saying connect to the class given by the id parameter.

The next line of code changes the top CSS style to a new given value in the parameter t.

```
sal[0].style.top = t;
```

Each of the remaining lines of code inside the function's open and close curly braces { and } changes a specific style for the given parameter, as our comments suggest.

```
t – top – How many pixels from the top of the screen
l – left – How many pixels from the left of the screen
c – colour – What colour to display the text in
s - font-size – The desired font size
z - z-index – Overlay level – 1 appears in front of 0
id – class – Our chosen class name
```

That brings us neatly on to our set of function calls. Without calling the function the code within it will not run. Each of our function calls does the same thing, with exception of changing the given parameters. Let's examine the first function call to get a better idea.

```
sals('5%', '50%', '#1b95cd', '250px', '20', "GB");
```

Okay, so firstly to call a function we need the following code:

```
sals();
```

This will call the function called sals but passes no parameters. In our case, it would throw an error that can be viewed in your browser console, but otherwise, it would appear to do nothing because we need to pass our parameters for our code to do what we want to achieve.

When we call it correctly like this:

```
sals('5%', '50%', '#1b95cd', '250px', '20', "GB");
```

sals(); calls the function

'5%' represents the t parameter

'50%' represents the l parameter

'#1b95cd' represents the c parameter

'250px' represents the s parameter

'20' represents the z parameter

'GB' represents the id parameter

Therefore, using our function allows us to change all of these parameters in just one line of code, per salutation.

Add the following to the h1 rules in our CSS file, and save it (CTRL + S):

```
opacity: 60%;
position: absolute;
```

`opacity: 60%;` does what it says on the tin. It makes each of the salutations slightly transparent, 60% opaque, or 40% transparent, depending on what perspective you choose to view it from. Opacity can also be set with decimals. 1 being equal to 100% opaque, while 0 is invisible. Opacity can be set in increments between 0.0 and 1 with lower values being more transparent.

`position: absolute;` allows the top and left parameters in our JavaScript function to update our CSS position property. There are seven properties for positioning in CSS. Namely, static (this is the default value), absolute, fixed, relative, sticky, initial, and inherit. For now, all you need to know is that position: absolute allows us to position elements on the screen in exact locations, based on top and left coordinates.

Now refresh your browser and take a look at the results.

It's starting to look more interesting!

STEP 7 - ADDING IMAGES

Our initial program said 'Hello, World', so let's do something with the world part. That's where our images folder comes in.

Okay, download the world.gif file from:

https://wddtrw.co.uk/resources/learntocode/images/world.gif

Save the image into your images folder within your file structure. To do so, go to the given link address, right-click on the spinning globe and choose 'Save As', from the context menu. Then choose the images folder within your file structure and hit the save button.

Note! If you decide to change the file name, you'll also have to change it in your code. It is named 'world.gif', so the image must be referenced as 'images/world.gif'. It is crucial that these match, otherwise your image won't display.

Next, edit your 'hello.html' file and add the following line of code just below your opening <body> tag, as shown below. Then save your file.

```
<body> ◄ Don't add this.
<div class="globe"><img src="images/world.gif" class="globe_image"></div>
```

Here we have placed our image inside a division. Think of this as a division of the visible screen. We have given the the class of globe. We will use this to place the div and its contents where we want it to appear on the screen. We have also given our image a class. This is so we can control how we want our image to display.

** :**

The div tag is known as the Division tag. The div tag is used in HTML to make divisions of content in the web page like (text, images, header, footer, navigation bar, etc). ... It is used to group various tags of HTML so that sections can be created and styles can be applied to them.

Next, open the app.css file and add the following lines of code.

Firstly inside the h1 { } add the following line of code:

```
-webkit-text-stroke: 2px white;
```

This will add a 2px or 2-pixel wide stroke (outline) around the text. This was added to make the words stand out better from the background.

Pixel:

A pixel is the smallest unit of a digital image or graphic that can be displayed and represented on a digital display device.

A pixel is also known as a picture element (pix = picture, el = element)..

Next, add the following two classes to the bottom of the app.css file:

```
.globe {
    text-align: center;
    margin-top: 150px;
}
.globe_image {
    height: 70vh;
}
```

The globe class has been given 2 rules to follow. Firstly it has been aligned to the centre of the page and secondly, it has a margin of 150 pixels set from the top of the visible screen.

The globe image class has been given one rule. This has set the height of our globe to 70vh or 70 percent of the view height.

Save the file and preview your results. If you have done everything correctly you should see the following with a spinning animated globe in the background of your display.

Hey, what a great start. Well done for getting this far!!

Next, we're going to develop something useful. That is the point of coding. Making your life or others' lives easier or more entertaining.

All files and images associated with this exercise can be found at:

https://wddtrw.co.uk/resources/learntocode/helloworld.zip

Make yourself a well-deserved cuppa and let's move on!

FAST ACCESS WEB MENU

LESSON OBJECTIVE:

Design and develop a well-structured webpage incorporating a fast access menu for all of your favourite websites.

STEP❶ - INFORMATION GATHERING

The first part of any good design is information gathering. This project is going to be personal to you, so although your result will follow the same logic for the version in the book, yours will look a little different.

Firstly, you need to choose up to 12 websites that you visit the most frequently. This can be any website. Take a look at my list below for some ideas.

Santander Online banking	Facebook
YouTube	Amazon
AOL Email	Zoom
PayPal	LinkedIn
Google Maps	Twitter

STEP❷ - ARRANGING THE FILE STRUCTURE

Next, we need to construct the file structure. For this project, it's a little simpler than the last. Create a new folder on your desktop called 'website'. Open it and within it create a folder called 'css' and another

called ‘images’. Finally, open your chosen coding editor and make a new file. Save it as ‘web_menu.html’.

If you have followed the instructions correctly, you should have the following file structure:

STEP❸ - DOWNLOADING AND PREPARING IMAGES

Next, for each of your chosen websites do a google search and find suitable images to represent each of them. Choose clear images. Don’t worry too much about the size right now because I’m going to explain how you can edit and optimise your images.

Using the same method we used to save the ‘world.gif’ image, search google images for suitable images and right-click, choose ‘Save As’ from the context menu, and save each of your images into your images folder.

This time, I recommend that you change the names of the images, for the sake of clarity and ease of use. For instance, in my case, I’ve named my first image ‘santander.jpg’. There are many different file formats for images, therefore, the one you’ve chosen might not be a jpeg. If that is the case don’t worry, just name it ‘[your filename].[your extension]’. We can just change the filenames and extensions in our code, to suit.

So I have 10 websites and therefore I also have 10 images to represent each of those websites saved into my images folder. If you have done this

correctly, you should have similar to the following in your images folder, obviously aside from potential different filenames and extensions.

Notice I chose all rectangular landscape images. This will make my result more uniform. This is a personal choice, it doesn't really matter. All that matters is that you have an image to represent each of your chosen websites.

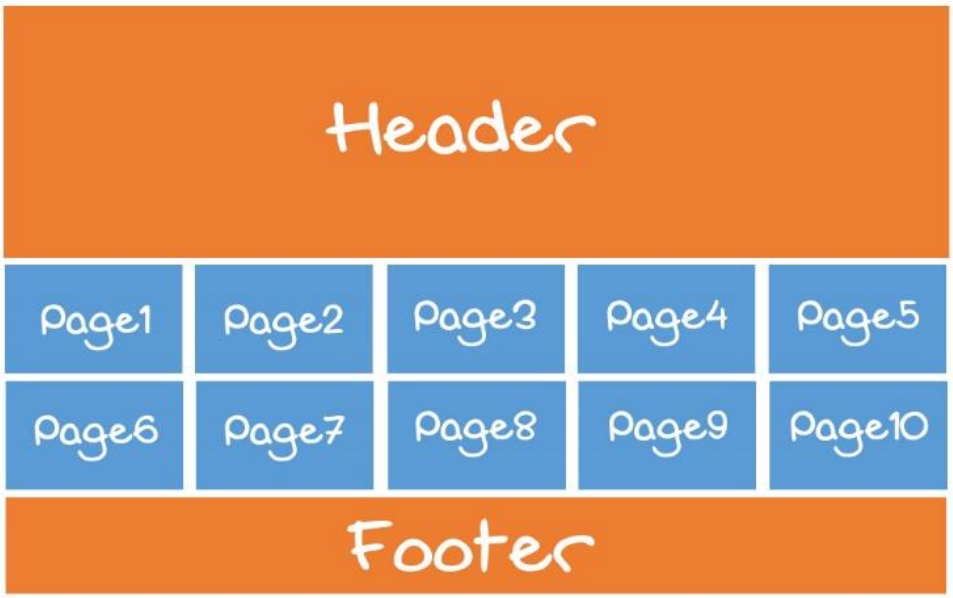
STEP 4 - IMPLEMENTING THE HTML FILE STRUCTURE

If it's not already, open 'website.html' in your chosen coding editor. As with the last exercise, firstly we need to implement our file structure, as shown below.

```
<html>
    <head>

    </head>
    <body>

    </body>
</html>
```

STEP 5 - PAGE LAYOUT DESIGN

At this stage, we need to think about our page layout. It is a good idea to make a block diagram of this, to represent what you want to achieve with your CSS. In my example, I have 10 websites, therefore I have decided on the following page structure:



STEP 6 - IMPLEMENTING THE PAGE LAYOUT WITH HTML

Next, we'll update 'website.html' with a title, a link for the CSS file we'll create, and explore how we are going to implement the page structure for our web menu.

Some of this code you have seen before in the last exercise, so you should already be getting a little familiar and recognising various keywords and mark-up.

In the <head></head> part of your code add the following:

```
<head>
    <title>Fast Access Web Menu</title>
    <link rel="stylesheet" href="css/app.css">
</head>
```

Then, in the <body><body>, add the following:

```
<body>
    <header>
        <h1>Fast Access Web Menu</h1>
    </header>
    <div class="wrapper">
        <div class="">
            <a href="" alt="" title="">
                <img src="" title="" alt="" class="">
            </a>
        </div>
    </div>
    <footer>
       <p>&copy; Copyright [Your Name Here] 2022</p>
    </footer>
</body>
```

Okay, let's examine what we have so far. We have a section named <header></header>. The <header> element represents a container for introductory content or a set of navigational links.

A <header> element typically contains:

- one or more heading elements (<h1> - <h6>)
- logo or icon
- author information

You can have several <header> elements in one HTML document. However, <header> cannot be placed within a <footer>, <address> or another <header> element.

Next, we have a with the class 'wrapper'. This will be used to enclose all of the page links into a division (a page section).

The next part is where the main work gets done.

```
<div class="">
    <a href="" alt="" title="">
        <img src="" title="" alt="" class="">
    </a>
</div>
```

Each website link we add will have this code replicated. I have 10 web pages, therefore I will need to reproduce this 10 times. Once for each of my page links. You will notice that I haven't named the class or added any details to the href, alt title, or src tags yet. That is because for each link that information will be different.

Let's take a deeper look.

We have a with a class. The class will be named appropriately for each page link so that it can be used in the CSS. You will already be familiar with this code if you've completed the last exercise.

Next, we have an <a> anchor tag, otherwise known as a hyperlink or link. The href carries reference information where we want the user to be directed when the link is clicked. Hyperlinks can use internal and external URL's or even an anchor to a particular section of the same web page. We will be using external URL's for third-party websites in our example. We will explore other options with this later in the book.

Next, we have alt="", you will notice that this is also shown within the image tag. This is for alternative text. Or in other words, text that is shown

to the user if, for some reason, the information or image cannot be displayed. This can occur when there is a slow connection or an error with the source file, as an example. Alt tags are also read by screen readers. With hyperlinks, it is good practice to describe where the link directs to. They are not strictly necessary, but it is a highly recommended best practice and forms part of good search engine optimisation (SEO).

The title="", similar to the alt tag, should carry relevant title information. This is displayed when hovered with a mouse. A hyperlink and image carrying these tags are recognised as being well-formed.

Also, we have a class="", in our image tag. This will be used to style our images accordingly.

Notice that the `<img src="" title="" alt="" class="">` is within the <a>. This can be text or, as we've done here, an image. It makes the image clickable. When clicked the user will be redirected to the given URL.

Finally, we have the footer. The <footer> tag defines a footer for a document or section.

A <footer> element typically contains:

- author information
- copyright information
- contact information
- sitemap
- back to the top links
- related documents

You can have several <footer> elements in one document.

We are keeping things simple and just adding copyright information. Update with your details to suit.

Taking all of that on board, update your code accordingly. See my example, below:

```
<html>
    <head>
        <title>Fast Access Web Menu</title>
        <link rel="stylesheet" href="css/app.css">
    </head>
    <body>
        <header>
            <h1>Fast Access Web Menu</h1>
        </header>
        <div class="wrapper">
            <div class="page">
                <a href="" alt="" title="">
                    <img src="" title="" alt="" class="">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="" title="" alt="" class="">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="" title="" alt="" class="">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="" title="" alt="" class="">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="" title="" alt="" class="">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
```

```
                <img src="" title="" alt="" class="">
            </a>
        </div>
        <div class="page">
            <a href="" alt="" title="">
                <img src="" title="" alt="" class="">
            </a>
        </div>
        <div class="page">
            <a href="" alt="" title="">
                <img src="" title="" alt="" class="">
            </a>
        </div>
        <div class="page">
            <a href="" alt="" title="">
                <img src="" title="" alt="" class="">
            </a>
        </div>
        <div class="page">
            <a href="" alt="" title="">
                <img src="" title="" alt="" class="">
            </a>
        </div>
        <div class="page">
            <a href="" alt="" title="">
                <img src="" title="" alt="" class="">
            </a>
        </div>
        <div class="page">
            <a href="" alt="" title="">
                <img src="" title="" alt="" class="">
            </a>
        </div>
    </div>
    <footer>
        <p>&copy; Copyright Garry Owen 2022</p>
    </footer>
</body>
</html>
```

Notice how we now have 10 divs with the class name page, 10 links, and 10 images within those links. In this example, I have chosen to style my

page links the same, it can easily be changed to individual styles, if you prefer.

Okay, so with that explanation absorbed, let's move on.

STEP ❼ - CHOOSING THE COLOUR DESIGN AND IMPLEMENTING THE CSS

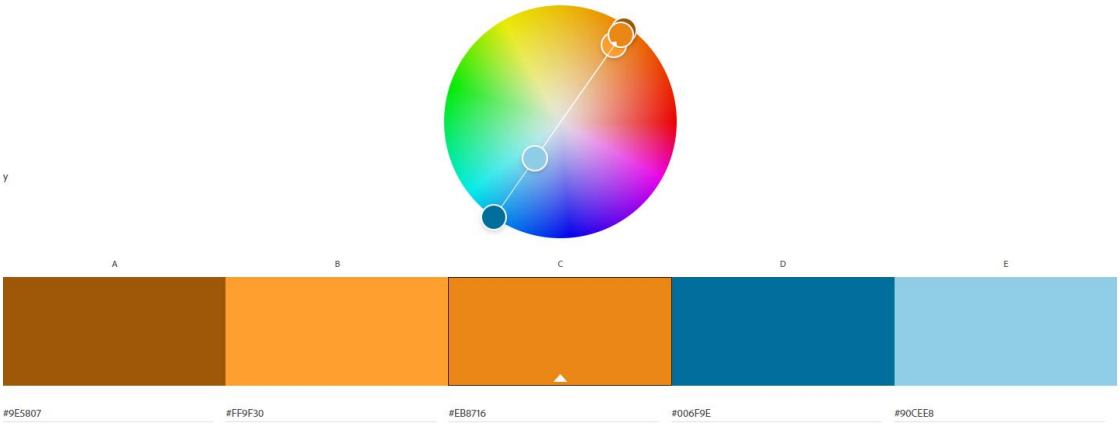
Now it's time to get to grips with our CSS. Open your chosen coding editor and make a new file called 'app.css' and save it in your 'css' folder, within your file structure.

Before we get into that though, let's choose some colours. I've already mentioned that using hexadecimal codes or RGB codes for colours will give far greater options.

Many designers use a colour wheel. It is said that opposite colours generally look good together. There are many examples of these online.

I particularly like the Adobe Colour Wheel Generator. Take a look, via the link below:

https://color.adobe.com/create/color-wheel



Alternatively, choose some colours from this table:

#ac1917	#b75420	#c0982b
#d70e17	#e46828	#f0be39
#e54a50	#ea8553	#eec76b
#768b45	#237f5d	#1c7873
#93ae55	#359d73	#27958f
#a9be77	#6fb293	#4eaaa6
#156b8a	#3f5d82	#704776
#1887ab	#4f73a1	#8c5792
#5d9db9	#728fb4	#926ca0
#874f80	#a15284	#aa3653
#a9629f	#c867a5	#d34467
#b881b1	#d386b7	#dc6986

I'm keeping my colour scheme fairly straightforward.

#EB8716 – for my header

#90CEE8 – for the wrapper around my main content

#006F9E – for my footer

Okay, with that decided let's move on to building the CSS.

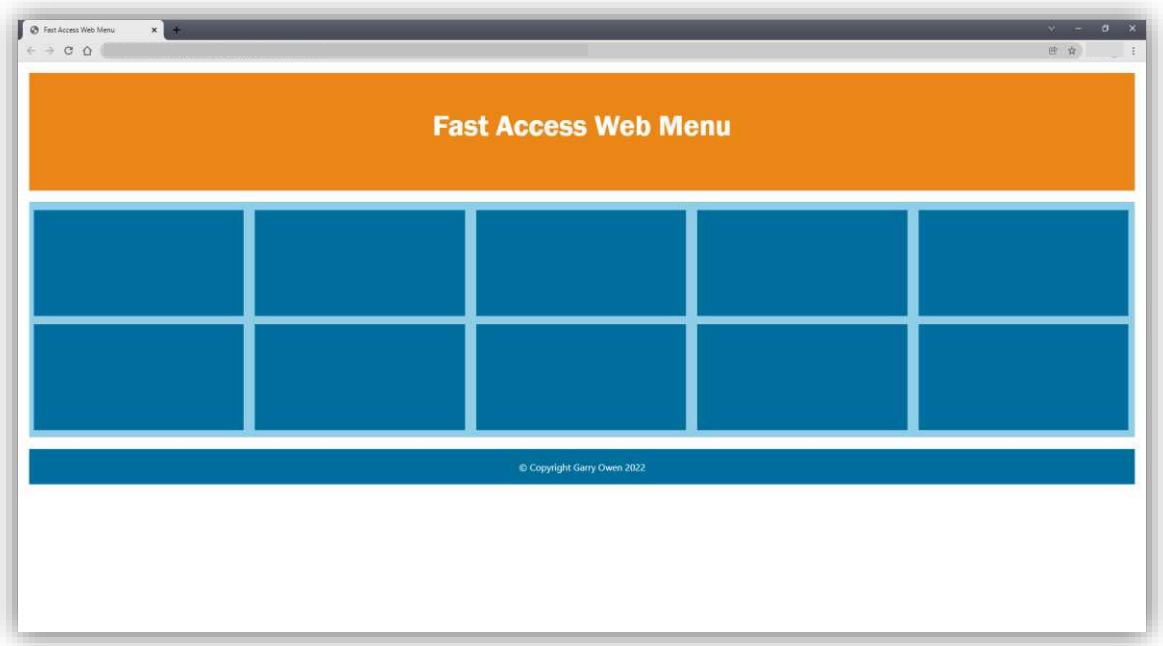
If it's not already, open 'app.css' and add the following code:

```
html, body{
    margin: 0px;
}
header {
    background-color: #EB8716;
    color: #fff;
    width: 98%;
    height: 200px;
    margin: 1%;
}
header h1{
    text-align: center;
```

```
    font-family:'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-
serif;
    padding: 60px;
    font-size: 50px;
}
.wrapper {
    background-color: #90CEE8;
    width: 98%;
    height: 400px;
    margin: 1%;
}
.wrapper .page{
    width: 19%;
    height: 180px;
    background-color: #006F9E;
    display: inline-block;
    margin: 0.4%;
}
footer {
    background-color: #006F9E;
    color: #fff;
    width: 98%;
    height: 60px;
    margin: 1%;
}
footer p{
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    text-align: center;
    padding: 20px;
}
```

Feel free to change the colour hex codes for your own.

When we refresh our page we have the following result:



Let's examine the CSS in more detail.

You can see that we have used two different types of selectors here. The first type is an element selector and the second is a class selector. There is also a third that I should mention and that is an id selector.

So what's the difference?

An id selector uses the form - #id

A class selector uses the form - .class

An element selector uses the form - element e.g. p = style all <p> elements

But it doesn't stop there. Selectors can be joined together to form complex patterns.

You can see a simple example of this here where we have used:

header h1{

This tells the browser rules that apply only to h1 elements within the header. This pattern is straightforward, but selectors can be joined into many forms. We will see more selector patterns as we work through the book. However, if you want to know more about selectors, a simple google search will reveal a host of options.

Most of the rules are repeated, so we'll just examine distinct rules. You can then apply that knowledge to others too. Some of the rules, you have also seen before. That said, let's take a look.

```
    background-color: #EB8716;
//CHANGES THE BACKGROUND COLOUR

    color: #fff;
// CHANGES THE TEXT COLOUR

    width: 98%;
// CHANGES THE WIDTH OF THE ELEMENT

    height: 200px;
// CHANGES THE HEIGHT OF THE ELEMENT

    margin: 1%;
// SETS ALL MARGINS OF THE ELEMENT

    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
// CHANGES THE FONT FACE OF THE ELEMENT
```

```
    text-align: center;
// ALIGNS THE ELEMENT TO THE CENTRE

    padding: 20px;
// ADDS PADDING AROUND THE ELEMENT

    display: inline-block;
// DISPLAYS THE ELEMENT ACROSS THE PAGE AND WRAPS IT AT THE EDGE OF THE
DISPLAY
```

It is recommended for you to experiment with these values and see the results for yourself. That is the best way to learn.

You can rest easy in the knowledge that all of the files are available to download, if you make a mistake you cannot resolve.

STEP 8 - ADDING WELL-FORMED IMAGE ELEMENTS

Okay, next we're going to add the images source information, the alternative text, titles, and image class.

Let's start with the image source URL's.

Open your images folder and add all of your image information to each of the image src="" placeholders, one at a time, in the form:-

'images/[image name].[extension]'

In my case, as an example, they will be:

images/amazon.png images/aol.jpg images/facebook.png images/googlemaps.png images/linkedin.png	images/paypal.png images/santander.png images/twitter.png images/youtube.png images/zoom.jpg

Fill in the alt text and title as appropriate and add ‘image_link’ as the class.

Now your ‘website.html’ file should look something like this:

```
<html>
    <head>
        <title>Fast Access Web Menu</title>
        <link rel="stylesheet" href="css/app.css">
    </head>
    <body>
        <header>
            <h1>Fast Access Web Menu</h1>
        </header>
        <div class="wrapper">
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/amazon.png" title="Amazon"
alt="Amazon Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/aol.jpg" title="AOL Mail" alt="AOL
Mail Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/facebook.png" title="Facebook"
alt="Facebook Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/googlemaps.png" title="Google Maps"
alt="Google Maps Link" class="image_link">
                </a>
            </div>
```

```
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/linkedin.png" title="LinkedIn"
alt="LinkedIn Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/paypal.png" title="PayPal"
alt="PayPal Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/santander.png" title="Santander"
alt="Santander Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/twitter.png" title="Twitter"
alt="Twitter Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/youtube.png" title="Youtube"
alt="Youtube Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="" alt="" title="">
                    <img src="images/zoom.jpg" title="Zoom" alt="Zoom
Link" class="image_link">
                </a>
            </div>
        </div>
        <footer>
            <p>&copy; Copyright Garry Owen 2022</p>
```

```
        </footer>
    </body>
</html>
```

STEP ❾ - ADDING STYLES TO THE IMAGES

Let's make a quick update to our CSS to address the image class we just added.

Open 'app.css' and add the following code to the bottom:

```
.image_link{
    height: 180px;
    width: 18.5vw;
    margin: 1% 0 1% 1%;
}
.image_link:hover {
    filter: grayscale(70%);
}
```

So what's going on here?

```
    height: 180px;
    width: 18.5vw;
```

This is to make all of your images appear uniformly the same size.

```
    margin: 1% 0 1% 1%;
```

This sets the margin with a slight offset to give a kind of 3D drop shadow effect. Remember margin settings are Top, Right, Bottom, and Left (TRBL). I always think of the word 'trouble' to help remember this. So the next time you're having **TR**OU**BL**E trying to remember the order, hey presto!!

The next part has a selector pattern we haven't seen before.

```
.image_link:hover {
```

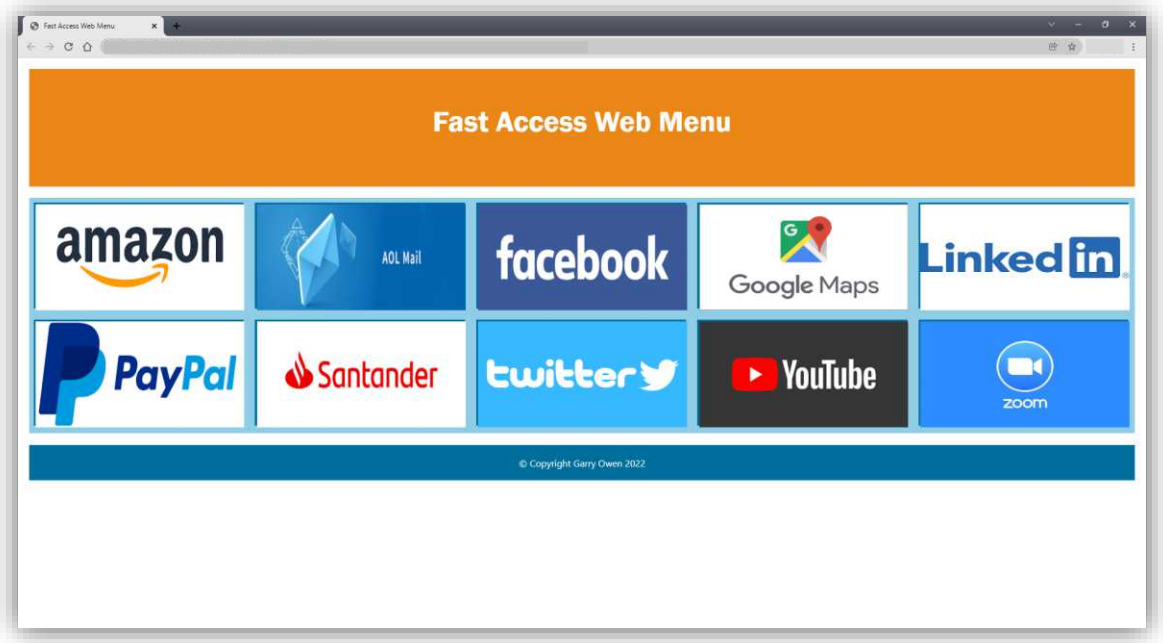
This simply means, when you hover an element, with the class of 'image_link' with a mouse, the rules are applied.

Lastly, we set a rule for when we are hovering the element:

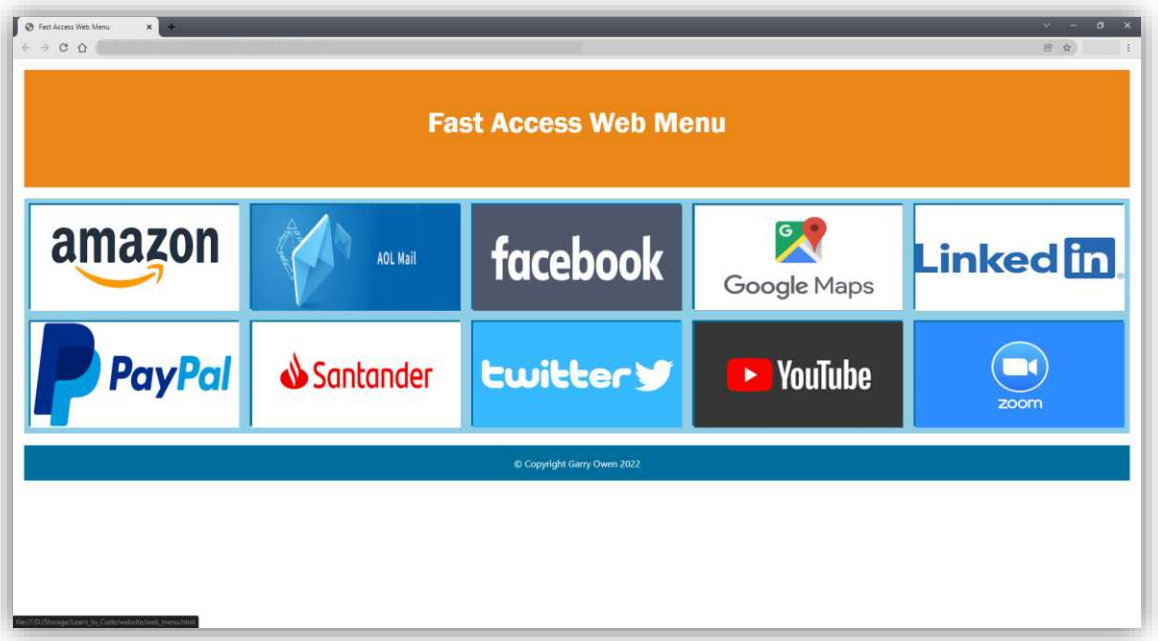
```
filter: grayscale(70%);
```

This sets the colour of the element hovered to 70% greyscale. The same as opacity, you can set this with decimals or percentages. 1 = 100%, 0.7 = 70%.

So, how does it all look now? Let's take a peek.



Looks pretty good. And if we hover over an element, what does that look like?



You can see here, we are hovering over the Facebook link.

Moving swiftly on…

STEP ❿ - IMPLEMENTING WELL-FORMED HYPERLINKS

The final step.

```
<a href="" alt="" title="">
```

All we have left to do now is fill in the URL information in the href="" placeholders, fill in the alternative text, and add our titles. Let's take one example:

```
<a href="https://amazon.co.uk" alt="Visit Amazon" title="Visit Amazon">
```

Now, fill in the rest.

If you don't want to open your links on the same page, you can add a target attribute to your anchor tag: `target="_blank"` This will open the link in a new tab when clicked.

Amazing stuff!!

Now your final 'website.html' mark-up should look like this:

```
<html>
    <head>
        <title>Fast Access Web Menu</title>
        <link rel="stylesheet" href="css/app.css">
    </head>
    <body>
        <header>
            <h1>Fast Access Web Menu</h1>
        </header>
        <div class="wrapper">
            <div class="page">
                <a href="https://amazon.co.uk" alt="Visit Amazon"
title="Visit Amazon">
                    <img src="images/amazon.png" title="Amazon"
alt="Amazon Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="https://mail.aol.com/webmail-std/en-gb/suite"
alt="Visit Aol Mail" title="Visit Aol Mail">
                    <img src="images/aol.jpg" title="AOL Mail" alt="AOL
Mail Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="https:facebook.com" alt="Visit Facebook"
title="Visit Facebook">
```

```
                    <img src="images/facebook.png" title="Facebook"
alt="Facebook Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="https://www.google.com/maps" alt="Visit Google
Maps" title="Visit Google Maps">
                    <img src="images/googlemaps.png" title="Google Maps"
alt="Google Maps Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="https://linkedin.com" alt="Visit LinkedIn"
title="Visit LinkedIn">
                    <img src="images/linkedin.png" title="LinkedIn"
alt="LinkedIn Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="https://paypal.com" alt="Visit PayPal"
title="Visit PayPal">
                    <img src="images/paypal.png" title="PayPal"
alt="PayPal Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="https://santander.co.uk" alt="Visit Santander"
title="Visit Santander">
                    <img src="images/santander.png" title="Santander"
alt="Santander Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="https://twitter.com" alt="Visit Twitter"
title="Visit Twitter">
                    <img src="images/twitter.png" title="Twitter"
alt="Twitter Link" class="image_link">
                </a>
            </div>
```

```
            <div class="page">
                <a href="https://youtube.com" alt="Visit Youtube"
title="Visit Youtube">
                    <img src="images/youtube.png" title="Youtube"
alt="Youtube Link" class="image_link">
                </a>
            </div>
            <div class="page">
                <a href="https://zoom.us" alt="Visit Zoom" title="Visit
Zoom">
                    <img src="images/zoom.jpg" title="Zoom" alt="Zoom
Link" class="image_link">
                </a>
            </div>
        </div>
        <footer>
            <p>&copy; Copyright Garry Owen 2022</p>
        </footer>
    </body>
</html>
```

As was said earlier in the exercise, your version will look a little different, have different links, and may have different colours and fonts chosen, but the functionality should be the same.

