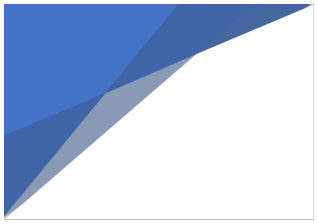


Mobile Systems Development

Theory + Exercise Booklet

Responsive Web Apps

Table of Contents

Overview

Welcome!!

This booklet will cover the basic theory for each week, as well as exercises for you to complete.

The exercises each week are designed to take 3-4 hours to complete, so plan ahead for the time you need to allocate.

This is an intensive unit. In just 10 weeks, I'll be pushing you to go from zero to making a complete responsive web app. That means each week you'll need to keep pushing ahead, making sure you not only cover all theory material and complete all lab exercises, but also write up your summaries and lab answers into your Portfolio document each week, **and** most importantly, be working on your Project each week.

This will be extremely rewarding once you get to the end.

Good luck!

Topics

Week 1	• What are apps / desktop apps / mobile apps / web apps / software. UI VS Events VS Code. UI VS UX. • Network theory (Google Chrome network) • HTML (document) • HTML linking to other documents. • HTML (app) content within same document
Week 2	• CSS for a document. • External vs Internal style sheets. • Colour theory (always write in CSS comments at top, and use CSS variables for how many colours you want, in BODY tag). • spacing (padding vs margin) • layout (relative / absolute)
Week 3	• JavaScript basics. • Functions • Event handlers.
Week 4	• JavaScript variables. • Objects & Arrays
Week 5	• If-statements • Loops
Week 6	• Drag & Drop events
Week 7	• Bonus material

Week 1

Week 1 Theory

What is the Web

The “web” is a combination of several concepts:

1. The first concept is the “Internet”. The Internet is a large (massive) collection of computers, devices, cables, routers and servers. Cables (and wifi) connect your computer to a router. Routers connect to other routers at your Telecom provider, via any number of cables (copper in phone line cables is used for ADSL, different copper is also used for Cable Internet and Cable TV, and fibre optic cables are used for fibre connections). These routers then connect to other routers (to other Telecom providers), to make connections between you and other computers or servers.
2. The second concept is “servers”. A server is simply a computer, or often a stack of computers, that perform requests or store & send data. Anytime you hear the word “cloud”, that simply means a server (or a group of servers), which means a group of computers. The difference between your home computer and a server isn’t the hardware (they use the same CPUs, RAM and disk drives that you do). Rather they usually don’t bother with a monitor and keyboard, and they’re usually stored in a cool room with backup battery power, redundant network connections, and security doors. That’s because servers are designed to be “always on”. E.g. when you access google.com.au, no one wants that site to be offline, you want it always available.
3. The third concept is “addresses”. In order to send a message from your computer to a server, or from your computer to a friend’s computer, you need a way of identifying them. This is the same as sending mail at the Post Office. Everyone needs an address to send to, and a reply address to receive responses. The Internet has the same concept. There are two primary (not the only two) address systems that you need to be aware of:
    a. IP addresses (Internet Protocol addresses). These are 4-chunk numbers, each part a value from 0-255. E.g. 127.0.0.1. Each address is unique, and each country runs its own governing body assigning bundles of addresses (called “blocks”), who then give them to ISPs (Internet Service Providers), which are your Telecom companies, who then allocate one to you (and many to businesses who run servers).
    b. URL addresses (Universal Resource Locator). An IP address only says “which computer”. It doesn’t say which file or which application on a computer to access. Nor is it very easy to remember a number. Also, often those numbers might change as servers move locations, but you don’t want to have to remember a new number each time that happens. So, another system was developed, which maps hostnames (such as “google.com”, or “facebook.com”) to IP addresses. The URL also includes additional information, such as the

protocol to use (the language of communication the computers will use to talk to each other), and also any specific file path to access, and also any specific query information to include. Example:

```
https://www.google.com.au/search?q=deakin
```

The above URL includes 4 parts:

`https://`	The protocol. It's saying "talk to me in encrypted https language"
`www.google.com.au`	This is the address of a computer (a server) on the internet. Your computer will look this up in DNS and convert it into an IP address.
`/search`	This is the file location on the server. It's saying access the file "search", which won't be a static file, but a code file to run and do something for you (e.g. search), and return the result.
`?q=deakin`	Anything after a question-mark is additional query information. The exact variables depend on each individual server, and how they work.

4. The final concept we are concerned about is the "web". The web is the World Wide Web (WWW), a collection of protocols that define how to serve and retrieve "pages" of content, which are then rendered (visually drawn) by a web browser. This includes some other concepts:
    a. **HTTP / HTTPS**: HyperText Transfer Protocol. Ignore the word HyperText, and just focus on this: "Transfer Protocol". This is a language, as in, a standard of communication, for requesting and then receiving files. That's all it is. HTTPS is the encrypted version, whereas HTTP is in plain (readable) text. A "web server" is a piece of software that understands this language (usually running on a server, but you can run them on your desktop too). A "web browser" also understands this language, and simultaneously understands how to read and visually draw "html" and related pages. A web browser talks to a web server using HTTP/HTTPS, to ask for files (HTML files, CSS files, JS files, image files, sound files, and many others).
    b. **HTML**: HyperText Markup Language. This is a "language" for how to render (how to draw) a web page. It's basically a set of instructions, saying things like "draw a header with the text 'Table of Contents', then draw a table with these details…, etc.".
    c. **CSS**: Cascading Style Sheet. This one has a weird name, I know. A Style Sheet is like a sheet of paper, with a list of rules on how to "style" elements. It's like if you were building a house and you wanted "window frames in dark red, door frames to be dark blue, doors to be light blue, the front door to be white, etc.". Styles don't just say the colours of components, but also can do the sizes, such as "window frames to be dark red and 4cm width, front door to be white and 120cm width, etc." A style sheet is the set of rules used to tell a web browser or web app how each component / element should look. How should a heading look, how should a table look, how wide should the article text be, etc.

d. **JS**: JavaScript file. This is a file of program code. Program code is used to tell the computer what to do when certain events occur. For example, if the user clicks on the login button, we tell the computer to go to the login page. Program code always happens on events. Like the user clicking a button, or email being received (and thus needs to be drawn in HTML), or even simply the load event, when the page loads up and we want to draw (create) a game interface.

What we will focus on entirely is the HTML, CSS and JS files of this, because using those we can create web apps. A web app is simply a single web page that operates dynamically. Usually web pages are static, each one containing text, like pages in Wikipedia. But a web app operates dynamically, changing the content you see based on the commands you use, such as a search engine, an email client, even Facebook is a web app, because the content changes based on your settings. It is "dynamic".

Responsive Web Apps

You want to know the big secret about responsive web apps? They're actually just regular software applications. They are design + program code + data.

There are many ways (many programming languages) to create applications. What we'll be doing is using web technologies to make web apps, which we can then test in any web browser. Later if we want to compile them to real apps, we can use Electron (for desktop apps) and Cordova (for mobile apps), both open source tools.

UI and UX

The User Interface, and User eXperience, are concerned with how our apps look (UI) and feel (UX). The UI is simply how our app looks. This involves where we place things (called layout), the spacing between things (called padding and margin), the fonts we use, and most importantly, the colours we use. We should never choose a series of colours randomly. Rather, we should either select colours from an existing design we have seen before, or we should use Colour Theory. There are many colour theories to choose from.

I'm going to recommend a site:

- http://paletton.com

This is one of many calculators you can use online to find matching colours to any primary colour you select. E.g. if your primary colour is red, what are the related colours that complement the primary colour, that you can use for headings or borders, etc.

The most common "theories" for calculating complementary colours are:

1. **Monocromatic**: a single colour with multiple shades of that colour.

2. **Adjacent**: two additional colours, on each side of the primary colour on a colour wheel.
3. **Triad**: two additional colours, on opposite sides of the primary colour on a colour wheel.
4. **Tetrad**: A total of 4 colours, each equally spaced on a colour wheel.

Now, I'm aware at first glance this probably seems quite confusing. And that's expected. You can't know anything until you learn it and use it. So, whenever you are confused, your task is to immediately jump onto Google and start asking questions, and then summarising what you find in your Portfolio.

Layout, spacing and fonts are very large topics on their own, and not something I can condense to you simply. Rather, I want you to look at layouts, spacing and fonts in games that you like, observe, and make notes of which ones work well and why. Also look at games/apps you don't like the layout of, and think about **why** it doesn't work. Is there not enough vertical spacing between headers to easily gauge the different sections at a glance? Does the game/app use too many different fonts which make it look inconsistent? Are the buttons or menus in a place that you don't expect? Make notes on all your personal discoveries in your Portfolio throughout all weeks.

Now, the UX is another concept. This is the user feel within an app, and is concerned with:

1. Does the layout make sense such that a first-time user can use it without knowing anything about the app.
2. Can the most common operations / actions be performed with the fewest number of taps.
3. Is the most wanted information easily available on the main screen, or very near the main screen (e.g. one tap away).

Presentation

Please make your work **look** amazing. The look (and feel) of any document / web app has been proven again and again to be the biggest influencer in success. If your work looks good, statistically you are far more likely to get a higher paying job as well. Always use MS Word styles for document consistency. Always apply Colour Theory to your app colour choices.

HTML Document

Every single HTML file has a certain structure that web browsers understand (which render, or "draw", the file as instructed). You must adhere to this standard correctly, otherwise your browser won't render the page as you intend. Note that one of the challenging issues with the HTML language is that, if there are mistakes, your web browser won't tell you. Rather it will try to "guess" your meaning. And every browser has a different way it guesses mistakes. This is one of the main reasons why web pages often look different in different browsers.

A HTML file is simply a text file, that is, a file that looks like plain English text, and has the file extension ".html" (in earlier times it was simply ".htm", as early versions of Windows limited file extensions to 3 characters in length). A quick note here is that file extensions are how your computer (the operating system, the web browser, etc.) is being told to interpret the file. A ".jpg" extension tells your computer "interpret this as a JPEG-formatted image". A ".docx" extension tells your computer "interpret this as a Microsoft Word 2005+ document.

The language of HTML allows you to write both text you want to appear, and instructions for how that text should be structured (e.g. into paragraphs / headings / bullet points / tables). The language of HTML differentiates between text you want to appear on screen, and text that are instructions for the web browser, by using what are called "**tags**". A tag is any text inside <> brackets. Normally each tag will also be in pairs, where we have an opening tag, <tagName>, followed later by a closing tag </tagName>. We almost always need a closing tag because we need a way to tell the browser where a certain component's information stops.

Consider this example of a paragraph:

```
<p>This is a single paragraph.
      All this text appears on a single line.
</p>
```

The P tag is the HTML specified code to tell the browser to render a paragraph. All of the text in that paragraph is anything between the opening tag (<P>) and the closing tag (</P>). If we didn't have the closing tag, the browser wouldn't know where our paragraph ended. I will mention an important side note here: HTML pages don't care about text on different lines, it ignores any line breaks.

So, now we know that HTML consists of tags, and these tags tell the browser to do something. Before we start designing our HTML document, we have an overall structure for every single HTML file:

1. The entire file wraps inside a HTML tag. Inside the HTML tag we have:
    a. A HEAD tag, for general information about the file.
    b. A BODY tag, with the actual contents that we render as the web page.

So, every single time you create a HTML file, always start with this structure:

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

      <body>

      </body>
</html>
```

HTML Head Section

The HEAD tag of a HTML file is for general document information. This can include things like a title (which the browser may use for a bookmark title, for example), the language the document is in, and other metadata to describe the document. When we get to CSS styling and JavaScript programming, those will also go in the HEAD tag.

In the following example, I will add a document title in the HEAD tag, and a P tag.

```
<html>
	<head>
		<title>My First Webpage</title>
	</head>

	<body>
		<p>This is my first homepage.</p>
	</body>
</html>
```

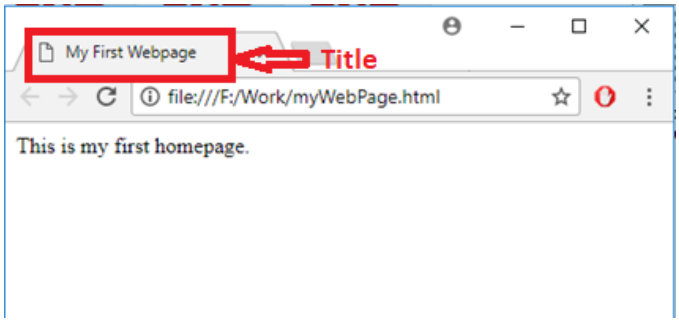


Fig: Title displayed on the web browser.

HTML Body Section

HTML was originally designed to be a language to describe documents. A web page is thus capable of being exactly like a Microsoft Word document. The same elements that you find in Microsoft Word documents, you'll generally find in the HTML language.

All content you want to render, i.e. "display", visually, must be in the BODY tag. We can do this in two ways:

1. Statically: by physically writing in the BODY tag within our HTML file.
2. Dynamically: by using JavaScript to write in the BODY tag for us.

We'll be looking at dynamic content when we get to JavaScript programming in a few weeks. For now, we'll just look at statically creating BODY content.

The HTML language understands several tags for different elements. The most common examples are:

<H1>	This is a Heading 1
<H2>	This is a Heading 2 (we can go all the way to H6).

<P>	Paragraph
<OL>	Ordered List (a numbered list)
<UL>	Unordered List (a bullet list)
<LI>	List Item (an item in a list)

You can try all of these with the following example:

```
<html>
	<head>
		<title>Project Plan</title>
	</head>

	<body>
		<H1>Introduction</H1>
		<P>This is my introduction paragraph.</P>

		<H2>Subsection</H2>
		<P>This is my first sub-section. I'll now demonstrate
some lists.</P>

		<OL>
			<LI>This is item #1</LI>
			<LI>This is item #2</LI>
			<LI>This is item #3</LI>
		</OL>

		<UL>
			<LI>This is a bullet point</LI>
			<LI>This is another bullet point</LI>
			<LI>This is a bullet list with a sub list:
				<UL>
					<LI>Sub bullet point</LI>
				</UL>
			</LI>
		</UL>
	</body>
</html>
```

In the above example, I demonstrate some of the main components of a document. Here I hope you can see why we need open and closing tags, because it's possible to put almost any amount of content "inside" the open and close tag. For example, I have one bullet point that inside it then has another list. The closing </LI> doesn't occur until after the <UL> and </UL> tags within it, with that sublist containing an <LI> for the sub bullet point of its own.

Simple Text Styling

As well as defining the components, the HTML language also has some basic tags for styling. Note that styling is mostly the task of CSS, which we'll cover later. But we can define blocks of text to be a certain style with the following:

****	Defines important (bold) text
****	Defines emphasised (italic) text
<code>	Defines a piece of computer code
<samp>	Defines sample output from a computer program

<blockquote>	Defines a block quote

Tables

We can also draw tables, which take a few different tags to create. These tags are:

<table>	Defines the table itself
<tr>	Defines a table row
<td>	Defines a table cell
<th>	Defines a header table cell

Here is a complete example you can experiment with to see how each tag works:

```
<html>
	<head>
		<title>Table Experiments</title>
	</head>

	<body>
		<TABLE>
			<TR><TH>Tag Name</TH><TH>Definition</TH></TR>
			<TR><TD>ul</TD><TD>unordered list</TD></TR>
			<TR><TD>ol</TD><TD>ordered list</TD></TR>
		</TABLE>
	</body>
</html>
```

Special Characters

What if we wanted to display “<” or “>” inside a paragraph (or any other tag)? The browser will assume “<” and “>” are part of tags, so it won’t render them.

Instead, we need to use special encodings, called HTML Entities. These are special codes that render any special character. A list of the most common are defined below.

Character	Entity Name	Description
non-breaking space		
<	&lt;	less than
>	&gt;	greater than
&	&amp;	ampersand
"	&quot;	double quotation mark
'	&apos;	single quotation mark (apostrophe)
¢	&cent;	cent
£	&pound;	pound
¥	&yen;	yen
€	&euro;	euro
©	&copy;	copyright
®	&reg;	registered trademark

Example Usage:

```
<html>
	<head>
		<title>Table Experiments</title>
	</head>

	<body>
		<H1>My Product&copy;</H1>
		<P>A tag looks like this: &lt;P&gt;</P>
	</body>
</html>
```

Spaces in HTML

Note, the very first one is a space, as in " ". This is because multiple spaces don't mean anything in HTML, the browser doesn't render all line breaks, tabs, and spaces, except for a single space between words.

```
<html>
	<head>
		<title>Table Experiments</title>
	</head>

	<body>
		<H1>A title with	lots	of spaces</H1>
		<H1>A title with	spaces
			and


			lines
		</H1>

		<H1>A title using a non-breaking space entity:  
          tada!</H1>

	</body>
</html>
```

Comments

One habit that is absolutely **essential** when creating apps is to add comments. Comments are notes left for humans (you and me) to read. They are ignored by the web browser and by the computer when rendering your app. Almost every single computer language has a syntax for adding comments to your code. In HTML, it is done with two special tags, the open tag: "<!--", and the closing tag "-->"

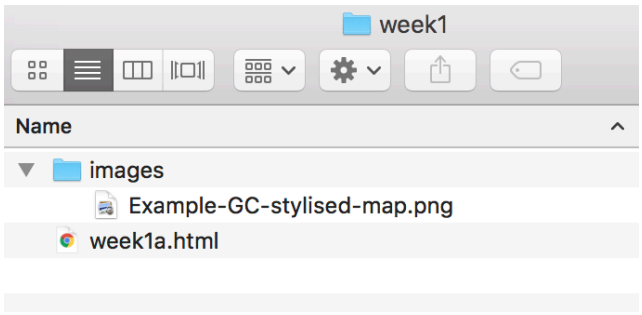
```
<!--  This is my comment.

	It can be as many lines as I want.
-->
```

Normally we would add comments to explain sections. Such as:

```
<!-- Table of
<TABLE>
      <TR><TH>Tag Name</TH><TH>Definition</TH></TR>
      <TR><TD>ul</TD><TD>unordered list</TD></TR>
      <TR><TD>ol</TD><TD>ordered list</TD></TR>
</TABLE>
```

Images

Now another feature we usually want is images. Images are slightly more complicated, as the "data" for the image is in a separate file. We first need to obtain an image (a JPG or PNG file, for example), and copy it into our directory. Usually we create a sub-directory called "images" to store all images. The following is an example of a week1 directory, in it a HTML file for week1 exercise A, and an images directory containing my image.



The image tag is one of the few tags that **does not** have a closing tag. This is because there's nothing that can possibly be "inside" the image, it is simply the image. Here is the code that would add the image inside a HTML document:

```
<img src="images/Example-GC-stylised-map.png">
```

Here we use a concept called an **attribute**. An attribute is additional information related to a tag. It's similar to the concept of the HEAD tag inside HTML, containing meta information related to that element. In our case, we use the attribute **src** to define the source of the image.

Some tags have multiple attributes, which we list inside a tag's opening tag, and separate by spaces. For example, let's say we have an image, and we want to provide alternate text to display for browsers that maybe can't display images (such as Braille text browsers for those with visual impairments). We can use the **alt** attribute for this purpose.

```
<img src="images/Example-GC-stylised-map.png" alt="Map of GC">
```

Note, it's actually a requirement of the official HTML standard to include an **alt** attribute with every image, so make sure you include it. You can think of it like a caption, except it doesn't display if the image displays normally, so you'll also want a separate caption on your images.

To caption an image, we can use the <FIGCAPTION> tag. However, in order to let the browser understand which image the caption relates to, we need to wrap both tags inside a <FIGURE> tag. Otherwise the browser wouldn't know that they are related. Example:

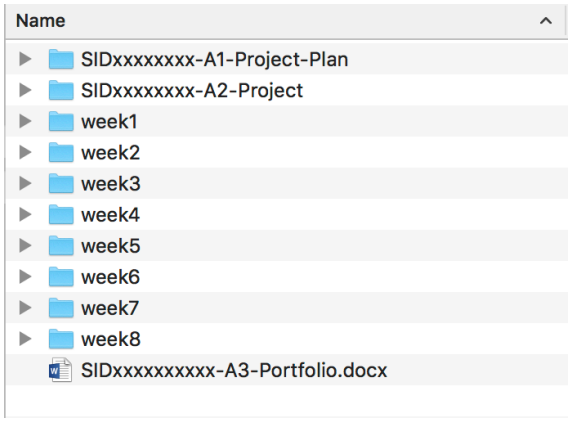
```
<figure>
      <img src="images/Example-GC-stylised-map.png" alt="Map of GC">
      <figcaption>Fig1. - A map of the GC.</figcaption>
</figure>
```

Week 1 Exercises

Before we start, you'll need to create a directory (folder) on your computer for use in this unit.

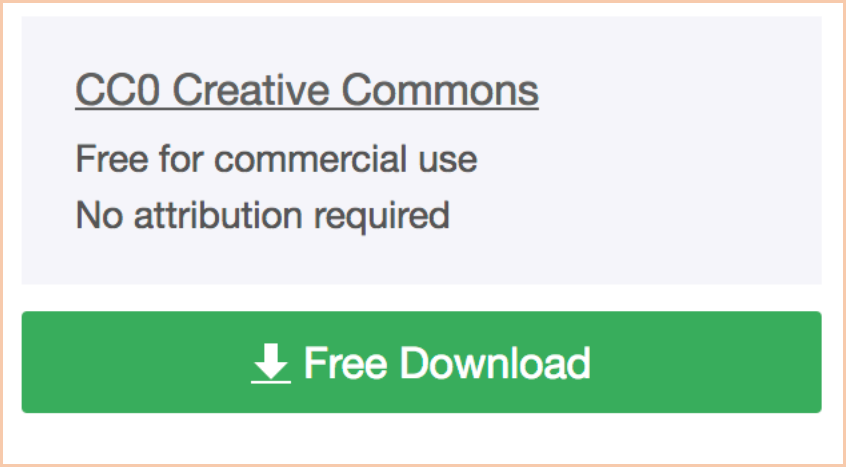
Inside that directory (folder), you'll create a separate directory for each week's exercises, as well as a directory for your project plan and project. Your portfolio doesn't need a directory normally, as it's just a single MS Word document.

Your unit directory will look like this:



Exercises:

A	Image searching. It is important that you are able to find images that you can legally use, even for commercial purposes. Such images are either in the Public Domain, or the owners have specifically said you can use them for commercial purposes. Note, a useful site to search other search engines for images with the correct license type, is at Creative Commons: https://search.creativecommons.org/ 1. Search for the word "car", and select Pixabay. Ensure you have ticked the two boxes below the search box. 2. Now click on any image of a car you like, to go to that image's page. 3. On this page, you should now see options for downloading the image, as well as license information. Look for text like this:



The text says it is a CC0 license type (which is one of the ones we can use, along with CC-BY and Public Domain). You'll note it also says "Free for commercial use", summarising what the CC0 license allows.

Also note it says "No attribution required". Some images will require you to attribute (credit) the author within your document or app. If you've ever seen websites, such as news sites, where in the caption they say "Photo credit: Person's Name". That's attribution. Legally you must do it if the image requires attribution.

Note, when you use a search engine to search for creative commons images, not all images will be correctly tagged with the license you require, so for each image, you need to check the image page to ensure the license does allow commercial use. This is extremely important to ensure you are legally in the clear to use the image.

Add all information to a licenses.txt file inside your week1 directory.
Then copy the image into an images/ sub-directory inside your week1 directory.

B

For these exercises, you'll need to use a text editor to edit HTML files. I recommend the open-source Atom text editor, which is detailed in Appendix 1.

Open the week1 directory (folder) for this week's work inside the text editor.

Then create a new file named "week1a.html".

Inside this new file, I want you to create a basic HTML page, and then add in **every** HTML element described in the Theory section of this week, one by one. As you add each one, test how it looks in a browser, so that you understand the concept. Play around with the text inside each tag. Play around by putting elements (tags) inside other elements, to see how the result works. Your finished page should not just be a copy of each of my examples, but demonstrate that you've experimented with each of them.

You'll need to include:

1. All basic HTML elements.
2. Experiment with spaces and HTML entities.
3. Add a table with headers both in the first row and first column ☺
4. Add an image with alt attribute, caption, and width + height (setting them to half the actual width + height, so that your image will look crisp on high-density devices).

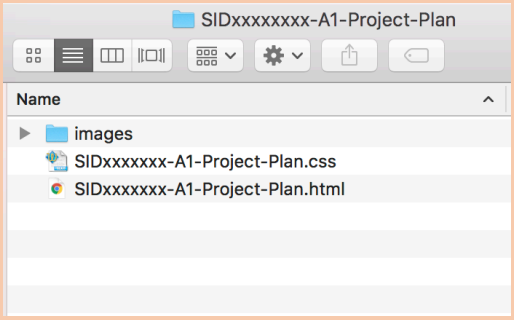
When you are done, paste a screenshot of all components rendered on your page into your Portfolio.

C

Now you're going to start work on your first assessment item: the Project Plan.

First, you'll need a directory for your project, named:
"SIDxxxxxxxxx-A1-Project-Plan"
(replace xxxxxxx with your Student ID number).

Inside it, you'll create an images directory (to store all your images), and then use the Atom text editor to create your HTML file for the document, named:
"SIDxxxxxxxxx-A1-Project-Plan.html"



Now you can start work on the HTML document itself.

Your tasks:

1. Create each heading (a H1 tag) that you need, as per the Project Plan: Required Sections.
2. In the first section, Marking Justification, create a 3-column table, with a row for each rubric. This should look like the following:

Criteria	Grade I Want	Evidence
Presentation	C	
Rich Text	P	

You can leave the "Grade I Want" and "Evidence" sections blank for now, and complete them once you have finished your document.

D

Now that you have a document ready for your Project Plan, you are to start thinking about what kind of educational game appeals to you the most to create. Then start

by researching competitors, as that is usually the easiest place to start and to get ideas from.

As you research competitors, you'll keep filling out your Project Plan HTML document, practicing your HTML tag skills as you go.

Remember to keep saving your document, then refreshing it in your browser, to see if you have the tags correct and everything is rendering correctly.

That's it for this week's exercises. Now you need to spend most of your time continuing your Project Plan, as you want most of the content finished in Week 1, before Week 2.

You definitely want to have your Competitor Analysis done, **and all images captured**, this week, as that takes at least 4 hours by itself.

Week 2

Week 2 Theory

Styles

Consistency is key to professionalism. When you look at a professional book / pamphlet / application, you'll notice there is consistency everywhere. In a book, all page numbers are the same font, same font size, same font colour. Each paragraph has the same space (padding) on the left and right sides of the text, between the text and the page edge. In an application, all buttons of the same type have the same colour and shape. When you look at a menu, all menu items have the same text colour, background, and font. When you're reading a pamphlet, all text within each paragraph is the same font & font size. This is consistency. It is essential to all design.

In order to make web pages, and by extension web apps, consistent, we need to ensure all our HTML elements look consistent. All H1 headings should look identical (same size, same underline, same font, etc.). To do this, we use what are called **styles**.

Styles are the descriptions used to define how elements **look**. They are defined once, and apply to all instances of an element, in order to ensure all elements visually appear consistent. We don't have to go to every single heading and alter each one, as that would be silly and take hours. Imagine a 100-page book, if you wanted to change the font of all paragraphs, and you had to manually click on every paragraph, one-by-one, and change the font. That would be absurd. Instead, we use styles, and can change all instances of a style (e.g. all paragraphs) by simply changing the description of that particular style. Many languages and applications have styles. Microsoft Word, for example, has styles that you can edit in the Format -> Styles menu, where you can adjust how your normal paragraph text looks, how your headings look, how your bullet points will look, etc.

In the world of HTML, styles are defined in a stylesheet using a language called CSS (Cascading Style Sheet). This file format allows us to define descriptions of how each element will be visually drawn.

Let's take a look at what the CSS language looks like. Below is an example of a CSS rule. It says that all H1 elements must have a border at the bottom, of 4mm thickness, be solid (not dashed), and be the colour black.

```
h1 {
      border-bottom: 4mm solid black;
```

```
}
```

A CSS file can list any number of style rules, in any number of CSS files, and these rules can overlap (thus the "cascading" part of the name "Cascading Style Sheet"). A CSS file comprises of style rules that are interpreted by the browser and then applied to the corresponding elements in your document. A style rule is made of three parts:

1. **Selector:** A selector is an HTML tag at which a style will be applied. This could be any tag like <h1> or <table> etc.
2. **Property:** A property is a type of attribute of HTML tag. Put simply, all the HTML attributes are converted into CSS properties. This could be color, border, etc.
3. **Value:** Values are assigned to properties. For example, color property can have the value either red or #F1F1F1 etc.

selector { property: value; }

Each selector can have multiple properties. For example, we may say that a H1 has a border and has a larger font size:

```
h1 {
      border-bottom: 4mm solid black;
      font-size: 24px;
}
```

Common CSS Properties

color	The foreground colour (of text usually). Note the American spelling of the word "color".
background-color	The background colour of the element's area.
font-size	
font-family	The name of the font to use.
text-align	Text alignment (left / center / right / justified)
margin	The space around each element (if there's a border, the space after the border).
padding	The space inside each element (if there's a border, it is the space just inside of the border). If there's no border, then margin and padding are much the same.
border	The type of border to draw. Generally there are 3 words to describe a border: the thickness, the style, and the colour. Examples: • 6px solid red • 1px dashed rgb(255,255,255) • 2px dotted rgb(255,0,0)

Here is an example of a CSS file using the CSS attributes just described:

```
body {
    background-color: lightblue;
}

h1 {
    color: darkgreen;
    text-align: center;
    border: 6px solid red;
}

p {
    font-family: verdana;
    font-size: 20px;
    text-align: center;
    border: 1px dashed black;
    margin: 30px;
    padding: 60px;
}
```

Style Sheet Locations

Now, one thing we haven't yet mentioned, is where we can actually type CSS definitions.

HTML definitions go inside a ".html" file. CSS definitions go inside a… ".css" file! Simply create a new blank file, name it "myStyles.css" or any name ending in ".css", and you can type your definitions there.

Now, there is one final step. The HTML file doesn't actually know about the CSS file, unless we link it. Inside the HEAD of any HTML file is where we specify information related to our HTML. Here is also where we link to other files:

```
<html>
      <head>
            <title>Table Experiments</title>

            <link rel="stylesheet" type="text/css"
            href="myStyles.css" />

      </head>

      <body>
            <H1>My Product&copy;</H1>
            <P>A tag looks like this: &lt;P&gt;</P>
      </body>
</html>
```

The syntax consists of a single tag, saying that the link is a stylesheet, that it's specific format is in text + CSS, and the location of the file/filename (href stands for Hypertext REFerence, basically the link's address).

Now, I will add that there is a way we can type CSS directly into the HTML document, but such an approach is never recommended. Keep your CSS in a separate ".css" file.

Sizes

In CSS, we can define sizes in many ways. These are the most common:

px	Pixels (the number of dots / LEDs that a screen is composed of).
%	The percentage of the parent container (Note: not necessarily the percentage of the whole window).
vw	A percentage (out of 100) of the horizontal width of the window. Example to make the width half the screen size: `width: 50vw;`
vh	A percentage (out of 100) of the vertical height of the window.
cm	Centimeters
mm	Millimeters
rem	Relative size to the document's font size. A value of "1rem" is equal to the document's font size (thus if the document's font size is 14px, then this will equal 14px.

My favourites are **rem**, **vw/vh**, and of course **px**.

Colours

In CSS, colours can also be defined in several ways. The 3 most common are using the colour name, a Hexadecimal code for Red Green Blue values, or decimal codes for Red Green Blue values. The Red Green Blue pattern is commonly abbreviated to RGB. So, if you see RGB, it means a colour definition of Red Green and Blue, where mixing those 3 colours together (like paints), you can create any other colour on Earth.

Colour Name	Hex Code RGB	Decimal Code RGB
Maroon	#800000	rgb(128,0,0)
Red	#FF0000	rgb(255,0,0)
Orange	#FFA500	rgb(255,165,0)
Yellow	#FFFF00	rgb(255,255,0)

Some notes:

- The hex codes are identical to the decimal codes. If you open up a calculator, go to scientific mode, and type in "128" in decimal, then switch to hexadecimal mode, it will appear as "80".
- The minimum value of any colour is 0. The maximum is 255 (in hexadecimal notation, 255 translates to FF). So, if you see 255 in the Red section of a colour, that means "maximum red".
- Colours combine together, similar to the way paints are mixed together. So rgb(255,255,0) means fully red + fully green, and red + green mixed together create yellow.

- It is possible to also include alpha. Alpha is the transparency. By default, everything is opaque (fully solid). We can define RGB + Alpha with the colour command “rgba”, and adjust the transparency between 0 (fully transparent) and 1 (fully opaque / solid), as such:
    - rgba(255, 0, 0, 0.5) : half transparent red
    - rgba(0, 0, 255, 0.1) : almost completely transparent blue.
- There is one other way to define colours, using Hue, Saturation and Lightness (HSL / HSLA for the alpha version). This is usually only used by graphic professionals. If you’ve ever used graphic drawing programs with a colour wheel, the 360 degrees of the colour wheel is the Hue, in a range 0 – 359. The angle 0 is red, 120 is green, and 240 is blue. The saturation and lightness are expressed in percentages, and determine the shade + brightness of the colour. Examples:
    - hsl(120, 100%, 50%) : green
    - hsl(120, 100%, 75%) : light green
    - hsl(120, 100%, 25%) : dark green
    - hsla(120, 100%, 75%, 0.5) : half-transparent light green (alpha version of hsl)

Responsive CSS

Our styling allows us to decide how our website / WebApp looks. A big part of styling is on the sizes, padding, margins and positioning of elements, and these are all dependent on the screen size that the user is viewing our website / WebApp in.

In order for us to write specific CSS rules for specific device dimensions, we use @media tags. The following is an example. In this example, we are specifying an @media tag with rules that the user is on a screen (e.g. this is not being printed), and the rule applies as long as the screen’s logical pixel width is less than or equal 600px. In which case, all rules within the { } will take effect.

```
@media screen and (max-width: 600px) {
    h1 {
        text-align: right;
    }
}
```

Usually we would use @media rules to reduce the margin + padding, to reduce the wasted white space, on smaller devices. These are also often different depending if the user’s small device (e.g. a mobile phone) is in portrait or tablet mode (e.g. if it’s the width that is small, or the height that is small).

One important tidbit of information is that for us to have control of our WebApp on a mobile device, we need to prevent the width/height being automatically adjusted by a mobile web browser. This happens both automatically, and when the user pinches-and-zooms on our website / WebApp. To prevent this, we use a special meta-tag inside our HTML’s HEAD section:

```
<meta name="viewport" content="width=device-width, user-scalable=no,
initial-scale=1.0, maximum-scale=1.0" />
```

This line sets some properties for the browser, which ensure that the width we use is equal to the device-width, so that there's no weird conversions going on. At the same time, we usually like to prevent user scaling (pinch-and-zoom finger action), and ensure both the initial and maximum scale is 1.0 (e.g. that it can't change). This basically covers all potential options for the user trying to change the size of our window to be different than the physical size of the window.

Selectors

Selectors are, at first, one of the more difficult concepts to understand. In any style description, we need to **select** what elements the style will apply to. In CSS, these are called **selectors**, and are part of the CSS specification. There are many ways we can use & combine selectors for our styling.

The simplest form of selector is the Type Selector, which is simply the type of element we want the style applied to. This is the one we've been using so far, with element types H1, P, BODY, etc.

```
h1 {
    color: darkgreen;
    text-align: center;
    border: 6px solid red;
}
```

The next type of selector is the Grouping Selector. It allows us to apply a style to multiple elements in a single description, by separating each one with a comma. Consider this example:

```
h1, h2, h3 {
    color: darkgreen;
}

h1 {
    font-size: 24px;
}

h2 {
    font-size: 20px;
}

h3 {
    font-size: 16px;
}
```

In the example above, all heading types in use (H1, H2 and H3) will all be dark green text colour. But each heading type will have a different font size. This saves us effort as we don't need to specify the text colour for every heading type descriptor.

The next type of selector we'll introduce is the Descendant Selector. This allows us to target elements within elements. Consider this HTML:

```
<h1>Example of Descendant Selector</h1>
<p>This is a normal paragraph.</p>
<table>
	<tr>
	<td>First column</td>
	<td><p>Paragraph in 2nd column.</p></td>
	</tr>
</table>
```

Now consider the following CSS description:

```
table p {
	font-size: 8px;
	padding: 16px;
}
```

The above CSS description specifies that P tags (paragraphs) **within** tables, get a smaller font size and more padding. But all other P tags won't be affected.

The final most common selector is the Class Selector. We haven't yet introduced CSS classes (which are different to programming "classes"). A class is simply a programmer-defined style type. That means, it is one that you invent. Let's say you want to have a special paragraph type that is used for quotations. You specify which paragraph tags in HTML that you want to be in your new style class:

```
<p class="quotations">This paragraph is a quotation.</p>
<p>This is a normal paragraph.</p>
```

And then you define the CSS definition of a class, using a period/decimal point before the name:

```
.quotations {
	margin-left: 80px;
	padding-left: 8px;
	border-left: 1px solid grey;
}
```

And that's it! Your new CSS class is defined with ".quotations" in CSS, and then can be used on any tag by adding ' class="quotations" ' to the tag.

Note that the "quotations" class we've created doesn't necessarily only apply to P tags. We can apply it to any tag. If we want the style to only apply to P tags with class "quotations", we use this syntax:

```
p.quotations {
	margin-left: 80px;
	padding-left: 8px;
	border-left: 1px solid grey;
```

```
}
```

The above syntax says that **only** P tags with the "quotations" class name will have this style applied.

There are many types of selectors for a variety of special situations. I'll quickly show you one called the nth-of-type. This allows you to declare that a style occurs on alternating instances of an element. Let's say we have a list of items, and we want odd numbered rows to be grey, even-numbered rows to be light grey:

```
li {
      background: slategrey;
}

/* select alternating items starting with the second item */

li:nth-of-type(even) {
      background: lightslategrey;
}
```

We use the syntax ":nth-of-type(even)" to any element (or any class) to select the even-numbered repeats of an element. We can also use ":nth-of-type(odd)" for odd numbers, or ":nth-of-type(4n)" for every 4th element, ":nth-of-type(5n)" for every 5th element (not the movie), etc. etc.

As a final piece of information in the above example: comments! Notice the text inside **/*** and ***/** characters? This is how the CSS language defines programmer comments. They are ignored by the CSS language, and are just for our (human) benefit, to leave little programmer notes to remind us what on Earth our complex bit of CSS is doing (in case we forget – which will always happen).

Fonts

Ahhh fonts. One of the most overlooked aspects of design. Fonts are the style of characters within an alphabet. They are a collection of pre-made drawings (images) of every character within what is called a "character set". First, let's cover some terminology:

Character Set	A character set is the list of characters available. Different languages have different characters, or "letters" of the language's alphabet. For example, the English character set has: "A B C D E … Z" as well as "a b c d e … z". Lowercase and uppercase letters are different characters – they are drawn differently, so each has its own character. We also have numbers 0-9 as characters, and several punctuation markers, such as ", . ( ) $ @ !". Other languages have different characters. Japanese has hiragana, katakana, and kanji. Chinese has hanzi characters (of which there are several thousand). Even French and Spanish have different character sets, with their accents such as á. A

	side note: most accent characters are mixed in with English in what is commonly called the "Latin" character set. So if you see "Latin", you know it means English + some other European characters.
Character Encoding	A character encoding is the mapping of characters of a language (e.g. the "letters" of a language's alphabet) to numbers. In computers, everything is stored as a number. When we see things as "text", what is happening is that the computer is looking up that number in a font table and then drawing the character equivalent to that number. The original character encoding that was used throughout the western world is called ASCII. It has 255 characters it maps, with the letter/character "A" mapped to number 65 (and thus "Z" is at 90), and the letter "a" at 95 (and "z" at 122). A space is mapped to character 32. These days, the most well-known character encoding is called Unicode. It is a worldwide effort to map every single in-use known character in all languages across the world. The current version has 136,755 characters representing 139 languages.
Font	A font is simply a collection of images (called **glyphs**), one per character for a language. Some fonts will skip some characters, that's why you'll often see a little empty square box □ inside Microsoft Word documents or on web pages. That means a character is trying to be displayed, but there's no drawing for it within that font. Note that there are some fonts that aren't characters, but images. This is like Microsoft Word Wingdings font. On the web, a popular image font is the Google Material Icons font, which contains 900 icons (900 glyphs of symbols).
Font file	A file containing the font. The most common formats you'll see for font files are: 1. WOFF: used by web browsers 2. TFF: used by Microsoft Word (and operating systems) 3. OTF: An open font format Note that a font may have variants (such as italic, bold), but these are contained in different font files. Each font file generally only has one glyph (drawing) per character.
Font Family	This is what the CSS language calls the name of a font.
Font Face	The font face is the specification used within the CSS language to link a font family name to a font file.

How to Download Fonts

To obtain fonts unfortunately takes a few steps, sometimes more than usual depending on the font format and details.

And here comes our first extremely important point: We can **never** link to external URLs in web apps. This is because the user may not have their network connection active (e.g. if we've compiled our web app to a desktop or mobile application). So, we must download anything we need and include it into our directory, usually into a directory called "fonts".

Steps:

1	Find a font you like. I'm going to use the site FontSquirrel.com, and browse for the popular Montserrat.
2	Next, we'll check the license type (the SIL Open Font License is fine). Now we download that font using the links on the page. It will download a zip file with all variations of that font (bold, italics, etc), each one in its own file. Now this font format is not the one we need for HTML pages, which need to be WOFF file format.
3	So now we need to use the Webfont Generator link on the FontSquirrel page to convert the OTF font format to a WOFF or TTF font format. We need WOFF or TTF format for our fonts for the most cross-platform browser compatibility. We upload a single font file we want (e.g. "Montserrat-Black.otf"), then on the Webfont Generator page, we tick the box saying we can legally use it (which we can as it's licensed as such), and then we can download the kit.
4	Inside the zip file of the kit we download, will be several files. The only one we need is the WOFF file: montserrat-black-webfont.woff – this is the font file we will use We will copy this file into a "fonts" directory that is within our project directory. We can then delete the downloaded zip and the extracted directory, we no longer need those.

5	Now we edit our CSS code to be able to use this font, by creating a font-face declaration. What we care about is creating a font-family name, and then linking to the file (which we will put inside our fonts directory.
	
	`@font-face {`
	`  font-family: 'montserrat';`
	`  src: url('fonts/montserrat-black-webfont.woff')`
	`format('woff');`
	`}`

6	Now anytime we want to use that font (e.g. for the entire BODY of our HTML document), we use the following CSS syntax:
	
	`body {`
	`  font-family: 'montserrat';`
	`}`

As you can see, working with fonts can be quite troublesome.

Week 2 Exercises

Exercises:

A

Create two files in this week's directory (folder): week2a.html, week2a.css

1. Create a link in your week2.html HEAD section, to link to the week2.css file.
2. Set up week2.html's content to display a heading "My Font Test", followed by a paragraph of text "Hello there. This is my paragraph in a new font."

3. Then in order to test that it worked, we'll need to put some content into week2.css. Use the following:

```
h1 {
      border-bottom: 2px solid rgb(100,200,100);
}
```

4. Modify the H1 CSS rule to increase the letter spacing between letters of the heading.
5. Modify the H1 CSS rule to add a dashed grey line (border) above the text.
6. Create a rule so that paragraph text has a line-height of 20px
7. Modify the paragraph rule so that the width of each paragraph is 100px, and has 20% left margin.
8. Download the font "Montserrat" from FontSquirrel.com, following the steps listed in the Theory section previously. Link it into your week2 CSS file and refresh your week2 HTML file. Your text in the whole document should now render in the font.
9. Using a colour theory calculator (e.g. Paletton's), find 3 Adjacent colours where the primary colour is anywhere within the blue range.
    a. Set the foreground colour of the H1 style to the primary colour.
    b. Set the foreground colour of the P style to the 2nd colour.
    c. Set the border colour of the H1 style to the 3rd colour.

As with all exercises, when you're done, take a screenshot of the final page, and also take a screenshot of your CSS file and put both into your Portfolio document.

B

Create two new files, week2b.html and week2b.css.

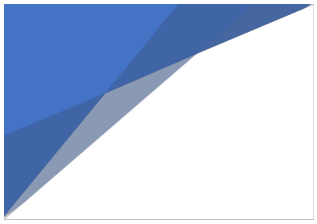
In these files I want you to create an example of a responsive web page, using @media tags.

I want you to demonstrate:

1. That your paragraphs have a margin at the bottom normally, but not on small landscape devices.
2. That your headings are centred normally, but are left-aligned on small landscape devices.
3. That your headings have a top+bottom border normally, with 8px padding and 8px margin, but only a bottom border on small landscape devices, and

	with 2px top+bottom padding & margin, but still 8px left+right padding & margins. You'll need to take two screenshots to put in your portfolio this time, one in a phone landscape mode, and one normally.
C	Now also complete these quick CSS selector exercises: http://toolness.github.io/css-selector-game/ Make notes about what you have learned about CSS classes inside your Portfolio (and any other lessons you feel will be useful to you for your Project Plan or Project)
D	OPTIONAL: Complete the following quick CSS selector exercises: https://flukeout.github.io/ (this exercise is not required for your Portfolio)
E	Create two new files, week2c.html and week2c.css 1. Create 3 paragraph tags, and fill with random text. Style them to each have a blue rounded border. Also give it a left and right margin of 80px. 2. Next create a 4th paragraph, and give it a class name "footer". 3. Add CSS code to make the footer paragraph style have a thick top border, but no other borders, and also give it an absolute position 20px from the bottom of the screen. You'll need to research absolute positioning to make this work, and to write about what you learn in your Portfolio. It will be essential for your project. Screenshots into your Portfolio, as always.
F	Now we'll continue with our Project Plan, this time adding styling. Complete the following tasks: 1. Browse FontSquirrel or Google Fonts and find a suitable font for your Project Plan. Download the woff file into a directory called "fonts" inside your Project Plan directory. Modify your Project Plan's CSS file so that your document now displays in the font you have selected. 2. Add a table styling such that your table rows have alternating colours between each row. Also make table header cells have a darker background colour and a different font colour. Remember to use a Colour Theory to get similar shades of the background colour you wish to use, so that they logically look good. 3. Modify your paragraph text to be justified, and have spacing after each paragraph of at least 8px. 4. Modify your H1, H2 and H3 styles to have a professional feel to them.

That's it for this week's exercises. Now finish your Project Plan.



Week 3

This week is the beginning of many lessons on programming, specifically JavaScript programming. Programming is our way of writing English-like instructions for the computer to follow (execute) whenever certain events occur. For example, when a Logout button is clicked, what do we want the computer to do? Probably hide or clear the current page, and show the login page. These are instructions that should happen every single time the Logout button is pressed, and we define these instructions using programming.

There are thousands of programming languages, though only about 20 that are really popular. Most programming languages have quite a lot of similarities in their ***grammar***, though their API (their vocabulary) varies greatly.

In the world of HTML, JavaScript is the default, and most widely-used, language. JavaScript is an interpreted, client-side, event-driven, object-oriented scripting language that you can use to add dynamic interactivity to your web pages. All those words are a mouthful, and it's not something you need to understand at this point. JavaScript scripts are written in plain text, like HTML, XML, Java, PHP and just about any other modern computer code. JavaScript is not Java, though if you come from a Java background, you will notice that both languages look similar when written.

Week 3 Theory – "Doing Stuff"

Computers can do two things: they can process (follow) a sequence of instructions, and they can store information. The sequence of instructions is called "program code", and the information is called "data".

When we "program", we are writing instructions (a recipe or algorithm) for the computer to follow at a later point in time (when the program runs). More specifically, we write program code to run at certain points of program execution, such as when a button is pressed. We do this because we don't want our application to simply "do everything in 1 second then end", rather we want it to be "alive", which usually involves:

1. setting up the application (when it loads)
2. running further code whenever certain events occur (such as a button press)

Every piece of program code we write falls into one of these two categories. And in fact, both categories are called "events", because the process of "loading" is an event too.

JavaScript is a language for telling the computer to "do stuff". We always get the computer to "do stuff" on "events". That means that program code always executes when something happens which triggers the events. In technical terms, these are called "event handlers".

Whenever we write "code", we are telling the computer "when/each time this event occurs, run this sequence of instructions I have pre-programmed into you."

The two most common events are:

- window.onload – this occurs when our visible window (essentially our entire HTML file) has completed loading. You can think of this as "all our scripts and CSS files have loaded, we are ready to start". Note that there is only one "window", so you always write "window.onload".
- button.onclick – this occurs whenever the user clicks (or taps, for touch devices) a button. Note that the word "button" here will change, depending on **which** button you are referring to. The way we **refer to** buttons (or any HTML element/node) is with variables. We learn about these next week. Just keep that in mind.

And that's it. There are many others (for dragging, for setting timers, etc), but 90% of what we use are the above two.

As well as run program code, the computer needs to "remember things". We have two ways of remembering things: temporary (while the application is running), and persistent (is remembered even if the application is closed down and reloaded). These are stored in physically different hardware within a computer system (or mobile phone):

1. temporary information is stored in RAM (Random Access Memory), special computer chips which only store information, but allow access to that information at extremely fast speeds).
2. Persistent information is stored on either hard drives of solid-state storage (SSD drives / chips). This information is retained even if the power is turned off on your computer / phone. It is permanent (until deleted).

In this unit, we won't be working with **saving** any persistent information. But we will be working with loading persistent information, which we'll write in files (and save to disk) and load those files when the application loads. Interestingly, "program code", such as JavaScript, is actually also data, and is also loaded from the disk when the application loads.

As well as loading persistent information, we will be storing temporary information in RAM, and each "piece" of information is called a **variable**. A **variable** gets its name because the information contained inside it is usually… variable, it varies depending on what's going on in our application.

A variable requires you to know 4 things:

1. What are you going to call it – this is called the "variable name". Imagine you're writing down friends' phone numbers on a piece of paper. If you just write the numbers, and nothing else, you have no way of recalling which number belongs to which person. So you need some **identifying** piece of information for each piece of data you store. Note that in almost all programming languages, the name cannot have spaces. So to make variable names readable, we capitalize the beginning of each word except for the first word (which always starts with a lower-case letter). For example currentLevel, currentPlayerName, etc.

2. What **type** of information are we going to store in the variable. Knowing what **kind** of information is vital. For example, if we want to add a player's age to their difficulty level, we need both the player's age and difficulty level to be Numbers, as other data types aren't capable of addition. Note that in JavaScript, unlike other languages, you don't actually have to say what the data type is to the computer. But **you** need to know and remember what data type it will contain.
3. Where is the information needed. This is another key piece of information that **you** need to decide. Generally there are only two choices: is the variable used only within a single function and then discarded, or is it used multiple times or in multiple functions. If it's used within a function, we declare it (tell the computer about it) within the function. If not, we declare it to a special place called **window**, which in JavaScript represents your entire running application (a "global variable", as it is commonly known).
4. What is this for. It is really essential that you know exactly what purpose you are creating and using a variable for, so that you don't get it confused with other variables, and so that you don't get confused with how to use it. While some variables may be obvious, such as playerName, others aren't clear, such as currentLevel. Is currentLevel going to contain a number referring to the current level the user is on, or is it going to actually contain the entire level data itself (e.g. all the questions of a quiz). This is really important that you write down and clarify somewhere (but again, the computer itself doesn't need this information).

The computer itself only needs to know the variable name, though you also need to say specifically where the variable will be "declared" (created), and thus you do need to immediately know if it is going to be a global variable, or a function variable/parameter (just to confuse you - functions have two types of variables, but we'll cover that much later).

Phew. That was a lot of explaining. I bet not much of it makes sense yet. That's fine, you haven't started programming yet. But go back and remember the keywords I used, and what they mean, so that when you come across them then you can start slowly making sense of things.

Where to Write Program Code

Let's take a step back for a moment. We want to write JavaScript code, to tell the computer (1) what to do, (2) when an event occurs. But… where do we actually write our code instructions for the computer to read?

What we do is we write JavaScript code, either directly inside the HTML document, or in a separate file with the file extension '.js', and then link to that file from the HTML document. These two approaches I'll refer to as "Internal JavaScript" and "External JavaScript". We'll always use External JavaScript in this unit, but I'll show you both approaches anyway.

Internal JavaScript

Defining JavaScript inside the HTML document requires use of <script> tag and all the programming code we will simply write within these <script> tags. Almost always these tags will go in the <HEAD> section.

These script tags take the following form:

```
<!-- this is HTML -->
<head>
	<script type="text/javascript">

		// this is JavaScript
		alert("popup message");

	</script>
</head>
```

The script element above can be placed virtually anywhere you could place any element in an HTML page – in other words, in either the head element or the body element. However, it is almost always written in the HEAD section. There is no arbitrary limit on the number of script elements that can be contained in an HTML page.

External JavaScript File

We can create a new blank file, give it a name with the extension '.js', and we have an external JavaScript file. We can use (and re-use) this file in as many HTML pages as we want. This provides a nice way to write code once and re-use it across projects.

Example
Here is an external JavaScript file app.js:

```
alert("popup message");
```

Super simple. We simply write the code directly. There are no special tags or anything. As long as the file ends in ".js", which stands for JavaScript, and as long as we link it from the HTML (so that it loads), then it works.

To link a JavaScript file from a HTML file, we use the following code:

```
<script type="text/javascript" src="app.js"></script>
```

We need the whole line, but the only important part is the part in bold, where we say the source location of the file we want to load. Note in the above syntax, the filename is defined

with the “src” attribute, but with CSS files, we use the “href” attribute. This is one of the weird quirks about HTML, and is simply something you need to get used to.

JavaScript Syntax / Grammar

The syntax, or grammar, of any language is the way in which words (commands) should be ordered, separated, and punctuated. This set of rules is usually quite simple. Think about HTML and CSS previously.

The grammar of the language HTML is simple:

- Tags go inside < > brackets.
- Pairs of tags must have both an open and a close.
- Attributes for a tag are inside the open tag, and are separated by a space.

The grammar of the CSS language is also simple:

- A selector always comes first.
- Multiple selectors can be separated by a comma
- The rules for a selector go inside { } braces.
- Each rule has a property name, followed by a colon “:” , followed by a value, then ending in a semi-colon.

The JavaScript language has a few more rules than HTML and CSS, but the most common ones are:

- Each single instruction ends with a semi-colon “;”
- A group of instructions are put inside { } braces.
- We can save a group of instructions by using the keyword **function**, followed by (), then wrapping the code inside { } braces.
- We can execute a named group of instructions (a function) by writing its name, followed by () parentheses.
- We can save information by creating a memory space for it, using the keyword **var**, and then giving the memory space a name (any single word, without spaces).
- We can save information into a memory space using the “=” equals sign.

Let’s look at these one by one.

Calling Functions, AKA “Doing Stuff”

In JavaScript, most of what we want to do is to execute commands. Commands in JavaScript are called **functions**, and they are bundles of more complex code that instruct the computer how to do things. When we want to execute a **function**, it is called “calling” the function. To “call” means to execute, in JavaScript terms.

The following is a line of code. It is a single instruction, and thus it ends with a semi-colon. We know it is calling a command, because there is a word (which is the name of an inbuilt function), followed by parentheses (brackets). What this **line of code** does is to tell the computer to **call** (to execute) the **function** called alert, which will be a subset of instructions that draw a box on the screen, draw a button, and add code so that when the button is pressed, the alert box closes.

```
alert();
```

Now when we call functions, we can pass in additional information, which are called parameters. Parameters are separated by commas if we need more than one. For example, the alert function allows for an additional piece of information, a text message. We always put text to display inside quotation marks. They can be double quotation marks or single quotation marks, either is fine. BUT, they **cannot** be “curly” quotation marks, which is what Microsoft Word generates when you write quotes in Word documents.

```
alert("popup message");
```

Some functions accept more than one parameter. The way we find out what parameters are accepted is we look at the **documentation** for a function, and we look for something called a **method signature**. A method signature is the description of how to call the function.

One of the best places to get documentation on JavaScript is the Mozilla developer network. I usually use Google to find the page within the Mozilla Developer Network, e.g. if I want to find out about the **prompt** function, I would google “Mozilla developer network prompt”, or “site:developer.mozilla.org prompt”, which will then find the specific page I’m after:

- https://developer.mozilla.org/en-US/docs/Web/API/Window/prompt

Mozilla (who make the Firefox web browser) have been kind enough to document all of the inbuilt JavaScript functions, which is incredibly helpful to everyone who make web apps and web sites.

In the page I just looked up, I will see the following syntax documentation, giving an example of the function signature. It shows that the function **prompt** has two parameters, a message, and a default. It also **returns** a value, that means that when it finishes running, there is an answer that we get back, that we can use (if we want to).

Syntax

```
result = window.prompt(message, default);
```

- `result` is a string containing the text entered by the user, or null.
- `message` is a string of text to display to the user. This parameter is optional and can be omitted if there is nothing to show in the prompt window.
- `default` is a string containing the default value displayed in the text input field. It is an optional parameter. Note that in Internet Explorer 7 and 8, if you do not provide this parameter, the string "undefined" is the default value.

Now one small note here. Instead of just calling prompt(), this example shows window.prompt(). “window” is a namespace where many functions are collected. Normally when functions are collected into a namespace, or “object”, we need to prefix the name of that namespace or object first. However, there’s one single exception to this rule: everything under the “window” collection is available without needing the “window” keyword. It’s what we call “globally” available. The explanation for that is a little beyond this week’s level of understanding, so for now just know that “window” is special, and move on.

So back to prompt. To call it, we can provide two parameters, or we provide just one, or even none, as both “message” and “default” are both optional (according to the documentation), which means we don’t have to put anything there.

In other words, all 3 of these **function calls** are valid and will work:

```
prompt();
prompt("Are you ok?");
prompt("Are you ok?", "yes");
```

Creating Functions

So far, we have called two functions: **alert** and **prompt**. These are inbuilt into the JavaScript language. What is amazing about programming is that we can make our own functions. What we mean by “make our own” means to define a sequence of instructions, to be called at any later point in time.

Here is an example:

```
window.promptIfOk = function() {
      prompt("Are you ok?", "yes");
}
```

What these 3 lines tell the computer, is:

1. Please save a function (a series of instructions), and store it in the window namespace under the name “promptIfOk” (names can be any single word).
2. The function contains instructions within { } braces.
3. There’s a single instruction, which calls another function **prompt**, with two parameters.

That’s it. Now if we want to run that function, our entire JS file might look like this:

```
// make the declaration (as in, save the set of instructions)
window.promptIfOk = function() {
	prompt("Are you ok?", "yes");
}

// now call the function
promptIfOk();

// let's call it again
promptIfOk();
```

Now there are actually two ways to write functions. One is to save them into an object (in the previous case, into “window”, so that it can be accessed throughout our application in case we have multiple JavaScript files). The other is to declare it locally, which limits its scope (where it is accessible). That concept is a bit advanced for now, but I’ll demonstrate the syntax as it’s very common. That second approach is used like this:

```
function promptIfOk() {
	prompt("Are you ok?", "yes");
}
```

In the above approach, we give the function keyword first, and then we create a name for our function. This is different from the syntax we used previously, where we specified the variable name first, and then assigned the function (using the equals sign “=”).

Now it’s time to two important points you always need to remember when you write program code:

1. Indentation. When something (e.g. instructions) are inside a grouping (what we call scope in JavaScript terminology), we intent (press the TAB key), so that visually we can see what is supposed to be inside and what is outside.
2. Comments! We always need to leave notes throughout our program code so that we, and other future developers, can understand what our code does. There are two ways to write comments in JavaScript:
    a. // A single line comment starts with two slashes
    b. /* A multi-line comment is the same syntax as CSS uses */

Now, back to calling our functions. What we’ve done so far is call our function **promptIfOk** (twice). Note how I capitalise the “I” in “If” and the “O” in “Ok”? Because we aren’t allowed to use spaces in a name, we need some way to visibly make it easier for us to read words, but without spaces. Otherwise our function would look like “promptifok”, and it would be very

hard to read and decipher the meaning. The approach now recognised as the "standard" for doing so is called camelHumps. Just like a camel, we create a "hump" at the beginning of each word, by capitalising that letter. You will see it used whenever we name a **function** or **variable**.

Event Handlers

JavaScript is what is called an event-driven language. Remember at the beginning of this week when we said "JavaScript is an interpreted, client-side, event-driven, object-oriented scripting language"? Well, this is the "event-driven" part. **Everything in JavaScript only happens when an event occurs**. Everything. Nothing will happen without an event. If you load a web page and no events occur, nothing about the web page changes. If the web page registers for events such as new network information received (like a new email), or when a button is pressed, or when a timeout occurs, then things will happen. These "things will happen" are all functions that only execute because an event has occurred. Even in the code we had above, the event it occurs on (although not explicitly stated), is the "load" event, when the script has been loaded.

When we want to assign a function to be called when an event occurs, this is called "assigning an event handler". We are going to assign some function (a sequence of instructions), to be called (executed), when an event occurs. The function will **handle** what happens when that **event** occurs.

Our previous promptIfOk function will be our event handler, it will handle our event. But, which event? This is where we need to assign the function to happen on an event. There are two ways of doing this: through JavaScript code, and through HTML.

For now, I'm just going to show you how to do it with HTML. Each tag can have several attributes. Some of them are event attributes. One event attribute is called **onclick**. If we assign code to that attribute, it will be called "on a click event", i.e. the function will run every single time it is clicked. Here is the example in HTML:

```
<html>
      <head>
            <script type="text/javascript" src="app.js"></script>
      </head>
       <body>
              <h1 onclick="promptIfOk();">Click here</h1>
       </body>
</html>
```

And our app.js file (we'll remove the calling of the two functions, and just declare the function):

```
// make the declaration (as in, save the set of instructions)
window.promptIfOk = function() {
```

```
        prompt("Are you ok?", "yes");
}
```

And that's a complete, working example.

There are many events we can assign things to, but the most common is click, and it works in the browser, on desktop and on mobile devices (whereas mouse moving functions only work on desktops and laptops, not on mobile devices, as there is no mouse).

The next most common is when the document has finished loading (our WebApp). We can do this in two ways.

With HTML:

```
<body onload="ourFunctionName()">
```

Or with JavaScript:

```
window.onload = ourFunctionName; // note, no () here
```

Note that sometimes, instead of writing and naming a separate function, we can create an anonymous function. An anonymous function's code runs identically to a named function. The only difference is that we never give it a name, we instead create it and immediately assign it into a property.

```
window.onload = function() {
      console.log("Application has started");
};
```

As a bonus, there are also timer events, such that we can set a timer to come back after a few seconds and then run code, or we can have code repeat at certain intervals. I'll show you them now:

```
// run promptIfOk after 5000 milliseconds (5 seconds)
setTimeout(promptIfOk, 5000);
```

We can also set an interval, **but be careful with this one**. If we do it with promptIfOk, then the prompt will continue to keep popping up, and you won't be able to close your browser window. I'll just show you the outline of it, to be safe (don't copy and paste):

```
// run a function (repeatedly) every 5 seconds
setInterval( functionNameHere, 5000);
```

I should also say that we can cancel a timeout or an interval timer, but we need to save a reference to the ID of that timer (as we are allowed to have multiple timers running simultaneously). Both **setTimeout** and **setInterval** both **return** a value, which is the ID of the timer that is created by the function call. We can save that to a variable, and pass the value

of it later when we want to clear the timer (using clearTimeout() and clearInterval(), respectively).

```
// start a continuous timer (every 2 seconds)
var savedIntervalTimerID = setInterval( functionNameHere, 2000);

// immediately clear the interval timer (so it'll never run)
clearInterval(savedIntervalTimerID);
```

Note, here we are introducing variables. That's a topic for next week, so for now, we'll conclude our talk of functions.

Logging

One of the essential techniques we use for "debugging" our code (as in, making sure it works, and more importantly, finding and removing any "bugs"), is to log each major step that our program is executing, to work out where it gets up to (before any errors) and which statements it runs. The most common way to do this is to log text messages to the "console", which is a special logging area that you can access in web browsers (though not on mobile devices, just FYI). You can view it in Google Chrome -> Inspector

```
console.log("This is my log message.")
```

You'll use this statement **all** the time in programming, no matter how experienced you become. It is possibly the most used programming statement in JavaScript of all time.

Every single time you write a function, you should write a message to console saying that function has started, and finished. That way you know at least it is running, which is essential.

So in our previous code, we wrote a promptIfOk function. Let's see it with console.log's added.

```
// make the declaration (as in, save the set of instructions)
window.promptIfOk = function() {
      console.log("promptIfOk(): Started");
      prompt("Are you ok?", "yes");
      console.log("promptIfOk(): Ended");
}
```

Logging is HUGE. ESPECIALLY when you don't know why something doesn't work. You absolutely (really, absolutely) must log EVERY STEP.

For example, let's say you wrote a function that was suppose to remove a character from a String of text (e.g. you're making a hang-man style game, where each time they guess a correct letter, you remove it from your lettersRemaining string).

When you look at the console view, you want to see a list of messages. Each message should start with the function name you are currently inside. And you should also put in brackets, the values of any parameters you have. Then write out what is going on (in plain English), so that you can visually see what should be happening.

For example, assume you had a function with the function signature:

```
window.removeCharacter = function(mainString, searchString) {
      console.log("removeCharacter (" + mainString + ", " +
searchString + "): Started");


      // code would go here


      console.log("removeCharacter (" + mainString + ", " +
searchString + "): Ending");

      // 'return' ALWAYS comes last
      return finalString;
}
```

Now if we had the fully written function code, I would want to see the following displayed on console, so I could tell what was going on at each step of the algorithm:

```
removeCharacter("bus stop", "o"): Started

removeCharacter("bus stop", "o"): The parameter mainString is: "bus
stop", the search string is: "o"

removeCharacter("bus stop ", "o"): The character is found at
position: 6

removeCharacter("bus stop ", "o"): Before found character string is
"bus st"

removeCharacter("bus stop ", "o"): After found character string is
"p"

removeCharacter("bus stop ", "o"): Final, joined string is "bus stp"

removeCharacter("bus stop", "o"): Ending
```

Note, the above is an example of what I'd like to see from an exercise in Week 7. I don't expect you to fully understand how to write a removeCharacter function yet. I just want to emphasize the process of always logging messages to the console. Regular users won't ever see these, they are only when you have the Inspector view open in a web browser, to look at your web app "behind the scenes". But you have to log any message you want displayed, otherwise nothing will appear (except for errors, called "exceptions" which we cover later).

I want you to always be using console.log(), so we can see what your code is doing behind-the-scenes, throughout each of the functions. You can comment them out anytime you don't need them (e.g. once a function is working):

```
// console.log("removeCharacter (" + mainString + ", " + searchString
+ "): Ending");
```

The two slashes at the beginning of a line "comment out" that line, which means it is no longer interpreted by your computer, it is ignored. "Comments" are only for humans to read, so a computer ignores them completely. So when we have code that we want the computer to "ignore", we "comment it out" by putting comment marks before the line.

A big hurrah for getting this far!

Week 3 Exercises

Exercises:

A	Create files week3a.html, a JavaScript sub-directory called 'js', and a file js/week3a.js Remember to test your program code after every step, to ensure you haven't introduced any programming bugs, and to narrow down your focus of what could need fixing in case something goes wrong. 1. Have your JavaScript file run a startup function called startup when the document has loaded. 2. The startup function should output a message to console "startup code running". 3. The startup function should then call another function createHeading1. This function will take one parameter, "titleText", and place that text inside the H1 that the function creates. The function should then append the created H1 to the document body. 4. Create functions to create H2, H3, and P tags, and test them in your startup function. Take a screenshot of your JavaScript file, as well as your browser output (with console view in Developer Tools), and put both screenshots in your Portfolio.
B	Create two new files, week3b.html , and js/week3b.js 1. Copy an existing image you've used in previous weeks, into a directory: week3/images. 2. Link the JavaScript file, week3b.js, with your HTML HEAD section. 3. Create a loadApp function that runs when your document is loaded. 4. Inside the loadApp function, I want you to call a function: createImage("images/imageName.png") 5. Next, create the function previously mentioned, createImage(imageSource). You'll need to investigate how to add attributes to a HTML element using JavaScript. 6. Add an anonymous function to handle the onclick event of your created image. Use something similar to this code:

```
imageElement.onclick = function() {
      console.log("The user just clicked on the image.");
      imageElement.style.opacity = 0.5;
}
```

Take a screenshot of both your week3b.js file and your browser window, and place both inside your Portfolio.

Week 4

Week 4 Theory: Variables

Ok! In my very last example last week, I did something new. Take a look:

```
var savedIntervalTimerID = setInterval( functionNameHere, 2000);
```

I wanted to **remember** the timer ID that setInterval() **returned** to me. I can do that using 3 steps (which I did all on the same line of code):

1. Use the "var" keyword to tell JavaScript I want to save a **var**iable.
2. Give the memory space a name that I can use later. savedIntervalTimerID is the name I chose. Generally you want to have a meaningful name that makes sense to you.
3. **Assign** the result **to** the memory space. The "=" equals sign is used for all assignment in JavaScript. The same way that we **assigned** a function to the label window.promptIfOk before, we use the same equals sign here, assigning whatever setInterval() returns, into the savedIntervalTimerID memory space.

Whenever we assign data into a memory space, it overwrites anything already there. Let's look at an example:

```
var myName = "Henry";
myName = "Bob";
myName = "Charles Smith";

console.log(myName);
```

In the above code, I **declare** a variable (create a memory space) **just once**, with the **var** keyword. I give it a name "myName", and I assign into it the text "Henry". Then I end that first instruction. Next, I assign the text "Bob" into myName. It will replace the "Henry" that is already there. Next, I assign the text "Charles Smith", which overwrites the "Bob" that was previously in there.

Now, anytime I want to retrieve the value of that memory space, I simply write the name of it. So, if I want to retrieve the value, and give that value as a parameter to the console.log function, I use the syntax:

```
console.log(myName);
```

What this really does, is look-up the current (live) value of the memory space named myName, and retrieve that value. When the computer is running, this would occur temporarily:

```
console.log("Charles Smith");
```

The **value** inside myName will be retrieved, and used when that console.log function is called.

That is variables in a nutshell. There are two more important factors that you need to understand however: Scope and Variable Types.

Scope

In programming terminology, the scope is where the memory space is allocated. There are restrictions placed on variables, in that **where** you declare the variable is where you are allowed to use it. You cannot use variables outside of **where** you declare them (outside of the scope), unless you make that variable globally available. Let's look at an example. I'm going to log out 3 variables to the console: localMyName, globalMyName and myName.

```
window.displayName = function() {

	var localMyName = "Bob";

	console.log(localMyName);

	console.log(globalMyName);

	console.log(myName);

}

// put something into the global namespace:
window.globalMyName = "Henry";

// now call the function (make it run once)
displayName();
```

If you run this code, you'll get 3 console.log lines: (1) "Bob", (2) "Henry", and (3) undefined (or possibly your program will crash at that point).

1. The first instruction retrieves the data held in memory called localMyName. This has been declared inside the displayName function (inside the { } brackets), which means it is available anytime within that function. Every time the displayName function is called, that memory space is re-created new. And a side note: when the function ends, that memory space is automatically deleted, in a process called "garbage collection".
2. The second instruction retrieves data held in memory called globalMyName. Do you recall earlier when I said everything in the "window" namespace is special, in that it can be accessed without specifying the "window" part? That applies to variables (memory spaces) as equally as functions. We assign something to window.globalMyName a few lines down. It doesn't matter that we do this **after** we declare the function, because **a declared function doesn't run**. A declared function only runs when it is **called** (executed), and we don't call the function until the very last line.
3. The final instruction tries to find a memory space named "myName". As there isn't any memory space (locally within the **function scope**, or globally within **window**) then this will fail.

Side note: When dealing with “window”, we don’t need to use “var”, because the “var” only takes place when we create the first word, e.g. the “window” word, and that word is already declared (it is inbuilt into JavaScript). To give you a super-quick example, let’s say we created our own scope called app, and assigned a variable name:

```
var app = {};
app.name = "My App Name";
```

In the above, once we have created the “app” reference with the keyword “var”, we don’t need to do it for any properties within that space. The {} used above isn’t for creating code, but for creating an Object data type. I’ll cover that in Variable Types next.

Operators

Within JavaScript, there are several symbols that perform instructions. These are slightly different than our usual way of running instructions using a function’s name. Operators exist as a kind of short-hand approach to perform extremely common inbuilt functions, which are almost always math-based operations. The most common operators are:

Symbol	Meaning	Examples

+	• Add two numbers, or
	• Append anything to a String (if one of the items is a String of text).
		`4 + 5`
		`"a" + "b"`
		`"a" + 4`

-	Subtract the second number from the first number	`4 - 5`
*****	Multiply two numbers	`4 * 5`
/	Divide two numbers	`20 / 5`

%	Modulus (division remainder)	`var remainder = 10 % 2;`
		`// remainder will be 0`
		
		`var remainder = 11 % 2;`
		`// remainder will be 1`

=	Assignment. Place any calculated result on the right side, to the variable name on the left side.	`var answer = 4 + 5;` `answer = answer + 1;`
++	Increment by 1	`var answer = 5;` `answer++;`
--	Decrement by 1	`var answer = 5;` `answer--;`

Example usage:

```
var a = 2;
var b = a * 10;
```

Variable Types

Variables are **memory spaces** within your application. Memory is essential in programming, as we need to ask the computer to temporarily remember information so that we know what state / stage our application is up to, what user information we know about, etc. Note that variables are only concerned with run-time memory, stored in the RAM of computer hardware. When your application restarts, or when a browser window refreshes, all that information is gone. This is different from permanent storage, which is a topic for another day.

Because different **types** of information take up different **amounts** of space in memory, it is necessary for JavaScript to know memory space it needs to store information. Some information, such as "yes" / "no" information, only takes a single electronic bit (not even a "byte", which is 8 whole bits, but a single tiny bit itself). Other information, like Strings, can grow to immense lengths (usually 65,565 bytes as an upper maximum, though this varies). JavaScript is what's known as a **dynamic typed language**, which means that it will dynamically (as the program runs), guess the types. In other languages, we need to specify exactly what type each variable is when we declare it. Not in JavaScript. But we do need to be aware of what types of data we put into variables, as when we access them later, if we assume a type that is incorrect, then our program code may not run as we expect it to.

There are 7 types of data that JavaScript is innately aware of, then a few other newer class types that you may learn about outside of this unit.

Type	Meaning	Examples
Null	A null value means an empty value void of any specific type	`null`
Boolean	A Boolean is a true or false (yes or no) value.	`true` `false`

Number	A number is any floating-point (decimal) number.	`4`
		`4.4`
		`-5.56789`

String	A String is any text, inside quotation marks.	`"This is a String"` `'This is another String'`

Object	An object is a data type that has key:value pairs of properties within it. In other words, it can define names of sub-items, and assign values of any type.	`{}`
		
		`{`
		`  name: "Henry",`
		`  score: 10`
		`}`

Array	An array is a list of other types. It can contain any other type, and it can mix & match those types.	`[]`
		
		`[1, 2, 3]`
		`["Henry", "Bob"]`

Function	Yes, function itself is a type of data, although it is special. We can assign a function to any variable in memory, though they don't save to disk or transfer over networks.	`function() { }`

Note, there is also a special reserved keyword “undefined”, which is used when a variable does not have any data in it yet.

JSON

JSON is a way to represent (write) data as a programmer. JavaScript actually understands JSON natively, and uses the same syntax for declaring all data types.

This format is both human-readable (to a programmer), and understandable to a computer interpreting JSON (which JavaScript interpreters can do).

In the previous sub-section on Variable Types, I included examples of each data type. Let’s look further at how to combine them.

Let’s say I want to store a list of words for a crossword puzzle. How would I provide that list to a computer (using JavaScript)? The JSON format is simple in that it uses characters on the keyboard (you don't need any special database software, you don't need a special application to write the data, you can write it as text directly into any file).

To give a simple example, the following is a data example, in JSON, of a list of words:

```
["cat", "dog", "lion", "mouse"]
```

Let’s say I wanted to store a list of questions and answers. Well I have a list (example above), and I have two pieces of information per list item, the question, and the answer. So I'll need some way to reference them (called "keys"). I'll call them "q" and "a".

```
[
  {q:"Capital of France", a:"Paris"},
  {q:"Colour of water", a:"blue"}
]
```

In the above example, I have a list of 2 items (a list is always inside square brackets []). I have more than one piece of information to store per list item, so I need to use curly braces {} to group together any & all information related to that item (called an “object”, and I have two of them, as I have 2 list item examples).

We can combine those rules as much as we want. There's no limit to how "deep" our constructs go.

```
[
  {q: "What is the capital of France",
   a: "Paris",
   options: ["Paris", "Berlin", "London"]
  },
  {
    q: "What is the colour of water",
    a: "Blue",
```

```
        options: ["Blue", "Red", "Orange"]
    }
]
```

In the above example, I added another list (using []) to one of my list items, to give a list of options. The list type is used in two areas.

Objects

Objects are more complex than most primitives (such as Boolean, Number and String). They can contain multiple values, and thus we have ways to both assign and retrieve those sub-values.

There are two ways we can put data **into** an object. The first is at declaration time, when we create the object:

```
var myObject = {
      name: "Henry",
      score: 10,
      finished: true
}
```

In this example, we have 3 values inside the object, saved using the keys “name”, “score” and “finished”. Each key:value pair is separated by a comma (not a semi-colon like instructions).

We can also assign values (or replace values) after the fact, using the dot notation:

```
var myObject = {};

myObject.name = "Charles";
myObject.score = 10;
myObject.finished = true;
```

In each of the 3 assignment instructions, we are first looking up the memory space “myObject”, then we are using the “.” character to say we want to access data inside that object. Then we give the name of the key to use inside that object (if it doesn’t yet exist, it is created).

There’s another syntax we can use which does the same thing:

```
var myObject = {};

myObject['name'] = "Charles";
myObject['score'] = 10;
myObject['finished'] = true;
```

This syntax uses [ ] brackets to access the same properties, although now we use a String (in quotations) to access the key. This syntax is more used with arrays, but can also be used with Objects equally.

Whenever we want to get a value out, we simply specify the name of the memory space, followed by a dot, followed by the property name (the **key** in JavaScript terminology):

```
alert( myObject.name );
```

To get even more advanced, there's nothing stopping us from putting any type of data inside an object. We can put objects inside of objects:

```
var myObject = {
      name: "Henry",
      address: {
            street: "221 Burwood Highway",
            suburb: "Burwood",
            state: "Victoria"
      }
}

alert( myObject.address.street );
```

We can create as **deep** an object as we want, and we simply use multiple dot character to go deeper.

Accessing an Object's field using another variable

This is a very common, and critical learning point.

Let's say we have some data:

```
window.gameLevels = {
      "Room A": {
            "name": "Room A",
            "description": "This is our starting room"

      },
      "Library": {
            "name": "Library",
            "description": "This is the library"
      }
};
```

And let's say we store the current location inside a variable:

```
window.currentRoom = "Room A";
```

If we want to access the description of the current room, the following **won't** work:

```
var currentRoomDescription = gameLevels.currentRoom.description;

console.log("Current room description: " + currentRoomDescription);
```

We can't type a variable name here, as any text you type is assumed to be inside the data. You don't want to search for "currentRoom". You want to search for what's currently

contained inside that variable, that you're calling currentRoom. So you want to extract the contents from currentRoom.

Whenever we work with objects or arrays (both in JavaScript and in almost all programming languages), we use a special syntax:

```
var currentLocation = gameLevels[currentRoom];
```

Then you can do either:

```
currentLocation.description
```

or

```
gameLevels[currentRoom].description
```

I prefer the first syntax, as it's easier to test (the second syntax has too many parts all combined, so if something goes wrong, you don't know which part of the expression it goes wrong at).

The square brackets is used to refer to a specific field inside the object/array, but it can be an expression. An expression in programming terminology means it gets processed first, before it gets used.

So you can type:

```
gameLevels["Room A"]
```

(in which case, the textual String "Room A" is immediately the result).

You can type:

```
gameLevels["Room" + " " + "A"]
```

(which appends the 3 strings together first, to get one result "Room A", which is then the result, and is used as the key for gameLevels)

Or you can use any variable, e.g.