I use a similar program to this every day with over 50 of my most visited sites. I have so many sites that I use often, I find the tools that are built into browsers are not as convenient. I hope you can make great use of this going forwards, as I have.

Copy the URL for your menu and put it on your book marks bar for fast access, whenever you need it!

The final result. Have a play around with it. See if you can improve it. Why not add another row of links or add animation? Be imaginative!

All files and images associated with this exercise can be found at:

https://wddtrw/resources/learntocode/fastaccesswebmenu.zip

Make yourself a well-deserved cuppa and let's move on!

CREATING A WEB APP

WHAT IS A WEB APP AND HOW IS IT DIFFERENT FROM A WEBSITE?

A web application also referred to as a web app, is a computer program with functionality and interactive elements. You use regular web technologies to build it but it also stores data and manipulates it according to a user's needs. Users can access it via the internet.

A website, on the other hand, is a collection of publicly accessible pages containing either documents, images, audio, text, or other files that users can access through the internet.

Okay, so they sound quite similar, right? From a user's perspective, essentially they are. They are both accessed via a web browser, you can search google and find either.

In a nutshell, the main differences are:

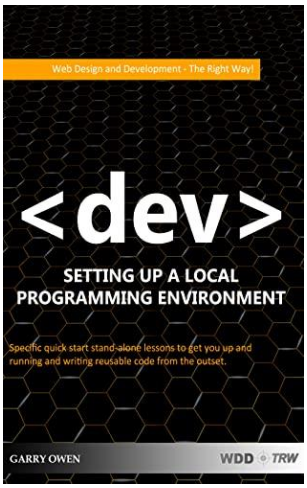
- A website provides information to view and read that cannot be manipulated.
- Authentication isn't mandatory with a regular website.
- Via a website information is publicly accessible – no sign-up required.
- A website uses only basic web technologies.
- It's cheaper to host.

Normally, a web app will be built user server-side technologies, such as PHP or Node.js, but for the purpose of example, we'll be using methods that can be run locally on your machine, in a browser.

Later in this book, I will demonstrate how to deploy your projects onto a remote server and make them publicly accessible.

It is possible to set up a local server to run on your PC or Mac, but that is beyond the scope of this book. If you'd like to learn more about setting up a local programming environment, you can get book 1 in my series 'Web Design and Development – The Right Way!' – Setting up your local programming environment, via the link below:

https://viewbook.at/wddtrw1



LESSON OBJECTIVE:

Design and develop a web-based quiz application with a mobile-first approach.

Build a bank of questions that the user can interactively answer and get their results.

Develop the system to allow users to add or delete questions and include a countdown timer.

Mobile First approach:

A "mobile-first" approach involves designing a desktop site starting with the mobile version, which is then adapted to larger screens (contrary to the traditional approach of starting with a desktop site and then adapting it to smaller screens).

STEP 1 - SETTING UP YOUR FILE STRUCTURE

Here goes nothing, guys! Let's start building our next exciting project. Firstly, let's implement our required file structure.

By now you should be getting familiar with setting up your file structure. This time we need a new folder on your desktop called 'web app'. Within that folder add a 'css' folder, an 'images' folder, a 'js' folder, and a 'json' folder. Finally, open your chosen coding editor and make a new file called 'quiz_app.html'. If you have done everything correctly you should have a folder called 'web app' with the following file structure:

So you've seen the other folders before, but what's that 'json' folder? Normally, we would use a database to allow a user to add and delete data. For this exercise, we are going to mimic that functionality by writing to a JSON file.

JSON-like documents are used as the data structure for a NoSQL text-based database system, called MongoDB. If you want to learn more about that, take a look at the link below. However, we won't be using it here.

https://www.mongodb.com/

WHAT IS A JSON FILE?

- JSON stands for JavaScript Object Notation
- JSON is a text format for storing and transporting data
- JSON is "self-describing" and easy to understand
- JSON is language-independent – meaning it can be used in many programming languages – therefore, it's easy to pass data between those different languages

A simple example of a JSON string would look like this:

```
{"name":"Garry Owen", "age":47, "car":"BMW"}
```

Simple, right? Notice, text strings are in double-quotes, while numbers aren't. This is so numbers can be recognised as values and allow us to use them in calculations. If that doesn't make sense, don't worry too much about that for now. We will look at different data types later and explain this in more depth.

This structure is called key-value pairs. A key-value pair consists of two related data elements: A key, which is a constant that defines the data set, and a value, which is a variable that belongs to the set.

E.g. "name":"Garry Owen" – "Key":"Value"

The JSON string defines 3 properties (name, age, and car) and each property has a value. If you parse the JSON string with JavaScript, you can access the data as an object, like this:

let personName = obj.name;

let personAge = obj.age;

We can now access that information and use it in the program.

WHY USE JSON?

The JSON format is syntactically similar to JavaScript code for constructing objects. Therefore, we can easily convert JSON data into JavaScript. Even better, JavaScript has a built-in function for converting JSON strings into

objects, and JavaScript also has a built-in function for converting a JavaScript object into JSON. Top class!

Syntactically:

In a way that relates to the structure of statements or elements in a computer language

JSON to JavaScript looks like this:

```
JSON.parse()
```

While, JavaScript object to JSON, looks like this:

```
JSON.stringify()
```

So let's put that all together and take a look at how we would pass our JSON string into a JavaScript object.

```
<html>
<head></head>
<body>
<h2>Convert JSON string into a JavaScript object.</h2>
<div id="result"></div>

<script>
//JSON String
const person = '{"name":"Garry Owen", "age":47, "car":"BMW"}';

//Make a new JavaScript Object
const obj = JSON.parse(person);

//Display the results in the div
document.getElementById("result").innerHTML = obj.name + " is " +
obj.age+ " and drives a " + obj.car;

</script>
</body>
</html>
```

If we take a quick look we can see that we have placed our JSON string into a constant called person. Then we have created a new JavaScript object and parsed our JSON 'person' string into it and displayed the results in our , by selecting each of the key-value pairs by referencing the key properties of our object (obj).

This results in:

You're doing great! Let's move on…

STEP ❷ - IMPLEMENTING THE HTML STRUCTURE

You should be fairly familiar with the HTML structure by now. Before we add any trimmings, add the mark-up below to our 'quiz_app.html' file.

```
<html>
    <head>

    </head>
    <body>

    </body>
</html>
```

With that done, let's add some meat on the bones.

How is this web app going to function?

1. We want our web app to have a mobile-first approach – so it's got to fit nicely and look good on a mobile phone.
2. We want to use JavaScript to fetch questions from a JSON text file.
3. We want the user to be able to interact and answer the questions and get their results.
4. We want the user to be able to add or delete questions and update the JSON file.

So, let's take a look at a rough layout.

Okay, so first we want to style the layout with a header, footer and content attributes. We have a question and 4 possible multi-choice

answers and prevision for a previous and next button. With that decided let's add some mark-up to our HTML file.

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="user-scalable=no, width=device-width"
/>
    <link rel="stylesheet" type="text/css" href="css/app.css">
    <title>Question Time</title>
    <script type="text/javascript"
src="json/questions.json"></script>
<body>
    <div id="header">
        <h1>Question Time<br>The Quiz App</h1>
    </div>
    <div class="quiz-wrapper">
      <div id="quiz"></div>
    </div>
    <button id="previous"><p>Previous Question</p></button>
    <button id="next"><p>Next Question</p></button>
    <button id="submit"><p>Submit Quiz</p></button>
    <div id="results"></div>
    <div id="footer">
        <h5>Written by Garry Owen &copy Copyright December 2021</h5>
    </div>
    <script src="js/app.js"></script>
</body>
</html>
```

Okay, so we've got our basic mark-up. Let's examine it more closely. Much of this mark-up you have seen before, but we have a few new tags in there too.

The first line of code we hit inside the head is:

```
<meta name="viewport" content="user-scalable=no, width=device-width" />
```

This line tells the browser that on a touch screen device the user shouldn't be able to zoom in and out. This is intentional so that the web app fits nicely on a phone screen. It also set the width of the web page to the device width. That will mean that it will fill 100% of the width of a device. By default, the user is allowed to scale the screen, but they won't need to for our web app because it'll be easy to read and navigate.

Next we have:

```
<link rel="stylesheet" type="text/css" href="css/app.css">
<title>Question Time</title>
```

You should be familiar with these two lines by now. The first connects the external CSS file and the next gives the page its title.

Then, just inside the <body> tag we have:

```
<div id="header"><h1>Question Time<br>The Quiz App</h1></div>
```

This opens a new div with the id of header and adds a <h1> title within the header.

Then, we have:

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

This is similar to what we did to pull in / reference the JavaScript file, in our first exercise. The only difference this time is that we are referencing JSON.

For good search engine optimisation, you should only ever have one <h1> title on any given web page. <h1> is seen as being the most important and with each decrement i.e. <h2>… <h3>… and so on, the algorithm of a search engine deciphers it as less and less important.

Okay, next we have two nested tags.

```
<div class="quiz-wrapper">
  <div id="quiz"></div>
</div>
```

This is where we will display our quiz. Note we have given one div a class of quiz-wrapper and the other an id of quiz.

Nested:

One element can be placed inside another element. Nesting allows you to apply multiple HTML tags to a single piece of content.

Next, we have three <button> tags. These will perform our user interaction requirements for our quiz.

```
<button id="previous">Previous Question</button>
<button id="next">Next Question</button>
<button id="submit">Submit Quiz</button>
```

Each one is given an id that represents its purpose.

Then, we have another div where we will display our results.

```
<div id="results"></div>
```

Next, we have our footer. Change the information here to reflect your name and the correct date.

```
<div id="footer">
    <h5>Written by Garry Owen &copy Copyright December 2021</h5>
</div>
```

Notice that we have used a <h5> tag. This will display the text as smaller text and set the hierarchy for our search engine, as explained previously.

Finally, just before we close the <body>, we have a connection to our 'app.js' file. Sometimes we can place this in the <head></head> part of our HTML. It is important in this case that it is placed just inside the closing body tag.

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

This is because the order in which the code is read is paramount. HTML loads from top to bottom. The head loads first, then the body, and then everything inside the body. We are making sure it is read last after we have declared our HTML tags with various classes and ids. If we put it in the head, the JavaScript wouldn't recognise those attributes, as they wouldn't have been read by the browser yet.

STEP ❸ - BUILDING THE QUESTION BANK JSON FILE

Building our JSON question bank is fairly straightforward. Each question needs the question itself, four possible answers, and the correct answer. Let's take a simple empty JSON structure with these elements:

```
{
  question: "",
   answers: {
        a: "",
        b: "",
        c: "",
        d: ""
        },
  correctAnswer: ""
}
```

As our question bank will have a number of questions, we can structure each question on one line, if preferred, like this:

```
{question: "", answers: {a: "", b: "", c: "", d: ""}, correctAnswer: ""},
```

That will keep our file much shorter. For the purpose of our web app we'll call all of the questions 'data', by declaring the name and wrapping the whole thing in square brackets, like this:

```
data = [
  {question: "", answers: {a: "", b: "", c: "", d: ""}, correctAnswer: ""},
  {question: "", answers: {a: "", b: "", c: "", d: ""}, correctAnswer: ""},
  {question: "", answers: {a: "", b: "", c: "", d: ""}, correctAnswer: ""},
  {question: "", answers: {a: "", b: "", c: "", d: ""}, correctAnswer: ""},
  {question: "", answers: {a: "", b: "", c: "", d: ""}, correctAnswer: ""}
];
```

This demonstrates how 5 questions would look. Notice that the last question has the comma removed from the end.

I'm going to add 20. You can add as many or as few as you like.

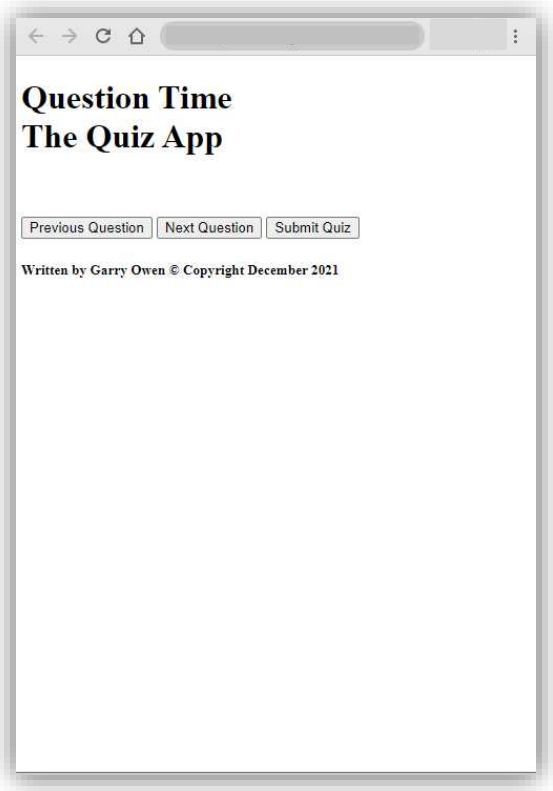
Take your time and add your questions, as below before moving on to the next step. Note! You can download my questions via the link at the end of this section, if preferred.

```
  {
      question: "The Plaka is the oldest quarter of which city?",
      answers: {
            a: " Athens",
            b: "Prague",
            c: "Rome",
            d: "Vienna"},
      correctAnswer: "a"
   },
```

I have laid it out here so that it is easier for you to see what I have done. Of course, if you prefer to keep the layout like this, then that's great too!

STEP❹ - ADDING SOME STYLE

If you take a look at how our web app looks at this stage, it's very plain and dull. The elements are there, but it doesn't have the look and feel of a web app just yet.



Next, make a new file called 'app.css' and save it in the 'css' folder. Referring back to our layout diagram and our HTML we need to style the following elements.

body, header, h1, quiz-wrapper, quiz, rad_butn, button, previous, next, submit, results and footer.

Let's build it!

Add the following code to your 'app.css' file and save.

```
body{
  margin: 0;
  padding: 0px;
  font-family: Arial;
  font-size: 20px;
  text-align: center;
  background-color: #eee;
  overflow-x: hidden;
}
#header{
    border-bottom: 1px solid #666;
    color: #222;
    display: block;
    padding: 10px 0;
    text-align: center;
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#ccc), to(#999));
}
#header h1{
  text-shadow: 0px 1px 0px #fff;
  font-weight: 300;
  margin: 0px;
  padding: 10px;
  font-size: 25px;
}
.question{
  font-size: 30px;
  margin: 0 5% 20px 5%;
}
.answers {
  margin-bottom: 20px;
  text-align: left;
  display: inline-block;
}
.answers label{
  display: block;
  margin-bottom: 10px;
```

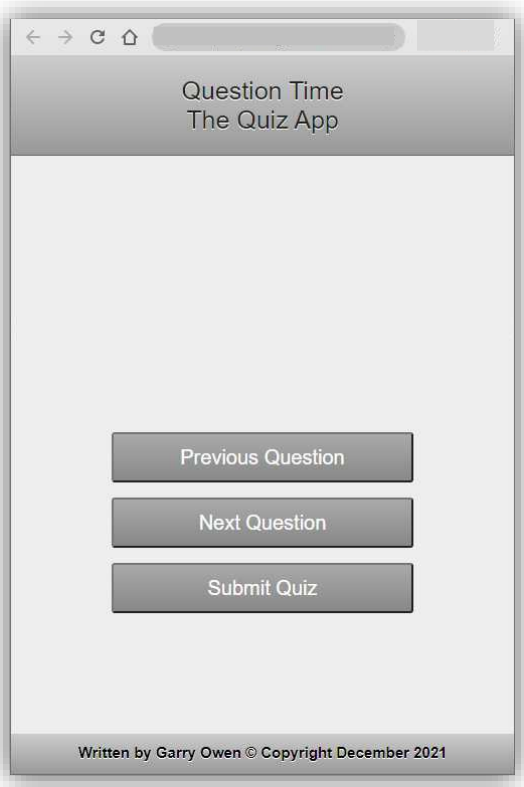
```
}
button p{
    margin-top: -8px;
}
button{
    font-weight: 200;
    font-size: 20px;
    width: 300px;
    height: 50px;
    padding: 20px;
    cursor: pointer;
    border-radius: 3px;
    color: #fff;
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#aaa), to(#888));
    margin-top: 15px;
}
button:hover{
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#bbb), to(#999));
}
.quiz-wrapper{
  position: relative;
  height: 200px;
  margin: 30px 0 60px 0;
}
.rad_butn {
    transform: scale(200%);
    margin: 0 30px 0 -20px;
}
#results{
    margin-top: 15px;
    padding: 20px;
}
.slide{
  position: absolute;
  left: 0px;
  top: 0px;
  width: 100%;
```

```
  z-index: 1;
  opacity: 0;
  transition: opacity 0.5s;
}
.active-slide{
  opacity: 1;
  z-index: 2;
}
#footer {
    position: absolute;
    bottom:0;
    width:100%;
    height:40px;
    text-shadow: 0px 1px 0px #fff;
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#ccc), to(#999));
 }
 #footer h5{
     text-align: center;
     margin-top: 10px;
     font-size: 15px;
 }
```

At first glance, this looks quite long and complex, but you'll soon see when we break it down that each element is straightforward.

Just a quick reminder, you don't have to type all of this in. All files are available to download. A download link is at the end of every exercise.

Before we break it down, let's take a look at the result, so that we can visualise our expected result so far. However, keep in mind that the result will look a little different because we will also be manipulating our CSS with our JavaScript code in the next section.



It's looking much different to our initial preview.

So let's take a look at the CSS. We'll break this into elements, as we go. Starting off with the <body> element rules.

```
body{
  margin: 0; // Set all margins to 0px (TRBL)
  padding: 0px; // Set padding to 0px (Space around an element's content)
  font-family: Arial; // Set font to Arial font face
  font-size: 20px; //Set the font size to 20px
  text-align: center; //align text to the centre
  background-color: #eee; // set the background colour to off white
  overflow-x: hidden; // prevent scrolling on the X axis
}
```

Okay, aside from the overflow-x command, if you've been following through the book step-by-step, you will have seen all of these commands before.

In a nutshell, if the content is taller or wider than the screen, your browser will automatically add scrollbars. You can prevent both of these by adding the overflow: hidden command for both the X and Y-axis. Or you can control each axis independently with overflow-x and overflow-y. The hidden command is not the only option. Choose from:

- `visible` - Default. The overflow is not clipped. The content renders outside the element's box
- `hidden` - The overflow is clipped, and the rest of the content will be invisible
- `scroll` - The overflow is clipped, and a scrollbar is added to see the rest of the content
- `auto` - Similar to `scroll`, but it adds scrollbars only when necessary

On to the next section, the header:

```
#header{ //These rules apply to all elements within the header div
    border-bottom: 1px solid #666;
// Display a solid border on the bottom edge of the element
    color: #222;
// Make the text colour dark grey
    display: block;
// Display the element as a block. Starts a new line and takes up the
whole width
    padding: 10px 0;
// Set padding top and bottom to 10px and right and left to 0
    text-align: center;
// Align text to the centre
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#ccc), to(#999));
// Create a gradient light-grey to dark-grey, from top to bottom
```

```
}
#header h1{ //These rules only apply to <h1> elements within the header
div
  text-shadow: 0px 1px 0px #fff; // text shadow - offset-x | offset-y
                                    | blur-radius | colour
  font-weight: 300; // Set font thickness to thin (900 max)
  margin: 0px; // Set all margins to 0px (TRBL)
  padding: 10px; // Set padding to 10px
  font-size: 25px;// Set the font size to 25px
}
```

On to the next section, the questions and answers:

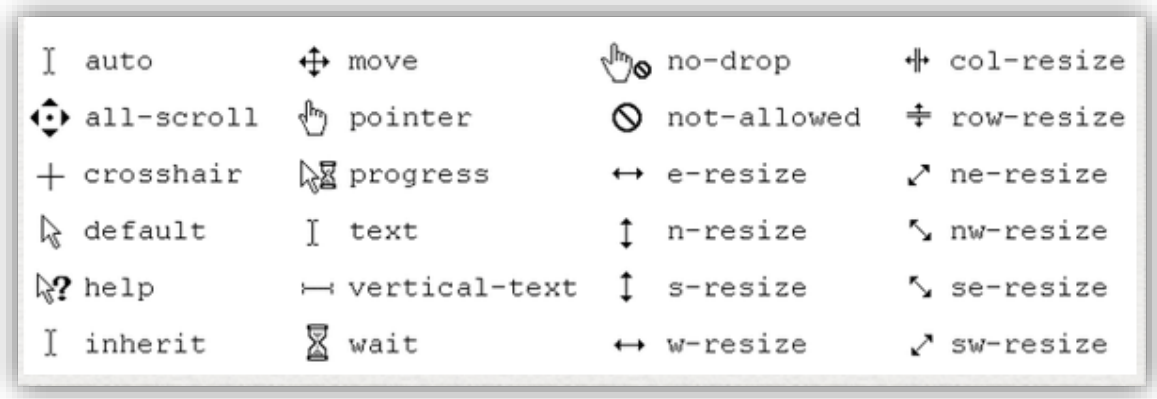
```
.question{
  font-size: 30px;
  margin: 0 5% 20px 5%;
}
.answers {
  margin-bottom: 20px;
  text-align: left;
  display: inline-block;
}
.answers label{
  display: block;
  margin-bottom: 10px;
}
```

All of these commands you have seen above. These won't do anything yet but will affect the questions and answers layout. They will be used with JavaScript in the next section. More on that later.

```
button p{ // This only affect elements inside the button and p tags
    margin-top: -8px; //Moves button text up 8px
}
button{ // This affects all elements within the button tags
    font-weight: 200; // Font thickness
    font-size: 20px; // Font size in pixels
    width: 300px; // Width of the buttons
    height: 50px; // Height of the buttons
    padding: 20px; // Padding around the elements content
```

```
    cursor: pointer; // Change the cursor to a pointer – more settings
                        explained over the page
    border-radius: 3px; // Set all corners of the element to a radius of
                        3px
    color: #fff; // Set text colour to white
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#aaa), to(#888)); // Set background of the button to a gradient
    margin-top: 15px; // Make a 15px space above all button elements
}
button:hover{ // This affects only button elements hovered with the mouse
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#bbb), to(#999)); // Change the background colour to a lighter
                          gradient
}
```

The cursor CSS command has many settings for different purposes. See the diagram below for details:



Simply change to suit your project.

On to the next section, the quiz wrapper and results:

```
.quiz-wrapper{ // Applies to everything within the quiz-wrapper class
  position: relative; // Position element relative to its normal position
  height: 200px; // Set element to 20px tall
```

```
  margin: 30px 0 60px 0; // Set margins (TRBL)
}
.rad_butn { // Applies to everything with the rad_butn class
    transform: scale(200%); // Double the default size of the element
    margin: 0 30px 0 -20px; // Set margins (TRBL)
}
#results{ // Applies to everything within the results div
    margin-top: 15px; // Set top margin to 15px
    padding: 20px; // Set padding to 20px
}
```

On to the next section, question slides:

```
.slide{
  position: absolute;
  left: 0px;
  top: 0px;
  width: 100%;
  z-index: 1;
  opacity: 0;
  transition: opacity 0.5s;
}
.active-slide{
  opacity: 1;
  z-index: 2;
}
```

Each question is separated and displayed independently. We will use this with JavaScript to fade each question in and out. More on that later.

Finally, the footer:

```
#footer { // Applies to everything that falls within the footer section
    position: absolute; //position the element precisely
    bottom:0; // Set the element position to 0px from the bottom of the
                parent element (in our case the body)
    width:100%; // Set the width of the element to 100%
    height:40px; // Set the height of the element to 40px
    text-shadow: 0px 1px 0px #fff; // text shadow - offset-x | offset-y
                                   | blur-radius | colour
```

```
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#ccc), to(#999)); // Set background colour to a gradient image
 }
 #footer h5{ // Applies only to <h5> elements within the footer
     text-align: center; // Align text to the centre
     margin-top: 10px; // move element down 10px
     font-size: 15px; // Set font size to 15px – this over-rides the
                          default size for <h5> elements
 }
```

You're doing great! Before you move on to the next step have a well-deserved break.

STEP 5 - BUILDING THE JAVASCRIPT ENGINE

Okay, so hopefully you've had a pleasant break and you're now ready to face the next part of this exercise. We're going to tackle this in parts. We'll break it down into functions. Each function will perform a particular task and together they will form the engine to allow the user to interact with the quiz and get their results.

Open your chosen coding environment and make a new file called 'app.js', and save it in the 'js' folder.

Let's do a quick recap of our HTML structure for our quiz. The main three elements we're interested in here are:

```
<div id="quiz"></div>
<button id="submit"><p>Submit Quiz</p></button>
<div id="results"></div>
```

In our JavaScript we can select these elements and make references to them like so:

```
const quizBox = document.getElementById('quiz');
const resultsBox = document.getElementById('results');
const submitButton = document.getElementById('submit');
```

What's the purpose `const`?

In JavaScript, you can set variables in different ways. Each way has slightly different properties. However, the differences are mainly focussed on the scope of the variable. Scope refers to the visibility of variables. In other words, which parts of a program can see or use it.

Other than scope, in a nutshell:

- `var` variables can be updated and re-declared within their scope

- `let` variables can be updated but not re-declared
- `const` variables can neither be updated nor re-declared

JavaScript has 3 types of scope:

- Block scope
- Function scope
- Global scope

Before 2015, JavaScript only had Global Scope and Function Scope. JavaScript ES6 introduced two new important keywords: `let` and `const`.

Block Scope

Variables declared inside a { } block cannot be accessed from outside the block.

Function Scope

Variables declared within a JavaScript function, become local to the function and have Function Scope.

Global Scope

A variable declared outside a function, becomes global and therefore has Global Scope.

Next, we need to build the quiz, render the results and allow user interaction. We can start by laying out the functions and then filling them in as we go.

```
function buildQuiz(){}
function showResults(){}
buildQuiz();
```

```
submitButton.addEventListener('click', showResults);
```

Here we have a function to build the quiz and another to show the results. We are calling or invoking the buildQuiz function with buildQuiz(), and when the user clicks on the submit button we are calling or invoking the showResults function.

Okay, with that understood let's take a look at how we are going to use our JSON question bank. In our HTML we have included the 'questions.json' file. As we saw earlier we wrapped that in square brackets and gave the whole dataset the name 'data'. With that in mind it is now easy to bring all of our questions into a JavaScript array, like so:

```
const myQuestions = [];
 for(i=0; i < data.length; i++){
     myQuestions.push(data[i]);
 }
```

Let's examine what we've done here. Firstly, we make a new empty array.

```
const myQuestions = [];
```

Secondly, we grab all of the information from our JSON dataset and push it into our new array. To do this we have used a For Loop. A For Loop executes a block of code as long as the specified condition is true.

Array:

An array is a data structure, which can store a fixed-size collection of elements of the same data type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

A For Loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

In our case:

```
for(i=0; i < data.length; i++){
```

This breaks down into:

`i=0;` - 'i' has its initial value set to 0.

`i < data.length;` - when 'i' is less than the length of data

`i++` - Increment i by 1 each time the code block is executed

For each loop iteration the following code block is executed:

```
        myQuestions.push(data[i]);
```

This line of code pushes the next question into our array. The loop cycles until it reaches the end of the dataset. In our case called 'data'.

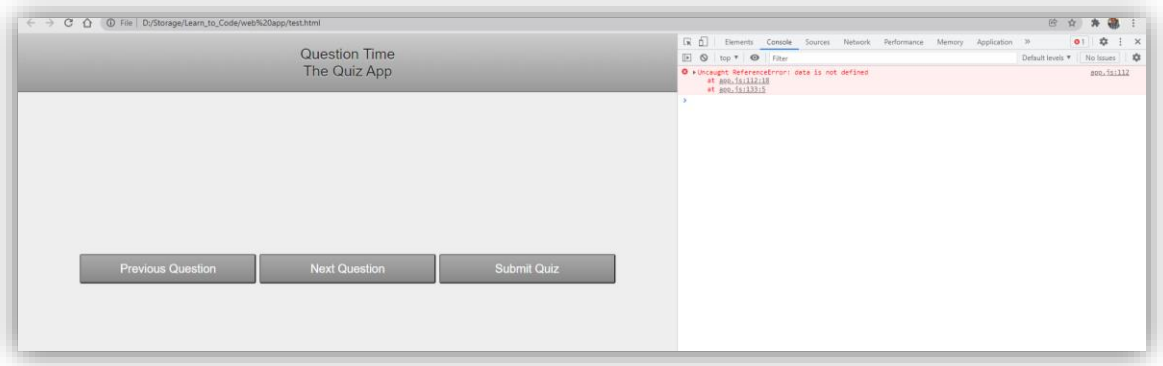
STEP ❻ - THE BUILD QUIZ FUNCTION

Add the following code to your 'app.js' file and save it.

```
function buildQuiz(){}
function showResults(){}

    const quizBox = document.getElementById('quiz');
    const resultsBox = document.getElementById('results');
    const submitButton = document.getElementById('submit');

    const myQuestions = [];
     for(i=0; i < data.length; i++){
         myQuestions.push(data[i]);
     }

buildQuiz();
submitButton.addEventListener('click', showResults);
```

This structure, the structure of your code, is super important. Please ensure you follow it closely. Due to a behaviour of JavaScript called 'The Temporal Dead Zone'. I know this sounds like a sci-fi phrase, but it is something that occurs in JavaScript when you try to access a variable or an array before it has been declared or defined.

If at any point your code doesn't behave the way you expect, you can take a look at your browser console and get some error information. You can access it via menu options in your browser or, in Google Chrome, for example, use shortcut keys 'CTRL + Shift + I'. Below is an example of a Google Chrome Console error relating to a Temporal Dead Zone.



Okay, now replace the `function buildQuiz(){}` with the following function:

```
function buildQuiz(){

 const output = [];
 myQuestions.forEach(
     (currentQuestion, questionNumber) => {
     const answers = [];
     for(letter in currentQuestion.answers){
         answers.push(
         `<label>
```

```
                <input type="radio" name="question${questionNumber}"
value="${letter}" class="rad_butn">
                ${letter} :
                ${currentQuestion.answers[letter]}
            </label>`
            );
        }
          output.push(
            `<div class="slide">
              <div class="question"> ${(questionNumber+1)}.
${currentQuestion.question} </div>
              <div class="answers"> ${answers.join("")} </div>
            </div>`
          );
        }
    );
    quizBox.innerHTML = output.join('');
    }
```

With that done, let's take a closer look at what each line of the 'buildQuiz' function does.

First, we create an empty array and store it in a variable called 'output' to contain all of the HTML output, including questions and user answer choices.

```
const output = [];
```

Next, build the HTML for each question, looping through each question like so:

```
myQuestions.forEach(
          (currentQuestion, questionNumber) => {
//Code block to be executed here
}
```

We're using an arrow (=>) function to perform our operations on each question. Because this is in a forEach loop, we get the current value, the index (the position number of the current item in the array), and the array itself as parameters. We only need the current value and the index, which for our purposes, we'll name currentQuestion and questionNumber respectively.

Now let's look at the code block we want to execute inside our loop. For every question, we want to generate the correct HTML, and so our first step is to create an array to hold the list of possible answers.

```
const answers = [];
```

Next, we'll use a loop to fill in the possible answers for the current question.

For each possible answer, we'll create a radio button, which we're enclosing in a <label> element. This is so that users can click anywhere on the answer text to select that answer. If the label was omitted, then the users would have to click on the radio button itself.

However, you may recall earlier that we made our radio buttons larger. This will improve overall usability, especially because we are designing this with a mobile-first approach in mind.

```
for(letter in currentQuestion.answers){
    answers.push(
    `<label>
        <input type="radio" name="question${questionNumber}"
value="${letter}" class="rad_butn">
        ${letter} :
        ${currentQuestion.answers[letter]}
    </label>`
    );
```

```
}
```

Once we have our list of answer buttons, we can push the question HTML and the answer HTML onto our overall list of outputs. Let's break it down some more.

```
for(letter in currentQuestion.answers){
 // Code block to execute

}
```

You'll recognise the For Loop from before. However, this is in a slightly different arrangement and is known as a For…In Loop.

It has the syntax:

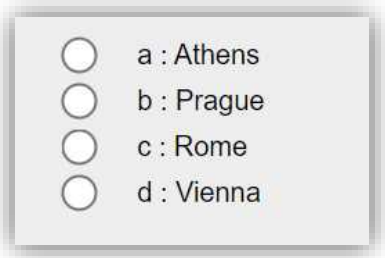
```
for (property in object) {
  // Code block to be executed
}
```

Then, the same as with our JSON file, we need to push the current question answers into the array.

```
answers.push( DATA TO PUSH INTO ARRAY );
```

This adds a block of data into the array. In our case, we are adding:

- A radio button
- The letter associated with the answer
- The answer associated with that letter



All wrapped neatly inside a <label></label> tag.

```
    `<label>
        <input type="radio" name="question${questionNumber}"
value="${letter}" class="rad_butn">
        ${letter} :
        ${currentQuestion.answers[letter]}
    </label>`
```

The join expression takes our list of answers and puts them together in one string that we can output into our answers div.

Next, we have to bring it all together and push the question and possible answers into the output array.

```
          output.push(
            `<div class="slide">
              <div class="question"> ${(questionNumber+1)}.
${currentQuestion.question} </div>
              <div class="answers"> ${answers.join("")} </div>
            </div>`
          );
```

Notice here we have wrapped our output in a with the class of slide. We are using this to separate each of the questions, instead of having them presented as a sequential list. More on that later.

Let's break it down:

Inside our slide we first have our question, presented in its own div with the class of question.

```
<div class="question"> ${(questionNumber+1)}. ${currentQuestion.question}
</div>
```

Inside that division, we have our question number `${(questionNumber+1)}.` followed by a point and a space. You will notice we have added 1 to the questionNumber value. This is because

questionNumber represents the position index of the data in the array, and as the array index starts at zero (0), we have added 1 to make our question numbers start from 1. The dot or point and space is so that each question number is displayed in the desired format I.E. 1. [question text?], 2. [question text?] and so on. Next, we have the question text itself - `${currentQuestion.question}`, which forms the overall desired numbered question format, as below:

1. The Plaka is the oldest quarter of which city?

Next, we grab the data from the answers array, like so:

```
<div class="answers"> ${answers.join("")} </div>
```

And join it together with the question.

Now that we've generated the HTML for each question and answer set, we can join it all together and show it on the page:

```
quizContainer.innerHTML = output.join('');
```

Inside each of our push commands, we have wrapped our code block with back ticks (`). In JavaScript, this allows code to be spread over several lines. This is much the same as how we can use /*…*/ for wrapping comments over several lines. This is super important. Double quotes (") and single quotes (') only allow code or data to be presented over a single line.

Template Literals

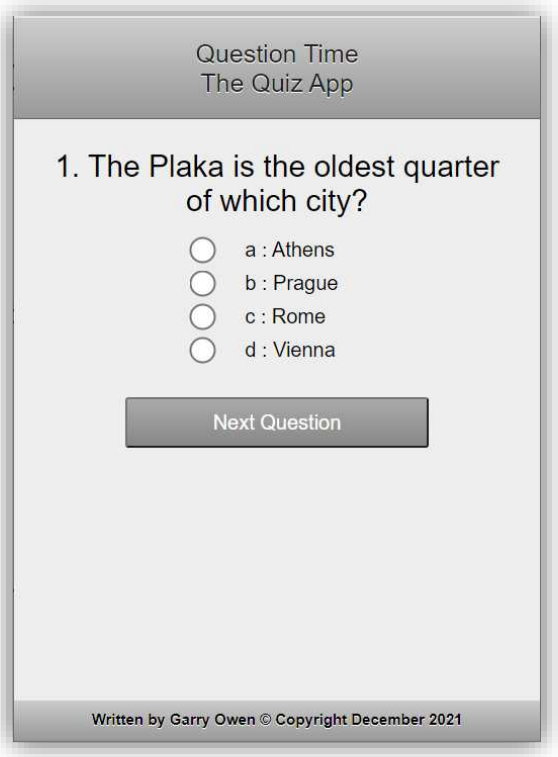
`` `${expression}` `` A template literal is an easy way to interpolate variables into strings. Interpolation is when you add variables within a string. E.g `` `Welcome ${firstName} ${lastName}!` ``. As shown in the example here, template literals can only be used in JavaScript when the

string is enclosed with back ticks. Informally they are sometimes called template strings.

Their use is very straightforward. After setting any variable you can use it within a string.

```
let age = 37;
let text = `John is aged ${age} years old.`;
console.log(text);
//output - John is aged 37 years old
```

At this point, you should be able to run the quiz and see your questions displayed.



You're doing great!

Before we move on to the next step, let's wrap our heads around arrays a little better.

WHAT IS AN ARRAY?

The simplest way to describe it is, an array is a special variable, which can hold more than one value. Okay, doesn't sound too complicated, right?

So a variable can hold one value and an array can hold multiple values. Great! So why use an array? Why not just store everything as variables?

When you are using a handful of values, for instance:

```
let colour1 = "Red";
let colour2 = "Green";
let colour3 = "Blue";
```

Variables could work just fine. However, now imagine you need to use 500 values or even 1000. Our list would be very long and all of the variable names would be difficult to remember.

An array can hold many values under a single name, and you can access the values by referring to an index number.

For example:

```
const colours = ["Red", "Green", "Blue"];
```

It is a common practice to declare arrays with the const keyword.

Now if we want to access any of our colours we only have to remember one variable name, instead of three. Each of the values can be accessed using its index number.

```
let colour = colours[0]; //Index position 0 is "Red"
```

The index always begins at 0 and for each additional value added to the array, the index will increment by 1.

It is also easy to change the values in an array by simply assigning a new value, like so:

```
colours[0] = "Pink";
```

Now our array would look like this:

```
const colours = ["Pink", "Green", "Blue"];
```

There's loads of other cool stuff we can do with arrays.

Finding the length of an array:

```
console.log(colours.length);// Returns a value of 3
```

Finding items in an array:

```
console.log(colours.indexOf("Blue"));   //  Returns a value of 2
```

Adding items.

```
colours.push("Black");//Adds "Black" to the end of the array
colours.unshift("Black");//Adds "Black" to the beginning of the array
```

Let's stick with adding "Black" to the end. Now we have:

```
const colours = ["Pink", "Green", "Blue", "Black"];
```

Removing items:

```
colours.pop();//Removes last item
colours.shift();//Removes first item
colours.splice(index, 2);//Removes item at index 2
```

Accessing every item:

```
const colours = ["Pink", "Green", "Blue", "Black"];
for (const colour of colours) {
    console.log(colour);
}
```

Splitting an array, in this case by the (,) delimiter:

```
const myData = colours.split(',');
```

Joining an array:

```
const myNewString = colours.join(',');

//Or another simpler way

colours.toString();
```

The output returned from both of these is the same:

```
Pink, Green, Blue, Black
```

There is one final point I wish to make about arrays, in our example, each item is a string, but in an array, we can store various data types:

- strings
- numbers
- objects
- and even other arrays

We can also mix data types in a single array — we do not have to limit ourselves to storing only numbers in one array and in others only strings.

There is much more we could look at regarding arrays, but that is beyond the scope of this book. Hopefully now though, you should feel clearer about arrays and their use.

Before you move on to the next step, have a quick break.

STEP ❼ - THE SHOW RESULTS FUNCTION

Let's fill in the code we need for our showResults function. Update your function with the code below and save (CTRL + S):

```
function showResults(){
    const answerBoxs = quizBox.querySelectorAll('.answers');
    let numCorrect = 0;
    myQuestions.forEach( (currentQuestion, questionNumber) => {
    const answerBox = answerBoxs[questionNumber];
    const selector = `input[name=question${questionNumber}]:checked`;
    const userAnswer = (answerBox.querySelector(selector) || {}).value;
    if(userAnswer === currentQuestion.correctAnswer){
        numCorrect++;
        answerBoxs[questionNumber].style.color = 'green';
    }
    else{
        answerBoxs[questionNumber].style.color = 'red';
    }
    });
    resultsBox.innerHTML = `${numCorrect} out of ${myQuestions.length}`;
}
```

As usual, let's break it down.

```
function showResults(){
      //Code to execute
}
```

Firstly, we declare our function which wraps around the code to be executed using curly braces { and }.

Inside the function, our first job is grab all of the answer boxes from our quiz:

```
const answerBoxs = quizBox.querySelectorAll('.answers');
```

Next, we need to keep track of the user's answers. To do so we must first set the users score (number of correct answers) to 0, like so:

```
let numCorrect = 0;
```

Next, for each question, we need to find the selected answer, check if the answer is correct, unanswered, or incorrect, and then format the output accordingly. Green for correct or Red for a missing or incorrect answer.

This is all done inside a forEach Loop, like so:

```
//For each question execute the code within the loop
    myQuestions.forEach( (currentQuestion, questionNumber) => {

//Determine the selected answer
    const answerBox = answerBoxs[questionNumber];
    const selector = `input[name=question${questionNumber}]:checked`;
    const userAnswer = (answerBox.querySelector(selector) || {}).value;

//Set the colour of the answers based on the users answer
    if(userAnswer === currentQuestion.correctAnswer){
        numCorrect++;
        answerBoxs[questionNumber].style.color = 'green';
    }
    else{
        answerBoxs[questionNumber].style.color = 'red';
    }

//close the forEach Lopp
    });
```

Finally, display the results. How many correct answers out of total questions:

```
resultsBox.innerHTML = `${numCorrect} out of ${myQuestions.length}`;
```

Okay, some of this code we haven't seen before. Let's look a little deeper.

```
    const answerBox = answerBoxs[questionNumber];
//Look inside the amswerBox for the current question

    const selector = `input[name=question${questionNumber}]:checked`;
//Define a CSS selector that will allow us to find which radio button has
been checked by the user

    const userAnswer = (answerBox.querySelector(selector) || {}).value;
//Determine which radio button has been checked and get the value of the
answer or || if no value exists, an empty object {}.
```

Next, we have a conditional, if-else statement. These can be much more complex, but what we have is very straightforward.

- We are using **if** to specify the block of code to be executed if the specified condition is true
- We are using **else** to specify the block of code to be executed if the same condition is false

```
if (condition) {
//  block of code to be executed if the condition is true
} else {
//  block of code to be executed if the condition is false
}
```

In our case we are doing two things:

- We are simply changing the text colour of our answers, depending on the condition met.
- We are adding 1 to the variable numCorrect for a correct answer.

```
    if(userAnswer === currentQuestion.correctAnswer){
        numCorrect++;
        answerBoxs[questionNumber].style.color = 'green';
    } else {
        answerBoxs[questionNumber].style.color = 'red';
    }
```

STEP 8 - SHOW EACH QUESTION ON ITS OWN PAGE

Okay, in this step we are going to be doing a little catch-up. We have already set a few things in place in our CSS and our buildQuiz(){} function.

As a quick reminder, we wrapped our output inside a with a class of slide.

```
<div class="slide">
      //Output
</div>
```

And we applied some styles in our CSS for transitioning from one slide to the next:

```
.slide{
  position: absolute;
  left: 0px;
  top: 0px;
  width: 100%;
  z-index: 1;
  opacity: 0;
  transition: opacity 0.5s;
}
.active-slide{
  opacity: 1;
  z-index: 2;
}
```

Here we can see that the active-slide has an opacity of 1 or another way of putting it 100% visible, while our other slides have an opacity of 0, in other words invisible. Plus, we have stacked the active slide on top of the other slides using the z-index command. The active slide is 2, while the other slides are 1.

That part is pretty straight forward. Now let's take a look at the logic that makes it all happen.

Open ‘app.js’ and add the following code just after the showResults(){} function:

```
function showSlide(n) {
  slides[currentSlide].classList.remove('active-slide');
  slides[n].classList.add('active-slide');
  currentSlide = n;
  if(currentSlide === 0){
    previousButton.style.display = 'none';
  }
  else{
    previousButton.style.display = 'inline-block';
  }
  if(currentSlide === slides.length-1){
    nextButton.style.display = 'none';
    submitButton.style.display = 'inline-block';
  }
  else{
    nextButton.style.display = 'inline-block';
    submitButton.style.display = 'none';
  }
}

function showNextSlide() {
  showSlide(currentSlide + 1);
}

function showPreviousSlide() {
  showSlide(currentSlide - 1);
}
```

Then after the buildQuiz(); function call, add the following:

```
buildQuiz();// Add the following code after this function call

const previousButton = document.getElementById("previous");
const nextButton = document.getElementById("next");
```

```
    const slides = document.querySelectorAll(".slide");
    let currentSlide = 0;

    showSlide(currentSlide);
```

Finally, add the following at the end of the script:

```
    submitButton.addEventListener('click', showResults);
// Add the following after this line of code

    previousButton.addEventListener("click", showPreviousSlide);
    nextButton.addEventListener("click", showNextSlide);
```

Save your file (CTRL + S).

Let's have a closer look at the logic here:

```
function showSlide(n) {

// Note! We are passing the value 'n' into this function. The value 'n'
will be used to represent the value of the current slide.

      slides[currentSlide].classList.remove('active-slide');
//Hide the current slide by removing the active-slide class

      slides[n].classList.add('active-slide');
//Show the new slide by adding the active-slide class for slide n

      currentSlide = n;
//Update the current slide number

      if(currentSlide === 0){
//If we're on the first slide
        previousButton.style.display = 'none';
//Hide the previous slide button
```

```
} else{
        previousButton.style.display = 'inline-block';
//Otherwise, display the previous button
      }

      if(currentSlide === slides.length-1){
//If we're on the last slide
        nextButton.style.display = 'none';
//Don't display the next button
        submitButton.style.display = 'inline-block';
//Instead, display the submit button
      }
      else{
        nextButton.style.display = 'inline-block';
//Otherwise, display the next button
        submitButton.style.display = 'none';
//And, don't display the submit button
      }
```

You're doing great! Next, let's take a look at the showNextSlide() and showPreviousSlide() functions:

```
    function showNextSlide() {
      showSlide(currentSlide + 1);
//Increment current slide by one
    }

    function showPreviousSlide() {
      showSlide(currentSlide - 1);
//Decrement current slide by one
    }
```

This logic is called when we click the 'Next Question', and 'Previous Question' buttons respectively, using the event listeners at the end of the code.

```
    previousButton.addEventListener("click", showPreviousSlide);
//Listen for a previosButton click. If clicked call the showPreviousSlide
function
```

```
    nextButton.addEventListener("click", showNextSlide);
//Listen for a nextButton click. If clicked call the showNextSlide
function
```

Unlike when calling a function normally, we forgo using the parenthesis () when we call a function in this way.

Finally in this step, it is sometimes recommended to wrap your JavaScript code in an IIFE, an immediately invoked function expression. This is a self-calling function. It runs as soon as you define it. This will keep your variables out of global scope to ensure it doesn't interfere with other scripts running on the page. There are occasions when you want your variables in global scope though, as we'll see a bit later.

The syntax is as follows:

```
(function(){
    // put the rest of your code here
})();
```

Okay, with that done your quiz should now be up and running.

Wouldn't it be great if we included a countdown timer and some feedback for the user so they know how many questions there are and how many are left? Let's move on.

STEP ❾ - ADDING A COUNTDOWN TIMER

Following the `<div id="results"></div>` in your 'quiz_app.html' file, add the following mark-up:

```
    <h2>Countdown Timer</h2>
    <div id="clockdiv">
//The wrapper where the countdown timer will reside
        <div>
            <div class="smalltext">Minutes</div>
```

```
// A label to show where 'Minutes' will be displayed
            <span class="minutes"></span>
//This is where the number of minutes for our countdown will reside
        </div>
        <div>
            <div class="smalltext">Seconds</div>
// A label to show where 'Seconds' will be displayed
            <span class="seconds"></span>
//This is where the number of seconds for our countdown will reside
        </div>
    </div>
```

Save the file (CTRL + S)

This mark-up is where we will display our countdown clock. Next, add the following line of code, just inside the closing body tag </body>, ensuring it is after the reference for our 'app.js' file.

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

This will make the necessary reference to our countdown script, in the same way, we referenced our 'app.js' file.

Next, open the 'app.css' file and add the following code to the bottom of the file (excluding the comments (//…)), then save the file (CTRL + S):

```
 /* ================= COUNTDOWN TIMER ================== */

h2{
   color: #333;//Set text colour
   font-weight: 100;// Set font thickness
   font-size: 40px;// Set font size
   margin: 40px 0px 20px; // Set margins (TRBL)
 }
 #clockdiv{
   font-family: sans-serif;//Set font face
   color: #fff;//Set text colour to white
   display: inline-block; // Set display property
   font-weight: 100;//Set font thickness
```

```
    text-align: center;//Align text to the centre
    font-size: 30px;//Set font size
  }
  #clockdiv > div{
    padding: 10px;//Set space around the element to 10 pixels
    border-radius: 100px; // Make borders rounded
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#ccc), to(#999)); // Set background to a graduated colour
    display: inline-block; // Set display property
    border: solid 1px #333;
//Make a solid border around the element of 1 pixel wide and in a dark
grey colour
    width: 100px;//Set the width of the element
    height: 100px;//Set the height of the element
  }
  #clockdiv div > span{
    padding: 15px;//Set space around the element to 15 pixels
    border-radius: 3px;//Make border slightly rounds at corners
    background: #333;//Set background colour to dark grey
    display: inline-block;//Set display property to inline
  }
  .smalltext{
    padding-top: 3px;//Set space at the top of the element to 3 pixels
    font-size: 16px;//Set the font size to 16 pixels
  }
```

Great job!

Now, make a new file called ‘countdown.js’ and save it in the ‘js’ folder.

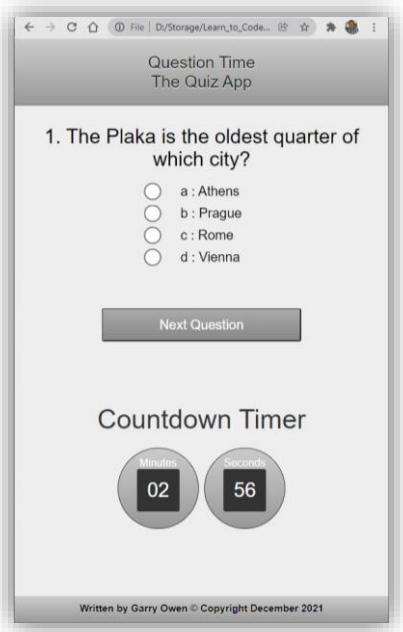
Add the following code (minus the comments (//…)), and then save the file (CTRL + S). Of course, you can add your own comments, but they should be much more succinct than these fuller explanations, and the general rule of thumb, comments should come before the code, not afterwards.

```
function getTimeRemaining(endtime) {
//Open function to get the remaining time with a parameter of 'endtime'
so that we can return the result and use it outside of this function

  const total = Date.parse(endtime) - Date.parse(new Date());
//Calculate the end time minus the time now using the computer clock

  const seconds = Math.floor((total / 1000) % 60);
//Calculate the number of seconds using our 'total' variable. In
JavaScript 1000 is equal to 1 second. Since we'll be setting our timer to
3 minutes, that equals 180000 (180 seconds x 1000). As you can see here
we are converting the total into seconds (total / 1000), then the %
(modulus) 60 ensures the result returned in whole seconds.

  const minutes = Math.floor((total / 1000 / 60) % 60);
//Here we are doing the same as above with the addition of a further /
60, or in other words converting total into the number of minutes.

  return { //This then returns the results
    total,
    minutes,
    seconds
  };
}

function initializeClock(id, endtime) {
//Open a function to initialise the clock

  const clock = document.getElementById(id);
//Reference the ID of the clock

  const minutesSpan = clock.querySelector('.minutes');
//Reference the minutes class
  const secondsSpan = clock.querySelector('.seconds');
//Reference the seconds class

  function updateClock() {
//Open the update clock function
```

```
    const t = getTimeRemaining(endtime);
//Set t variable to the remaining time

    minutesSpan.innerHTML = ('0' + t.minutes).slice(-2);
//Dislay minutes in the minutes <span>
    secondsSpan.innerHTML = ('0' + t.seconds).slice(-2);
//Display seconds in the seconds <span>

    if (t.total <= 0) {
      clearInterval(timeinterval);
//If the total time is equal to or less than 0 reset the clock
    }
  }

  updateClock();
//Call the updateClock function

  const timeinterval = setInterval(updateClock, 1000);
//Update the clock every second (1000 = 1 second)
}

const deadline = new Date(Date.parse(new Date()) + 3 * 60 * 1000);
initializeClock('clockdiv', deadline);
//Set the countdown timer to 3 minutes and initialise the clock
```

- Here we are calculating the date/time right now and then adding 3 minutes
- (3 * 60 * 1000) means (3 x 60 x 1000) or in other words 180000.
- Then we send the value with our call to initialise the clock `initializeClock('clockdiv', deadline)` so that the initializeClock function knows where to start from.

Let's see what we have so far:



Okay, looking pretty good, right?

- We've got a nice looking layout that works for a mobile phone
- We have our question wrapper
- We're loading in our question bank
- We are taking user input and collecting their answers
- We can show results when the user submits the quiz
- We have a 3-minute countdown timer

So happens when the timer hits 00 00. Well, right now, nothing at all. Let's make it submit the quiz and show us our results.

```
if (t.total <= 0) {
  clearInterval(timeinterval);
        ◄ Add a function call here to submit the quiz
}
```

How do we do that?

Well, since we have ordered our JavaScript files in our 'quiz_app.html' to load the questions and then the timer, we have access to all of the functions within 'app.js' from 'countdown.js'. Cool right! So all we need to do is a normal function call, like this:

```
showResults();
```

So what are we doing here? When the timer is either less than (<) or equal to (=) zero (0) we are clearing the timer and then showing the results. Save your file and refresh your browser.

Ah, but it didn't work. That's simply because we have used an IIFE. Sometimes an IIFE is great, other times it prevents us from accessing variables globally, as we need to. Remove your IIFE, save the file, and run your test again. Now you can see first-hand what an IIFE actually does.

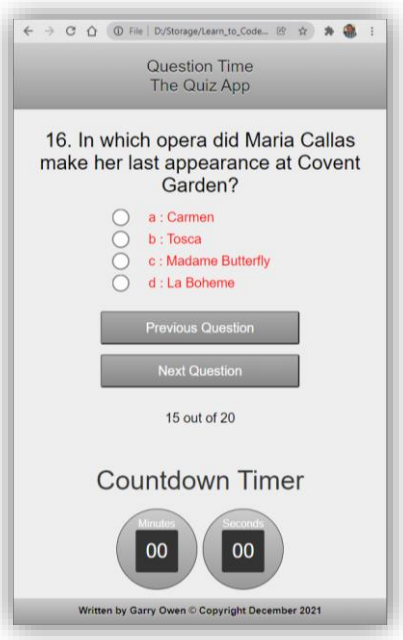
If you've forgotten what the IIFE is, it's simply this:

```
(function(){
    // put the rest of your code here
})();
```

Wrapped around your code. Simply remove it and save your file, then refresh your browser.

Or we could of course still use the IIFE and copy and paste the countdown script into our 'app.js' file and just have all of our logic in one longer JavaScript file. However, when we are working with possibly hundreds of functions, I find it better to structure your code in separate and specifically named files. That way your code will be cleaner and easier to bug fix. You will see exactly what I mean in our next exercise when we start building our platform game.

Now you should have the following result:



Okay, so when the timer runs out we no longer need the buttons. We should remove them to prevent further user input and add a try again option for the user. We'll need to add another button to our 'quiz_app.html' file and then update our 'countdown.js' file. We need to add the JavaScript logic in the same place as our function call to submit the quiz, like this:

```
if (t.total <= 0) {
  clearInterval(timeinterval);
  showResults();
  submitButton.style.display = 'none';
  previousButton.style.display = 'none';
  nextButton.style.display = 'none';
  tryAgainButton.style.display = 'inline-block';
}
```

And then open our 'quiz_app.html' file and add the following mark-up just below the results :

```
<button id="tryAgain"><p>Try Again</p></button>
```

But, we don't want to show the button unless the timer is at 00 00. So we also need to add another condition to `if (t.total <= 0) { }` to be able to apply the logic for what happens when the clock is greater than 00 00. This is simple since we have the logic for the true condition, the false is just an else condition, like so:

```
    } else {
      tryAgainButton.style.display = 'none';
    }
```

This is telling the browser not to display the try again button unless the timer is at zero - 0. This won't work just yet though. We need to define the button in the same way we defined the buttons in our 'app.js' file. Due to TDZ, we need to define it before the function call is made to inializeClock function, like so:

```
const tryAgainButton = document.getElementById("tryAgain");
```

Okay, so now the quiz control buttons are removed and we are displaying our 'Try Again?' button. Great so far, but the button currently has no functionality. To make the button interactive we need to make a function that will run a block of code to make the button reset the quiz and then add an event listener to have JavaScript listen for a button click, which we will use to call our function. The event listener logic we have seen before in the 'app.js' file. Add the following to the bottom of the 'countdown.js' file:

```
tryAgainButton.addEventListener("click", resetQuiz);
```

As you can see, when the button is clicked we are calling a function called 'resetQuiz'. Great!

Now let's create that function:

```
function resetQuiz(){
 //reset quiz logic
}
```

Okay, so what does our reset quiz function need to do? For our purposes, it simply needs to refresh the page, like this:

```
function resetQuiz(){
    location.reload();
}
```

JavaScript has a built-in method called reload(). Reload() does the same as the reload button in your browser.

So what about the user knowing how many questions are left? Well, if you use the same logic as we've used for the final results to find the length of the question bank, we can inform the user how many questions there are in a variety of ways. As we need to be conscious of available screen space, I think it will work well to dynamically update the title, depending on the question bank length. At the moment our web app is entitled 'Question Time – The Quiz App'. How about we update that to: '[number of questions] Questions – The Quiz App'. So in my case, it would read '20 Questions – The Quiz App'. Sounds spot on!

First, we need to create a new within the title <h1></h1> and apply an ID to it. So why are we using a element and not a , for example? The tag is an inline container, whereas the is a block-level element. Meaning a would make a new line,

while keeps everything on the same line. So using a would scupper our desired layout.

Open the 'quiz_app.html' file and update the title to reflect the following:

```
<div id="header">
    <h1><span id="quizLength"></span> Questions<br>The Quiz App</h1>
</div>
```

Since we have set an ID of 'quizLength', we can now use this in our JavaScript. We want this to be present all of the time, so placement within our code is important. Add the following to the 'app.js' file:

```
document.getElementById('quizLength').innerHTML = data.length;
```

Ensure it is placed after this:

```
const myQuestions = [];
for(i=0; i < data.length; i++){
    myQuestions.push(data[i]);
}
```

Up to this point we have:

It's looking great!

If you want to make the quiz easier or more difficult, try adjusting the timer to more or less time.

```
const deadline = new Date(Date.parse(new Date()) + 3 * 60 * 1000);
initializeClock('clockdiv', deadline);
```

All you need to do is change the `3` to an alternative value. The 3 here means 3 minutes, 2 would be 2 minutes, and so on. This is your quiz, have fun, play around with it. Challenge your friends to complete your quiz.

I think now would be a great time for another break, before we move on to the final part of this program. Allowing the user to add or delete questions.

STEP ❿ - ADDING USER ADMINISTRATION FUNCTIONS

Finally, we want the user to be able to add new questions and delete existing ones. Essentially to be able to edit the quiz and make it their own.

We have a few questions we need to ask ourselves here:

1. How are we going to collect data from the user?
2. How do we add the user data to our JSON question bank?
3. How will we remove a question from the JSON question bank?

All good questions. In programming, there are always many ways to tackle a problem. Let's start by taking a look at question 1.

Something we haven't used yet is HTML forms. HTML forms are the obvious way to collect data from the user and allow us to make our data collection more user-friendly. We know we need the following information:

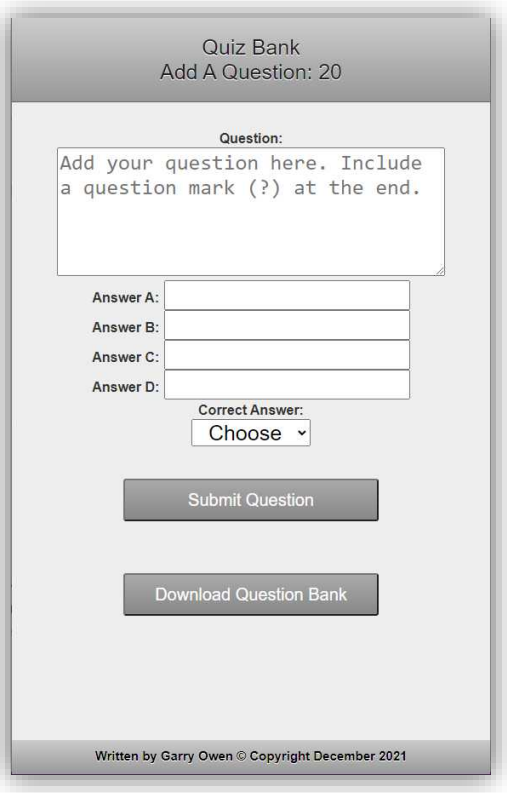
- The question
- Four possible answers
- The correct answer represented by a, b, c or d

Once again our design falls into three areas.

1. The view provided by our HTML
2. The styles and page formatting, provided by our CSS
3. The program logic, provided by our JavaScript.

We will design a new HTML page for this and then add appropriate page links to allow the user to navigate between both. Then we will make a new JavaScript file to accommodate the logic that will make it all function.

Considering our data capture requirements, our form might look something like this:



Normally, when creating a program like this you would use a database and server-side languages, such as PHP or Node.js, etc. However, this book is exploring the use of HTML, CSS, and JavaScript, which are all client-side technologies. We will be using other techniques that will allow us to code everything and run it locally in your browser, without the use of any other outside technologies.

Client-side and Server-side:

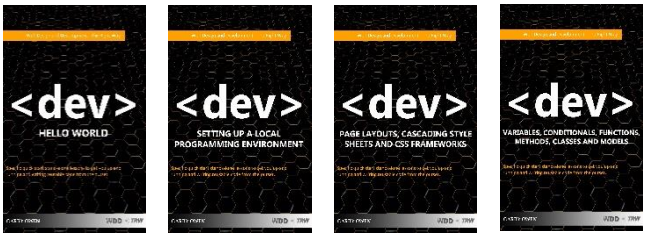
Client-side and server-side are sometimes referred to as front-end and back-end. The client-side of a website refers to the web browser and the server-side is where the data and source code is stored.

For your information, there are many databases available. Some of the most used and well-known are MySQL, MariaDB, and PostgreSQL. These all use a specially developed language to query them called Structured Query Language (or SQL, pronounced sequel). Then there is another database type known as NoSQL databases, such as MongoDB, ApacheDB, and Cassandra, which utilise alternative structures and query techniques, such as the JSON-like data structure with MongoDB or CQL with Cassandra.

For our purposes, we will be writing our questions into our JSON file instead, but it is important for you to know that if you wanted to create a program like this one and deploy it on a remote server, you would need to use other technologies to maintain security over your code and data. Using HTML, CSS, and JavaScript any user can access all of your code, just by hitting 'CTRL + U' ('Cmd + U' on Mac), in many browsers or using the menu and choosing the 'view source' option.

That said, the logic you'll learn by doing it this way is a very valuable learning exercise and the same methods can be used in many of your future projects, perhaps with the addition of server-side technologies to better handle security aspects, etc.

Server-side languages are translated by the browser and displayed as HTML, therefore the user can't read the actual code. If you want to learn more about that, I would highly recommend my book series called, 'Web Design and Development – *The Right Way!*'.



….and many more

HOW TO CONSTRUCT OUR HTML FORM

Once again, open your chosen coding editor and make a new file called 'add_questions.html'. Then add the following mark-up and save the file in the 'web_app' directory (CTRL + S).