```
gameLevels[currentRoom]
```

In this case, the variable currentRoom is accessed, the contents extracted (currently "Room A", but that could change), and then it applies to gameLevels.

Arrays

The next advanced topic is Arrays. In JavaScript, Arrays are actually Objects, so this can be both useful (in that accessing them is identical) and problematic (to work out when something is an Object vs when it is an Array). But ignore any downsides for now – Arrays are awesome!

Arrays let you store any number of data elements. We can further add elements using the **push** keyword.

```
var listOfStudents = ["Henry", "Bob"];

listOfStudents.push("Charles");
listOfStudents.push("Xavier");
```

When it comes to arrays, we access them with the [ ] notation, using what is called an **index** number. The index is the **position** in the list that we want to access. And the most important fact to remember about index positions in almost all programming languages: index positions start at 0. The first item in the list is [0]. The second is [1], the third is [2], the fourth is [3], etc.

To display the first list item (“Henry”):

```
alert( listOfStudents[0] );
```

To find out the length of the array:

```
alert( listOfStudents.length );
```

To delete in arrays, we have a special syntax:

```
listOfStudents.splice(index, 1);
```

The first parameter is the index position (e.g. to delete the first element: “splice(0, 1)”). The second parameter is how many elements to delete. We just want to delete one in this example, so we put “1” in the second parameter.

UI Components

Note: In my video I append to document directly in this function, which is <u>not</u> what I want you to do from now on.

When creating UI components, we want to make things as simple as possible to create code using functions we understand.

```
/*
      Creates a paragraph
*/
```

```
window.createParagraph = function() {
	console.log("Creating paragraph");

	// create a P tag, and refer to it locally as "paragraph"
	var element = document.createElement("P");

	return element;
}
```

We actually should also allow some basic parameters to be passed in. So let's modify the above function to make it more detailed:

```
/*
	Creates a paragraph, with the given text

	text: any text to include on the paragraph
*/
window.createParagraph = function(text) {
	console.log("Creating paragraph with text: " + text);

	// create a P tag, and refer to it locally as "paragraph"
	var element = document.createElement("P");

	// check if text is not undefined
	if (typeof(text) !== 'undefined') {
		// the variable [text] does exist, so use it
		var textNode = document.createTextNode(text);
		element.appendChild(textNode);
	}

	// return the element so the caller can use it
	return element;
}
```

Now, anytime we want to create a paragraph:

```
var questionElement = createParagraph("Question: blah");
var answerElement = createParagraph();

// remember to display the two paragraphs somewhere
document.body.appendChild(questionElement);
document.body.appendChild(answerElement);
```

Now what if we want to append the paragraphs to a DIV (a general container)? We need a create DIV function

```
window.createDiv = function() {
	console.log("Creating DIV");

	var element = document.createElement("DIV");

	// return the element so the caller can use it
	return element;
```

```
}
```

And then use it:

```
var questionElement = createParagraph("Question: blah");
var answerElement = createParagraph();
var container = createDiv();

// add two paragraphs to container
container.appendChild(questionElement);
container.appendChild(answerElement);

// display the container on the screen
document.body.appendChild(container);
```

The advantage of wrapping the elements in a container is that then we can move or style that container using a class name and CSS. For example, let's give the container a class name so we can style it with CSS:

```
var questionElement = createParagraph("Question: blah");
var answerElement = createParagraph();
var container = createDiv();
// add a CSS class (no "." before the name in Javascript)
container.classList.add("questionAnswerBox");

// add two paragraphs to container
container.appendChild(questionElement);
container.appendChild(answerElement);

// display the container on the screen
document.body.appendChild(container);
```

Then in CSS we can write a style to say, move the container to the top-right corner (the container contains both question and answer elements, so they will continue to be inside the container):

```
.questionAnswerBox {
      position: absolute;
      top: 10px;
      right: 10px;
      border: 1px solid black;
      border-radius: 16px;
      padding: 32px;

}
```

You know what, wouldn't it be cool to have a container append elements automatically?

Say I want to do this:

```
var questionElement = createParagraph("Question: blah");
var answerElement = createParagraph();
```

```
var container = createDiv(questionElement, answerElement);

// add a CSS class (no "." before the name in Javascript)
container.classList.add("questionAnswerBox");

// display the container on the screen
document.body.appendChild(container);
```

What code would I need to write for createDiv to work? Well I want parameters, yes. But I don't know how many parameters, as it will vary based on the programmer (in the above case, 2 parameters, but in other usages of calling createDiv, there may be **more or less** parameters).

Luckily JavaScript has something called **rest parameters**, which allows you to define any length of parameters:

```
/*
	Creates a div.
	Any parameters (HTMLElements or Strings) are appended.
*/
window.createDiv = function(...otherElements) {
	console.log("Creating DIV with other elements:");
	console.log(otherElements);

	var mainElement = document.createElement("DIV");

	// now iterate over otherElements (which is an array)

	for (var subElement of otherElements) {

		// check what type of subElement we have

		if (subElement instanceof HTMLElement)
			// add any HTMLElement
			mainElement.appendChild(subElement);

		else if (isString(subElement)) {
			// we have a String, so create a TextNode
			var textNode = document.createTextNode(subElement);
			mainElement.appendChild(textNode);

		} else
			console.log("ERROR: createDiv() failed with
subElement: " + subElement);

	}

	// return the mainElement so the caller can use it
	return mainElement;

}
```

Note the code here needs a special utility function called isString(), to determine if a given variable is a String or not. Here is the definition:

```
/*
      This is a utility function to tell if a given parameter
      is a String or not.

      Returns: true/false
*/
window.isString = function(s) {
    return typeof(s) === 'string' || s instanceof String;
}
```

Week 4 Exercises

Exercises:

A

Create 4 new files: week4a.html , js/week4a.js , js/playerSettings.js, js/gameData.js

1. Create a function loadApplication inside week4a.js, and call it when the document has loaded. It should call configureGameUI()
2. Create a function configureGameUI, which creates two HTML elements on the window, a Paragraph for the player's name, and a Paragraph for the player's location. Place both variables into window properties, so that you can refer to them in other functions:

```
window.playerNameElement = ...
window.playerLocationElement = ...
```

3. Inside playerSettings.js, declare an object:

```
window.playerSettings = {
      name: "Your name",
      height: 160,
      location: "Room A"
};
```

4. Inside gameData.js, declare an object:

```
window.gameLevels = {
      "Room A": {
            description: "This is the starting room.",
            connectsTo: "Library"
      },
      "Library": {
            description: "You are now in the library.",
```

```
                connectsTo: "Room A"
            }

        };
```

5. Inside loadApplication() function, call two functions: displayPlayerName, and displayCurrentLocation.
6. displayPlayerName should output to console the current player name.
7. displayCurrentLocation should output to console the current location.
8. Change displayCurrentLocation to not just get the current location, but to look up the location's description inside gameLevels.
9. Change both displayPlayerName and displayCurrentLocation to write text into the P tags you created previously, instead of outputting to console.

Take a screenshot of both your running browser window and your week4a.js code file contents, and add them to your Portfolio.

B

For this exercise, simply write your answer as text inside your Portfolio.

Write an explanation as to when it is useful to assign variables to "window." scope, and why that was necessary in the previous exercise.

Aim for about 200 words.

C

For this exercise, I want you to start building your very own ui library.
Create a new file js/ui.js , as well as week4b.js to test each of your UI functions.

For each of these functions, you should create **and** return the element (don't append it directly to the body within the creating function).

Create functions to generate each of the following UI elements:

- P
- H1, H2, H3
- DIV
- SPAN
- BUTTON
- TEXT INPUT
- IMG

Then I also want you to create a complex UI element. I want you to create a labelledTextField function. In it, you will create a label + text input + button, just like a search bar might have "Search: [ ] (Search)".

You function should take text for the label, text for the button, and return a custom object with each of the 3 fields:

```
return {
    label: /* label element goes here */ ,
```

```
            input: /* input element goes here */ ,
            button: /* button element goes here */
        };
```

Test out all of these functions in your week4b.js code.

Take a screenshot of your working application, as well as the specific code for the labelledTextField function.

D

Now create a file week4c.html and js/week4c.js . Make both a copy of week4a, and re-use the existing gameData.js . Also make use of the js/ui.js you just created.

I want you to add a button below the player's location, displaying where the user can move to next. E.g. if they are in the Library, then according to the data, the button would have the text "Move to Room A". On clicking the button, the user's location would move to "Room A", and that room's description, as well as next location (the "Library") would be displayed.

Take a screenshot of your working app, as well as the modified displayCurrentLocation function code.

By this stage, you should start to be having some solid ideas of how you may possibly write some aspects of your Project. So it's time to start writing bits of program code, and testing them as you go. Good luck!

Week 5

Week 5 Theory

This week we are looking at **control structures**. Control structures are common across almost all programming languages, and actually have almost identical syntax, that's how common and important they are.

Control structures allow us, as the programmer, to write decision-making code. We write code that may or may not run, depending on some **condition** when the user is using our app.

For example, we may write code which loads a user's previous game state, IF they saved previous game state. Another example, we may run code multiple times, depending on some factor, like we might display the score for each game level the user has completed. How many levels have they played? We as a programmer don't know. That will be data that varies from user to user, and varies based on how many levels they've completed.

In other words, control structures are vital to create code that **adapts** to the specific circumstance that our app happens to be in at any particular point in time, for any kind of user.

There are two main types of control structures: if-statements and loops. And in regards to loops, there are two sub-types, a for-loop and a while-loop. These structures will solve 99% of all your decision-making code problems. So, when in doubt about how to program if a block of code will run, or how many times it will run, you only have 3 options to think of:

1. if-statement (used 60% of the time),
2. for-loop (used 35% of the time),
3. while-loop (used 4% of the time).

On very rare occasion (< 1% of the time), you may need something more fancy. But this would be highly unlikely at beginner or intermediate programming skill levels.

Conditional Programming

Conditional programming is an essential element of programming. Without it, every application would literally do exactly the same thing every time, with no variation for user actions nor user data. This is because unless the computer has the ability to choose between two or more paths, then the same path of program logic / program code will run every single time.

In JavaScript, as with most languages, we use “if statements”. Here are several examples, for you to see the different syntaxes we can use.

```
var x = 4;
if (x > 5) console.log("X is bigger than 5");


var city = "Melbourne";
if (city == "Melbourne")
      console.log("The city is Melbourne");


var gameStarted = true;
if (gameStarted) {
      console.log("The game has started");
}
```

The 3 examples above all give different examples of conditions:

1. if the value stored in variable x is greater than 5, execute 1 line of code.
2. If the value stored in variable city is exactly equal to the String "Melbourne" then execute 1 line of code.
3. If the value stored in the variable gameStarted is a Boolean (true/false value) that is set to true, then run a block of statements within { } braces (of which there is a single line of code in this example).

Another useful if statement is to check if a variable is undefined or not, which is useful in function parameters:

```
function test(cityName) {
      if (typeof cityName === "undefined") {
            console.log("The parameter cityName is NOT defined.");
      }
}
```

Note in the above we use triple equals. I won’t go into the specifics on why we do that right now, but you are encouraged to look it up if you’re going for above a pass/credit in this unit.

Note that we can check the opposite of a condition using “!”, which means “not …”.

```
if ( !(x > 5)) console.log("X is NOT bigger than 5");

if (city != "Melbourne")
      console.log("The city is NOT Melbourne");

if (!gameStarted) {
      console.log("The game has NOT started");
}
```

We can also combine multiple operators, with special combination operators:

- && : means “and”, as in, both the first condition AND the second condition must both be true.
- || : means “or”, as in, if either (or both) of the conditions are true, the code will execute.

Examples:

```
if ( (city != "Melbourne") && (city != "Sydney") )
      console.log("Your city doesn't seem popular");


if ( (age < 18) || (age > 65) )
      console.log("You aren't of working age.");
```

We can also have multiple if statements linked together:

```
var a = 4;
var b = 5;

if (a > b)
      console.log("X is larger");
else if (b > a)
      console.log("Y is larger");
else
      console.log("X and Y are the same");
```

Notice the "else if" in there. We can have **as many** else-if statements as we want. We can only have a single "if" and a single "else" within an if-statement block, but sandwiched between those two can be as many "else if" conditions as we wish (technically we never need the "else" on the end if it's not relevant).

Also, whenever we check Strings, these are case-sensitive. This means that the case (uppercase / lowercase) has to be exact. If we don't want that, then we need to convert both Strings to a specific case, like this:

```
if ( (city.toLowerCase() != "melbourne") && (city.toLowerCase() !=
"sydney") )
      console.log("Your city doesn't seem popular");
```

In this case, I converted city to lowercase letters temporarily. I didn't change the existing "city" variable, which is why I had to do .toLowerCase() twice.

If I was comparing my answer String to a user's answer String, I would do:

```
if ( userAnswer.toLowerCase() == actualAnswer.toLowerCase() ) {
      console.log("Answers are a match!");
}
```

Loops

Looping is an essential programming structure, much like if-statements, that occur in almost all programming languages.

There are generally two main types of loops, for-loops and while-loops. We use for-loops when we can know, at program runtime, specifically how many times a loop can run. This is most commonly used with arrays. Whereas while-loops are used where we don't know how many times a loop may run.

Unfortunately, a for-loop looks quite complex at first, because there are 3 pieces of information all crammed into a single line. But once you get accustomed to the structure, you'll be able to write it off the top of your head with ease. You want to remember this structure by heart, as it's so common you will write it thousands of times within each year of programming.

```
for (var counter = 0; counter < 10; counter++) {
      console.log("Counter value: " + counter);
}
```

The above is the structure, and it's virtually the same across situations you will encounter. We start by **declaring a variable** (and that variable's scope is only available **within** the for-loop, not outside it. We also assign it to a starting value (in the above case, 0). We then have a semi-colon separator, and next we have the **loop condition**. The loop condition is a true/false condition value (the same conditions that we use in if-statements). The loop will keep running in each iteration that this condition is true. The final segment inside a for-loop declaration is the **counter**.

Usually we simply increment by 1 each iteration with our counter. But we can increment by 2 each time, if we so facing, to do say, even numbers between 10 and 20 (inclusive):

```
for (var counter = 10; counter <= 20; counter += 2) {
      console.log("Even numbers: " + counter);
}
```

Notice above how the condition is "counter <= 20". I put "<=" because I also want "20" to be a valid counter value. It's only when counter gets to 22 that execution would stop running.

What about running backwards?

```
for (var counter = 10; counter >= 0; counter--) {
      console.log("Counter: " + counter);
}
```

In the above example, I **start** at 10, I keep going while we are **above or equal to** 0 (so we include 0 in our final result), and the counter **decrements**.

Now, the other type of loop is extremely simple and generic, but requires more thought in our program. By more thought, I mean that **you must make sure your loop eventually ends**. This means thinking about the **condition** expression of the loop, and when that condition will change during each iteration of the loop (each repeat cycle of the loop), such that eventually the loop will stop running. Otherwise your computer will freeze.

Let's say we want to work out how many times a number can be divided by 10. The number may come from user input, so we don't know initially how many times it can be divided or anything.

```
var userNumber = 120; // we should get this from a field

while (userNumber > 1) {
	var remainder = userNumber % 10;
	console.log("Digit: " + remainder);

	// here is where I change userNumber each loop iteration
	userNumber = userNumber / 10;
}
```

In the above example, I am running a while-loop. A while-loop only contains a **condition**, nothing else. In the above example, the condition is "if the value inside the variable userNumber is still above 1, repeat another cycle of the loop". **The most important factor to remember with while-loops is that we ABSOLUTELY must adjust the variable within our while-loop condition** (the part inside ( )). If we fail to adjust userNumber, then the loop will run forever, and your app will sit there stuck and you have no choice but to force-close the browser window / app. So, if the while-loop uses a variable "userNumber", then that variable "userNumber" MUST change within that loop code (it must be assigned to a different value within the looping code). Notice also I put > 1, because JavaScript numbers are always decimal, so we may end up with countless fractions that run on forever. An alternative would be to ensure userNumber stays as a whole number:

```
while (userNumber > 0) {
	var remainder = userNumber % 10;
	console.log("Digit: " + remainder);

	// here is where I change userNumber each loop iteration
	userNumber = Math.floor(userNumber / 10);
}
```

This way we would be a lot safer to use the condition (userNumber > 0), as there wouldn't be any decimal places after the Math.floor() operation.

A Refresher on Functions

Whenever you want code to run later, as in, not immediately at this point in time, but later on some event (an event handler), we always create a function. That function (whether named or anonymous) says "bunch this code up together as a single item that I'll run later".

Let's say we're setting up a game with the following function, but I'll include one line of incorrect code at the onclick assignment:

```
window.currentItem = null;
```

```
window.createGameLevel1() {
      // pick a random number, from 0 – gameLevel1.length (exclusive)
      var itemNumber = Math.floor(Math.random()*gameLevel1.length);

      // save the current item (object) in global variable
      currentItem = gameLevel1[itemNumber];

      var questionElement = createParagraph("Question" +
currentItem.question);

      var answerElement = createButton(currentItem.answer);
      var incorrectElement1 = createButton(randomAnswer[0]);
      var incorrectElement2 = createButton(randomAnswer[1]);

      answerElement.onclick = console.log("correct");
}
```

Currently:

```
answerElement.onclick = console.log("correct");
```

This runs console.log("correct") immediately when generating the level 1 question (so the user hasn't done anything yet, only the level has been generated). This is wrong.

What you want to do is tell onclick to **run some bundle of code later** (when the click event actually occurs). So we always have a function in this situation.

There are two ways to declare functions: named and anonymous functions.

Named function:

```
answerElement.onclick = level1AnswerButtonClickHandler;
```

(and before this function is declared, write somewhere the following):

```
window.level1AnswerButtonClickHandler = function() {

    console.log("correct");

}
```

Another approach is an anonymous function (what I prefer if the function is small):

```
answerElement.onclick = function() {

    console.log("correct");

}
```

And that's it. That will then correctly run the function when the answerElement node is created.

Capturing Loop Variables with Anonymous Functions

This is one of the most advanced JavaScript concepts out there. It's also, somewhat unfortunately, one of the most necessary. So, we have no choice but to learn it.

Whenever we run through any kind of loop, occasionally we want to "save", or **capture**, the variable, for use in an event handler. When we have an event handler, we are setting up a function to run **later**, and to run later means we need to save any parameters. A for-loop and a while-loop variables change every iteration, we can't "freeze them", because the loop keeps running immediately until it finishes.