```
<html>
<head>
    <title>Quiz Bank Editor</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='stylesheet' type='text/css' media='screen'
href='css/app.css'>
    <script type="text/javascript" src="json/questions.json"></script>
</head>
<body>
    <div id="header">
        <h1>Quiz Bank<br>Add A Question: <span id="qCount"></span></h1>
    </div>
    <div class="quiz-wrapper">
        <form id="addQuestion" onsubmit="submitForm(event)">
            <label class="formLabel">Question Number:</label><br>
            <label class="formLabel">Question:</label><br>
            <textarea maxlength="200" cols="50" rows="5" class="formArea"
placeholder="Add your question here. Include a question mark (?) at the
end." id="q" name="q" required></textarea><br>
            <label class="formLabel">Answer A:</label>
            <input type="text" maxlength="100" size="100"
class="formInput" id="a" name="a" required><br>
            <label class="formLabel">Answer B:</label>
            <input type="text" maxlength="100" size="100"
class="formInput" id="b" name="b" required><br>
            <label class="formLabel">Answer C:</label>
            <input type="text" maxlength="100" size="100"
class="formInput" id="c" name="c" required><br>
            <label class="formLabel">Answer D:</label>
            <input type="text" maxlength="100" size="100"
class="formInput" id="d" name="d" required><br>
            <label class="formLabel">Correct Answer:</label><br>
            <select id="ca" name="ca" class="formSelect" required>
```

```
                <option value="" selected disabled> Choose </option>
                <option value="a"> - A -</option>
                <option value="b"> - B -</option>
                <option value="c"> - C -</option>
                <option value="d"> - D -</option>
            </select><br><br>
            <button id="submitQuestion"><p>Submit
Question</p></button><br><br><br>
          <button id="saveQuestionBank"><p>Save Question
Bank</p></button>


        </form>
    </div>
       <div id="footer">
        <h5>Written by Garry Owen &copy Copyright December 2021</h5>
    </div>
    <script src='js/addquestions.js'></script>
</body>
</html>
```

You've seen some of this mark-up before, so instead of repeating those aspects, let's concentrate on the <form> itself.

```
        <form id="addQuestion"> …. </form>
```

Firstly, we've opened a new form element and closed it at the end of the form structure. We have added an ID to the form so that we can manipulate it with our JavaScript code and/or our CSS.

```
<label class="formLabel">Question:</label><br>
```

Next, we have a form label <label></label>. We have given each label a class so that we can choose styling rules with our CSS. The text between the <label></label> tags will be displayed (rendered) in the users view.

The
 element simply means bridge return. Its function is to go to a new line, before rendering the next element. I have added this here for the sake of simplicity, we could get the same effect by adding necessary rules to our CSS.

Each of the label elements has pretty much the same structure, shy of a
 here and there.

```
<input type="text" maxlength="" size="" class="" id=" " name="" required>
```

Next, I want to discuss the most common type of form input. Inside each form input we can set various rules. Some rules are dependent on the type of <input> we choose, while others apply to any of the available options.

First, we need to set the input type. Here we have chosen “text”, however, there are many input types to choose from. The number input type allows only numbers as input, text allows text and numbers. We won’t explore them all here, but you can see the range of options in the list below and over the page:

INPUT TYPE FORM ATTRIBUTES

INPUT TYPES	
<input type="button">	<input type="password">
<input type="checkbox">	<input type="radio">
<input type="color">	<input type="range">
<input type="date">	<input type="reset">
<input type="datetime-local">	<input type="search">
<input type="email">	<input type="submit">
<input type="file">	<input type="tel">
<input type="hidden">	<input type="text">

<input type="image"> <input type="month"> <input type="number">	<input type="time"> <input type="url"> <input type="week">

As can be seen, there are many HTML form input types. We will explore the ones we are using in greater detail as we progress.

OTHER FORM ATTRIBUTES

ATTRIBUTE	VALUE	DESCRIPTION
name	text	Name of the input element.
value	text	Value of the input element.
id	identifier	Defines a unique identifier for the input.
class	classnames	Sets one or more CSS classes to be applied to the input.
style	CSS-styles	Sets the style for the input.
data-*	value	Defines additional data that can be used by JavaScript.
hidden	hidden	Specifies whether the input is hidden.
title	text	Sets a title that displays as a tooltip.
tabindex	index	Sets a tab sequence relative to the other elements.
checked	checked	For types: checkbox or radio. Makes the option checked or chosen.
placeholder	text	A short hint which describes the expected value
maxlength	number	The maximum number of characters allowed
required	no value	Sets the input to the required field

readonly	readonly	Make input read-only
disabled	disabled	Disables input element
autofocus	no value	Sets the focus on the element after page loads
autocomplete	on | off	Presents the user with previously entered values
form	form-id	Refers to the id of the form the <input> element belongs to
formaction	URL	For types: submit image. URL or path of the file the submitted data will be processed
formtarget	_blank _self _parent _top framename	For types: submit and image. Indicates where the response should be displayed
formenctype	application/x-www-form-urlencoded, multipart/form-data, text/plain	For types: submit image. How the form data submitted shall be encoded to the server.
formmethod	get post	For types: submit and image. The HTTP method for sending form data
formnovalidate	formnovalidate	Avoids the form being validated after submission
accept	file-extension audio/* video/* image/* media_type	For type: file. Indicates what file types user can pick to upload

min	number date	Minimum value
max	number date	Maximum value
step	number any	The interval between legal numbers
multiple	multiple	Allows users to choose more than one value from the selection
pattern	regexp	A regular expression that an <input> element's value is checked against
size	number	Input control's width in number of characters
width	pixels	Width of the input element in pixels. Used by image input types.
height	pixels	Height of element in pixels. Used by image input types
list	datalist-id	<datalist> element that contains pre-defined options for an <input> element
dirname	inputname	Text direction to be submitted

As with input types, there are also a great many HTML form attributes to choose from, with many different uses. We will explore the ones we need in more detail, as we progress.

Next we have <textarea></textarea> mark-up.

```
<textarea maxlength="200" cols="50" rows="5" class="formArea"
placeholder="Add your question here. Include a question mark (?) at the
end." id="q" name="q" required></textarea><br>
```

Okay, this is slightly different than a form input in that it can occupy more than one line, and as its name suggests it has an input type of text pre-set.

`maxlength="200"` is just the same as with the last input element. It restricts the number of characters, in this case, to 200.

`cols="50"` is setting our <textarea> to 50 characters wide.

`rows="5"` is setting our <textarea> height to 5 characters tall.

`class="formArea"` is what we will use as a reference for our CSS rules.

`placeholder="Add your question here. Include a question mark (?) at the end."` a string that provides a brief hint to the user as to what kind of information is expected in the field. In our case, I wanted to ensure the user adds a question mark.

`id="q"` is an ID we can use with either our CSS or JavaScript. In this case, it is intended for use with our JavaScript. The 'q' simply stands for 'question'.

Finally, `name="q"`, is the name of the input element and can be used to pass data from a form as reference. Again, more on that later.

Unlike with an <input>, we close a textarea with the close </textarea> tag.

The next 4 inputs are almost identical, with the exception of changing the values to a, b, c and d respectively:

```
<input type="text" maxlength="100" size="100" class="formInput" id="a"
name="a" required><br>
```

So we've restricted the user to text input only, then we have `maxlength="100"`. That will restrict the user input to a maximum of 100 characters.

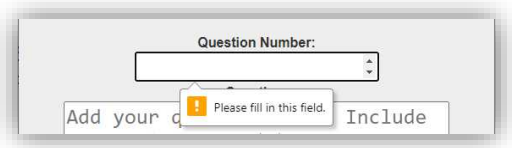
Next, we have `size="100"`. That sets the size (width) of the user input field in the number of characters.

Then we have `class="formInput".` This is what we will use as a reference for our CSS rules.

Next, we have an ID of 'a'(`id="a"`). That is an ID we can use with either our CSS or JavaScript. In this case, it is intended for use with our JavaScript. 'a' simply stands for answer a. We can name classes and IDs pretty much anything we want, outside of existing keywords and tags within our code or mark-up.

Then we have `name="a"`, this is the name of the input element and can be used to pass data from a form as reference. We'll see more on that later.

Finally, we have the `required` attribute. Which tells the browser that the form element MUST be completed by the user, otherwise the form will not submit and the user will be prompted for input, like this:



We'll bypass the <label> and look at the next input. This time we have a different type of input, <select></select>.

```
<select id="ca" name="ca" class="formSelect" required>
    <option value="" selected disabled> Choose </option>
    <option value="a"> - A -</option>
    <option value="b"> - B -</option>
    <option value="c"> - C -</option>
    <option value="d"> - D -</option>
</select>
```

The attributes the <select> carries are much the same as before, ID, name, class and required, so I won't go over those again.

Next, we have dropdown options enclosed in <option></option> tags. Each one of these passes its value when chosen and the form submitted.

You'll notice the first option is a little different:

```
<option value="" selected disabled> Choose </option>
```

The settings here make it impossible to choose (`disabled`) and also show it as the current selection (`selected`). That means that the user is presented with the word 'Choose' in the dropdown.

Great stuff! Finally we have a <button> to submit our form and parse the data within the form:

```
<button id="submit"><p>Submit Question</p></button>
```

And then another, to download our question bank file:

```
<button id="downloadQuestionBank"><p>Download Question Bank</p></button>
```

These are in the same format as all of the previous buttons we have created and therefore they will automatically be styled the same because we've referenced our CSS in the head of the mark-up.

I think it's that time again…. Or do I just drink too much coffee? ☺

STYLING OUR HTML FORM

Okay we already have our CSS file created, so open 'app.css' and add the following code to the bottom of the file:

```
/* ========================== FORM STYLES ====================== */
.formLabel{
    color: #333;
    font-size: 16px;
    font-weight: bold;
    width: 10%;
}
.formArea{
  color: #333;
  font-size: 25px;
  font-weight: bold;
  width: 80%;
}
.formInput{
  color: #333;
  font-size: 25px;
  font-weight: bold;
  width: 50%;
}
.formSelect{
  color: #333;
  font-size: 25px;
  font-weight: bold;
  width: 25%;
  text-align: center;
}
```

Notice here, we have some of our rules repeated in more than one class. I am showing you this on purpose because you should always be thinking, how can I make my code cleaner and more optimised? This won't only make your code run more efficiently, but it will be easier to read.

In these circumstances, firstly ask yourself, what rules apply to everything in the form?

```
form{
  color: #333;
  font-weight: bold;
}
```

Next, are there any rules that apply to multiple classes (or IDs / Keywords)?

```
.formArea, .formInput, .formSelect{
  font-size: 25px;
}
```

Next, what's left over?

```
.formArea{
  width: 80%;
}
.formInput{
  width: 50%;
}
.formSelect{
  width: 25%;
  text-align: center;
}
.formLabel{
    font-size: 16px;
    width: 10%;
}
```

Ensure you have optimised it as much as you can and then bring it all together, and add the code to the end of your 'app.css' file.

You should have something that looks like this:

```
form{
  color: #333;
  font-weight: bold;
}
.formLabel{
    font-size: 16px;
    width: 10%;
}
.formArea, .formInput, .formSelect{
  font-size: 25px;
}
.formArea{
  width: 80%;
}
.formInput{
  width: 50%;
}
.formSelect{
  width: 25%;
  text-align: center;
}
```

In this case, not a great amount of fewer lines of code, but I'm sure you get the idea. We started with 26 lines and reduced it to 21 – not bad! Save your file (CTRL + S). Let's move on…

GET DATA FROM THE HTML FORM AND ADD IT TO THE JSON FILE

Let's think about the title of this section. It has 2 parts but requires 3 steps.

1. Get data from the HTML form
2. Convert it to JSON format
3. Add it to the JSON file

Make a new file called 'addquestions.js' and save it in the 'js' folder.

In the same way, we've tackled our JavaScript logic before, we will place each element for what we want to achieve into JavaScript functions.

```
function submitForm(event){
    event.preventDefault();
}
```

The first function is combined with some code added to the HTML form tag. I.E. `onsubmit="submitForm(event)".` Together they prevent the form from submitting in the default fashion and allow us to be able to gather the form data in the next function.

Why do we need to do this? By default, when a form is submitted the web page is refreshed, and therefore the data is cleared.

```
function getMyData(){
    const question = document.getElementById("qn").value +". " +
document.getElementById("q").value;
    const a = document.getElementById("a").value;
    const b = document.getElementById("b").value;
    const c = document.getElementById("c").value;
    const d = document.getElementById("d").value;
    const correctAnswer = document.getElementById("ca").value;
    const newData = { question, answers: { a, b, c, d }, correctAnswer }
    if(question != "" && a != "" && b != "" && c != "" && d != "" &&
correctAnswer != ""){
        questionBank.push(newData);
        document.getElementById('qCount').innerHTML =
questionBank.length;
        document.getElementById("addQuestion").reset();
    }
    return questionBank;
}
```

We are gathering the values here from our form, one by one, and then putting them back together in the object form that we require. Each form

element is attained using the attributed ID and the value is stored in a variable. E.G.

```
const a = document.getElementById("a").value;
//store the value from the input with the ID ‘a’, in a variable named ‘a’. We have repeated this structure to gather the data from all form fields.

const newData = { question, answers: { a, b, c, d }, correctAnswer }
//form our object using the collected values

    if(question != "" && a != "" && b != "" && c != "" && d != "" && correctAnswer != ""){
//here we have a conditional which is only true if all form fields have a value when the form is submitted

        questionBank.push(newData);
//pushes our new question, answers and correct answer (newData) into an array named questionBank

        document.getElementById('qCount').innerHTML = questionBank.length;
//displays the current number of questions in the bank in the <h1> title element

        document.getElementById("addQuestion").reset();
//clears the form ready for another question to be added

    }
//closes the conditional

return questionBank;
//return the updated question bank, making it accessible outside of the function

}
//Finally, we close the function
```

Next, we have a function that is called when we save our question bank to local storage. This will allow us to update the question bank and display the newly updated question set to the user. However, to do that we'll need to make some small changes to our 'app.js' file too. For now, though, let's take a look at our saveMyFile function:

```
function saveMyFile(){
    localStorage.setItem("questionBank", JSON.stringify(questionBank));
    location.replace("quiz.html")
}
```

This function is very simple. We are setting a local storage item with the name `"questionBank"` and inserting our question bank into it after converting it to a JSON string `JSON.stringify(questionBank)`

WHAT IS LOCAL STORAGE?

The localStorage object allows you to save key/value pairs in the browser. It stores data with no expiration date. This means the data is not deleted when the browser is closed and will be available for future use. The concept behind it is very simple. The syntax is as follows:

```
localStorage.setItem(key, value); // Store data
localStorage.getItem(key, value); //Retrieve data
localStorage.removeItem(key); //Remove data
```

An example of its usage is shown below:

```
localStorage.setItem("fullname", "Garry Owen"); //Sets fullname
localStorage.getItem("fullname");// Returns Garry Owen
localStorage.removeItem("fullname");// Clears storage
```

In the last part of our code we have our variable declarations and event listeners, to listen for when our form is submitted and when the save question bank button is clicked.

```
const addQuestionButton = document.getElementById("submitQuestion");
//reference the submitQuestion button and store it in a variable called
addQuestionButton

const saveButton = document.getElementById("saveQuestionBank");
//reference the saveQuestionBank button and store it in a variable called
bankButton

const questionBank = [];
//create a new empty array called questionBank

const newObject = localStorage.getItem("questionBank");
//retrieve the question bank from localStorage

let dataStored = JSON.parse(newObject);
//parse the JSON data into a new JavaScript object

  if(dataStored != ""){
//if dataStored is not empty…

    for(i=0; i < dataStored.length; i++){
      questionBank.push(dataStored[i]);
    }
//iterate through the dataStored array and push each question into the
questionBank array
  } else {

    for(i=0; i < data.length; i++){
        questionBank.push(data[i]);
    }
//otherwise, iterate through the data array (brought in in the
<head></head> section of our code) and push each question into the
questionBank array

  }// close the conditional
```

```
document.getElementById('qCount').innerHTML = questionBank.length;
//update the <h1> title tag with the current number of questions in our
question bank

addQuestionButton.addEventListener("click", getMyData);
//Listen for the submit form button and when clicked call or invoke the
getMyData() function

saveButton.addEventListener("click", saveMyFile);
//Listen for the save question bank button and when clicked call (invoke)
the saveMyFile() function
```

Okay, so we can now add new questions and save them to local storage.

What's left to do?

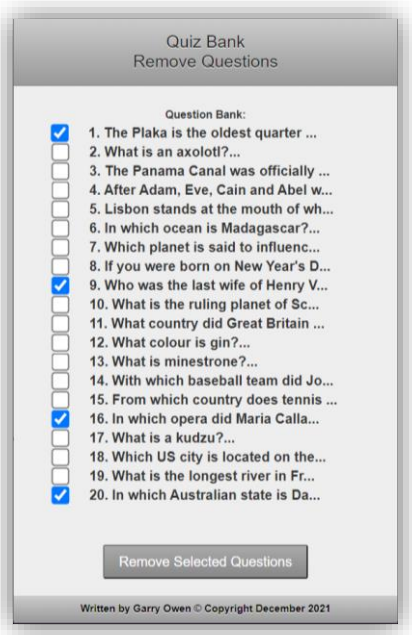
- We need a way to remove unwanted questions
- We need a way for the user to access the admin functions
- We need to give our quiz access to local storage updates

CONSTRUCTING THE 'REMOVE QUESTIONS' FORM MARK-UP

Okay, so we already know that push() adds data to the end of an array, and pop() will remove the last item of an array.

We have already added our new questions to the end of the array, but unlike with adding questions, a user will not only want to be able to remove the last one. In fact, we need to be able to remove multiple questions. From a user's perspective, it would be easier if all questions were displayed on a list and they could click a tick box and have a button with the function of removing all of the questions selected in one hit.

Like this:



Okay, let's build the HTML mark-up. Make a new file called 'remove_questions.html' and save it in the main web app directory. Add the following code and then save (CTRL + S).

```
<html>
<head>
    <title>Quiz Bank Editor</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='stylesheet' type='text/css' media='screen'
href='css/app.css'>
    <script type="text/javascript" src="json/questions.json"></script>
</head>
<body>
    <div id="header">
        <h1>Quiz Bank<br>Remove Questions</h1>
    </div>
    <div class="wrapper">
```

```
        <form id="removeQuestion" onsubmit="submitForm(event)">
            <label class="formLabel">Question Bank:</label><br>
            <!-- Add all questions here with check boxes -->
            <div id="showQstn"></div>
            <br><br>
            <button id="removeQuestions"><p>Remove Selected
Questions</p></button><br><br><br>
            <button id="saveQuestions"><p>Save Question Bank</p></button>
        </form>
    </div>
    <div id="footer2">
        <h5>Written by Garry Owen &copy Copyright December 2021</h5>
    </div>
    <script src='js/removequestions.js'></script>
</body>
</html>
```

Pretty straightforward, right? You've seen pretty much everything here before. However, let's take a deeper look at the form to understand what's going on.

```
<form id="removeQuestion" onsubmit="submitRemoveForm(event)">
    <label class="formLabel">Question Bank:</label><br>
    <!-- Add all questions here with check boxes -->
    <div id="showQstn"></div>
    <br><br>
    <button id="removeQuestions"><p>Remove Selected
Questions</p></button><br><br><br>
    <button id="saveQuestions"><p>Save Question Bank</p></button>
</form>
```

Aside from the element and attributes that you have already seen, the main difference here is that we intend to populate the form inside our show question div `<div id="showQstn"></div>` using our JavaScript coding. Therefore, what's left is a very simple form. The other attributes have had their id names changed to ensure we don't get any conflict issues from other scripts.

STYLING THE 'REMOVE QUESTIONS' FORM

Open the 'app.css' file and add the following to the bottom:

```
  #showQstn {
      text-align: left;
      display: block;
      margin: 0 auto 0 auto;
      width: 420px;
  }
  #showQstn label:first-of-type{
      margin-left: 10px;
  }

  .chk_butn {
    transform: scale(200%);
    margin: 10px 30px 5px -20px;
    }
    .wrapper{
        position: relative;
        margin: 30px 0 20px 0;
    }
    #footer2 {
    position: relative;
    bottom:0;
    width:100%;
    height:40px;
    text-shadow: 0px 1px 0px #fff;
    background-image: -webkit-gradient(linear, left top, left bottom,
from(#ccc), to(#999));
    }
    #footer2 h5{
        text-align: center;
        padding-top: 10px;
        font-size: 15px;
    }
```

Okay, apart from a couple of rules, you have seen all of this CSS by now. In the vein of avoiding too much repetition, I'll explain just those differences.

```
margin: 0 auto 0 auto;
```

We've seen margin before, but 'auto' is something we haven't explored. Used together with 'display: block;' and 'width: 420px;' this displays the element at the centre of the display, as a block.

```
#showQstn label:first-of-type{
    margin-left: 10px;
}
```

`label:first-of-type` - this sets rules only for the first label element

The rest we've already seen. However, note we have made another wrapper class and another footer class. This is because existing rules won't work for our new HTML page. That is because the layout here is longer (potentially much longer depending on how many questions the user decides on) than the original structure for our question and answer slides.

UPDATING QUIZ JAVASCRIPT FILE TO ACCOMMODATE LOCAL STORAGE

Open 'app.js' and exchange this code:

```
const questionBank = [];
    for(i=0; i < data.length; i++){
        questionBank.push(data[i]);
    }

document.getElementById('quizLength').innerHTML = data.length;
```

with this:

```
    const myQuestions = [];
    const newObject = localStorage.getItem("questionBank");
    let dataStored = JSON.parse(newObject);

      if(dataStored != ""){
        for(i=0; i < dataStored.length; i++){
          myQuestions.push(dataStored[i]);
        }
        document.getElementById('quizLength').innerHTML =
dataStored.length;
      } else {
        for(i=0; i < data.length; i++){
            myQuestions.push(data[i]);
        }
        document.getElementById('quizLength').innerHTML = data.length;
      }
```

This works exactly the same way as explained in the 'addquestions.js' file, with the exception that we are updating the 'quizlength' attribute depending on whether we are using local storage 'dataStored' or the JSON file we created for our questions 'data', as below:

```
document.getElementById('quizLength').innerHTML = dataStored.length;
document.getElementById('quizLength').innerHTML = data.length;
```

This simply updates our title with the number of questions.

BUILDING THE JAVASCRIPT TO DELETE QUESTIONS

Make a new file called 'removequestions.js' and save it in the 'js' folder, then add the following code and save (CTRL + S).

```
function submitRemoveForm(event){
    event.preventDefault();
}
function showQuestions(){
const questionHolder = document.getElementById('showQstn');
questionToShow = [];
const newObject = localStorage.getItem("questionBank");
let dataStored = JSON.parse(newObject);
  if(dataStored != ""){
    for(i=0; i < dataStored.length; i++){
        questionToShow.push(
            `<label>
            <input type="checkbox" name="${i}" value="${i}"
class="chk_butn">
            ${i+1}. ${dataStored[i].question.substring(0, 32)}...
            </label><br>`
        );
    }
  } else {
    for(i=0; i < data.length; i++){
        questionToShow.push(
            `<label>
            <input type="checkbox" name="${i}" value="${i}"
class="chk_butn">
            ${i+1}. ${data[i].question.substring(0, 32)}...
            </label><br>`
        );
    }
}
    questionHolder.innerHTML = questionToShow;
    return questionToShow;
}
function removeQuestion(){
    const questionHolder = document.getElementById('showQstn');
    questionsToRemove = [];
    let checkboxes =
document.querySelectorAll('input[type=checkbox]:checked')
    for (var c = 0; c < checkboxes.length; c++) {
           questionToShow[checkboxes[c].value] = "";
```

```
      questionBank.splice(checkboxes[c].value,1);
    }
    questionHolder.innerHTML = questionToShow;
}
function saveMyFile(){
    localStorage.setItem("questionBank", JSON.stringify(questionBank));
    location.replace("quiz.html")
}
showQuestions();
const questionBank = [];
const newObject = localStorage.getItem("questionBank");
let dataStored = JSON.parse(newObject);
  if(dataStored != ""){
    for(i=0; i < dataStored.length; i++){
      questionBank.push(dataStored[i]);
    }
  } else {
    for(i=0; i < data.length; i++){
        questionBank.push(data[i]);
    }
  }
const removeButton = document.getElementById("removeQuestions");
const saveButton = document.getElementById("saveQuestions");
removeButton.addEventListener("click", removeQuestion);
saveButton.addEventListener("click", saveMyFile);
```

As usual, let's break it down:

```
function submitRemoveForm(event){
    event.preventDefault();
}
//When the form is submitted prevent the default operation

function showQuestions(){
const questionHolder = document.getElementById('showQstn');
//reference the showQstn div
```

```
questionToShow = [];
//make a new empty array called questionToShow for storage of questions
and check boxes for the users view

const newObject = localStorage.getItem("questionBank");
//get the questions from local storage

let dataStored = JSON.parse(newObject);
//parse the questions as a JSON object into a variable called dataStored

  if(dataStored != ""){
//if dataStored contains data

    for(i=0; i < dataStored.length; i++){
//for every question stored….

        questionToShow.push(
//push into the questionToShow array

            `<label>
            <input type="checkbox" name="${i}" value="${i}"
class="chk_butn">
//pass the index number if the checkbox is ticked

            ${i+1}. ${dataStored[i].question.substring(0, 32)}...
//display the question number (index + 1) and 32 characters of the
question, so that it will fit neatly onto a mobile phone screen without
wrapping onto more than one line

            </label><br>`
//wrap the input in a <label</label> to make the whole thing clickable
        );
    }
  } else {
//otherwise….if there is no questions stored in local storage

    for(i=0; i < data.length; i++){
//for all data obtained from the original JSON file
```

```
        questionToShow.push(
//push each question and checkbox into the questionToShow array
            `<label>
            <input type="checkbox" name="${i}" value="${i}"
class="chk_butn">
//pass the index number if the checkbox is ticked

            ${i+1}. ${data[i].question.substring(0, 32)}...
//display the question number (index + 1) and 32 characters of the
question, so that it will fit neatly onto a mobile phone screen without
wrapping onto more than one line

            </label><br>`
//wrap the input in a <label</label> to make the whole thing clickable

        );
    }
}
    questionHolder.innerHTML = questionToShow;
//display the questions and checkboxes in the question holder

    return questionToShow;
//return questions data
}
```

Okay, it is important to note that so far we have simply displayed the questions for the user depending on whether there are updates available and stored in local storage, or if the questions are being accessed from the original JSON question bank we created.

Next, let's take a look at how we will remove those questions:

```
function removeQuestion(){
//Open the function

    const questionHolder = document.getElementById('showQstn');
//reference the showQstn div ID and store it in a variable named
questionHolder
```

```
    questionsToRemove = [];
//make a new empty array where we will store the index numbers of
questions we want to remove

    let checkboxes =
document.querySelectorAll('input[type=checkbox]:checked');
//get all indexes from ticked check boxes and store them in a variable
called checkboxes

    for (var c = 0; c < checkboxes.length; c++) {
//iterate through each check box index

      questionToShow[checkboxes[c].value] = "";
//remove each ticked question from display

      questionBank.splice(checkboxes[c].value,1);
//remove each ticked question from the question bank

    }
    questionHolder.innerHTML = questionToShow;
//display the remaining questions and checkboxes in the question holder

}// close function
```

Okay, that was fairly straightforward. Now all that's left to do is save the changes in local storage, set our variables, and set up our listeners to capture any button clicks.

Firstly, the saveMyFile() function:

```
function saveMyFile(){
    localStorage.setItem("questionBank", JSON.stringify(questionBank));
//when the save button is clicked, this updates our local storage with
the question bank data with chosen questions removed.

    location.replace("quiz.html"); //this redirects back to the quiz page
}
```

Then we call the main showQuestions() function and set the variables, like so:

```
showQuestions();
//call (invoke) showQuestions function

const questionBank = [];
//creates an empty array called questionBank ready to store our data

const newObject = localStorage.getItem("questionBank");
//gets the question bank from local storage

let dataStored = JSON.parse(newObject);
//parse JSON data into a JavaScript object and store in a variable called
dataStored

  if(dataStored != ""){
//if dataStored is not empty…


    for(i=0; i < dataStored.length; i++){
//iterate through the dataStored JSON object

      questionBank.push(dataStored[i]);
//push each question into the questionBank array
    }//close for loop
  } else {
//otherwise… if local storage is empty

    for(i=0; i < data.length; i++){
//iterate through the data object pulled in from the JSON file in the
<head></head> of our HTML mark-up

        questionBank.push(data[i]);
//push each question into the questionBank array
    }//close for loop

  }// close conditional
```

Last, but not least, we reference the buttons for removing and saving questions as well as setting up event listeners to listen for button clicks.

```
const removeButton = document.getElementById("removeQuestions");
const saveButton = document.getElementById("saveQuestions");
//reference button by ID and store them in appropriately named variables

removeButton.addEventListener("click", removeQuestion);
//listen for button clicks of the remove question button. When clicked
call the removeQuestion function

saveButton.addEventListener("click", saveMyFile);
//listen for button clicks of the save button. When clicked call the
saveMyFile function
```

Top class! Now all that's left to do is enable navigation between the quiz and admin functions.

HOW TO SET UP NAVIGATION BETWEEN PAGES

For the final part of this exercise, we will need to add a little mark-up to each of our pages and add a few new rules to our CSS file.

Open the 'quiz_app.html' file and add the following inside the following the </h1> closing tag:

```
<span id="menu"></span>
<div id="menubox" style="display: none;">
    <br><a href="add_question.html" alt="add questions" title="add
questions">  Add Questions  </a><br><br>
    <a href="remove_questions.html" alt="remove questions" title="remove
questions"> Drop Questions </a>
    <span id="menudismiss"> X </span>
</div>
```

Then scroll to the bottom of the file and just inside the closing </body> tag, add the following:

```
    <script src='js/menu.js'></script>
```

Then save the file (CTRL + S).

Next, open the 'add_question.html' file and add the following inside the following the </h1> closing tag:

```
<span id="menu"></span>
<div id="menubox" style="display: none;">
    <br><a href="quiz.html" alt="back to quiz" title="back to quiz"> Back
to Quiz </a><br><br>
    <a href="remove_questions.html" alt="remove questions" title="remove
questions"> Drop Questions  </a>
    <span id="menudismiss"> X </span>
</div>
```

Again, scroll to the bottom of the file and just inside the closing </body> tag, add the following:

```
    <script src='js/menu.js'></script>
```

Then save the file (CTRL + S).

Next, open the 'remove_questions.html' file and add the following inside the following the </h1> closing tag:

```
<span id="menu"></span>
<div id="menubox" style="display: none;">
    <br><a href="quiz.html" alt="back to quiz" title="back to quiz"> Back
to Quiz </a><br><br>
    <a href="add_question.html" alt="add questions" title="add
questions">  Add Questions  </a>
    <span id="menudismiss"> X </span>
</div>
```

Again, scroll to the bottom of the file and just inside the closing </body> tag, add the following:

```
    <script src='js/menu.js'></script>
```

Then save the file (CTRL + S).

We will take a closer look at one of these examples, as the logic is all the same, although they are slightly different. The differences are that each menu set navigates to specific pages.

```
        <span id="menu"></span>
//used to display a clickable menu icon – a burger

        <div id="menubox" style="display: none;">
//used to style and enclose the menu

            <br><a href="quiz.html" alt="back to quiz" title="back to
quiz"> Back to Quiz </a><br><br>
//hyperlink 1 – back to quiz page – specific to this page

            <a href="add_question.html" alt="add questions" title="add
questions">  Add Questions  </a>
//hyperlink 2 – to add questions page – specific to this page

            <span id="menudismiss"> X </span>
//displays a close menu control

        </div> // close menubox div
```

Time for a quick break, before we finish off the exercise.

STYLING OUR SIMPLE NAVIGATION MENU

Hopefully, you have recharged your batteries a little. Next, we need to add styling rules for our menu elements. Open the 'app.css' file and scroll to the bottom, then add the following:

```
   #menu {
        position: absolute;
        top: 10px;
        right: 30px;
    }
    #menu:hover{
        background-color: rgba(255,255,255,0.5);
        font-weight: bold;
    }
    #menubox {
        position: absolute;
        background-image: -webkit-gradient(linear, left top, left bottom,
from(#ccc), to(#999));
        padding: 20px;
        border: solid 1px #333;
        width: 80vw;
        margin: 50px 0 0 7vw;
        height: 200px;
        z-index: 10;
        font-size: 1.7em;
    }
    #menubox a {
        color: #333;
        text-shadow: 0px 1px 0px #fff;
        text-decoration: none;
    }
    #menubox a:hover{
        background-color: rgba(255,255,255,0.5);
    }
```

```
#menubox #menudismiss{
    position: absolute;
    top: 10px;
    right: 15px;
    color: rgb(102, 100, 100);
}
#menubox #menudismiss:hover{
    position: absolute;
    top: 10px;
    right: 15px;
    background-color: rgba(255,255,255,0.5);
    font-weight: bold;
}
```

As with previous additions to our style sheet, you have seen much of the commands before, so I'll only explore those we haven't already covered:

```
border: solid 1px #333;

// apply a solid border, 1 pixel in width and dark grey in colour

margin: 50px 0 0 7vw;

// although we have dealt with margins before the 7vw is a size measurement I haven't mentioned vw stands for viewport width. You can also use vh or viewport height. Essentially, we are saying 7% of the viewport width as a left margin setting (TRBL)

#menubox a:hover

//this applies rules to <a> tags that are being hovered by the mouse inside an element with the ID of menubox
```

```
background-color: rgba(255,255,255,0.5);
//finally, we have seen color and background-color settings with named colours and hexadecimal colour codes, but here we are using 'r, g, b, a' (red, green, blue, alpha) settings. 'r, g and b' are all values from 0 – 255 or black through to white, while the 'a' represents the alpha channel or opacity channel. The value we have chosen 255, 255, 255, 0.5 is white in colour with an opacity of 50%.
```

JAVASCRIPT TO HANDLE SIMPLE NAVIGATION

Make a new file called 'menu.js' and save it in the 'js' folder. Then add the following code:

```
function htmlMenu(){
    const menu = document.getElementById('menu');
    const menudismiss = document.getElementById('menudismiss');
    menu.innerHTML = '<h1>&#9776;</h1>';
    menu.addEventListener("click", displayMenu);
    menudismiss.addEventListener("click", dismissMenu);
}
function displayMenu(){
    const menubox = document.getElementById('menubox');
    if(menubox.style.display == "block"){
        menubox.style.display = "none";
    } else {
        menubox.style.display = "block";
    }
}
function dismissMenu(){
    const menubox = document.getElementById('menubox');
        menubox.style.display = "none";
}
htmlMenu();
```

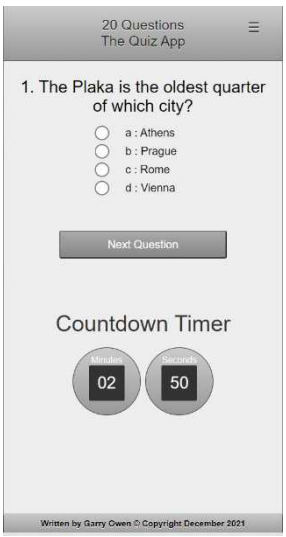
In usual form, let's break it down!

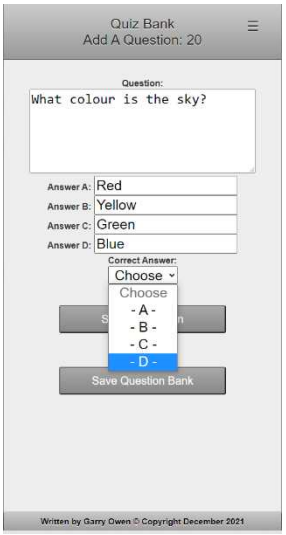
```
function htmlMenu(){
//open htmlMenu function

    const menu = document.getElementById('menu');
//reference the element with the ID of menu and store it in a variable
named menu

    const menudismiss = document.getElementById('menudismiss');
//reference the element with the ID menudismiss and store it in a
variable

    menu.innerHTML = '<h1>&#9776;</h1>';
//set the menu to a burger symbol

    menu.addEventListener("click", displayMenu);
//add an event listener to check for menu clicks. If clicked open the
menu

    menudismiss.addEventListener("click", dismissMenu);
//add an event listener to check for menu dismiss clicks. If clicked
close the menu

}//close function

function displayMenu(){
//open display menu function

    const menubox = document.getElementById('menubox');
//reference the element with the ID of menubox and store it in a variable
named menubox

    if(menubox.style.display == "block"){
//if the menu box is display.ed…

        menubox.style.display = "none";
//…remove it from display
    } else {
//otherwise, if the menu isn't displayed
```

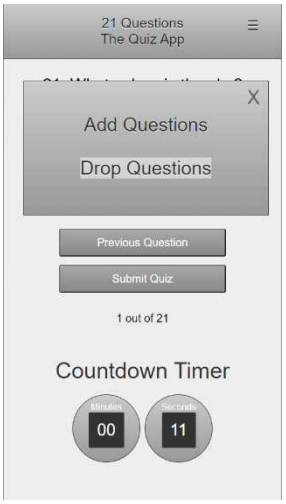
```
        menubox.style.display = "block";
//display the menu

    }//close conditional
}//close function

function dismissMenu(){
//open the dismiss menu function

    const menubox = document.getElementById('menubox');
//reference the element with the ID menubox and store it in a variable
named menubox

        menubox.style.display = "none";
//remove the menu from the display

}//close the function

htmlMenu();
//call the htmlMenu function
```

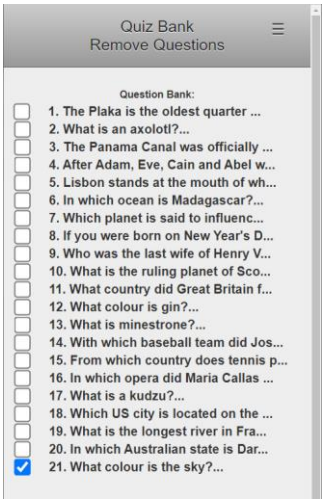
And we're done….you've done superbly. Very well done!

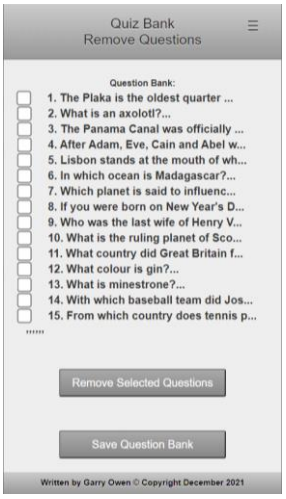
If you followed this exercise correctly you should now have the following great quiz application.











So we created a quiz app with a countdown timer, the ability to add new questions, and also the ability to remove questions. Have a play around with it. See if you can improve it! Try adding even more features. Maybe a high score table, the ability to choose from multiple question sets, perhaps? There are endless possibilities.

All files and images associated with this exercise can be found at:

https://wddtrw/resources/learntocode/quizapp.zip

CREATING A PLATFORM GAME

WHAT IS A PLATFORM GAME?

Platform games, also called platformers, are games in which the player controls a character that runs/walks and jumps on platforms. Some of the famous ones are Kong, Sonic the Hedgehog, and Mario Brothers, to name just a few.

We are going to create a game called 'Sorcerers Mountain'. That's what I named it, but of course, you may name it whatever you like.

The objective will be to collect all of the jewels while avoiding enemies and obstacles along the way. This is a multi-part exercise that follows on in subsequent books. However, by the end of this exercise, you will have a fully playable platform game and will have learned many skills. That said, without even knowing it, you have been developing many of the skills you need while following earlier exercises in this book.

LESSON OBJECTIVE:

Design and develop a web-based platform game using HTML, CSS, and JavaScript.

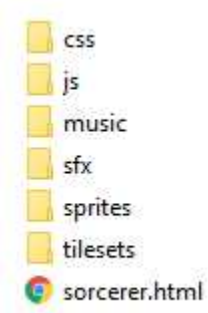
The game we develop will include:

- A tile map for each screen
- A game monitor
- The main character
- Three different enemies – 1 static and 2 animated
- Collectables – Jewels, Coins much more
- Obstacles – walls, jumps, barrels

- Collisions and Gravity
- Sound Effects (SFX)
- Sprites and Tiles
- Title Screen and Game Over Screen
- Background Images
- Music

STEP ❶ - SETTING UP YOUR FILE STRUCTURE

Here goes nothing, guys! Let's start building our final exciting project. Firstly, let's implement our required file structure. This time we need folders to accommodate different kinds of assets. Make a new project folder called 'game' and save it on your desktop or another desired location. Inside that folder, add the following file structure:



By now you should know how to create these, but in case you need a reminder. Double click on your 'game' folder. Then right-click, and from the context menu, choose new and then folder enter 'css' and hit enter. Repeat this process for 'js', 'music', 'sfx', 'sprites', and 'tilesets'. Finally, open your chosen coding editor and create a new file and save it as 'sorcerer.html' within the 'game' folder. Job done!

STEP ❷ - CREATING THE HTML FILE FOR OUR PLATFORM GAME

Open the 'sorcerer.html' file (if it's not already open) and add the following mark-up:

```
<html>
<head>
    <title>Sorcerers Mountain</title>
    <link rel="stylesheet" type="text/css" href="css/app.css"/>
</head>
<body>
    <div id="gameArea">
        <canvas id="canvas"></canvas>
    </div>
    <script type="text/javascript" src="js/assets.js"></script>
    <script type="text/javascript" src="js/tiles.js"></script>
    <script type="text/javascript" src="js/sprites.js"></script>
    <script type="text/javascript" src="js/main.js"></script>
    <script type="text/javascript" src="js/startScreen.js"></script>
    <script type="text/javascript" src="js/map1.js"></script>
</body>
</html>
</html>
```

The vast majority of this mark-up you will already be familiar with if you have been following the exercises throughout this book. As you can see, at this stage, it looks quite simple. The good news is that this HTML won't get any more complex. We will add references to more JavaScript files as we build our game, but that is all. Let's take a closer look at the mark-up.

```
<html>
//tell the browser we are using HTML

<head>
    <title>Sorcerers Mountain</title>
//Add a title to our HTML page
```

```
    <link rel="stylesheet" type="text/css" href="css/app.css"/>
//reference ‘app.css’ to allow us to set styling rules for our page

</head>

<body>
//open the body – the main content area

    <div id="gameArea">
        <canvas id="canvas"></canvas>
    </div>
//add a canvas element within a div. This will be used to display our
game and make necessary references for our CSS and JavaScript commands

    <script type="text/javascript" src="js/assets.js"></script>
    <script type="text/javascript" src="js/tiles.js"></script>
    <script type="text/javascript" src="js/sprites.js"></script>
    <script type="text/javascript" src="js/main.js"></script>
    <script type="text/javascript" src="js/startScreen.js"></script>
    <script type="text/javascript" src="js/map1.js"></script>
//reference JavaScript files. Bring it all together

</body>
</html>
//close the body and html tags
```

STEP ❸ - CREATING THE CSS FOR OUR PLATFORM GAME

Create a new file called ‘app.css’ and save it in the ‘css’ folder. Then, add the following CSS rules:

```
body, html {
    height: 100%;
    overflow:hidden;
    background: #1f1f1f;
}

canvas{
```

```
    border:1px solid black;
}

#gameArea{
    text-align: center;
    margin-top: 70px;
}
```

This is very simple CSS rules. All of which you have seen earlier in the book. Let's take a closer look:

```
body, html {
    height: 100%;
    overflow:hidden;
    background: #1f1f1f;
}
//set rules for the whole html element and the body. Set height to 100%, do not show scrollbars if the content is larger than the page, and set the background to dark grey in colour.

canvas{
    border:1px solid black;
}
//set rules for the canvas element – make a solid 1 pixel border with the colour black

#gameArea{
    text-align: center;
    margin-top: 70px;
}
//set rules for the element with an ID of gameArea. Align the game area to the centre and 70 pixels from the top of the screen.
```

As was already said, these are all very straightforward rule sets.

Moving on!!

STEP ❹ - CREATING ASSETS FOR OUR PLATFORM GAME

Next, create a new file called 'assets.js' and save it in the 'js' folder. Then add the following code and save it (CTRL + S).

```
var canvas = document.querySelector("#canvas"),
    ctx = canvas.getContext("2d"),
    width = 1152,
    height = 704,
    player = {
        screen: 15,//15 = START SCREEN
    },
    press_s = { width: 500, height: 40, f: 100 },
    instruct = { x: 126, y: 300, w: 900, h: 57, f: 103 };
```

What are we doing here? The 'assets.js' file is used to set all initial values and store them in a JSON object. Let's take a closer look:

```
var canvas = document.querySelector("#canvas"),
// reference the canvas element by its ID and store it in a variable
called canvas

    ctx = canvas.getContext("2d"),
// To enable canvas' 2D rendering context on the canvas element (allows
us to draw in 2D on the canvas)

    width = 1152,
//set the canvas width to 1152 pixels – very important that you use a
size that is a multiple of your intended tile size. Our chosen tile size
is 64 pixels x 64 pixels (1152 pixels will accommodate 18 tiles across)

    height = 704,
//set the canvas height to 704 pixels. As with the width, it must be a
multiple of your chosen tile size (704 pixels will accommodate 11 tiles
down)
```

```
    player = {
        screen: 15,
    },
//Set up a player object where we will store all start values for the
player. Here we have set the initial screen to 15 which represents the
title screen background, as we'll see in more detail in the next section

    press_s = { width: 500, height: 40, f: 100 },
//set the width, height and frame of a press 's' on screen message to
start the game

    instruct = { x: 126, y: 300, w: 900, h: 57, f: 103 };
//set the initial x coordinate, y coordinate, width, height, and frame of
our sprite. We will use this to show the user instructions when they
press and hold 'i'.
```

This won't do anything by itself as it works in conjunction with other JavaScript files we are yet to create.

STEP ❺ - CREATE AND REFERENCE TILES FOR BUILDING OUR TILE MAPS

Create a new file called 'tiles.js' and save it in the 'js' folder.

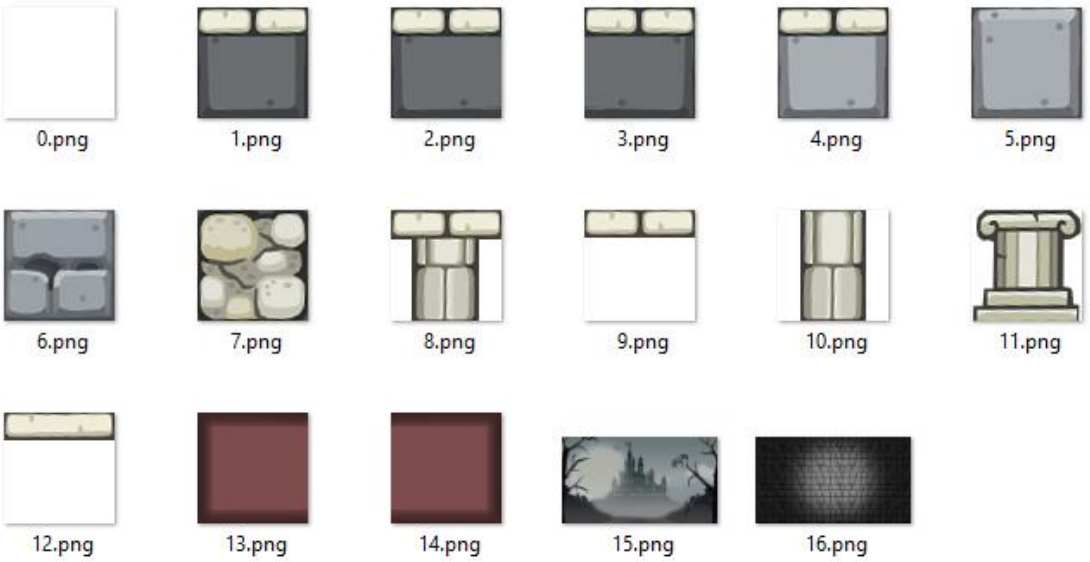
Before we add any code to this file you'll need to download the tile images to the 'tilesets' folder from the following URL:

https://wddtrw/resources/learntocode/platformgame_building_tiles.zip

Once downloaded, unzip it and add the contents to the 'tilesets' folder. To unzip the file, double click it and then select all of the contents (CTRL + A) and copy it (CTRL + C), then open the 'tilesets' folder and paste (CTRL + V).

If you have followed this correctly your 'tilesets' folder will now contain the following 64 x 64 pixel tiles, plus 2 background images. You can of

course choose to make your own. If you wish to do that or edit the ones from the file to make them a little different you will need an image editor. If you don't have access to one I recommend Gimp, which is more than adequate to do the job, and best of all it's free. You can download it for free at https://gimp.org.



We will use these tiles in our game screens using tile maps. More on that in a bit. For now, let's set up a JavaScript file to reference them and enable their usage in our game.

Create a new file called 'tiles.js' and save it in the 'js' folder, then add the following code. We will start small here, but add more tiles as we need them, when building the game.

```
/* ================================
TILE KEY:
0 - blank space
1 - base block dark grey
2 - base block left - lives
3 - base block right - lives
```

```
4 - base block light grey
5 - wall block light grey
6 - wall block bricks light grey
7 - stones block
8 - pedestal topper
9 - floating platform 2 bricks
10 - pedestal base
11 - low pedestal
12 - floating platform 1 brick
13 - game monitor left tile
14 - game monitor right tile
15 - title screen background
16 - game screen background stone wall
================================== */
    const tile = [];
    for(i=0; i < 17; i++){
        tile[i] = new Image();
        tile[i].src = `tilesets/building_tiles/${i}.png`;
    }
```

Let's take a look a closer look at the code:

Firstly, note the extensive comments. This is so that we know which index number to use when drawing our tile maps.

```
    const tile = [];

//create an empty array and store it in a variable called tile

   for(i=0; i < 17; i++){

//iterate through our tiles and add them to our array

    tile[i] = new Image();
//set index 'i' of the array to a new image object

tile[i].src = `tilesets/building_tiles/${i}.png`;
//set the source file of the image object
```

Note, for this method to work the file names need to be sequential and in the same format. We can also add tiles individually to our tile array using similar coding, but specifying the index instead of using the 'for loop', like so:

```
tile[99] = new Image();
tile[99].src = 'tilesets/building_tiles/[FILENAME HERE]';
```

STEP 6 - CREATE AND REFERENCE SPRITE OBJECTS

Next, create a new file called 'sprites.js' and save it in the 'js' folder.

Before we add any code to this file you'll need to download the images to the 'sprites' folder from the following URL:

https://wddtrw/resources/learntocode/platformgame_title_sprites.zip

Once downloaded, unzip it and add the content to the 'sprites' folder. To unzip the file, double click it and then select all of the contents (CTRL + A) and copy it (CTRL + C), then open the 'sprites' folder and paste (CTRL + V).

If all went well you should have the following:

Okay, now add the following code to your 'sprites.js' file:

```
/* ============================
SPRITES KEY:
100 - Press 's'
101 - Press 's' flash
102 - Instructions
103 - Title - Sorcerers Mountain
104 - Instruction Message
105 - Controls
106 - Copyright Message
107 - Get ready text
108 - Get ready text flash
109 - Game Over text
============================== */
var sprite = [];
for(i = 100; i < 110; i++){
    sprite[i] = new Image();
    sprite[i].src = `sprites/${i}.png`;
}
```

This works in the same way as to how we referenced our tiles. As with our tiles, we will also be adding many more sprites as we get further into our game development.

STEP❻ - CREATE THE MAIN JAVASCRIPT FILE TO TIE IT ALL TOGETHER

Open your chosen coding environment, if it's not already open, and create a new file called 'main.js' and save it in the 'js' folder. Then add the following code and save it (CTRL + S).

```
canvas.width = width;
canvas.height = height;
function update() {
    ctx.clearRect(0, 0, width, height);
    ctx.fillStyle = "#333";
    ctx.beginPath();
```

```
    if (player.screen === 15){ drawMyMap1000();}
    if (player.screen === 1){ drawMyMap(); }
    requestAnimationFrame(update);
}
window.addEventListener("load", function () {
    update();
});
document.body.addEventListener("keydown", function (e) {
    keys[e.keyCode] = true;
});
document.body.addEventListener("keyup", function (e) {
    keys[e.keyCode] = false;
});
```

The 'main.js' file will hold the code that will enable us to load different maps based on key presses and/or the player leaving the confines of the current game screen. We will be adding lots more logic here as we build the game. For now, let's take a deeper look at what we've done so far:

```
canvas.width = width;
//bring in the initial setting canvas width from our assets file

canvas.height = height;
//bring in the initial setting canvas height from our assets file

function update() { // Main screen update function

    ctx.clearRect(0, 0, width, height);// Clear the canvas
    ctx.fillStyle = "#333"; //fill canvas with dark grey colour
    ctx.beginPath();// begin or reset current path

    if (player.screen === 15){ drawMyMap1000();} // draw the title screen
    if (player.screen === 1){ drawMyMap(); }// draw the first game screen
    requestAnimationFrame(update);// animate the current frame
}//close update function

window.addEventListener("load", function () {//when web page has loaded…
    update();//call the update function
```

```
});
document.body.addEventListener("keydown", function (e) {
    keys[e.keyCode] = true;
});
//listen for key down events and get the relevant key code

document.body.addEventListener("keyup", function (e) {
    keys[e.keyCode] = false;
});
//listen for key up events and get the relevant key code
```

STEP 7 - HANDLING THE START / TITLE SCREEN

Next, we are going to create the first script that will display some results. Create a new file called 'startScreen.js' and save it in the 'js' folder. Now add the following code and save it (CTRL + S).

```
function drawMyMap1000(){
ctx.drawImage(tile[15], 0, 0, 1152, 704);
 if(keys[83]){
        player.screen = 1;
 }
 if(keys[73]){
    instruct.f = 102; instruct.x = 126, instruct.y = 25, instruct.w =
900, instruct.h = 567;
 } else {
    ctx.drawImage(sprite[104], 383.5, 380, 385, 24);
    instruct.f = 103; instruct.x = 126, instruct.y = 200, instruct.w =
900, instruct.h = 57;
    ctx.drawImage(sprite[105], 476, 435, 200, 137);
    ctx.drawImage(sprite[106], 342, 275, 469, 85);
 }
    ctx.drawImage(sprite[instruct.f], instruct.x, instruct.y, instruct.w,
instruct.h);
    ctx.drawImage(sprite[press_s.f], 326, 610, press_s.width,
press_s.height);
    if (press_s.f === 100){
        setTimeout(function(){ press_s.f = 101; },200);
```

```
    } else {
        setTimeout(function(){ press_s.f = 100; },200);
    }
}
```

Top class! Let's take a closer look and see what's going on:

```
function drawMyMap1000(){//open the draw start screen function

ctx.drawImage(tile[15], 0, 0, 1152, 704);
//draw the background image - tile[15] the stored title image, the rest
is coordinates 0,0 :top left - 1152, 704 :bottom right

 if(keys[83]){// if the user presses 's' to start
        player.screen = 1; //change the screen to map 1
 }

 if(keys[73]){// if the user presses 'i'….

    instruct.f = 102; instruct.x = 126, instruct.y = 25, instruct.w =
900, instruct.h = 567;
//show the instructions image – sprite 102

 } else {//otherwise….

    ctx.drawImage(sprite[104], 383.5, 380, 385, 24);
    instruct.f = 103; instruct.x = 126, instruct.y = 200, instruct.w =
900, instruct.h = 57;
//draw the instructions message

    ctx.drawImage(sprite[105], 476, 435, 200, 137);
//draw the user controls

    ctx.drawImage(sprite[106], 342, 275, 469, 85);
// …and draw the copyright message
 }
```

```
    ctx.drawImage(sprite[instruct.f], instruct.x, instruct.y, instruct.w,
instruct.h);
//Always draw the title - Sorcerers Mountain

    ctx.drawImage(sprite[press_s.f], 326, 610, press_s.width,
press_s.height);
//bring in the initial values for press_s from assets

    if (press_s.f === 100){
        setTimeout(function(){ press_s.f = 101; },200);
    } else {
        setTimeout(function(){ press_s.f = 100; },200);
    }
//make the 'Press s to Start message animate between image 100 and 101
every 0.2 seconds

}//close draw map function
```

Once this code has been saved you will now be able to view the title screen. Open 'sorcerer.html' in your browser (Google Chrome recommended). If you've followed the exercise correctly so far, you should have the following result:

It's looking great so far. Moving on…

STEP 8 - BUILDING OUR FIRST TILE MAP

Now let's build our first tile map!

Create a new file called 'map1.js' and save it in the 'js' folder. Okay, now add the following code, then save the file (CTRL + S).

```
function drawMyMap(){
ctx.drawImage(tile[16], 0, 0);
var xt = 0;
var yt = -64;
var tileMap = [];
var mapNo = 0;
tileMap[0] =  [6, 6, 6, 6, 6, 6, 0, 0, 0, 6, 6, 6, 6, 6, 6, 6, 6, 6];
tileMap[1] =  [6, 0, 0, 0, 10, 0, 0, 11, 0, 0, 10, 0, 6, 0, 0, 0, 0, 0];
tileMap[2] =  [6, 0, 0, 9, 9, 9, 9, 9, 9, 9, 8, 0, 0, 0, 0, 0, 0, 0];
tileMap[3] =  [6, 0, 11, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 9, 9, 0, 6];
tileMap[4] =  [6, 9, 9, 9, 0, 0, 0, 0, 0, 0, 8, 9, 9, 0, 0, 0, 0, 6];
tileMap[5] =  [6, 0, 0, 0, 0, 9, 9, 0, 0, 0, 10, 0, 0, 9, 0, 9, 9, 6];
tileMap[6] =  [6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 9, 9, 0, 0, 0, 0, 0, 6];
tileMap[7] =  [0, 0, 0, 0, 0, 0, 9, 9, 0, 0, 0, 0, 0, 8, 9, 8, 9, 6];
tileMap[8] =  [9, 9, 9, 8, 9, 0, 0, 0, 0, 0, 0, 0, 8, 10, 0, 10, 0, 6];
tileMap[9] =  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3];
tileMap[10] = [13, 14, 13, 14, 13, 14, 13, 14, 13, 14, 13, 14, 13, 14,
13, 14, 13, 14];
    for (mapNo=0; mapNo < 11; mapNo++){
        yt+=64;
        for (xt=0; xt < tileMap[mapNo].length*64; xt+=64){
        if (xt > 1152){ xt = 0; }
            var i = tileMap[mapNo][xt/64];
            ctx.drawImage(tile[i], xt, yt);
      }
    }
}
```

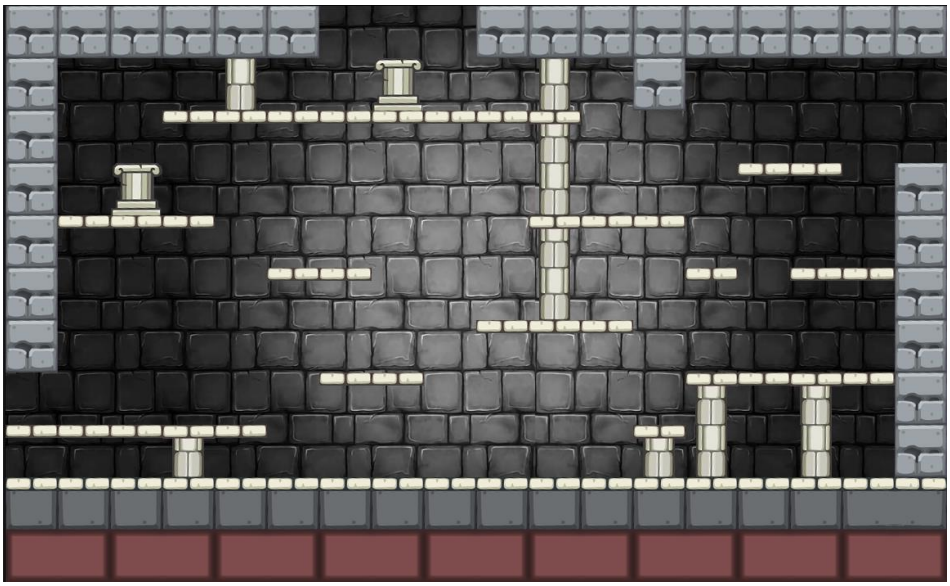
Each of our game maps will be drawn in the same way. Let's take a closer look:

```
function drawMyMap(){
//open the draw map function

ctx.drawImage(tile[16], 0, 0);
//draw the game screen background – a stone wall

var xt = 0; //tile x coordinate variable declaration
var yt = -64; //tile y coordinate variable declaration
var tileMap = []; //creates a new empty array to store the tile map
var mapNo = 0; //map rows variable declaration

tileMap[0] =  [6, 6, 6, 6, 6, 6, 0, 0, 0, 6, 6, 6, 6, 6, 6, 6, 6];
tileMap[1] =  [6, 0, 0, 0, 10, 0, 0, 11, 0, 0, 10, 0, 6, 0, 0, 0, 0, 0];
tileMap[2] =  [6, 0, 0, 9, 9, 9, 9, 9, 9, 9, 8, 0, 0, 0, 0, 0, 0, 0];
tileMap[3] =  [6, 0, 11, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 9, 9, 0, 6];
tileMap[4] =  [6, 9, 9, 9, 0, 0, 0, 0, 0, 0, 8, 9, 9, 0, 0, 0, 0, 6];
tileMap[5] =  [6, 0, 0, 0, 0, 9, 9, 0, 0, 0, 10, 0, 0, 9, 0, 9, 9, 6];
tileMap[6] =  [6, 0, 0, 0, 0, 0, 0, 0, 0, 9, 9, 9, 0, 0, 0, 0, 0, 6];
tileMap[7] =  [0, 0, 0, 0, 0, 0, 9, 9, 0, 0, 0, 0, 0, 8, 9, 8, 9, 6];
tileMap[8] =  [9, 9, 9, 8, 9, 0, 0, 0, 0, 0, 0, 0, 8, 10, 0, 10, 0, 6];
tileMap[9] =  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3];
tileMap[10] = [13, 14, 13, 14, 13, 14, 13, 14, 13, 14, 13, 14, 13, 14,
13, 14, 13, 14];
//stores tile index numbers in the tileMap array

    for (mapNo=0; mapNo < 11; mapNo++){
//iterate through each map row

        yt+=64;
//adds 64 to yt for every iteration of the for loop

        for (xt=0; xt < tileMap[mapNo].length*64; xt+=64){
//iterate through each map 64 pixel wide column

            if (xt > 1152){ xt = 0; }
//if at the end of the row, start another
```

```
            var i = tileMap[mapNo][xt/64];
//set the variable i to hold each tile reference

            ctx.drawImage(tile[i], xt, yt);
//draw the tiles spaced 64 pixels apart

 }// close column for loop

 }// close row for loop

}// close draw function
```

Great job! If you have followed the exercise correctly when you hold down the 'i' key, you should be presented with the following user instruction set:

…And if you hit the 's' key you should be presented with your first tile map, as shown over the page:

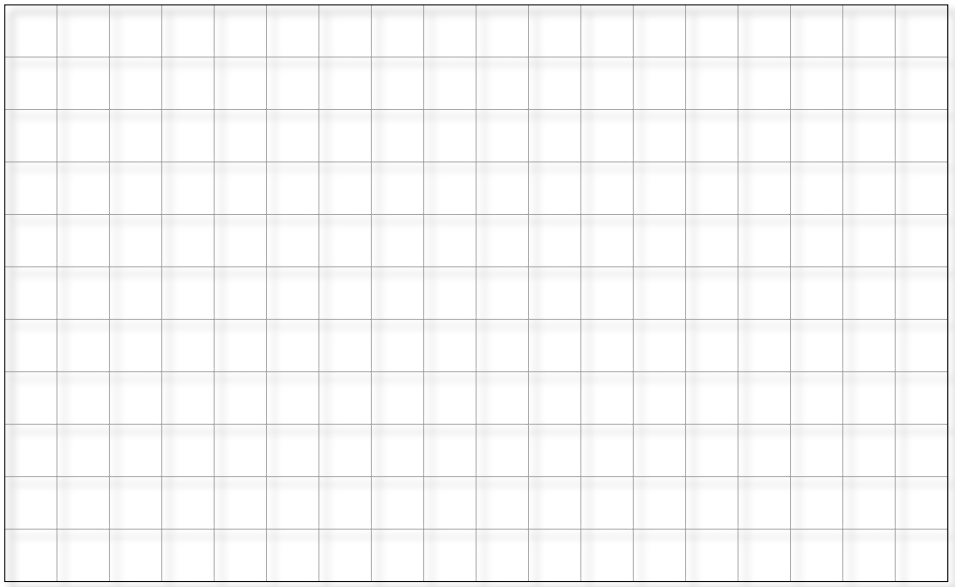


Awesome stuff!

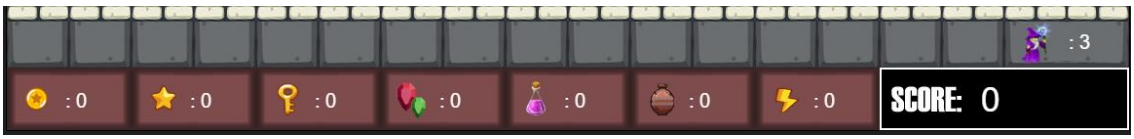
It’s a good time to have a break and recharge your batteries.

STEP 9 - DESIGNING TILE MAPS

In simple terms, to make the first tile map we have populated a grid that is 1152 pixels wide and 704 pixels tall, which is split into 64 by 64-pixel tiles. A representation is shown below:



Each 64 x 64 square is a placeholder for one of our tiles. I have designed each of the screens so that the bottom 2 rows are always the same. These tiles will eventually be used for a game monitor, to show lives, player score, collected items, and so on. The game monitor will eventually look something like that below:



The rest of the tiles, row 0 to row 8 will be changed to form the game screens. Where we want to be able to exit to allow our player into other

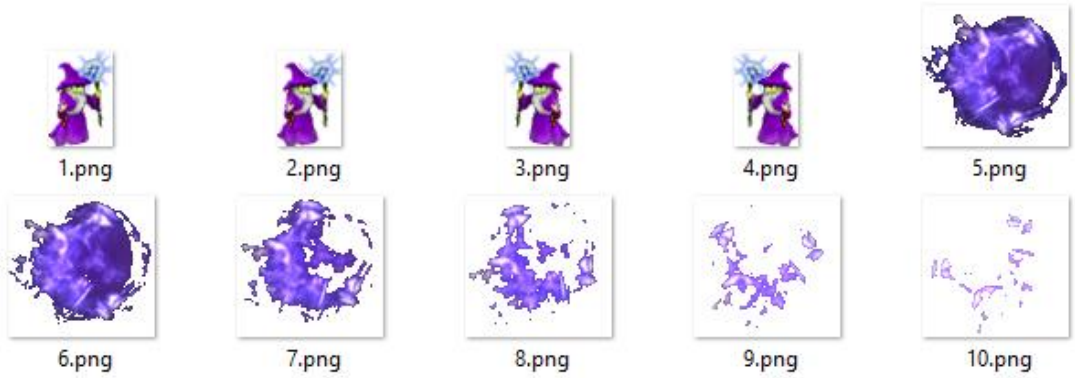
screens we'll need to leave gaps using tile[0]. Designing our next tile map is a little way off though, so for now, let's add our main player and give it a few attributes.

First, you'll need to download the sprite images to the 'sprites' folder from the following URL:

https://wddtrw/resources/learntocode/platformgame_wizard_sprites.zip

Once downloaded, unzip it and add the content to the 'sprites' folder. To unzip the file, double click it and then select all of the contents (CTRL + A) and copy it (CTRL + C), then open the 'sprites' folder and paste (CTRL + V).

If all went well you should have the following:



As with previous tiles and sprites, you'll see that they are numbered sequentially.

Open the 'sprites.js' file and add the following code to the bottom of the file:

```
/* ============================
WIZARD KEY:
1 - Wizard Right Frame 1
2 - Wizard Right Frame 2
```

```
3 - Wizard Left Frame 1
4 - Wizard Left Frame 2
5 - Wizard Aura Frame 1
6 - Wizard Aura Frame 2
7 - Wizard Aura Frame 3
8 - Wizard Aura Frame 4
9 - Wizard Aura Frame 5
10 - Wizard Aura Frame 6
============================== */
var wiz = [];
for(i = 1; i < 11; i++){
    wiz[i] = new Image();
    wiz[i].src = `sprites/${i}.png`;
}
```

This works the same as last time. With each iteration of the 'for…loop,' an image is placed in the `wiz = []` array with the relevant index number.

As with the other sprites and tiles, we have commented them with a key, so that we can quickly see which index number references which sprite.

Great job!

Next, open the 'assets.js' file and update the player object, as below and save the file (CTRL + S):

```
    player = {
        x: 608,
        y: height - 182,
        width: 35,
        height: 52,
        screen: 15,
        start: 0,
        f: 3
    },
```

Okay, so we've set initial variables for our player. We have set the X coordinate to 608 pixels across our canvas, the Y coordinate to 522 (704 –

182), our player height and width to 52 pixels and 35 pixels respectively, the screen was already set at 15 (the title screen) and finally our animation frame number (f) to sprite number 3.

Okay, with that done, now open 'main.js' and add the following two conditionals and save the file (CTRL + S):

```
    if (player.screen === 1){ drawMyMap(); } ◄ add the following after
this line of code

    if (player.screen < 15 && player.start === 0) {
        player.start = 1;
    }
    if (player.screen != 15){
        ctx.drawImage(wiz[player.f], player.x, player.y);
    }
```

So what have we done here?

```
    if (player.screen < 15 && player.start === 0) {
        player.start = 1;
    }
//if the screen is no longer the title screen and the player start
attribute is 0, set the start attribute to 1

    if (player.screen != 15){
        ctx.drawImage(wiz[player.f], player.x, player.y);
    }
//if the player screen is not 15 – no longer the title screen, then draw
the player
```

Refresh your browser and press 's'. You will see that your player is now displayed.

Now let's add some controls to make your player move. Once again, edit the 'assets.js' file and add the following highlighted attributes:

```
    player = {
        x: 608,
        y: height - 182,// Start height (Must be larger than tile, plus
character height
        width: 35,
        height: 52,
        speed: 3,
        velX: 0,
        velY: 0,
        screen: 15, //15 = START SCREEN
        start: 0,
        f: 3
    },
    press_s = { width: 500, height: 40, f: 100 },
    instruct = { x: 126, y: 300, w: 900, h: 57, f: 103 },
    keys = [],
    gravity = 0.3,
    friction = 0.8;
```

These initialise values for speed, velocity on the X-axis, velocity on the Y axis, a value for gravity to allow our player to fall back to the ground after jumping, and another for friction to dampen the players' movement making the player come to a stop gradually. Save the file (CTRL + S).

Next, edit the 'main.js' file and add the following, just after the opening line of the update() function:

```
    if (keys[39]) {
        if (player.velX < player.speed) {
            player.velX++;
            player.f = 1;
        }
    }
    if (keys[37]) {
        if (player.velX > -player.speed) {
            player.velX--;
            player.f = 3
        }
```

```
    }
    player.velX *= friction;
    player.velY += gravity;
    player.x += player.velX;
```

Save the file (CTRL + S). Now, let's take a closer look at the code:

```
    if (keys[39]) {
        if (player.velX < player.speed) {
            player.velX++;
            player.f = 1;
        }
    }
//if the right arrow is pressed on the keyboard move the player to the
right and set the player sprite to sprite[1] – facing right

    if (keys[37]) {
        if (player.velX > -player.speed) {
            player.velX--;
            player.f = 3
        }
    }
//if the left arrow is pressed on the keyboard move the player to the
left and set the player sprite to sprite[3] – facing left

    player.velX *= friction;
// Set the velocity on X-axis to gradually stop by applying the friction
coefficient

    player.velY += gravity;
// Set the velocity on the Y-axis to fall back to the ground by applying
the gravity coefficient

    player.x += player.velX;
//move the player x position based on velocity X
```

Save your file (CTRL + S).

Now refresh your browser, press 's', and try moving your player left and right using the left and right arrows on your keyboard. Of course, you can choose alternative keys if you wish. Just change the index number of `keys[]` to match your desired key code. See the table below:

Key	Code		Key	Code		Key	Code
backspace	8		e	69		numpad 8	104
tab	9		f	70		numpad 9	105
enter	13		g	71		multiply	106
shift	16		h	72		add	107
ctrl	17		i	73		subtract	109
alt	18		j	74		decimal point	110
pause/break	19		k	75		divide	111
caps lock	20		l	76		f1	112
escape	27		m	77		f2	113
page up	33		n	78		f3	114
page down	34		o	79		f4	115
end	35		p	80		f5	116
home	36		q	81		f6	117
left arrow	37		r	82		f7	118
up arrow	38		s	83		f8	119
right arrow	39		t	84		f9	120
down arrow	40		u	85		f10	121
insert	45		v	86		f11	122
delete	46		w	87		f12	123
0	48		x	88		num lock	144
1	49		y	89		scroll lock	145
2	50		z	90		semi-colon	186
3	51		left window key	91		equal sign	187
4	52		right window key	92		comma	188
5	53		select key	93		dash	189
6	54		numpad 0	96		period	190

Key	Code		Key	Code		Key	Code
7	55		numpad 1	97		forward slash	191
8	56		numpad 2	98		grave accent	192
9	57		numpad 3	99		open bracket	219
a	65		numpad 4	100		back slash	220
b	66		numpad 5	101		close bracket	221
c	67		numpad 6	102		single-quote	222
d	68		numpad 7	103			

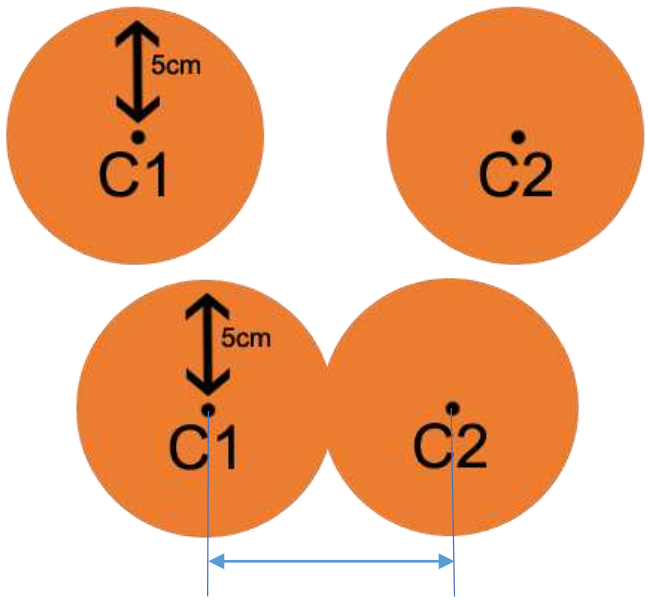
Although we can move our player, currently we can only move left and right.

Before we give our player the ability to be able to jump we need to set up collisions between the player and the environment. For instance, at the moment the player can walk through walls. Also, if we set gravity working right now, the player would fall straight through the floor. To be able to jump the player must also be able to return to the ground, so we need gravity. At the moment, although we have set the tile map so that we can see it, nothing is solid. So next, let's add some substance to the floor, walls, and platforms.

STEP 10 - COLLISION DETECTION

Let's first take a look at the concept of collision detection. It does involve some calculation, but if we use circles for our example, it's easier to grasp. Take two circles, both with a diameter of 10 cm, therefore a radius of 5cm. If we say the centre of circle 1 is C1 and the centre of circle 2 is C2. To be able to calculate if they have collided (overlapped) all we need to

work out is if the distance between C1 and C2 is less than 10cm. See the diagram below:



If it's more, then the circles cannot be touching, but less means that they have overlapped.
In order we need to:

1. calculate the size of the objects
2. find the centre of both objects
3. calculate the distance from the centre to the edge of both objects and find the total
4. calculate the difference between the current centre measurements compared with the total allowable distance

So our formula to detect a collision will look something like this:

```
let c1_middle = c1.width / 2; //5cm
let c2_middle = c2.width / 2; //5cm
let c_total = c1_middle + c2_middle;//10cm
if(((c1.x - c1_middle) - (c2.x - c2_middle)) < c_total || ((c1.y -
c1_middle) - (c2.y - c2_middle)) < c_total){
    //what to do when a collision has occured
}
```

We can of course simplify our equation further, but this is for the sake of example. Here c1.x c1.y c2.x c2.y represent current x and y coordinates of c1 and c2. So, we're saying if the current x positions of c1 and c2 is less than 10cm or the current y positions of c1 and c2 is less than 10cm, then a collision has occurred. The good news is that we only need to set up our collisions once. A single function will handle it all for us. It looks complicated on the face of it, but when it's broken down it's much more easily understood.

Until you can get your head around it, you can treat the function as sort of a black box. You only need to know the inputs required to get the desired outputs. However, mathematics plays a fundamental part in games programming so it is better if you can get to grips with it.

Okay, let's do it! Once again, open the 'assets.js' file and add the following code to the bottom of the file and save the file (CTRL + S):

```
    var boxes = [];
    var boxes1 = [];
    var boxesDrawn1 = 0;
//create 2 empty arrays for our collision coordinates and set up a
variable to check if the collision boxes have been drawn.
```

Next, create a new file called 'collisions.js' and save it in the 'js' folder. Add the following function and then save it (CTRL + S).

```
function colCheck(shapeA, shapeB)
    var vX = (shapeA.x + (shapeA.width / 2)) - (shapeB.x + (shapeB.width
/ 2)),
        vY = (shapeA.y + (shapeA.height / 2)) - (shapeB.y +
(shapeB.height / 2)),
        hWidths = (shapeA.width / 2) + (shapeB.width / 2),
        hHeights = (shapeA.height / 2) + (shapeB.height / 2),
```

```
        colDir = null;
    if (Math.abs(vX) < hWidths && Math.abs(vY) < hHeights) {
        var oX = hWidths - Math.abs(vX),
            oY = hHeights - Math.abs(vY);
        if (oX >= oY) {
            if (vY > 0) {
                colDir = "t";
                shapeA.y += oY;
            } else {
                colDir = "b";
                shapeA.y -= oY;
            }
        } else {
            if (vX > 0) {
                colDir = "l";
                shapeA.x += oX;
            } else {
                colDir = "r";
                shapeA.x -= oX;
            }
        }
    }
    return colDir;
}
```

Okay, here we are passing 2 parameters into our function, ‘shapeA’ and ‘shapeB’. We are checking for collisions in all directions between the two shapes and returning one value called ‘colDir’. The value ‘colDir’ return is simply ‘l’, ‘r’, ‘b’, or ‘t’, which represents the direction of any detected collision, left, right, bottom and top respectively. Using these values we will determine what we want to do in that instance. First, though, let’s take a closer look at what’s going on in our colDir function:

```
function colCheck(shapeA, shapeB) {
//first we pass the player and obstacle to check into the function -
shapeA and shapeB
```

```
        var vX = (shapeA.x + (shapeA.width / 2)) - (shapeB.x +
(shapeB.width / 2)),
         vY = (shapeA.y + (shapeA.height / 2)) - (shapeB.y +
(shapeB.height / 2)),
// get the vectors (x and y coordinates) to check against


         hWidths = (shapeA.width / 2) + (shapeB.width / 2),
         hHeights = (shapeA.height / 2) + (shapeB.height / 2),
         colDir = null;
// add the half widths and half heights of the shapes

        if (Math.abs(vX) < hWidths && Math.abs(vY) < hHeights) {
// if the x and y vectors are less than the half width or half height,
they must be overlapping one another (a collision)


         var oX = hWidths - Math.abs(vX),
             oY = hHeights - Math.abs(vY);
         if (oX >= oY) {
             if (vY > 0) {
                 colDir = "t";
                 shapeA.y += oY;
             } else {
                 colDir = "b";
                 shapeA.y -= oY;
             }
         } else {
             if (vX > 0) {
                 colDir = "l";
                 shapeA.x += oX;
             } else {
                 colDir = "r";
                 shapeA.x -= oX;
             }
         }
    }
// figure out on which side we are colliding (top, bottom, left, or
right)
```

```
    return colDir;
//return the direction of the collision

}// Close Collision Check Function
```

We have a math function here, `Math.abs()`. The Math.abs() method returns the absolute value of a number.

It returns:

- A number representing the absolute value of the specified number
- NaN if the value is not a number (a text string for instance)
- 0 if the value is *null* (doesn't have a value)

What does the absolute value of a number mean by definition?

1. MATHEMATICS - the magnitude of a real number without regard to its sign.
2. TECHNICAL - the actual magnitude of a numerical value or measurement, irrespective of its relation to other values.

In our case, if we take `Math.abs(vX)`, it doesn't matter if vX is a negative or a positive number, the method will return only the number.

E.g.

```
Math.abs(-89.25);//returns 89.25
Math.abs(89.25);//returns 89.25
```

We do this so that the number is prepared for comparison in our conditionals using various operators.

WHAT IS AN OPERATOR?

There are many types. An operator performs an operation on a single or multiple operands (data value) and produces a result. See detailed tables over the next couple of pages.

Arithmetic Operators

OPERATOR	DESCRIPTION
+	addition
-	subtraction
*	multiplication
/	division
%	modulus(division remainder)
++	increment
--	decrement

Assignment Operators

OPERATOR	USAGE	LONGHAND
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x – y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Comparison Operators

OPERATOR	DESCRIPTION
==	equal to
===	equal to value and type
!=	not equal to
!==	not equal to value or type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Logical Operators

OPERATOR	DESCRIPTION
&&	and
||	or
!	not

Conditional Operator

variable = (condition) ? value1 : value2

Bitwise Operators

OPERATOR	DESCRIPTION
&	AND
|	OR
~	NOT
^	XOR
<<	Left Shift
>>	Right Shift

Okay, now we have to do something with the returned result. Open 'main.js' and add the following code:

```
if (player.screen === 1){ drawMyMap(); }//◄ add the following after this

    if (player.screen < 15 && player.start === 0) {player.start = 1;}
    if (player.screen === 1 && boxesDrawn1 === 0){ drawBoxes1(); }
    if (player.screen === 1 && boxesDrawn1 < 2 ) {
        boxes = boxes1;
        if (player.entered < 1){player.entered = 1;}
    }
    player.grounded = false;
    for (var i = 0; i < boxes.length; i++) {
        ctx.rect(boxes[i].x, boxes[i].y, boxes[i].width,
boxes[i].height);
        var dir = colCheck(player, boxes[i]);
```

```
        if (dir === "l" || dir === "r") {
            player.velX = 0;
            player.jumping = false;
        } else if (dir === "b") {
            player.grounded = true;
            player.jumping = false;
        }
    }
    if(player.grounded){player.velY = 0;}
    player.x += player.velX;
    player.y += player.velY;
if(player.y > 600){ player.y = 522;}
if (player.screen != 15){
    ctx.drawImage(wiz[player.f], player.x, player.y);
}
```

Let's take a closer look at what's going on.

```
    if (player.screen < 15 && player.start === 0) {player.start = 1;}
// if the player is no longer on the title screen and start = 0, set
start to 1 – the game has began

    if (player.screen === 1 && boxesDrawn1 === 0){ drawBoxes1(); }
//if the game has started and collision boxes haven't been drawn then
draw them now

    if (player.screen === 1 && boxesDrawn1 < 2 ) {
        boxes = boxes1;
        if (player.entered < 1){player.entered = 1;}
    }
//is the player's screen is 1 (the first game screen) and the collision
boxes have started to be drawn set the boxes variable to hold the
collision boxes for screen 1. Player entered means the player has been
drawn, so set to 1.

    player.grounded = false;
// set the player to not on the ground

    for (var i = 0; i < boxes.length; i++) {
```

```
        ctx.rect(boxes[i].x, boxes[i].y, boxes[i].width,
boxes[i].height);
//draw rectangles that the player will use for collisions

        var dir = colCheck(player, boxes[i]);
//call the colCheck function to determine if there are any collisions and
from which direction

        if (dir === "l" || dir === "r") {
            player.velX = 0;
            player.jumping = false;
        } else if (dir === "b") {
            player.grounded = true;
            player.jumping = false;
        }
    }
//depending on which direction the player has collided from set rules for
the collision

    if(player.grounded){player.velY = 0;}
//if player is on the ground set the velocity for the player in the Y
direction to 0

    player.x += player.velX;
//allow the player to move the player on the X axis

    player.y += player.velY;
//allow the player to move the player on the Y axis

if(player.y > 600){ player.y = 522;}
//set play start Y axis position

if (player.screen != 15){
    ctx.drawImage(wiz[player.f], player.x, player.y);
}
//if the player is not on the title screen draw the main character
```

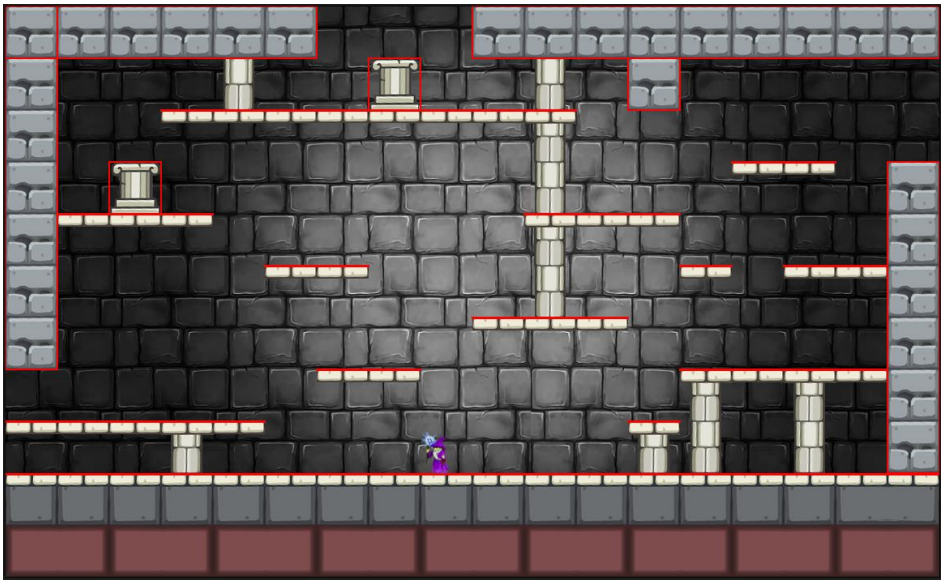
Lastly, open ‘map1.js’ and add the following code to the bottom of the file, then save it (CTRL + S):

```
function addBox(a,b,c,d){
    boxes1.push({ x: a, y: b, width: c, height: d });
    return
}
var boxes = [];
function drawBoxes1(){
    // BORDER WALLS
    addBox(0,577,1152,1);
    addBox(0,0,64,448);
    addBox(1088,192,64,384);
    addBox(0,0,384,64);
    addBox(576,0,576,64);
    addBox(768,64,64,64);
    // PLATFORMS
    addBox(0,512,320,1);
    addBox(768,512,64,20);
    addBox(832,448,256,1);
    addBox(384,448,128,1);
    addBox(128,450,57,62);
    addBox(576,384,192,1);
    addBox(320,320,128,1);
    addBox(832,320,64,1);
    addBox(960,320,128,1);
    addBox(64,256,192,1);
    addBox(640,256,192,1);
    addBox(128,192,64,64);
    addBox(896,192,128,1);
    addBox(192,128,512,1);
    addBox(448,64,64,1);
    boxesDrawn1+=1;
    return boxesDrawn1;
}
```

These functions draw the collision boxes at set coordinates. Let's take a closer look at how it works:

In our assets file, we have set an array called boxes1.

```
function addBox(a,b,c,d){
//open a function called addBox and allow paramters a,b,c and d

    boxes1.push({ x: a, y: b, width: c, height: d });
//each time the function is invoked, push a new object into the boxes1
array which sets the x and y axis to start drawing from and also the
length and height of the box to draw.

    return
//return the results
}

var boxes = [];
//make a new array called boxes
```



The drawBoxes1 function adds collision boxes as illustrated above.

Inside the drawBoxes1 function we have a bunch of function calls that add collision boxes, like so:

```
addBox(0,577,1152,1);
//call / invoke the addBox function and send the necessary parameters.
X coordinate to start from, Y coordinate to start from, Width in pixels,
Height in pixels
```

This, in conjunction with our colDir function, has the effect of making our platforms and walls appear solid. Note that the platforms only have a 1 pixel surface for collision detection. This is so that our main character can pass beneath the platforms. Our main character is 52 pixels tall, and each tile is 64 pixels tall. The actual platform is 16 pixels tall and if we take that away from 64 we are only left with 48 pixels of available space. Therefore, our player wouldn't be able to pass through. See the images below.

You can see in the first image that our player isn't able to pass beneath the platform, while in the second image our player can pass freely.

With collisions now set up, open 'main.js' and just inside the main update function opening add the following code:

```
if (keys[38] || keys[32]) {
    // up arrow or space
    if (!player.jumping && player.grounded) {
        player.velY = -player.speed * 2.5;
        player.jumping = true;
        player.grounded = false;
    }
}
```

Save the file. This enables the player to jump. Whenever you press the spacebar or the arrow up key. Combined with the arrow left and right, you can now navigate through the whole game screen. Try it out.

Awesome! Let's add some enemies!

STEP 11 - ADDING ENEMIES - BOTH STATIC AND MOVING

At the beginning of the exercise, I explained that the objective of the game was to collect all of the jewels while avoiding the obstacles. In this section, we are going to add ghosts, bats, and deadly spikes.
For the sake of giving them a label, we'll call all of them enemies.

The first thing we need to do is bring in the sprites we need. As before these are available for you to download via the link below:

https://wddtrw/resources/learntocode/platformgame_enemy_sprites.zip

Once downloaded, unzip it and add the contents to the 'sprites' folder. To unzip the file, double click it and then select all of the contents (CTRL + A) and copy it (CTRL + C), then open the 'sprites' folder and paste (CTRL + V).

If all went well you should have the following:

In the usual fashion, we're going to add them to the sprites file. Open 'sprites.js' and add the following code to the bottom of the file, then save it (CTRL + S):

```
/* ============================
ENEMIES KEY:
200 - Ghost Frame 1
201 - Ghost Frame 2
202 - Spikes
203 - Bat Frame 1
204 - Bat Frame 2
============================== */
for(i = 200; i < 205; i++){
    sprite[i] = new Image();
    sprite[i].src = `sprites/${i}.png`;
}
//this adds our 5 enemy sprites to the sprites array at stated indexes
```

The next thing we need to do is set them up in our assets file. Open the 'assets.js' file. Following the closing curly brace of the player object (},) add the following code, and then save the file:

```
enemy1 = { x: 64, y: 452, w: 59, h: 57, s: 2, f: 200, dest: 0 },
enemy2 = { x: 256, y: 68, w: 59, h: 57, s: 3, f: 200, dest: 0 },
enemy3 = { x: 1024, y: 64, w: 59, h: 57, s: 2, f: 200, dest: 0 },
bat1 = { x: 64, y: 482, w: 64, h: 17, s: 2, f:203, dest: 0 },
spikes1 = { x: 512, y: 556, w: 61, h: 20 },
spikes2 = { x: 684, y: 556, w: 61, h: 20 },
//Here we have set up with initial values three ghosts (enemy 1 – 3), one
bat, and two spike objects. Each has some or all of the following
variables:
```

- x - X-axis starting coordinate
- y - Y-axis starting coordinate
- w - Width in pixels
- h - Height in pixels
- s - Speed
- f – Sprite Frame
- dest – Destination

Importantly, we don't have to use all of these in every game screen, but they are now declared and set up in readiness. Also note, the spike objects don't need speed, frame, or destination because they're static.

Next, make a new file called 'enemy.js' and save it in the 'js' folder. Then add the following code:

```
function enemy(sn, e, ax, bx, f1, f2, p, xy, xos, yos){
    ctx.drawImage(sprite[sn], e.x+xos, e.y+yos, e.w, e.h);
    if (xy === 1){
        if (e.x < ax && e.dest === 0){ e.x+=e.s;
            if(e.x > ax-2){ e.dest = 1;}
        }
        if (e.dest === 1){ e.x-=e.s;
            if (e.x <= bx){ e.dest = 0; }
        }
    }
    if (xy === 2) {
        if (e.y < ax && e.dest === 0){ e.y+=e.s;
            if(e.y > ax-2){ e.dest = 1;}
        }
        if (e.dest === 1){ e.y-=e.s;
            if (e.y <= bx){ e.dest = 0; }
        }
    }
    if (e.f === f1){
        setTimeout(function(){ e.f = f2; },200);
    } else {
        setTimeout(function(){ e.f = f1; },200);
    }
    if (p.x < e.x+xos + e.w && p.x + p.width > e.x+xos && p.y < e.y+yos +
e.h && p.y + p.height > e.y+yos){ lifeLost(e, p, xos); }
}

function enemyStatic(sn, e, xos, yos, p){
    ctx.drawImage(sprite[sn], e.x+xos, e.y+yos, e.w, e.h);
    if (p.x < e.x+xos + e.width && p.x + p.width > e.x+xos && p.y <
e.y+yos + e.h && p.y + p.height > e.y+yos){ lifeLost(e, p, xos); }
```

```
}

function lifeLost(e, p, xos){
    player.lives-=1;
    player.x = 630;
    player.y = 522;
    if ((p.x + p.width) > e.x+xos && (p.x + p.width) < e.x+xos + e.w){
player.f = 1; }
    if (p.x < (e.x+xos + e.w) && p.x > e.x+xos){ player.f = 3; }
}
```

Okay, as you can see, we have three functions here. The first function draws enemies, sets their movement either on the X or Y axis as required, animates the enemies every 200ms, and handles collisions with those (animated and moving) enemies, the second function draws static enemies and handles collisions with those static enemies (the spikes) and the third function (lifeLost) gets called by the afore mentioned functions if a collision occurs between the player and an enemy.

Let's take a closer look:

```
function enemy(sn, e, ax, bx, f1, f2, p, xy, xos, yos){
//open function and pull in parameters

    ctx.drawImage(sprite[sn], e.x+xos, e.y+yos, e.w, e.h);
//draw the enemy on the canvas

    if (xy === 1){
// if horizontal movement is chosen move on X axis

        if (e.x < ax && e.dest === 0){
//if enemy x position < destination and destination not reached

            e.x+=e.s;
// move enemy at chosen speed from left to right

            if(e.x > ax-2){ e.dest = 1;}
```

```
//if destination is reached set destination to 1
        }

        if (e.dest === 1){
//if destination is reached…

            e.x-=e.s;
// move enemy at chosen speed from right to left

            if (e.x <= bx){ e.dest = 0; }
//if destination is reached set destination to 0


        }
    }

    if (xy === 2) {
// if vertical movement is chosen move on Y axis

        if (e.y < ax && e.dest === 0){
//if enemy y position < destination and destination not reached

            e.y+=e.s;
// move enemy at chosen speed from top to bottom

            if(e.y > ax-2){ e.dest = 1;}
//if destination is reached set destination to 1

        }

        if (e.dest === 1){
//if destination is reached…

            e.y-=e.s;
// move enemy at chosen speed from bottom to top

            if (e.y <= bx){ e.dest = 0; }
//if destination is reached set destination to 0
```

```
        }
    }

    if (e.f === f1){
        setTimeout(function(){ e.f = f2; },200);
    } else {
        setTimeout(function(){ e.f = f1; },200);
    }
//if the frame is the first frame wait 200 milli-seconds and change to
the second frame. Otherwise, if the frame is the second frame wait 200
milliseconds and change to the first frame

    if (p.x < e.x+xos + e.w && p.x + p.width > e.x+xos && p.y < e.y+yos +
e.h && p.y + p.height > e.y+yos){ lifeLost(e, p, xos); }
//if a collision is detected call lifeLost function
}
```

Okay, then we have the second function, which is much simpler because it is for static enemies.