```
// assume this is an array of Button elements, already created
var buttons = [];

// this function simply displays a given number
function displayNumber(number) {
	console.log(number);
}



/*
	This function CREATES a brand new, anonymous (no name)
function.
	We do not call (run) this function at this stage (so
displayNumber(number) is not called yet).
	We simply create the function, to run later.
	We are dynamically writing program code, that happens to
capture (save) the local parameter "number".
*/
function addDisplayNumberEventHandler(button, number) {
	// both "button" and "number" are captured, which means they
won't change here, so the correct number is related to the relevant
button
	button.onclick = function() {
		// if this button is clicked, then we call displayNumber
		displayNumber(number);
	}
}



for (var counter = 10; counter >= 0; counter--) {
	var button = createButton("whatever");

	// THIS WON'T WORK: (because "counter" keeps changing)
	button.onclick = displayNumber(counter);

	// THIS IS CORRECT: (button + counter get "saved" or "captured"
when a function is called, such that their parameters are
"permanent")
	addDisplayNumberEventHandler(button, counter);
}
```

The above example is quite complicated, all things considering. We have a loop (at the bottom of the code), that runs backwards 10 times. There are two lines of code, the first incorrect, the second correct. The first is incorrect because we are attempting to assign an event handler to a future function, but we are running the function immediately, not providing a function name for future calling. The function is already run, so when the button is clicked, nothing happens. The second example does provide a function, that hasn't yet being called, waiting to be called whenever the button is clicked. **This function does not have a name**, thus why we call it an "anonymous function", in programming-speak. The createDisplayNumberHandler function will create a brand new function, right there, and return it (so that it can be assigned to button.onclick). That anonymous function calls displayNumber with the given number. Because we are creating a written function, every line of code is saved (the same as when we write JavaScript code into a file, except an anonymous function is writing that code into RAM memory). Each time we call createDisplayNumberHandler, it creates (writes) a brand new function. So, we will end up with 11 new functions (10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 are the loop iterations). None of those 11 functions have names. Each one is slightly different (in that the "number" will be either 10, or 9, or 8, etc.). This is how we **capture** the value of a loop for the specific use of **event handlers**.

Week 5 Exercises

Exercises:

A

Create new files week5a.html and js/week5a.js . You may copy any existing ui.js and utils.js files that you have from previous exercises. You may create a week5a.css file if needed, as always. I also want you to create a folder "data", to store all your JavaScript files that are exclusively data only.

1. Create a file data/colourList.js , which creates a global array called colourList, containing a list of 5 colours (e.g. "red", "blue", etc).
2. In your week5a.js file, create a button "Randomise Colour" which calls a function randomiseBackgroundColour(). This function, when executed, will select a random colour from our colourList array, and change the document BODY element's background colour.
3. In addition to the aforementioned button, create a vertical list of buttons for each colour in the colourList. Each buttons text should be the colour name. When that button is clicked, the document BODY's background colour will specifically change to the button's colour.
4. Further to the last step, colour the background of each button to be the same as the button's intended colour. E.g. if a button says "blue", then the background colour of that button should be blue.

Take screenshots of both your application running, and your week5a.js file.

B

Create new files week5b.html, week5b.css, and js/week5b.js .

1. Create a data file data/studentList.js . Create a global array called studentList and populate it with 5 example students, each array element having an object for the student's data, containing:

```
{
      name: "student name goes here",
      sid: 12345678,
      wam: 64.5
}
```

2. Create a function called displayStudentTable(). The purpose of this function is to create a TABLE element, with 3 columns: SID, Name, WAM (as a header row). You are then to add a new row element for every student in studentList, and to fill each column (TD) as appropriate.
3. Create a 4th column "Grade". This column is not stored in the studentList data, but is auto-calculated based on the WAM. You will write text in this column: Fail, Pass, Credit, Distinction, High Distinction, based on each student's WAM score.
4. Depending on the grade of the user, assign a background colour for the entire row of red if the student is failing, otherwise green.
5. Create one final row, with only the WAM field having data, the rest being empty. Make this field the average value of all students' WAM scores. Note: to calculate the average, simply keep a sum of all WAM scores (start the sum at 0, and add to it for each student), then at the end, divide this sum by the number of students (the length of the studentList).

Take a screenshot of your final app's page, as well as of the displayStudentTable function code.

That's it for this week. You're doing awesome having reached this far! You've covered 80% of the topics you need for your project by now. Well done!!

Week 6

Week 6 Theory - Events

Programs don't run unless they're told. That's how computers are designed. Even if everything occurs on auto-pilot, that auto-pilot doesn't start unless the application is loaded (run by the user).

In JavaScript, all JavaScript code only ever runs when an event occurs. The code will run to completion before any other events will be processed. And JavaScript is a single-tasking language, not multi-tasking, which means that no matter how many CPUs or CPU cores your computer has, JavaScript won't use them. JavaScript runs execution on a single CPU core (thread), and runs each function to completion before checking for any awaiting events. That means that if your code takes too long to run (or in the case of an endless loop, never ends), then your interface will feel very sluggish, as the user may click a button, then another, but the second button action does not operate immediately because the first button click event code is still running.

A JavaScript event handler is any function that **you** define. If you have a HTML element in a variable, you can simply assign a function to the onXXX variable within that element. For example:

```
startButton.onclick = startMyGame;
```

The above statement assigns an event handler (the startMyGame function will "handle" the event, thus what we call an event handler, even though it's just a regular function). The function name is assigned to the field "onclick" inside a theoretical variable startButton (which we'll presume contains a HTML button). By the way, the onclick event can happen on any HTML element, whether it's DIVs, IMG tags, H1, H2, bullets… anything.

Event handling functions are regular functions. They can ask for one parameter, as one parameter is always given to them: an object containing the event that is currently occurring. This event object contains useful information about the event that is currently occurring (and that your function is handling), and also has some functions that allow you to stop the event from going further.

```
function startMyGame(event) {
      console.log(event.target);
}
```

The above is an event handling function for the previous startButton.onclick event. When the startButton is clicked, this function will now run. There is a single parameter that the function can receive (if it wants), which we will refer to locally as "event". One of the most useful fields inside this event object is the "target" field, which specifies which HTML element is the one

that triggered this event handler. This is quite useful if we have an event handler which handles multiple HTML elements (in other words, the same function is given as the event handler to multiple HTML elements' onclick events).

Drag & Drop

I have to be honest here. Drag & drop event handling is not as easy as the onclick event. But, it is one of the must-have interfaces we should design for if we are creating web app games.

There are 4 steps to accepting drag & drop events, two on the dragging element, two on the dropping element:

(1) The element to be dragged must have an attribute "draggable" set to true.

```
ballImage.setAttribute("draggable", "true");
```

(2) The element to be dragged must do something when the drag event starts (usually, save the fact that a drag is happening).

```
ballImage.ondragstart = onBallDragStart;
```

This assumes we have a function declared:

```
function onBallDragStart(event) {
	currentDraggingObject = event.target;
}
```

Or if we want to write the "onBallDragStart" function code right there:

```
ballImage.ondragstart = function (event) {
	currentDraggingElement = event.target;
};
```

(3) Now we can focus on the drop target. First, we need to accept drops by allowing dragging over the drop element. This is not done with a Boolean flag like the dragging element, but with a function. The reason there's a function is because we can actually write code to say if we accept a dropping element based on the type of dragging element. Let's say we have a kid's game, matching shapes. A triangle drop element should only accept a triangle drag element. So, we can actually prevent the triangle drag element from even triggering a drop. Alternatively (and commonly), we will check what is being dropped on the actual drop event handler, because then we can give some error / user feedback.

```
goalDiv.ondragover = function (event) {
	event.preventDefault();
};
```

This function only needs to call the special function preventDefault, which is inbuilt into all event objects. This will prevent the default HTML action (and the default action is to prohibit drops, so if we want to accept drops, we want to prevent this default prohibition). A bit of a double-negative going on there.

(4) Finally, and most importantly, is our drop code. This occurs on the element **receiving** the drop. It is this function that decides what to do with the given dragging element (which we need to keep track of somehow, e.g. in a global variable).

```
goalDiv.ondrop = function (event) {
	// error checking
	if (currentDraggingElement == null) return;

	var receivingElement = event.target;

	console.log("Receiving element: " + receivingElement + "
receives: " + currentDraggingElement);

	// MUST have this to allow the drop to complete
	event.preventDefault();
}
```

The ondrop event handler is the one where you'll write most of your code. Remember that, if this is in a loop of any kind (because you have multiple elements all accepting dropping), then you often need to use a function to create the anonymous function, in order to capture the looping variable. In the above case, there is nothing inside the function that relies on goalDiv, so there is nothing to capture, so it is fine.

Mobile Devices

On mobile devices, drag + drop isn't well supported. So instead we make use of:

- touchstart
- touchmove
- touchend

With these 3 events, we can get a similar effect to dragging (but note that it's not identical).

Week 6 Exercises

Exercises:

A	Create week6a.html, week6a.css, and js/week6a.js Create a small game:

1. At the left of the screen will be 3 shapes: square, triangle and circle (search for vector images to use as these), positioned vertically.
2. At the right of the screen will be 3 square boxes, also positioned vertically, each with centred text above each box: “square”, “triangle”, and “circle”.
3. Then write code to allow the user to drag each shape into any of the 3 boxes. If the shape is a match, then move the shape element into the box element.
4. Write code to slowly fade-in a background colour of green when the game is won.
5. Also write code to draw a big red letter X inside a square box whenever it receives an incorrect shape. If that occurs, leave the X there until the correct shape is dropped.

Note, you don’t have to use for-loop for any of the shape or box creation, if you don’t want to. But if you do, then create a “shapes” array of objects, each element containing shape text (e.g. “Triangle”) and a matching shape image (e.g. “images/triangle.png”).

Take screenshots both of your completed game in half-won and entirely won states, as well as your drop event handler code.

Alrighty! All theory and lab exercises are now done. By now you should have enough knowledge to tackle your project.

Note that there will still be much you will learn (and discover) as you run into issues and go about solving them. I’ve shown you many examples of how to search for help when you run into issues, and the main sites that I look for in search results. I’ve also shown you many examples of the “search query” / search text that I use when trying to describe my problem, to give Google as much information to give me the answer that I want within the first few matches.

You want to make sure you are above > 80% confidence in what we've taught. It's okay if it doesn't make 100% sense, as a lot of that will come with putting skills into practice (i.e. your project). But you should be able to achieve all the basics:

- Can write CSS for any HTML element or class (generally avoid id items).
- Can make easy use of console.log() to debug and find issues with your code using the Console output, and feel very confident to know how you would go about figuring out which line of code doesn't work if any issue came up at any time. Generally you are able to narrow down which line of code doesn't work in under 5 minutes, by placing console.log() lines when an issue comes up.
- Can write JavaScript functions, complete with comments and indentation, that can perform some function, and return a value if necessary.
- Can write JavaScript functions to create UI components (and return them to be added to the document or some other HTML element).
- Can add and remove CSS classes from a HTML element.

- Can add/remove CSS classes to show, hide, and animate elements dynamically on Javascript events.
- Can write event handler functions, and attach them to HTML elements (e.g. the "click" event).
- Can use container elements (e.g. DIV / SPAN) in creating complex UI components consisting of multiple HTML elements.
- Can add variables and functions to the window object, to access them between multiple Javascript files.
- Can write data structures in Javascript/JSON format, including objects, arrays of primitive types, and arrays of objects.
- Can access data structures using another variable to store the key (if an object) / index (if an array).
- Can use logic (if/else statements), to decide on two (or more) different paths / courses of action.

That's the bulk of what you should now be able to do. If you have any weak points, I do suggest you go back to an exercise, or simply create a new test directory, with a test HTML, CSS, and JS file in there, to create little exercises for anything you don't yet feel confident with. E.g. if you didn't quite feel confident adding or removing a class to a HTML element in javascript, you'd create a button to say "Test adding a CSS class to the paragraph", and then on click, add a paragraph and add a class to that paragraph dynamically. You'd have another button "Test removing a CSS class", which then checks if we have a paragraph (e.g. in some global variable window.testParagraph), and if not, says to click the first button first, and if it does exist, removes the class from it. Then in about 8 lines of JavaScript code you have tested and demonstrated to yourself that you can add and remove a CSS class. You then add buttons for anything else you want to experiment with (whether it's existing content we covered, or new concepts you want to play around with). This way you don't affect your Project code in your experiments, and you keep the code small and contained, so it's easier to know where something goes wrong, as most code will be contained to a single function. All great programmers do this when they're testing a new code concept / API that they haven't used before.

Week 7

This week is going to be fairly simple, as by now you should be well into your Project, and have all the core knowledge you need.

Week 7 Theory – Bonus Topics

The following are topics which you may find useful. There are no exercises from this point on.

On Screen Keyboard

When supporting mobile devices, we often can't rely on the pop-up keyboard for a number of reasons. The first is that the pop-up keyboard removes screen space that we may need (and triggers resize events which may annoy the user if the keyboard is constantly popping up and down). The second is that we can't style the keyboard in any way and make it look like part of our app. We always want consistency **within** our app, which means that all features + buttons + interactions within our app have the same look and feel. The inbuilt pop-up on keyboard does not allow this.

So what we can do is simply create our own "on screen keyboard". Which all it really is – is buttons. 3 rows of buttons to generate 26 alphabet letter options (in the same layout as a regular US-English keyboard – although you're welcome to shuffle the order of keys, or only display keys of letters that actually exist in the answer, etc. The options are endless.

We have two main things to do: create the buttons, and do something when the button is pressed. But this means 3 functions (because we need an event handler created, but we'll be using loops, so we need to use another helper function to add the event handler):

1. createAndDisplayOnScreenKeyboard()
2. addEventHandlerForOnScreenKey(button, character)
3. crosswordReceivesKey(character)

```
/*

      This function is called automatically whenever (and each time)
an on-screen keyboard button is pressed.

      character: a String with a single ASCII character of the "key"
button that was pressed. For example "q", "w", "e", etc.

*/
function crosswordReceivesKey(character) {
```

```
	// do something with character
}



/*
	Simply add an onclick handler for the button,
	To take relevant action with the given character.
*/
function addEventHandlerForOnScreenKey(button, character) {
	button.onclick = function() {
		// let's just pass the character onto our
crosswordReceivesKey function. That's all we need to do.
		crosswordReceivesKey(character);
	};
}



/*
	Create the 3 rows of a keyboard.
*/
function createAndDisplayOnScreenKeyboard() {
	var row1 = ["Q","W","E","R","T","Y","U","I","O","P"];
	var row2 = ["A","S","D","F","G","H","J","K","L"];
	var row3 = ["Z","X","C","V","B","N","M"," ",",","."];

	// create a container for the entire row1, and append it
	var row1Div = createDiv();
	document.body.appendChild(row1Div);

	// create EACH button for the first row
	for (var index in row1) {
		// get the key character string from the array
		var key = row1[index];
		// create the button element
		var button = createButton(key);
		// add the event handler, so the button click does
	something
		addEventHandlerForOnScreenKey(button, key);
		// add it to our row1div
		row1Div.appendChild(button);

	}

	// now repeat for row2.... (same thing)
	// create a container for the entire row2, and append it
	var row2Div = createDiv();
	document.body.appendChild(row2Div);

	// create EACH button for the first row
	for (var index in row2) {
		// get the key character string from the array
		var key = row2[index];
		// create the button element
		var button = createButton(key);
		// add the event handler, so the button click does
	something
		addEventHandlerForOnScreenKey(button, key);
		// add it to our row2div
		row2Div.appendChild(button);
```

```
        }

        // then row3…

    }
```

Notice how the code above for row2 (and for row3) is the same as row1?

That's usually a sign to introduce a loop, to shorten your code.

But we already use a loop, so how? Well, loops can go inside other loops. We can repeat any code (including code that itself has a repeating part), so long as:

1. the variables don't conflict (e.g. the loop variable names need to be different)
2. we re-structure our variables so that they can be iterated over (e.g. rows need to be in an array)

Let's try that function again:

```
/*
      Create the 3 rows of a keyboard.
*/
function createAndDisplayOnScreenKeyboard() {
      var rows = [
            ["Q","W","E","R","T","Y","U","I","O","P"],
            ["A","S","D","F","G","H","J","K","L"],
             ["Z","X","C","V","B","N","M","  ",",","."]
      ];

      for (var rowIndex in rows) {
            var row = rows[rowIndex];

            // create a container for the entire row, and append it
            var rowDiv = createDiv();
            document.body.appendChild(rowDiv);

            // create EACH button for the current row
            for (var index in row) {
                  // get the key character string from the array
                  var key = row[index];
                  // create the button element
                  var button = createButton(key);
                  // add the event handler, so the button click does
            something
                  addEventHandlerForOnScreenKey(button, key);
                  // add it to our rowdiv
                  rowDiv.appendChild(button);

            } // end of ROW for-loop

      } // end of ROWS for-loop



}
```

And that's it.

See how the power of loops (specifically FOR-loops) can condense code and make it much shorter and neater?

Also always focus on both (1) indentation, and (2) comments. See how my indentation (the TAB key on the keyboard) always keeps code vertically aligned. Also see that a comment every line of code, and what the function does, and what input (parameters) the function takes, with examples.

Randomising

We can generate random numbers using Math.random() . This special function returns a value between 0.000000000 and 0.999999999

Now, how do we get whole numbers from all those decimals? Simple. Using either:

- Math.floor(…)
- Math.round(…)
- Math.ceil(…)

Those 3 functions will take any decimal number, and return a whole number (what we call an "integer"). Floor will always round down (so 0.1 goes to 0, and 0.9 goes to 0). Round will round up or down, depending what's closest (so 0.1 goes to 0, and 0.9 goes to 1). Ceil goes to the ceiling (so 0.1 goes to 1, and 0.9 goes to 1).

Now how do we get a range from this? Here is where we use a little maths. Some examples:

```
// number to be between 0-2 inclusive:
var number = Math.floor(Math.random() * 3);

// number to be between 0-10 inclusive:
var number = Math.floor(Math.random() * 11);

// number to be between 1-10 inclusive:
var number = Math.floor(Math.random() * 10) + 1;

// number to be between 500-599 inclusive:
var number = Math.floor(Math.random() * 100) + 500;

// number to be between 500-600 inclusive:
var number = Math.floor(Math.random() * 101) + 500;
```

Random item from a list:

```
window.list = ["dog", "cat", "mouse"];

// get a random item from a list
var randomNumber = Math.floor(Math.random() * list.length);
window.currentItem = list[randomNumber];

console.log("Current Item: " + currentItem);
```

Quickly randomise an entire array:

```
var array = ["dog", "cat", "mouse"];
array.sort(function(a,b) {
	// 50% odds of a positive or negative number
	return 0.5 – Math.random();
});
```

Random item from a list of objects:

```
window.list = [
	{word: "bus", answer: "automobile"},
	{word: "yellow", answer: "colour"},
	{word: "dog", answer: "animal"}
];

// get a random item from a list
var randomNumber = Math.floor(Math.random() * list.length);
window.currentItem = list[randomNumber];

console.log("Question: " + currentItem.word);
console.log("Answer: " + currentItem.answer);
```

Even though Math.random() always returns numbers, we can use it for other data types. For example, we can use random to generate true/false options, by randomising a number between 0 and 1, and checking if it's 0 or not. Let's say you want to randomise the order of dynamically-created DIV elements on the screen. You want, say, 3 options, 1 correct, and 2 incorrect, but you want their order to be random. For logistical reasons, you happen to add the answer first, and then the incorrect options following. While you could simply put all divs into an array first, randomise the array, and then iterate over the array and append them, if you are insisting on appending the correct answer first, then alternatively you could append or prepend each incorrect div one by one, by randomising a number between 0 and 1, and checking if it's 0 or not and thus getting a true/false value from randomising.

Consider the following example:

```
var questionObject = {
	question: "Capital of France",
	answer: "Paris",
	options: ["Moscow", "Sydney"]
}
```

```
var numberOfIncorrectOptions = 2;

// add the correct answer
var correctDiv = createDiv(questionObject.answer);
answerSection.appendChild(correctDiv);

// add some incorrect answers
for (var i = 0; i < numberOfIncorrectOptions; i++) {
      var incorrectDiv = createDiv(questionObject.options[i]);

      // decide if we add the incorrect before or after
      var beforeOrAfter = Math.round(Math.random() * 1);
      if (beforeOrAfter == 0)
            answerSection.prepend(incorrectDiv);
      else
            answerSection.append(incorrectDiv);

}
```

In the above example, we Math.random() (which gives us a value between 0.000000000 and 0.999999999), and we round it (so below 0.5 goes to 0, above 0.5 goes to 1). Giving us 50:50 odds. Then we simply check if the result is 0, in which case we'll choose before, otherwise after. That will run in a loop twice in the above example.

Select Difficulty Level

The following is a small example of how you could support difficulty levels (or even numbered levels, like level 1, level 2, level 3), in your data, and have your code make use of each level without knowing how many levels there are, and without having any separate code per level.

Imagine a spelling type of game, or a select the word to match an image type of game.