```
function enemyStatic(sn, e, xos, yos, p){
    ctx.drawImage(sprite[sn], e.x+xos, e.y+yos, e.w, e.h);
//draw the enemy to the screen at the given parameters

    if (p.x < e.x+xos + e.width && p.x + p.width > e.x+xos && p.y <
e.y+yos + e.h && p.y + p.height > e.y+yos){ lifeLost(e, p, xos); }
//if a collision is detected call lifeLost function
}
```

Much simpler, but also easier to navigate for the player. Lastly, we have the lifeLost function.

```
function lifeLost(e, p, xos){
//open the function and pass parameter

    player.lives-=1;
//decrease player lives by 1
```

```
    player.x = 630;
//Set player x position to 630 pixels

    player.y = 522;
//set player y position to 522 pixels

    if ((p.x + p.width) > e.x+xos && (p.x + p.width) < e.x+xos + e.w){
player.f = 1; }
//if the player has collided with an enemy to the left set player sprite
frame to 1
    }
    if (p.x < (e.x+xos + e.w) && p.x > e.x+xos){ player.f = 3; ?}
//if the player has collided with an enemy to the right set player sprite
frame to 3
    }
}
```

Next, we need to call the enemy and enemyStatic functions just after we've drawn our map. Open 'map1.js' and add the following function calls just after the closing curly brace for the outer for loop (}), then save (CTRL + S).

```
enemy(enemy1.f, enemy1, 335, 0, 200, 201, player, 1, 0, 0);
enemy(bat1.f, bat1, 710, 64, 203, 204, player, 1, 0, 0);
enemy(enemy3.f, enemy3, 265, 64, 200, 201, player, 2, 0, 0);
enemyStatic(202, spikes1, 0, 0, player);
```

Okay, finally we need to set start positions and give the player some feedback. Open main.js'.

Add the following conditional to set enemy start positions:

```
    if (player.screen === 1 && boxesDrawn1 < 2 ) {
        boxes = boxes1;
        if (player.entered < 1){
            enemy1.x = 62; enemy1.y = 454; enemy1.dest = 0;
            bat1.x = 62; bat1.y = 70; enemy2.dest = 0;
```

```
            enemy3.y = 62; enemy3.dest = 0;
            player.entered = 1;
        }
    }
```

Insert it following this line of code:

```
    if (player.screen === 1 && boxesDrawn1 === 0){ drawBoxes1(); }
```

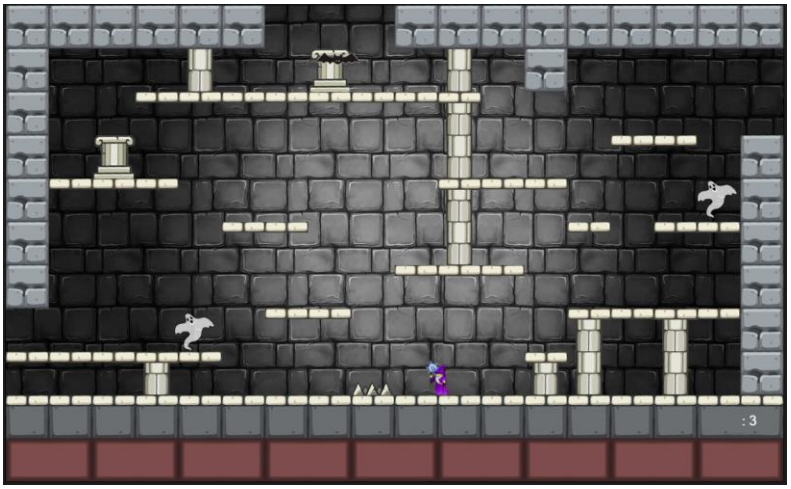
Add the following code just before the `requestAnimationFrame(update);` instruction before the update function closing curly brace (}) and save the file (CTRL + S):

```
ctx.font="20px Arial";
ctx.fillStyle = "white";
ctx.fillText(": "+player.lives, 1090, 621);
//displays current player lives at given coordinates
```

Now make sure to add a reference in the 'sorcerer.html' file for the 'enemy.js' file, in the usual manner. Add the following just inside the </body> closing body tag and save (CTRL + S):

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

If you've followed this correctly, you should now have the following:



At this point, we have set some basic rules in place, but as we move on they will need to be updated. For instance, on different game screens you will want your player to respawn in different positions, depending on your design, we'll need to do something when all lives are lost and your game is over and we'll need to add some animation for our main character for walking, respawning, etc.

I'm pretty sure now is a good time for a well-deserved break. Grab a drink and relax a while, before we move on.

GAME OVER

Open 'enemy.js' and add the following conditional to the end of the lifeLost function:

```
    if (player.lives === 0){
        gameOver();
    }
```

Then add the following gameOver function to the end of the file, and save it (CTRL + S):

```
function gameOver(){
    player.screen = 20;
    ctx.drawImage(tile[16], 0, 0, 1152, 704);
    ctx.drawImage(sprite[109], 250, 200, 688, 151);
}
```

Finally, open 'main.js' and add the following conditional inside the top of the update function:

```
    if (player.screen === 15){ drawMyMap1000();}◄ following this line
    if (player.screen === 20){ gameOver();}◄ add this line
```

Save the file (CTRL + S). Now when player.lives === 0 you get:

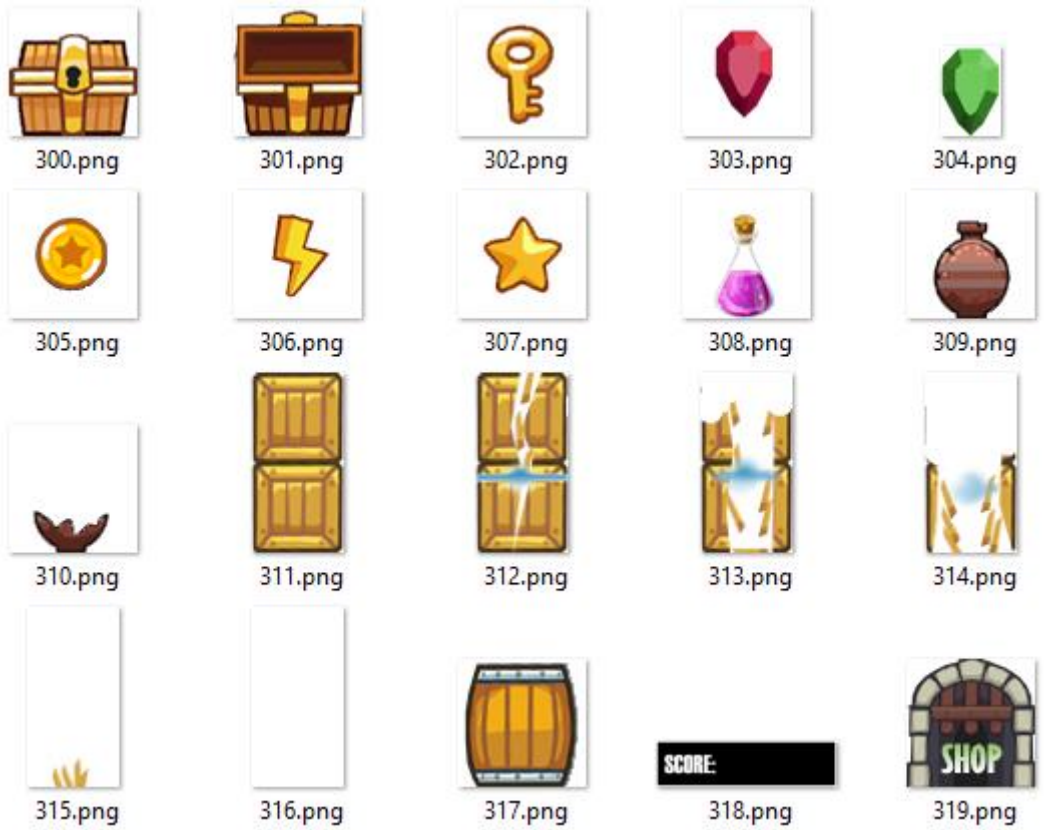
STEP 12 - ADDING COLLECTABLES

The first thing we need to do is bring in the sprites we need. As in all cases before, these are available for you to download via the link below:

https://wddtrw/resources/learntocode/platformgame_collectables.zip

Once downloaded, unzip it and add the contents to the 'sprites' folder. To unzip the file, double click it and then select all of the contents (CTRL + A) and copy it (CTRL + C), then open the 'sprites' folder and paste (CTRL + V).

If all went well you should have the following:



Some of these sprites this time will also be used for the game monitor in the next section.

In the now very familiar fashion, we're going to add them to the sprites file. Open 'sprites.js' and add the following code to the bottom of the file, then save it (CTRL + S):

```
/* ============================
COLLECTABLES KEY:
300 - Chest
301 - Open Chest
302 - Key
303 - Ruby
304 - Emerald
305 - Gold Coin
306 - Lightning Bolt
307 - Star
308 - Potion
309 - Flask
310 - Broken Flask
311 - Crates Frame 1 - Whole
312 - Crates Frame 2
313 - Crates Frame 3
314 - Crates Frame 4
315 - Crates Frame 5
316 - Crates Frame 6 - Destroyed
317 - Barrel
318 - Score Card
319 - Shop
============================ */
for(i = 300; i < 320; i++){
    sprite[i] = new Image();
    sprite[i].src = `sprites/${i}.png`;
}
```

SPRITE SHEETS

It is worth noting at this point that we could and should (for the finished game) load all of the sprites in as a single file called a sprite sheet. Sprite

sheets increase the performance of your game and reduce loading and start-up time. The game would be using a few larger images, instead of possibly hundreds of smaller ones.

We would load in and draw from a sprite sheet like this:

```
let img = new Image();
img.src = '/sprites/sorcerersmountain.png';
img.onload = function() {
  init();
};

let canvas = document.querySelector('canvas');
let ctx = canvas.getContext('2d');

function init() {
  drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight);
}
```

I have kept them separate in this book to keep each section more easily understood, as we built each part of the game. However, in the final game, a sprite sheet will be used and we'll update our code accordingly.

Moving on…

Make a new file called 'collectables.js' and save it in the 'js' folder and make sure to add a reference to the file in the 'sorcerer.html' file, as below and save it (CTRL + S):

```
<script type="text/javascript" src="js/collectables.js"></script>
//place this below the enemy.js reference
```

Next open 'assets.js' and add the following two lines of code, to initialise our coins settings. The first line declares a new coin object, sets the width and height of our sprite, and sets the initial coin count to zero (0). Secondly, we have a coinsCollected array with ten delimited values all set

to zero. Each one of these values represents a coin. Zero (0) means not collected and one (1) will mean collected. You'll see shortly how we use this to determine whether or not the coin in question should be drawn.

```
    coin = { width: 49, height: 50, count: 0 },
    coinsCollected = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

Okay, back to our 'collectables.js' file, let's tackle drawing and collecting coins.

Add the following code and then save the file (CTRL + S).

```
var coins = [];
function addCoins(mn, w, h, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6,
y6, x7, y7, x8, y8, x9, y9, x10, y10){
let xs = [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10];
let ys = [y1, y2, y3, y4, y5, y6, y7, y8, y9, y10];
for(i =0; i < 10; i++){
   coins[i] = { x: xs[i], y: ys[i], width: w, height: h, c: 0 };
   if(player.x < coins[i].x + coins[i].width
    && player.x + player.width > coins[i].x
    && player.y < coins[i].y + coins[i].height
    && player.y + player.height > coins[i].y && coinsCollected[i] == 0){
     coins[i].c +=1;
     coinsCollected[i] = 1;
   }
   if (coins[i].c == 0 && coinsCollected[i] == 0){
     ctx.drawImage(sprite[305], coins[i].x, coins[i].y, coins[i].width,
coins[i].height);
   } else if (coins[i].c > 0 && coins[i].c < 2){
     coin.count+=1;
     player.score+=10;
  }
}
```

The addCoins function does three jobs. It passes in required parameters, checks if the player had collided with any of the coins, and then draws the

coin if it is determined that the player hasn't collided (collected) the coin(s).

Let's take a closer look:

```
var coins = [];
//make a new empty array and store in it in the variable coins
function addCoins(item, w, h, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6,
y6, x7, y7, x8, y8, x9, y9, x10, y10){
//open the addCoins function and pass necessary parameters for 10 coins

let xs = [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10];
let ys = [y1, y2, y3, y4, y5, y6, y7, y8, y9, y10];
//separate x and y coordinate parameters into indexed arrays

for(i=0; i < 10; i++){
//iterate through 10 indexes

   coins[i] = { x: xs[i], y: ys[i], width: w, height: h, c: 0 };
//add a coin object for the coins array for each index with position and
size parameters

   if(player.x < coins[i].x + coins[i].width
    && player.x + player.width > coins[i].x
    && player.y < coins[i].y + coins[i].height
    && player.y + player.height > coins[i].y && coinsCollected[i] == 0)
   {
//if a player has a collision with a coin not yet collected…

     coins[i].c +=1;
//add 1 to the c parameter of coins for the given index

coinsCollected[i] = 1;
//update the coinsCollected array to 1 for the given index – indicating
the coin has been collected
   }
   if (coins[i].c == 0 && coinsCollected[i] == 0){
```

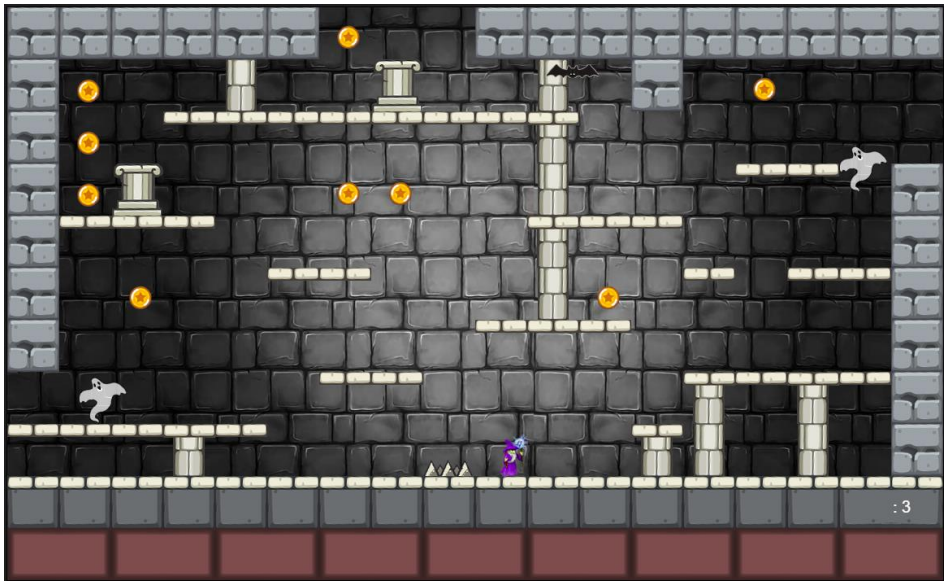
```
//if a coinis not collected…

     ctx.drawImage(sprite[305], coins[i].x, coins[i].y, coins[i].width,
coins[i].height);
//draw the coin at the given coordinates in the given size

   } else if (coins[i].c > 0 && coins[i].c < 2){
//otherwise, if the coin has been collected

     coin.count+=1;
//add 1 to coin count – for use with the game monitor

     player.score+=10;
//add 10 to the player score – for use with the game monitor
  }
}
```

Now, let's make it all work. Open 'map1.js' and add the following function call, immediately following the nested for loops that draw the tile map and save the file (CTRL + S):

```
addCoins(coin, 49, 50, 76, 78, 76, 142, 76, 206, 140, 332, 396, 204, 460,
204, 396, 12, 716, 332, 908, 76, 600, 78)

// call or invoke addCoins function with the given parameters
addCoins(mn, w, h, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6, y6, x7,
y7, x8, y8, x9, y9, x10, y10)

mn: map number
w: width of the sprite
h: height of the sprite
x and y: positions (x10)
```

We have a few more collectables to add to our game screen, but unlike coins having 10, we'll use less of the others.

If you have followed all of the instructions correctly so far you should have a game screen that looks like the example below. Experiment with the function call x and y parameters. You can place your coins anywhere and they are all collectable, and on the collection, they add to your coin count and score.



We could continue in the same vein and build almost identical functions for our other collectables. As shown for the gems below:

```
var gems = [];
function addGems(item, w, h, x1, y1, x2, y2){
let xs = [x1, x2];
let ys = [y1, y2];
for(i =0; i < 10; i++){
   gems[i] = { x: xs[i], y: ys[i], width: w, height: h, c: 0 };
   if(player.x < gems[i].x + gems[i].width
    && player.x + player.width > gems[i].x
    && player.y < gems[i].y + gems[i].height
    && player.y + player.height > gems[i].y && gemsCollected[i] == 0){
    gems[i].c +=1;
    gemsCollected[i] = 1;
   }
```

```
    if (gems[i].c == 0 && gemsCollected[i] == 0){
      ctx.drawImage(sprite[304], gems[i].x, gems[i].y, gems[i].width,
gems[i].height);
    } else if (gems[i].c > 0 && gems[i].c < 2){
     gems.count+=1;
      player.score+=50;
  }
}
```

Notice how the code is almost identical, aside from the variables being called gems instead of coins. That means we can optimise it. As much as possible we should remove repetitive code. For instance, here we could build one large array to hold all collectable objects. With that in mind, let's do a quick revisit.

Firstly, open the 'assets.js' file and add the following code instead of the coin assets, then save (CTRL + S):

```
coin = {width:49, height:50, count:0, mn:1, sn:305, fct:0, t:'coins'},
gem = {width:40, height:45, count:0, mn:1, sn:303, fct:12, t:'gems'},
key = {width:34, height:49, count:0, mn:1, sn:302, fct:14, t:'keys'},
potion = {width:30, height:52, count:0, mn:1, sn:308, fct:16, t:'potions' },
flask = {width:38, height:49, count:0, mn:1, sn:309, fct:18, t:'flasks' },
star = {width:39, height:39, count:0, mn:1, sn:307, fct:23, t:'stars' },
bolt = {width:39, height:39, count:0, mn:1, sn:306, fct:28, t:'bolts' },
itemsCollected = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
```

These are the initial settings for all collectables.

- width
- height
- count – how many collected
- mn – map number
- sn – sprite number

- fct – increase index factor
- t – the type of collectable, as a string

At the bottom of the same file, add the following for the screen 2 platform collision detection:

```
var boxes2 = [];
var boxesDrawn2 = 0;
```

Then save the file (CTRL + S).

Next, let's add our function calls. Open 'map1.js' add the following code, in place of the addCoins function call, then save it (CTRL + S):

```
//addCxs(item, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6, y6, x7, y7,
x8, y8, x9, y9, x10, y10)
addCxs(coin, 76, 78, 76, 142, 76, 206, 140, 332, 396, 204, 460, 204, 396,
12, 716, 332, 908, 76, 600, 78);
addCxs(gem, 720, 74, 1040, 532);
addCxs(key, 460, 10, 1024, 345);
addCxs(star, 525, 12, 973, 76, 1037, 268);
addCxs(potion, 1041, 398, 593, 332);
addCxs(flask, 13, 463);
addCxs(bolt, 425, 384);
//Now we can add up to 10 of each item, simply by stipulating x and y
coordinates and bringing in the item object from assets
```

Finally, open the 'collectables.js' file and replace all code with the following function, to handle all collectables, then save the file (CTRL + S):

```
var items = [];

function addCxs(item, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6, y6, x7,
y7, x8, y8, x9, y9, x10, y10){

    let iw = item.width;
    let ih = item.height;
    let plw = player.width;
    let plh = player.height;
```

```
    let plx = player.x;
    let ply = player.y;
    item.mn = player.screen-1;
    let mn = item.mn*28+item.fct;

    let xs = [];
    xs[1+mn] = x1;
    xs[2+mn] = x2;
    xs[3+mn] = x3;
    xs[4+mn] = x4;
    xs[5+mn] = x5;
    xs[6+mn] = x6;
    xs[7+mn] = x7;
    xs[8+mn] = x8;
    xs[9+mn] = x9;
    xs[10+mn] = x10;

    let ys = [];
    ys[1+mn] = y1;
    ys[2+mn] = y2;
    ys[3+mn] = y3;
    ys[4+mn] = y4;
    ys[5+mn] = y5;
    ys[6+mn] = y6;
    ys[7+mn] = y7;
    ys[8+mn] = y8;
    ys[9+mn] = y9;
    ys[10+mn] = y10;

  for(i=0; i < 10; i++){
    items[i+mn] = { x: xs[i+mn], y: ys[i+mn], width: iw, height: ih, c: 0
};
    if(plx < items[i+mn].x + iw && plx + plw > items[i+mn].x && ply <
items[i+mn].y + ih && ply + plh > items[i+mn].y && itemsCollected[i+mn]
== 0){
      items[i+mn].c +=1;
      itemsCollected[i+mn] = 1;
    }
    if(items[i+mn].c == 0 && itemsCollected[i+mn] == 0){
```

```
      ctx.drawImage(sprite[item.sn], items[i+mn].x, items[i+mn].y, iw,
ih);
    } else if (items[i+mn].c > 0 && items[i+mn].c < 2){
      item.count+=1;
      if(item.t == 'coins'){ player.score+=10; sfx[0].play(); }
      if(item.t == 'gems'){ player.score+=50; sfx[0].play(); }
      if(item.t == 'keys'){ player.score+=20; sfx[0].play(); }
      if(item.t == 'potions'){ player.score+=30; sfx[0].play(); }
      if(item.t == 'flasks'){ player.score+=100; player.lives+=1;
sfx[0].play(); }
      if(item.t == 'stars'){ player.score+=25; sfx[0].play(); }
      if(item.t == 'bolts'){ player.score+=15; sfx[0].play(); }
    }
  }
}
```

In the usual fashion, let's break it down. The logic here is the same as the addCoins function from before, but taking into consideration which collectable we are referencing.

```
var items = [];
//create an empty array and store it in a variable called items

function addCxs(item, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6, y6, x7,
y7, x8, y8, x9, y9, x10, y10){
//open the addCxs function and pass in the collectable item assets, plus
ten sets of x and y coordinates

    let iw = item.width;
    let ih = item.height;
    let plw = player.width;
    let plh = player.height;
    let plx = player.x;
    let ply = player.y;
// set short variable names to hold player and collectable attributes.
This will allow us to make collision calculations shorter and more easily
readable
```

```
    item.mn = player.screen-1;
// set the item map number to match the current player screen - 1

    let mn = item.mn*30+item.fct;
//calculate mn to ensure data is held at unique array indexes. For
instance, if the player is on screen number 1 the equation would be
0 * 30 + the item factor given in assets, screen 2 would be 1 * 30 + item
factor and so on. 30 is the chosen maximum number of collectables per
game screen, based on the following:
```

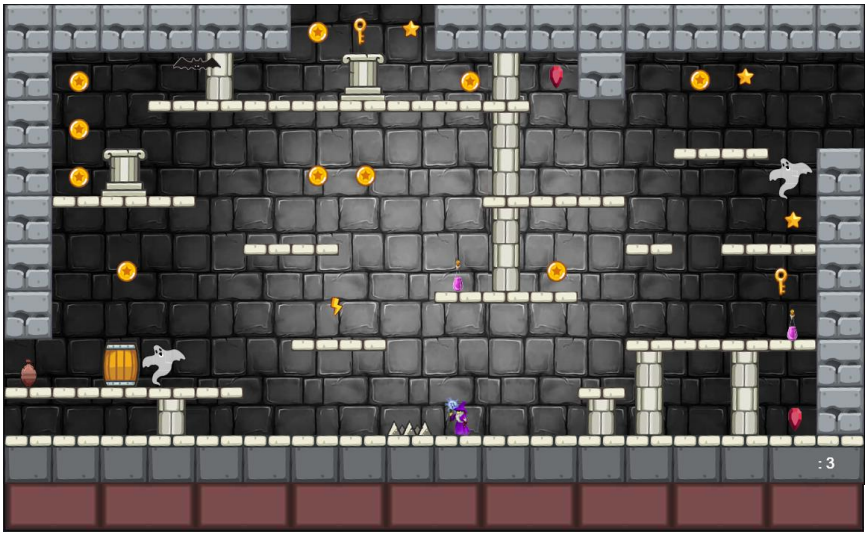
- coins x 10
- gems x 2
- keys x 2
- potions x 2
- flasks x 2
- stars x 5
- lightning bolts x 2
- spare indexes x 5 – if we want to add more collectables

```
    let xs = [];
    xs[1+mn] = x1;
    xs[2+mn] = x2;
    xs[3+mn] = x3;
    xs[4+mn] = x4;
    xs[5+mn] = x5;
    xs[6+mn] = x6;
    xs[7+mn] = x7;
    xs[8+mn] = x8;
    xs[9+mn] = x9;
    xs[mn] = x10;
//grab all X coordinates from the function parameters and add them to the
xs array at calculated indexes

    let ys = [];
    ys[1+mn] = y1;
    ys[2+mn] = y2;
    ys[3+mn] = y3;
    ys[4+mn] = y4;
    ys[5+mn] = y5;
    ys[6+mn] = y6;
```

```
    ys[7+mn] = y7;
    ys[8+mn] = y8;
    ys[9+mn] = y9;
    ys[mn] = y10;
//grab all Y coordinates from the function parameters and add them to the
ys array at calculated indexes.

  for(i=0; i < 10; i++){
//iterate through loop 10 times – 1 time per possible collectable item

 items[i+mn] = { x: xs[i+mn], y: ys[i+mn], width: iw, height: ih, c: 0 };
//add item object to the items array at given coordinates and calculated
indexes

    if(plx < items[i+mn].x + iw && plx + plw > items[i+mn].x && ply <
items[i+mn].y + ih && ply + plh > items[i+mn].y && itemsCollected[i+mn]
== 0){
//if player has had a collision with an item that has not yet been
collected

      items[i+mn].c +=1;
//set the item collected value to 1

      itemsCollected[i+mn] = 1;
//update the items collected array to show the item as collected

    }
    if(items[i+mn].c == 0 && itemsCollected[i+mn] == 0){
//if the item has not been collected…

      ctx.drawImage(sprite[item.sn], items[i+mn].x, items[i+mn].y, iw,
ih);
//draw the given sprite, at the given coordinates, width and height

    } else if (items[i+mn].c > 0 && items[i+mn].c < 2){
//otherwise, if the item has been collected…

      item.count+=1;
//add 1 to the item count – for the game monitor in the next section
```

```
      if(item.t == 'coins'){ player.score+=10; }
//if a coin, add 10 to the player's score

      if(item.t == 'gems'){ player.score+=50; }
//if a gem add 50 to the player's score
      if(item.t == 'keys'){ player.score+=20; }
//if a key, add 20 to the player's score

      if(item.t == 'potions'){ player.score+=30; }
/if a potion, add 30 to the player's score

      if(item.t == 'flasks'){ player.score+=100; player.lives+=1; }
//if a flask, add 100 to the player's score and add 1 life

      if(item.t == 'stars'){ player.score+=25; }
//if a star, add 25 to the player's score

      if(item.t == 'bolts'){ player.score+=15; }
//if a lightning bolt, add 15 to the player's score
    }
  }
}
```

Great job! If you've followed everything correctly, refresh your browser and press 's' to start and you should have the following:



Great! In the next section, we'll look at generating some user feedback. I think it's that time again. Take a break. Have a brew and refresh your batteries.

STEP ⓭ - THE GAME MONITOR

Okay, first of all open 'sorcerer.html' and add the following JavaScript file reference, just after the 'enemy.js' reference and save.

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

Next, create a new file called 'gameMonitor.js' and then add the code below and save the file (CTRL + S):

```
function gameMonitor(){
    ctx.drawImage(sprite[305], 19, 655, coin.width/1.5, coin.height/1.5);
    ctx.drawImage(sprite[307], 148, 658, star.width/1.5,
star.height/1.5);
    ctx.drawImage(sprite[302], 277, 656, key.width/1.5, key.height/1.5);
    ctx.drawImage(sprite[304], 400, 655, gem.width/1.5, gem.height/1.5);
    ctx.drawImage(sprite[303], 405, 670, gem.width, gem.height/1.5);
    ctx.drawImage(sprite[308], 533, 654, potion.width/1.5,
potion.height/1.5);
    ctx.drawImage(sprite[309], 660, 656, flask.width/1.5,
flask.height/1.5);
    ctx.drawImage(sprite[306], 788, 660, bolt.width/1.5,
bolt.height/1.5);
    ctx.drawImage(sprite[318], 896, 640, 256, 64);
    ctx.drawImage(wiz[1], 1042, 594, 30, 42);
}
```

All we are doing here is drawing sprites to the screen in specific locations and slightly resizing them. In order for them to get drawn, we have to call the function. Open 'map1.js' and add the following function call, just below the function calls for the collectables:

```
gameMonitor();
```

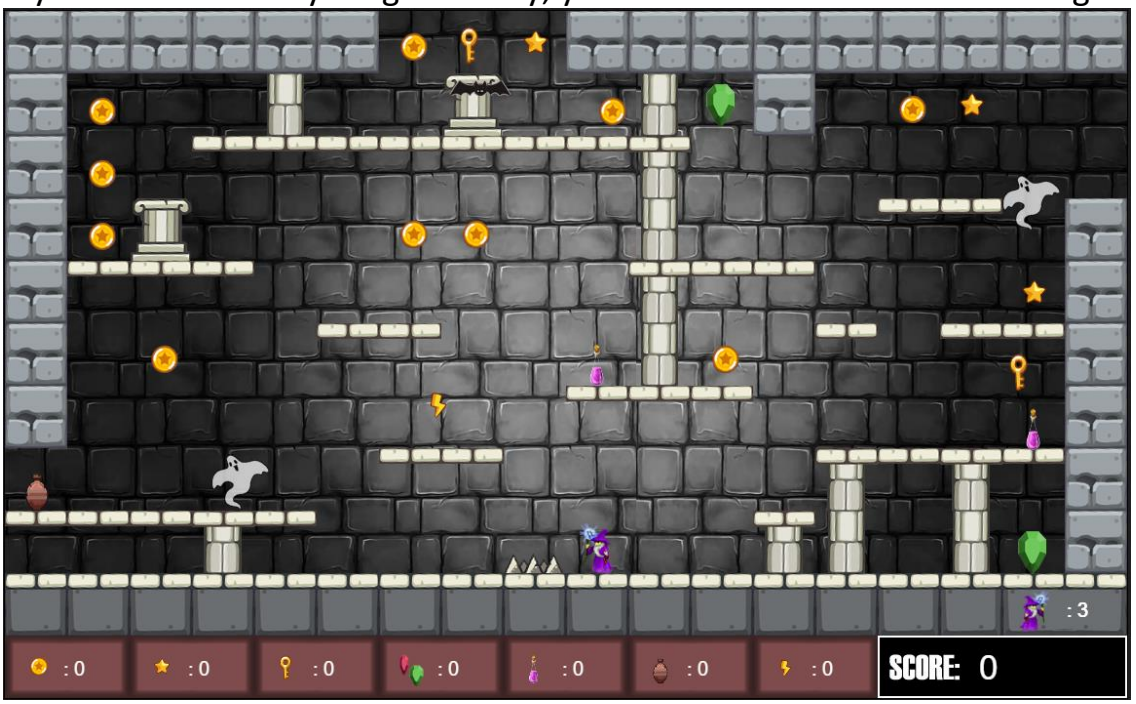
If you refresh your browser and press 's' to start the game you will see that we have placed sprites along the bottom of the screen to represent the game collections, score, and lives variables, as shown below:

Now, we need to add numbers to give feedback to the player. Open 'main.js' and add the following code inside the player.screen conditional, then, as always, save the file:

```
    if (player.screen != 15 && player.screen != 20){
        ctx.drawImage(wiz[player.f], player.x, player.y);
        ctx.font="20px Arial";
        ctx.fillStyle = "white";
        ctx.fillText(": "+player.lives, 1090, 621);
// after this add the following code

        ctx.font="35px Arial";
        ctx.fillStyle = "white";
        ctx.fillText(player.score, 1000, 685); // Update Score
```

```
        ctx.font="20px Arial";
        ctx.fillStyle = "white";
        ctx.fillText(": "+coin.count, 60, 681);// Update Coin Count
        ctx.fillText(": "+gem.count, 444, 681);// Update Gem Count
        ctx.fillText(": "+star.count, 188, 681);// Update Star Count
        ctx.fillText(": "+potion.count, 572, 681);// Update Potion Count
        ctx.fillText(": "+flask.count, 700, 681);// Update Flask Count
        ctx.fillText(": "+bolt.count, 828, 681);// Update Bolt Count
        ctx.fillText(": "+key.count, 316, 681);// Update Key Count
        ctx.fillText(": "+player.lives, 1090, 621);// Player Lives
    }
//add counts to the game monitor – updated with
requestAnimationFrame(update); 60 frames per second (every 16.7 ms)
```

If you followed everything correctly, you should now have the following:



Top class! So we've built one game screen. It looks pretty cool, right? In the next section, we'll explore how to add lots more screens and how to move the player from screen to screen.

STEP 14 - ADDING MORE GAME SCREENS

The great news here is that we've done a lot of the hard work. Our game screens will utilise much of the code we have already built.

Let's get cracking with game screen number 2. Firstly, we need to decide where each screen will sit in relation to each other. As we've already created screen 1, let's say screen 2 is to the left and screen 3 is to the right, like so:

Next, open 'sorcerer.html' and add a reference for our second map below the reference for map1, like so:

```
<script type="text/javascript" src="js/map2.js"></script>
```

Save the file (CTRL + S).

Now, create a new file called 'map2.js' and save it in the 'js' folder. Then add the following code:

```
function drawMyMap2(){
ctx.drawImage(tile[16], 0, 0);
var xt = 0; // Tile Map X Index
var yt = -64; // Tile Map Y Index
var tileMap = [];
var mapNo = 0; // Map Index
tileMap[0] =  [6, 0, 0, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6];
tileMap[1] =  [6, 9, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6];
```

```
tileMap[2] =  [6, 9, 9, 9, 9, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6];
tileMap[3] =  [6, 0, 0, 0, 0, 0, 0, 9, 0, 0, 4, 4, 4, 4, 0, 0, 4, 6];
tileMap[4] =  [6, 4, 4, 4, 4, 0, 0, 0, 0, 9, 5, 5, 5, 5, 0, 0, 5, 6];
tileMap[5] =  [6, 6, 6, 6, 6, 0, 9, 0, 0, 0, 5, 5, 5, 5, 0, 0, 5, 6];
tileMap[6] =  [6, 0, 0, 0, 0, 0, 0, 0, 0, 9, 6, 6, 6, 6, 6, 6, 6, 6];
tileMap[7] =  [6, 0, 0, 0, 0, 0, 0, 9, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0];
tileMap[8] =  [6, 9, 9, 9, 0, 0, 0, 0, 0, 9, 6, 0, 0, 0, 0, 0, 9, 8];
tileMap[9] =  [6, 1, 1, 1, 1, 1, 1, 1, 1, 1, 6, 0, 0, 1, 1, 1, 2, 3];
tileMap[10] = [13, 14, 13, 14, 13, 14, 13, 14, 13, 14, 13, 14, 13, 14,
13, 14, 13, 14];
for (mapNo=0; mapNo < 11; mapNo++){
    yt+=64;
    for (xt=0; xt < tileMap[mapNo].length*64; xt+=64){
    if (xt > 1152){ xt = 0; }
        var i = tileMap[mapNo][xt/64]; // Array Variable at current
iteration position
        ctx.drawImage(tile[i], xt, yt); // Draw row - Y position px from
top of canvas
    }
}

addCxs(coin, 76, 399, 140, 399, 206, 399, 652, 79, 716, 79, 780, 79, 846,
79)
addCxs(gem, 70, 200, 76, 532);
addCxs(key, 1040, 76, 585, 525);
addCxs(star, 1037, 140, 717, 460);
addCxs(potion, 590, 461, 0, -100);
addCxs(flask, 580, 207, 0, -100);
addCxs(bolt, 457, 386, 0, -100);

gameMonitor();

enemy(enemy1.f, enemy1, 390, 66, 200, 201, player, 1, 0, 0);
enemy(enemy2.f, enemy2, 576, 66, 200, 201, player, 1, 0, 0);
enemy(enemy3.f, enemy3, 320, 66, 200, 201, player, 2, -66, 0);
enemyStatic(202, spikes1, -129, 0, player);
enemyStatic(202, spikes2, 212, -192, player);
}
```

```
function addBox2(a,b,c,d){
    boxes2.push({ x: a, y: b, width: c, height: d });
    return
}

function drawBoxes2(){
addBox2(0,57,706,1);
addBox2(832,577,320,1);
addBox2(0,0,66,640);
addBox2(1089,0,66,384);
addBox2(192,0,960,64);
addBox2(640,386,66,256);
addBox2(640,192,256,192);
addBox2(706,386,386,64);
addBox2(1026,192,66,192);
addBox2(66,256,256,128);
addBox2(576,512,66,1);
addBox2(1024,512,128,1);
addBox2(66,512,192,1);
addBox2(449,449,66,1);
addBox2(576,386,66,1);
addBox2(386,320,66,1);
addBox2(576,256,66,1);
addBox2(66,129,320,1);
addBox2(449,192,66,1);
addBox2(66,66,66,1);
boxesDrawn2+=1;
return boxesDrawn2;
}
```

You should recognise all of this code. Apart from the changed parameters, this program code is the same as for screen 1. Just as a refresher, we have drawn the background image, then the tile map, added the collectables, then the game monitor, the enemies, and finally, the collision boxes.

If you need further clarification please turn back to page 162 and onwards, where we build ‘map1.js’. As the explanation would be virtually identical.

The last step we need to take is a small update to ‘main.js’ to allow the main character access to the other tile maps. For example, when the player moves out of the left-hand side of screen one, they need to arrive at the right-hand side of screen 2. Likewise, if the player moves out of the right-hand side of screen 2, they need to arrive at the left-hand side of screen 1. Open ‘main.js’ and add the following code, then save the file:

```
if (player.screen === 1){ drawMyMap(); }◄ following this line add the
code below

//==== SCREEN 1 ===========
if (player.x <5 && player.screen === 1){
    player.screen = 2;
    player.x = 1103;
    player.entered = 0;
}
if (player.x > 1103 && player.screen === 1){
    player.screen = 3;
    player.x = 15;
    player.entered = 0;
}
if (player.y <-20 && player.screen === 1){
    player.screen = 7;
    player.y = 525;
    player.entered = 0;
}

//==== SCREEN 2 ===========
if (player.x > 1103 && player.screen === 2){
    player.screen = 1;
    player.x = 5;
    player.entered = 0;
}
```

```
if (player.y > 635 && player.screen === 2){
    player.screen = 5;
    player.y = 5;
    player.entered = 0;
}
if (player.y < -20 && player.screen === 2){
    player.screen = 8;
    player.y = 576;
    player.entered = 0;
}
```

Then, after this code block:

```
if (player.screen < 15 && player.start === 0) {player.start = 1;}
if (player.screen === 1 && boxesDrawn1 === 0){ drawBoxes1(); }
if (player.screen === 1 && boxesDrawn1 < 2 ) {
    boxes = boxes1;
    if (player.entered < 1){
        enemy1.x = 62; enemy1.y = 454; enemy1.dest = 0;
        bat1.x = 62; bat1.y = 70; enemy2.dest = 0;
        enemy3.y = 62; enemy3.dest = 0;
        player.entered = 1;
    }
}
```

Add this:

```
if (player.screen === 2){ drawMyMap2(); }
if (player.screen === 2 && boxesDrawn2 === 0){ drawBoxes2(); }
if (player.screen === 2 && boxesDrawn2 < 2 ) {
    boxes = boxes2;
    if (player.entered < 1){
        enemy1.x = 62; enemy1.y = 448; enemy1.dest = 0;
        enemy2.x = 62; enemy2.y = 192; enemy2.dest = 0;
        enemy3.y = 62; enemy3.dest = 0;
        player.entered = 1;
    }
}
```

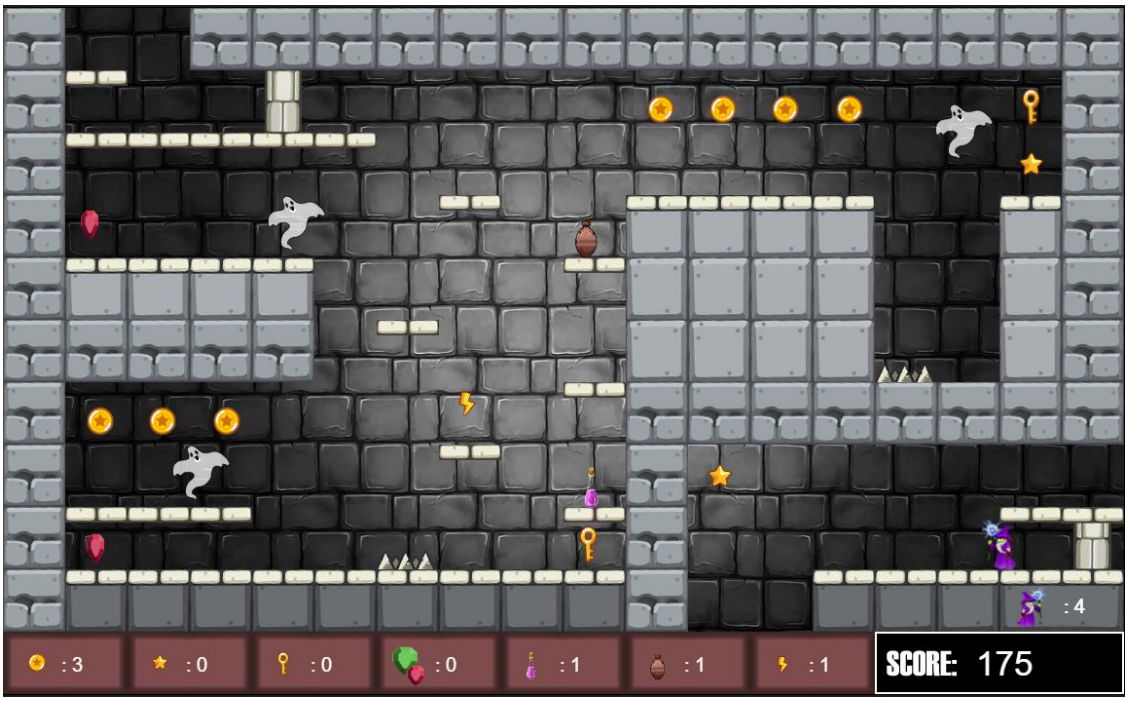
Okay, let's examine the code more closely.

```
if (player.x <5 && player.screen === 1){
    player.screen = 2;
    player.x = 1103;
    player.entered = 0;
}
//if the player's x position is less than 5 pixels and the player is on
screen 1, change the player's screen to 2, set the player's x position to
1103 pixel (right-hand side), and set player entered to 0
```

There are conditionals for every possible exit or entry point from and to each screen. Next, we have the following code block:

```
if (player.screen === 2){ drawMyMap2(); }
//if the player is on screen 2, draw the tile map

if (player.screen === 2 && boxesDrawn2 === 0){ drawBoxes2(); }
//if the player is on screen 2 and collision boxes are not drawn – draw
them

if (player.screen === 2 && boxesDrawn2 < 2 ) {
//if player is on screen 2…

    boxes = boxes2;
//add collision boxes for screen 2 to the boxes variable so that the
player will interact with them

    if (player.entered < 1){
//if player entered is 0…

        enemy1.x = 62; enemy1.y = 448; enemy1.dest = 0;
        enemy2.x = 62; enemy2.y = 192; enemy2.dest = 0;
        enemy3.y = 62; enemy3.dest = 0;
        player.entered = 1;
//draw the enemies and then set player entered to 1
    }
}
//close the conditional
```

Now, refresh your screen, press ‘s’ to start, and navigate the player into screen 2. If you followed everything correctly, your screen 2 tilemap should look like this:



Now, in the same manner, you can add as many screens as you wish. Ensure, as you design the game screens, that the entrances/exits match across screens, as below:

Amazing job! To finish this exercise in the next section we’re going to add some background music and sound effect (SFX).

STEP 15 - ADDING SOUND (SFX) AND MUSIC

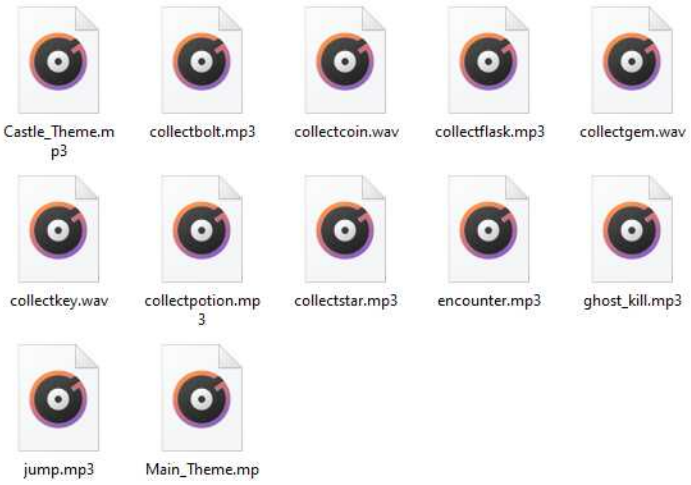
There are many ways to feedback to a player as they navigate through a game world. A game monitor, like ours, animations, and also sound. In the real world, we experience many things through our senses. It's the same in the gaming world. A sound for when a player dies, an animation for when they respawn, a sound effect when they eat something tasty, or another for something that tastes not so good. The possibilities are endless. The only thing that stands in your way is your imagination. The more layers and behaviours you add to your game, the more engaging it will become for players.

The first thing we need to do is bring in the sound files we need. As in all cases before, these are available for you to download via the link below:

https://wddtrw/resources/learntocode/platformgame_sfx.zip

Once downloaded, unzip it and add the contents to the 'sfx' folder. To unzip the file, double click it and then select all of the contents (CTRL + A) and copy it (CTRL + C), then open the 'sfx' folder and paste (CTRL + V).

If all went well you should have the following:



Next we need to add references to two new JavaScript files that we'll create to handle our sound. Open 'sorcerer.html' and add the following references just before the 'tiles.js' file reference:

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

Save the file.

Next, create a new file called 'sfx.js' and save it in your 'js' folder, as usual. Then add the following code:

```
var sfx = [];
sfx[0] = new Audio('sfx/collectcoin.wav');
sfx[1] = new Audio('sfx/collectgem.wav');
sfx[2] = new Audio('sfx/collectstar.mp3');
sfx[3] = new Audio('sfx/collectpotion.mp3');
sfx[4] = new Audio('sfx/collectflask.mp3');
sfx[5] = new Audio('sfx/collectbolt.mp3');
sfx[6] = new Audio('sfx/collectkey.wav');
sfx[7] = new Audio('sfx/ghost_kill.mp3');
sfx[8] = new Audio('sfx/jump.mp3');
sfx[9] = new Audio('sfx/encounter.mp3');
sfx[10] = new Audio('sfx/Castle_Theme.mp3');
sfx[11] = new Audio('sfx/Main_Theme.mp3');
```

Here, we have created a new array called sfx. Then we have added our sounds at chosen indexes so that we can use them in our game code with ease.

Save the file (CTRL + S).

Okay, now we have to add instructions to play the sound files in certain circumstances. Let's handle the collectables first. Open 'collectable.js' and edit the following conditionals:

```
if(item.t == 'coins'){ player.score+=10; sfx[0].play(); }
if(item.t == 'gems'){ player.score+=50; sfx[1].play(); }
if(item.t == 'keys'){ player.score+=20; sfx[6].play(); }
if(item.t == 'potions'){ player.score+=30; sfx[3].play(); }
if(item.t == 'flasks'){ player.score+=100; player.lives+=1;
sfx[4].play(); }
if(item.t == 'stars'){ player.score+=25; sfx[2].play(); }
if(item.t == 'bolts'){ player.score+=15; sfx[5].play(); }
```

Add the highlighted code. This will play a different sound when you collect a particular item.

Save the file and try it out!

Next, open 'main.js'. Just after the opening of the main update function, add the following two conditionals:

```
function update() {◂ add the following two lines of code after this

    if(player.screen === 15){ playMusic(2); }
    if(player.screen < 15 && player.lives > 0){ playMusic(1); }
```

Then, inside the jump controls conditional, add the following highlighted instruction, then save the file:

```
    if (keys[38] || keys[32]) {
        if (!player.jumping && player.grounded) {
            player.velY = -player.speed * 2.5;
            player.jumping = true;
            player.grounded = false;
            sfx[8].play();
        }
    }
```

Next, create a new file called ‘music.js’, add the following code and then save it in the ‘js’ folder:

```
function playMusic(m){
    if(m === 1){
    sfx[11].pause();
    sfx[11].currentTime = 0;
    sfx[10].addEventListener('ended', function(){
                    if(sfx[10].currentTime === sfx[10].duration - .34){
                        sfx[10].currentTime = 0.1;
                        sfx[10].play();
                    }}, false);
    sfx[10].play();
    }
    if(m === 2){
    sfx[10].pause();
    sfx[10].currentTime = 0;
    sfx[11].addEventListener('ended', function(){
                    if(sfx[11].currentTime === sfx[25].duration - .34){
                        sfx[11].currentTime = 0.1;
                        sfx[11].play();
                    }}, false);
    sfx[11].play();
    }
    if(m === 0){
    sfx[10].pause();
    sfx[10].currentTime = 0;
    }
}
```

In a nutshell, this function loops the music at the end of its duration and changes the music when the player is either on the title/game over screens, or alternatively, a game screen. There are a few new instructions here that we haven’t previously explored in this book.

They are currentTime, duration, pause() and play(). The play() method starts playing the current audio, while the pause() method pauses the audio, currentTime returns the current play time and duration returns the duration, in other words the total length. These are inbuilt JavaScript functions. They have the following syntax:

```
audioObject.play()

audioObject.pause()

audioObject.currentTime

audioObject.duration
```

Our playMusic function attempts to loop the music seamlessly by starting it over again when the music score reaches near the end.

Next, let's update 'enemy.js'. Open the file and add the following just inside the opening for the lifeLost function:

```
sfx[7].play();
```

And then add the following, just indise the opening for the gameOver function:

```
if(player.lives < 1){ PlayMusic(2);}
```

Save the file ((CRTL + S).

Now we have music and sound effects. Why not try experimenting and adding more of your own? If you don't have the means to edit sound files I would highly recommend Audacity, which you can download free of charge.

You will find it at:

https://www.audacityteam.org/download/

Just download the version you require for your particular system.

STEP 16 - CONCLUSION

We have come to the end of this exercise, but this is only the beginning of the journey.

Ideas of things to try:

- Add more game screens
- Make new tiles and tilemaps
- Add different enemies
- Try adding different collectables, such as berries or bread. You could also add something that you should avoid eating. Maybe if you eat poisonous berries the player will lose a life?
- Spawning and respawning animations
- Encounter and dying animations
- Main character animations – walking, jumping
- Casting spells
- Determine how far you can fall without dying
- Add a high scores table
- Doors – Collect keys to open
- …the list goes on

As usual, you can download all associated files via the following link:

https://wddtrw/resources/learntocode/platformgame.zip

DEPLOYING YOUR PROJECTS ONTO A WEB SERVER

To be able to deploy your projects onto a web server and to be able to access them via the internet, you will first have to choose a web provider.

There are some free web spaces available, but they usually come with many restrictions. There are a few cheaper web spaces available if you want a webspace for testing your personal projects.

You can get effective and very reasonably priced packages from the likes of Heart Internet and Ionos 1 & 1. Packages start from only a few pounds or dollars a month. To deploy your own code all you need is a domain name, a hosting package, and an SSL certificate.

You can get packages from Ionos (at the time of writing this book) from only £1 plus VAT per month for the first 6 months and then only £4 plus VAT thereafter, which comes with a free SSL certificate. Then you can buy a domain name for around £10 a year, but again the first year is usually discounted to as low as £1 plus VAT.

FILE TRANSFER PROTOCOL

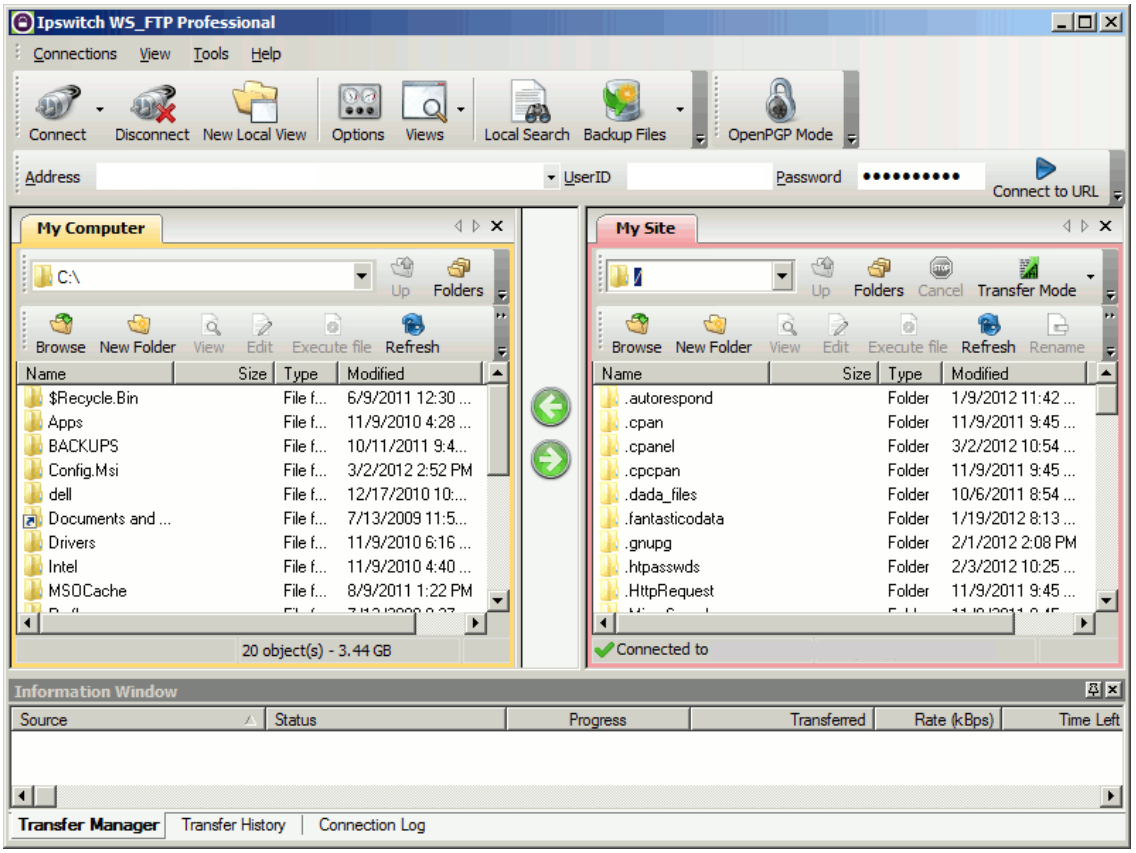
Once you have web hosting, you can use a system called file transfer protocol (FTP) to upload files with ease, or if dealing with development on a more professional basis you would use a repository, such as GIT hub, but that is beyond the scope of this book.

In the most basic of terms, when you upload a file to a web server, any files that are within the public html folder are available for public access. When you connect a domain name to your webspace, if you upload a file to the public folder called index.html or index.php, for example, you do not need to stipulate the file name.

E.g. https://wddtrw.co.uk opens a file called index.php in the public folder of the webserver.

Once you have established a connection, uploading files through FTP is much like transferring files from one folder to another on your PC or Mac. The only difference is that you are transferring the file to another computer.

Here's a screenshot of WS FTP Pro:



Once you've made a connection to the webserver, the left-hand window displays files on your local machine, while the right-hand window displays

files on the server. You can upload, download, and delete files and folders from the webserver easily in this way.

Internet services providers (ISP's) usually provide a control panel with FTP access, as well as there being FTP client tools available, such as WS FTP, FileZilla, Cyberduck, etc.

If you choose to buy web hosting you must be aware of what you wish to use it for. With some web spaces, you get a web builder with various tools, but they are restrictive. Generally speaking the fewer restrictions, the more technical the systems are to use. If you're unsure, almost all ISP's have great customer service and or huge databases full of frequently asked questions.

Web servers generally fall into the following categories:

- WordPress or other web builders – No secure shell access
- Web Hosting and SSL only – No secure shell access
- Virtual Private Servers – VPS – Secure Shell access
- Dedicated Servers– Secure shell access

If you want to build a regular website and you want some quick and easy tools and less coding, then a WordPress or web builder hosting would be suitable for you.

If you are new to web coding and you just want to learn more about hosting a website or web app, I would choose regular web hosting with an SSL certificate. Make sure the SSL certificate is included, otherwise you may find yourself paying around £60 - £100 + per year to make your website secure.

If you are a more competent web developer, then you won't go far wrong with a virtual private server. A VPS Hosting simulates the experience of a dedicated server even though you're still sharing the physical server with other users. A VPS can be very cheap but requires more technical proficiency. With SSH access you have more flexibility over how the server is configured.

Then, if budget is of no concern and you are an experienced developer, a dedicated server may be the right choice for you. The main reasons you would choose a dedicated server are security and heavy traffic. A dedicated server has all available power and flexibility all to itself, instead of sharing resources with others.

WHAT IS SSL?

SSL stands for Secure Sockets Layer and, in short, it's the standard technology for keeping an internet connection secure and safeguarding any sensitive data that is being sent between two systems, preventing intruders from reading and/or modifying any information transferred, including potential personal details.

WHAT IS SSH?

SSH, also known as Secure Shell or Secure Socket Shell, is a network protocol that gives users, particularly system administrators, a secure way to access a computer over an unsecured network.

It provides access to be able to execute commands and gives flexibility over how a server can be managed.

WHERE TO GO FROM HERE

In 'Book 2' (soon to be published) we'll explore three more great developments, plus we'll continue with our Sorcerer's Mountain game development by adding character animations using sprite sheets, magic powers, portals, a high score table, and much more.

For now, try designing some of your own game levels. See you in the next book.

I'd love to hear from you. You can email me at:

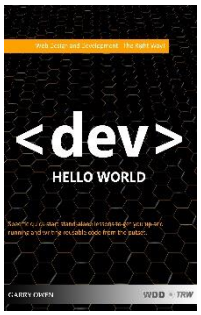
learntocode@wddtrw.co.uk

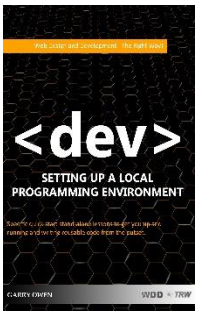
If you enjoyed this book, please leave me an honest review on Amazon. I'd very much appreciate it. If not, please contact me and tell me why, so that I can help you and improve future editions.

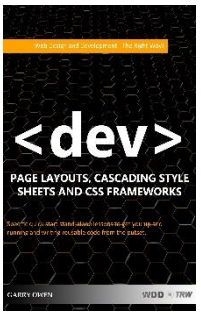
My very best wishes,

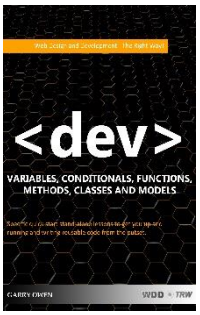
(Software Developer, Designer, and Author.)

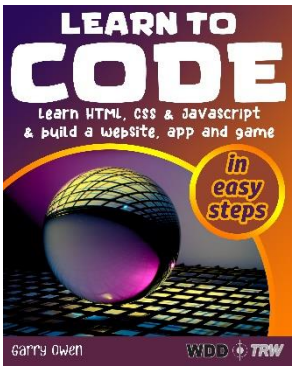
CHECK OUT SOME OF MY OTHER PUBLICATIONS (YES, I WRITE FICTION TOO!)

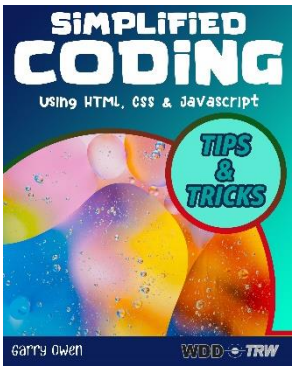


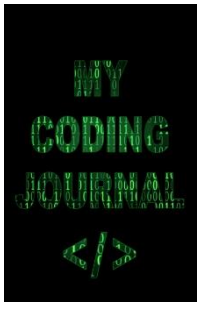


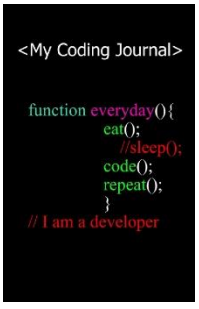


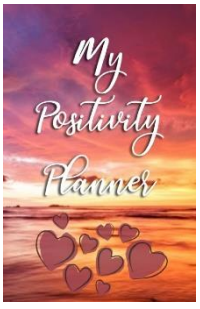




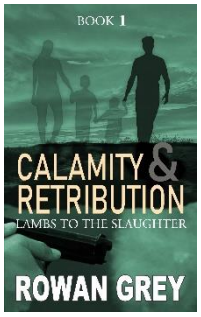


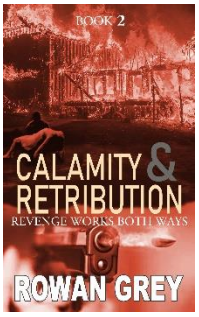


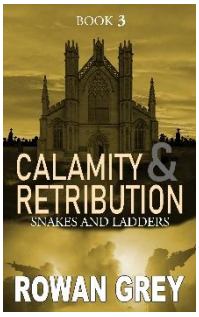


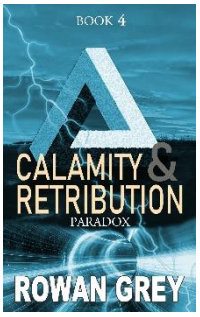


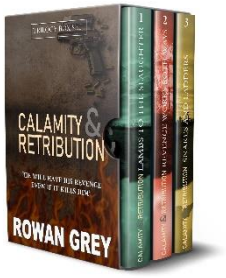
Learn To Code Book 2 is due out around April 2022. Follow me on Amazon to keep up-to-date and get notified when it's released.











For almost the same reason I fell in love with the art of computer coding, I also found a passion for writing.

At the age of 9 years old, I had my first computer as a birthday gift and fell in love with the art and creativity of computer coding. I found it enabled me to create characters, game worlds, and other magical features, only limited by the depths of my imagination.

Likewise, writing for me opened the door to the same wonderful universe of imagination and with me sitting at the helm, the ability to navigate and explore under my complete and unhindered control.

One of my favourite quotes which sums up the driving force behind my passions is:

"Everything you can imagine is real." - Pablo Picasso.

INDEX

LEARN TO CODE

Learn HTML, CSS & Javascript

Strap yourself in and get ready to have some great fun learning to code. Each exercise in this book is fully illustrated in colour, and in easy-to-follow steps.

First, write the 'Hello, World!' program with added Falir...

Next, design and code a Fast Access Web Menu. Get to all of your favourite websites in a single click!

Then, code an awesome multi-question quiz web app, with a countdown timer and admin functions

Last but not least, design and code a fully-fledged tile based platform game. Learn about tilemaps, sprites, collisions, gravity, collectables, enemies, player controls and more...

ISBN 9798417818691

WDD TRW