```
// this is actually not needed, as we can grab these from levels, but
I'll show you this approach first
window.difficultyLevels = ["easy", "medium", "hard"];

// this is the actual data per level (what words we have to choose
from)
window.levels = {
      "easy": ["dog", "cat", "horse"],
      "medium": ["elephant", "nephew", "system"],
      "hard": ["mortgage", "acquire", "randomize"]
};

function setDifficultyLevel(level) {
      window.difficultyLevel = level;
}

/*
      Creates (and returns) an event handler (function)
      for when a menu button is clicked to change
```

```
	difficulty level.
*/
function addEventHandlerForLevelChangeButton(button, level) {
	button.onclick = function() {
		setDifficultyLevel(level);
	};
}


/*
	Create and return a menu (to be put on the screen)
*/
function createDifficultyMenu() {
	var node = createDiv();
	node.classList.addClass("menu");

	for (var counter in difficultyLevels) {
		var level = difficultyLevels[counter];
		var button = createButton(level);
		addEventHandlerForLevelChangeButton(button, level);
	}

	return node; // in case someone wants it
}
```

Alternatively, we could re-write the above function in a simpler way, so that we don't need the difficultyLevels array at all, but rather we extract the difficulty levels directly from the levels variable (we can iterate over an object's keys very similarly to an array, but note the differences).

```
/*
	This is version 2 of the same function above.

	Create and return a menu (to be put on the screen)
	SAME AS ABOVE, but gets the difficulties
	Directly from the menu
*/
function createDifficultyMenu2() {
	var node = createDiv();
	node.classList.addClass("menu");

	for (var level in levels) {
		if (levels.hasOwnProperty(level)) {
			var button = createButton(level);
			addEventHandlerForLevelChangeButton(button, level);
		}
	}

	return node; // in case someone wants it
}
```

Working with Strings

Remove a letter from a String:

```
window.removeLetter = function(str, letter) {
	var positionOfLetter = str.indexOf(letter);

	// if the letter doesn't exist, return str as-is
	if (positionOfLetter < 0)
		return str;

	// get everything before the letter
	var beforePart = str.substring(0, positionOfLetter);
	// get everything after the letter
	var afterPart = str.substring(positionOfLetter + 1);

	// join the two parts together (missing the letter)
	return beforePart + afterPart;
}
```

Check if a letter:

```
/*
	Returns true if the character is a letter (ASCII table)
*/
window.isLetter = function(char) {
	if ((char >= 'A') && (char <= 'Z'))
		return true;
	else if ((char >= 'a') && (char <= 'z'))
		return true;
	else
		return false;
}
```

Create clickable DIVs (or Buttons) for each letter of a String:

```
/*
	Check if the game is won.
	Assumes all letters remaining are in window variable:
	lettersRemaining

	If won, takes action.
	If not won, does nothing.
*/
function checkIfGameWon() {
	if (lettersRemaining.length == 0) {
		// we have won


	}
}

/*
	Create (and return) an event handler (function)
	to do an action on letter click.
	Necessary because this is called in a loop.
```

```
*/
function createLetterClick(node, letter) {

	var eventHandler = function() {
		console.log("Letter clicked: " + letter);

		// here we would do something to check if the letter is
correct
		// e.g. assume we have window.lettersRemaining as a
String of the word

		if (lettersRemaining.indexOf(letter) >= 0) {
			console.log("Correct letter: " + letter);

			// remove the (first) matched letter
			lettersRemaining = removeLetter(lettersRemaining,
letter);

			// say this letter is correct
			node.classList.add("correct");


			// check if we've won
			checkIfGameWon();

		}
	};

	// return the event handler (to be attached to an onXXX event)
	return eventHandler;
}


/*
	Create clickable DIVs (or Buttons) for each letter of a String.
	Returns a container containing all the clickable elements.
*/
function createRowOfLetters(str) {
	var node = createDiv();

	// iterate over each letter
	for (var i=0; i<str.length; i++) {
		var letter = str.charAt(i);
		// only include letters
		if (isLetter(letter) == true) {
			// create the DIV (could also create a button)
			var letterNode = createDiv();
			letterTextNode = document.createTextNode(letter);
			letterNode.appendChild(letterTextNode);
			letterNode.onclick = createLetterClick(letterNode,
letter);
			node.appendChild(letterNode);
		}
	}
	// return the DIV (of clickable divs / buttons)
	return node;
}
```

Showing / Hiding Screens

You can create the idea of "scenes" or "pages" within your app by placing each scene inside a DIV, and hiding any DIVs that aren't currently active. Consider the following for inspiration:

```
window.currentPage = null;

function showGame1() {
	// hide any existing page
	if (currentPage != null)
		currentPage.classList.add("hide");

	// check if we haven't yet created this page
	if (typeof window.game1Page === "undefined") {
		window.game1Page = createDiv();
		game1Page.classList.add("page");
		// ... now fill in the game1Page contents
	}

	// set the new currentPage to this game
	currentPage = game1Page;

	// show the page (now it is definitely created)
	currentPage.classList.remove("hide");
}
```

And the important CSS:

```
.page {
	position: absolute;
	top: 0;
	left: 0;
	width: 100vw;
	height: 100vh;
	z-index: 1; /* above other elements */
}

.hide {
	display: none;
	z-index: -1; /* moves this behind others */
}
```

Overlaying One Element above Another

If you want one image / HTML element to be directly above another, the most common strategy is to use a container (with relative positioning), and two absolute-positioned elements within it, one on top of the other, and using z-index to control which one is on top.

Here is the HTML (you'd generate this with JavaScript, but this would be the end result):

```
<div class="imageContainer">
```

```
        <img class="actualImage" />
        <div class="imageOverlay"></div>
    </div>
```

And css:

```
.imageContainer {
    /* you probably want the elements to line up next to each other
*/
    display: inline;
    /* doesn't do much, EXCEPT that now any child elements can have
absolute coordinates relative to this one */
    position: relative;
}
.imageContainer .actualImage {
    display: absolute;
    top: 0px; left: 0px;
    z-index: 1; /* we are below the imageOverlay */
}
.imageContainer .imageOverlay {
    display: absolute;
    top: 0px; left: 0px;
    z-index: 2; /* we are above the actualImage, but in the same
position */
}
```

And that's how we would do it.

Note the container needs “position: relative”. When using position: absolute: the coordinates are always relative to the first parent container with "position: ..." (either "relative" or "absolute", doesn't matter). If none are found, then it's relative to the page. As we usually want the container to flow naturally on the page to wherever it ends up (e.g. if we had a grid of tiles), but inside that container we want exact control over positioning, then the parent (the container) gets relative positioning.

Also note that when using “position: absolute”, we must have coordinates (top or bottom, and left or right). Usually people use top and left (y + x coordinates). And we almost always specify these as pixels (e.g. “4px”).

Creating Reusable UI Components

We often have a UI component that is built of several HTML elements altogether. For example, we may come up with the concept of a CommandBox, which is a concept of a new UI component that we create. You can imagine it would be used for the user to type in any command, and have a button to then execute that command. It may be used as a search bar: [search input] (Search button) , or it may be used as a name input: [your name] (Save Name button)

Because this element is one that we create (of other HTML elements), we can simply create a function at any time to create one of our new elements, in essense, we are then creating a new UI component.

```
function createCommandBox(buttonText) {
	// create our main wrapping element, to contain others
	var element = createDiv();
	// our own UI component will probably need a unique style
	element.classList.add("commandBox");

	// we have two sub-elements: the input and the button

	// create the text input, and save the reference into element
	element.myTextInput = createInput();
	element.myTextInput.classList.add("textInput");
	// remember to append it visually to our element
	element.appendChild(element.myTextInput);

	// create the button, and save the reference into element
	element.myButton = createButton(buttonText);
	element.myButton.classList.add("actionButton");
	// remember to append it visually to our element
	element.appendChild(element. myButton);

	return element;
}
```

And that's it.

Notice how I added CSS classes. That's because we'll probably do some formatting to ensure the text input and command button are beside each other, maybe with some margin between them:

```
.commandBox .textInput {
	margin-right: 5px;
	display: inline-block;
	height: 1.2rem;
}

.commandBox .actionButton {
	margin-left: 5px;
	display: inline-block;
	height: 1.2rem;
}
```

The CSS above just adds some very basic styling. Mostly I have space between the text and button, and I'm setting them as inline-block to assist with this. Note I also want to ensure their height is identical. I don't know what height is appropriate, as it depends on the body's CSS font size, so I'll use the "rem" measurement, which is a 1x representation of the body's font size. So, if the font size is 14px, and I have a rem of 1.2rem, then 14 * 1.2 = 16.8px height. I make it a little larger than the font size, as usually a button will have some visual styling (like a border) around it, that will take up some space. Note that I'm not being very accurate here, I'm just taking an approximation.

Handling Bugs Gracefully

We should anticipate users giving us situations that don't work. This may be because the user doesn't yet know how to use our app, or because the input makes sense from the user's perspective.

As much as possible, we should check (1) user inputs, (2) function parameters, and (3) program state situations, all of them using if-statements, to ensure data is valid, and if it isn't, to act **appropriately**, i.e. that the application keeps working and the user is alerted to the issue to be fixed. This is THE MOST IMPORTANT way to check for errors.

For example, consider getting a number from a text input element:

```
var userInputText = inputElement.value;
var number = parseInt(userInputText, 10); // base-10 conversion

// check that we had a number, not a letter
if (!isNaN(number)) {
      // we have a valid number
} else {
      // set the feedback text
      feedbackElement.innerHTML = "Oops.. you didn't give me a
number? (something like 4)";
      // un-hide the feedback element
      feedbackElement.classList.remove("hide");
}
```

In the above code, we need to read up on parseInt, and realise that it will return a number, but if it can't work out the number, it will return a special NaN (Not-a-Number) answer. We can't use == to check for NaN, we need to call a special function isNaN(number), which will return true or false. We want to know if the number ISN'T NaN (in other words, it IS a valid number), so I use (!isNaN(number)). P.S. The reason why we have a special answer called NaN is because this is actually a number, a special number that acts like a code for invalid numbers, that is specified by all CPUs. It is not unique to JavaScript.

There may also be a case where our program code fails because of an issue we didn't anticipate, and these are called program bugs, or software bugs.

Consider this problematic code:

```
// function to create the sum of values from 1 up to the user's input

var userInputText = inputElement.value;
var number = parseInt(userInputText, 10); // base-10 conversion
// check that we had a number, not a letter
if (!isNaN(number)) {
      var sum = 0;
      // loop until the counter reaches the number
      for (var counter = 1; counter != number; counter++) {
            // add the sum of numbers for the user
            sum += number;
      }
} else {
      // set the feedback text
      feedbackElement.innerHTML = "Oops.. you didn't give me a
number? (something like 4)";
      // un-hide the feedback element
      feedbackElement.classList.remove("hide");
}
```

Analyse the code above. There's a programming bug, a logical bug, within our code. The code is syntactically correct and will execute. But, in certain situations, it will crash.

Guess the answer yet? It is…. What happens if the user enters a negative number? How many times will the loop then run? counter = 1, then 2, then 3, then 4… when will it ever equal -1? Never. The loop will never stop.

Now the above bug can't be safeguarded against because endless loops are one of those things we can't catch. So, in the above case, we mostly just need to analyse any loop we have very carefully, and make sure we brainstorm all possible scenarios as best as possible.

Try / Catch

There's another way to "catch" bugs, and that's with a special pair of instructions "try" and "catch"

```
try {
      // divide by 0 is impossible, so this will error

      var x = 5 / 0;

} catch (errorObject) {
      // display the error, without crashing our program
      console.log("Divide by zero error: " + errorObject);
}
console.log("This line continues as normal");
```

This code above will "try" any block of code, to see if it works. If it fails (crashes), the error will be "caught" inside the "catch" block. And we also capture a parameter from this catch

block, the errorObject (this can be any variable name, it's up to you). This way we can find out what the error actually was, and display it.

Note, when we catch an error, we lose the line number that the error occurred on. So often this is something you do later for a function, once you know it's working. But when you are still testing / debugging it, you wouldn't add try/catch blocks at that stage.

Audio – Playing Sound

Two play sound you need two things:

1. MP3 file to play (and put in a sounds/ directory that you've created in your project directory)
2. Code to load + run the file.

The code you want is this:

```
// create a new Audio object for one sound "ding",
// and store it in a variable called soundDing

var soundDing = new Audio('sounds/ding.mp3');

// now play it (can be called repeatedly)
soundDing.play();
```

If you want to load multiple sounds, and store them, perhaps in some global "sounds" object variable, you'll want something like this:

```
function loadSounds() {
   window.sounds = {
        "ding": new Audio("sounds/ding.mp3"),
        "gameOver": new Audio("sounds/gameOver.mp3")
   }
}
```

Then anytime you want to play:

```
sounds.ding.play();
sounds.gameOver.play();
```

Appendix 1: Setting up the Atom text editor

Overview

Atom is a free and open-source text and source code editor for macOS, Linux, and Microsoft Windows which can used to develop various applications and different programming purposes. We create and edit text files using Atom and the text files are used to store data in different format.

The entire Atom editor is free and open source and is available under the https://github.com/atom organization.

Installing Atom

Installing Atom is pretty straight forward and you can download it from https://atom.io/

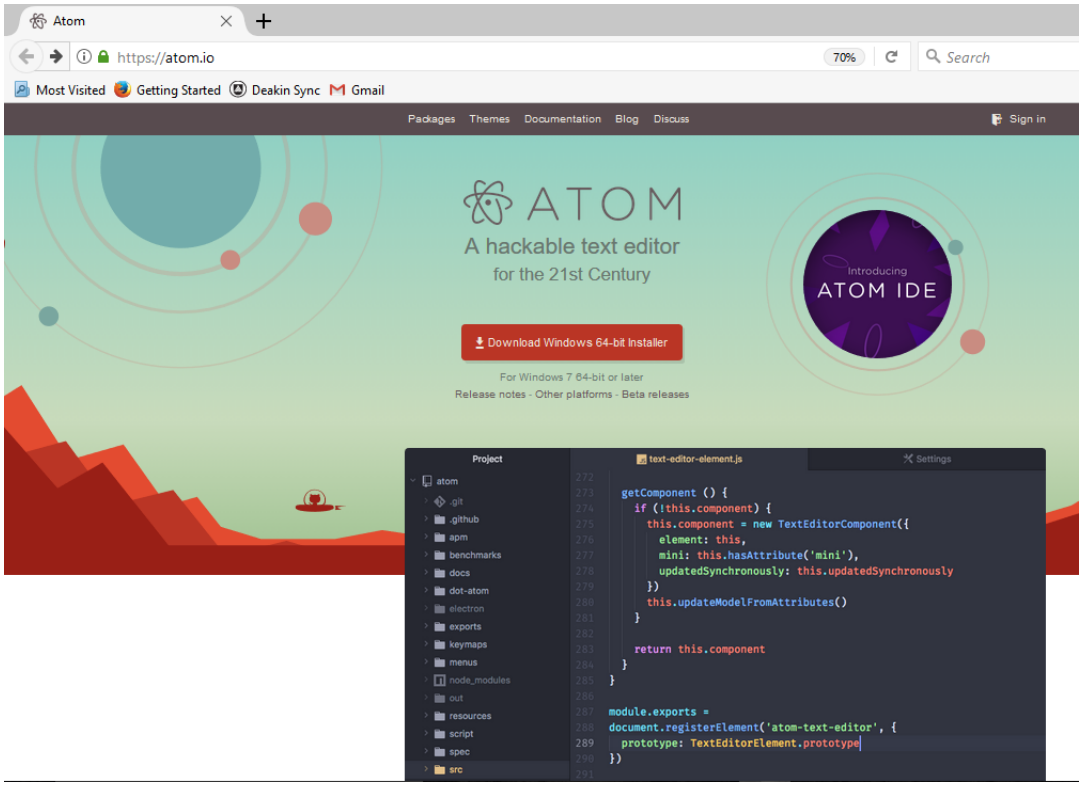


Fig: Atom Website

Other versions or for other platforms, you can download from https://github.com/atom/atom/releases/tag/v1.21.1

After installation, the atom can be opened and below screen will be displayed.

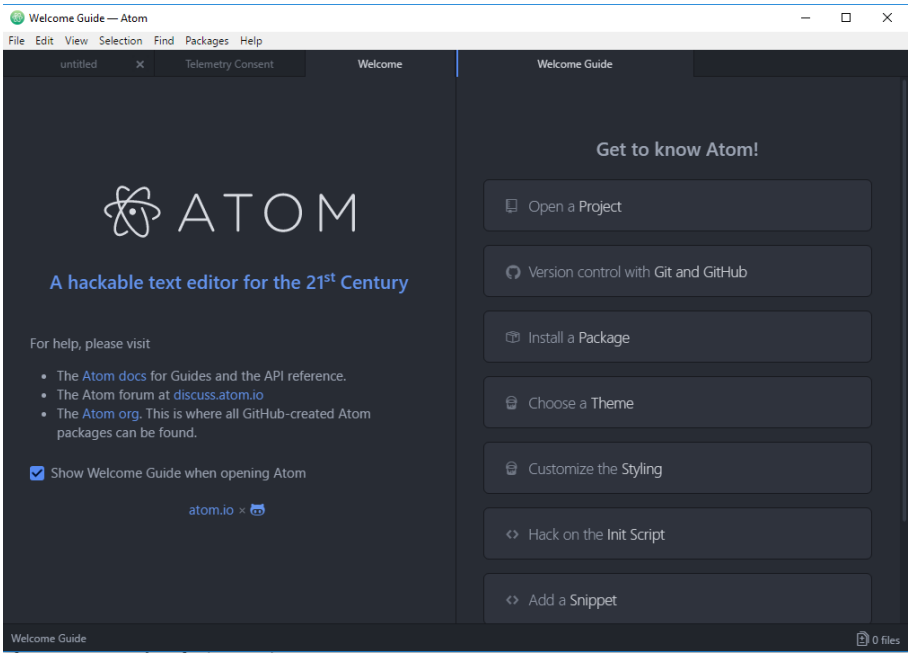


Fig: Atom Editor Welcome Screen

Atom Interface

Now let's familiarize the interface of Atom

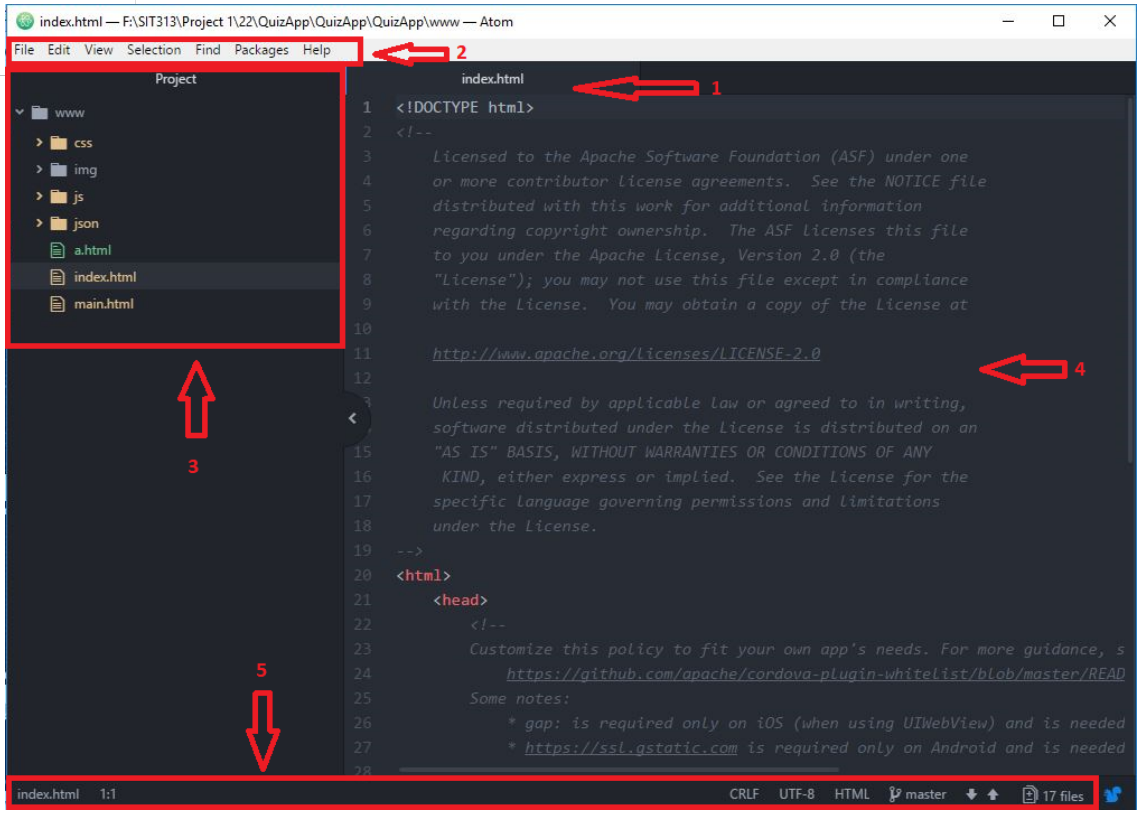


Fig: Atom Editor

Pane: A pane is a visual section of Atom (eg. the tab bar, the gutter (which has line numbers in it), the status bar at the bottom etc).

Basic Terminologies

1. Tab bar: Specifies the files which are selected are editing.
2. Menu Bar: To access various menu items.
3. Tree View: Gives the current working directory and the file structure.
4. Buffer: A buffer is the text content of a file in Atom.
5. Status Bar: This gives information about current working file.

Basic Operations

Opening a file

A file can be opened in Atom for editing. This can be opened by choosing File -> Open File or cmd + O (Mac) or ctrl + O (windows).

Saving a file

You can directly save the file after editing/adding the contents in any format as required.

To save current file: File -> Save / Ctrl + S (windows) / cmd + S (Mac)
To save a file with extension: File -> Save As / Ctrl + Shift + S (Windows) / cmd + shift + s (Mac)

Opening a project in Atom

You can also open a directory instead of a file so that you can work on all related files in a project.

For this, you can choose File -> Open Folder or Ctrl + Shift + O (windows) or cmd + shift + O (Mac).

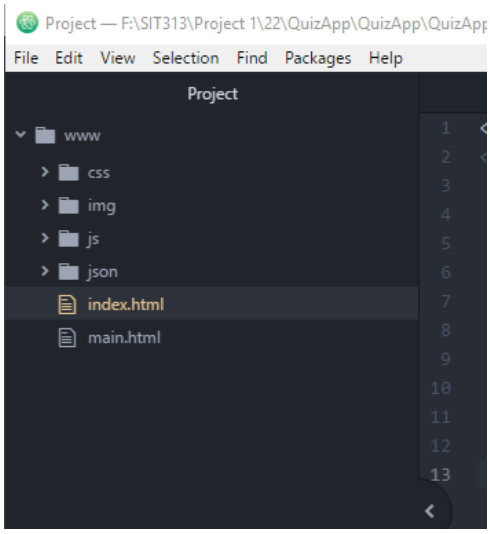


Fig: Project view in Atom

Customisation

Now let us see how we can change the settings of the editor or add packages to Atom.

Settings and Preferences

We can modify various settings and preferences of the editor by accessing the settings of the editor. You can access the settings by clicking File -> Settings

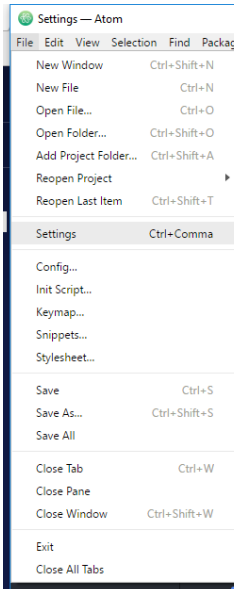


Fig 4: Settings Menu

In Mac:
To open the Settings screen, you can go to the Preferences menu item under the main “Atom” menu in the menu bar.

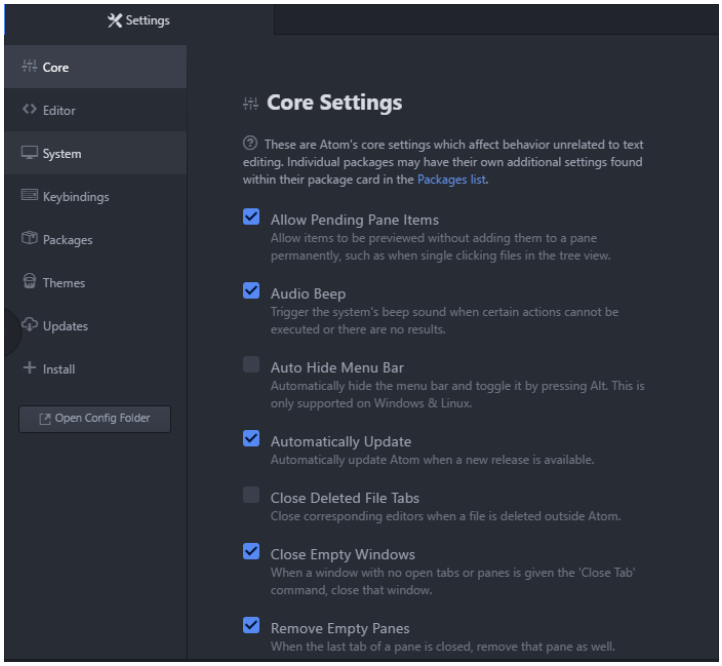


Fig: Settings pane

You can modify any settings and preferences which are necessary. This includes things like changing the theme, specifying how to handle wrapping, font settings, tab size, scroll speed and much more.

For example, let’s change the default tab size to 4. For doing so, access the editor item in settings window
File -> Settings -> Editor

Under the Editor menu, scroll down to Tab Length item and change it to 4.

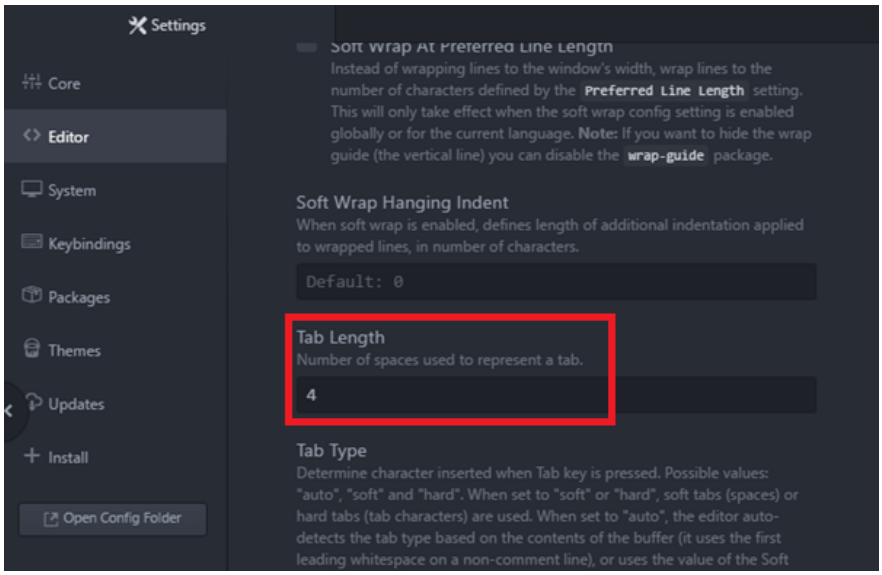


Fig: Settings pane with Editor tab

Adding Packages

You can add any package you want to the Atom editor. The package installer can be accessed from the settings tab in the File menu in Install tab.
File -> Settings -> Install

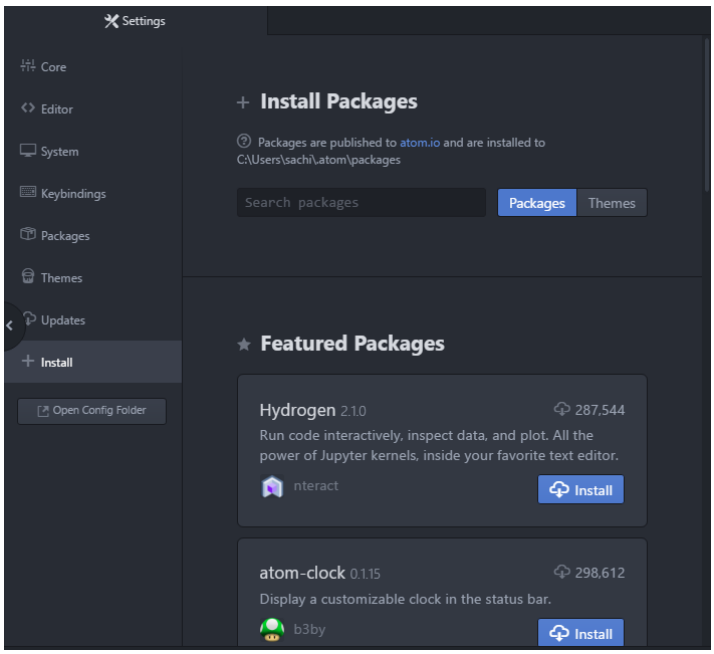


Fig: Install menu in Settings pane

All the packages will come up with an "Install" button. Clicking that will download the package and install it.

Let's install a package to our project say 'github-utils for Atom' which allows to view the url of all pull requests on GitHub.

For doing so, search for ‘github’ in the install tab and this will list all the packages related to GitHub. You can scroll down to gitnhu-utils and click on install for adding the package.

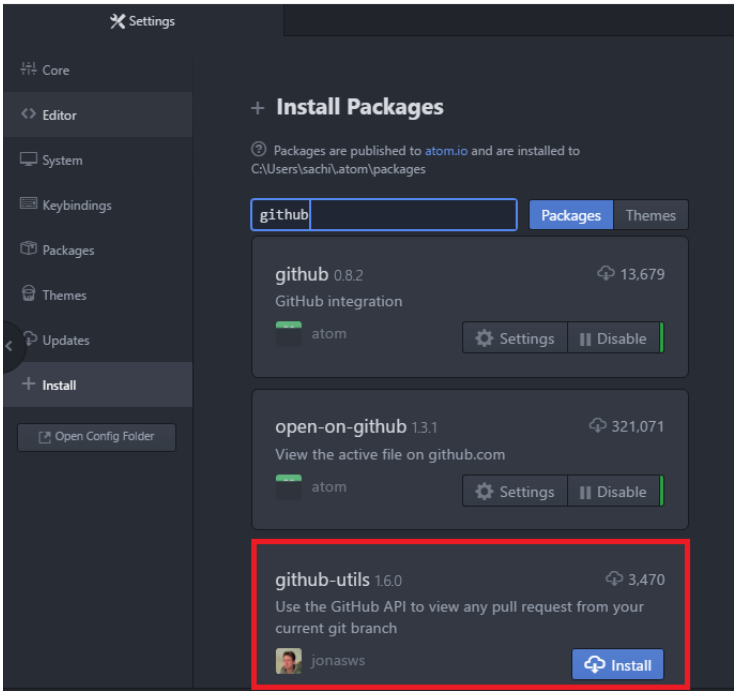


Fig: Install window for github package search

There are more than 80 packages that comprise all the functionality that is available in Atom by default. All installed packages can be accessed using the Packages item in the Settings pane. You can also disable any unwanted packages in this window.

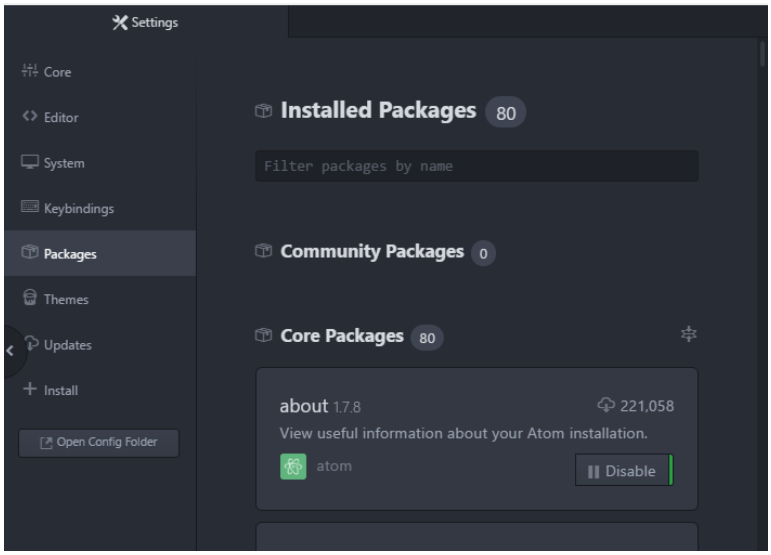


Fig: Packages menu in Settings pane

Bonus Operations

Fuzzy Finder

Fuzzy finder can used to locate any files in the current Project folder.

Windows
Ctrl + T or Ctrl + P

Mac
Cmd + T or cmd + P

Find and Replace

Finding and replacing text in your file or project is quick and easy in Atom.

Cmd + F (Mac) or Ctrl + F (windows)
Search within a buffer

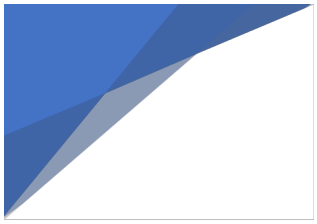
Cmd + shift + f (Mac) or Ctrl + shift + F (windows)
Search the entire project

Snippets

Snippets are an incredibly powerful way to quickly generate commonly needed code syntax from a shortcut.

Many of the packages come bundled with their own snippets that are specific to that mode. If you type html in a new HTML document and you hit tab, the basic snippet associated with html will be expanded.

Fig: A HTML file with expanded code snippet from Atom



Appendix 2: Additional HTML / CSS / JS

HTML Forms

HTML forms are used to collect information from the visitors of the website. There are different form elements used for this purpose.

<form>: This element should always carry the action attribute and will usually have a method and id attribute too.

action
Every <form> element requires an action attribute. Its value is the URL for the page on the server that will receive the information in the form when it is submitted.

Method
Forms can be sent using one of two methods: get or post.
With the get method, the values from the form are added to the end of the URL specified in the action attribute.
With the post method the values are sent in what are known as HTTP headers.

Form Elements

1. **<input>:** The <input> element is used to create several different form controls. The value of the type attribute determines what kind of input they will be creating.

   - *type="text"*: When the type attribute has a value of text, it creates a single line text input.
   - *type="password":* When the type attribute has a value of password it creates a text box that acts just like a single-line text input, except the characters are blocked out.
   - *type="radio":* Radio buttons allow users to pick just one of a number of options.
   - *type="checkbox":* Checkboxes allow users to select (and unselect) one or more options in answer to a question.
   - *type="submit":* The submit button is used to send a form to the server.
   - *type=”reset”:* This is used to reset the form. It will clear all the contents of input fields.

2. **<textarea>:** The <textarea> element is used to create a mutli-line text input.

3. **<select>:** The <select> element is used to create a drop-down list box. It contains two or more <option> elements.
   - <option>: The <option> element is used to specify the options that the user can select from.

Example

```
<select name="state" id="state" required="required">
        <option value="" selected="selected">Please select</option>
        <option value="vic">VIC</option>
        <option value="nsw">NSW</option>
        <option value="qld">QLD</option>
         <option value="nt">NT</option>
        <option value="wa">WA</option>
        <option value="sa">SA</option>
        <option value="tas">TAS</option>
        <option value="act">ACT</option>
</select>
```

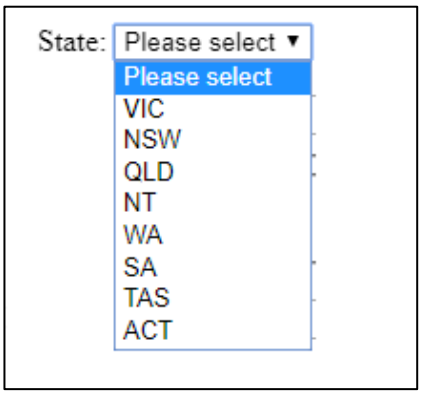


Fig: Select menu displayed in the web browser

Grouping elements

<fieldset>: You can group related form controls together inside the <fieldset> element. This is particularly helpful for longer forms.

<legend>: The <legend> element can come directly after the opening <fieldset> tag and contains a caption which helps identify the purpose of that group of form controls.

Commonly used CSS properties

Box Model Properties

Property Name	Description
float	The float property defines if a <**div**> or <**img**> element should float to the left or to the right, e.g.: {**float**: left;} or {**float**: right;}.
width	This property defines the width of a <**div**> element, e.g.: {**width**: 996px;}.
height	Defines the height of a <**div**> element, e.g.: {**height**: 100px;}.
margin	Defines the margin for all four sides in one declaration. First value relates to top margin, second value to right margin, third value to bottom margin and fourth value to left margin, e.g.: {**margin**: 10px 20px 8px 15px;}.
padding	Defines the padding for all four sides in one declaration. First value relates to top padding, second value to right padding, third value to bottom padding and fourth value to left padding, e.g.: {**padding**: 10px 20px 8px 15px;}.
display	Allows for an element to be displayed as inline element (e.g. <**span**> element) or as block element (e.g. <**p**> element). E.g.: {**display**: inline;} or {**display**: block;}.
clear	The clear property defines the side(s) that another floating HTML element is/are not allowed to be positioned, e.g.: {**clear**: left;}, {**clear**: right;} or {**clear**: both;}.

Background And Border Properties

Property Name	Description
background-color	This property defines the background color of a or <body> element. The colour may be specified using

Property Name	Description
	one of the predefined HTML color names, hex value or RGB value, e.g.: {background-color: silver;}, {background-color: #c0c0c0;} or {background-color: rgb(192,192,192);}.
background-image	Allows the use of an image as background image of a or <body> element, e g.: {background-image: url("../images/css/wrapper-bg.jpg");}.
background-position	Sets the position of a background image, e.g.: {background-position: left top;}, {background-position: left center;}, {background-position: left bottom;}, {background-position: right top;}, {background-position: right center;}, {background-position: right bottom;}, {background-position: center top;}, {background-position: center center;}, {background-position: center bottom;}, {background-position: 33% 33%;} or {background-position: 10px 20px;}.
background-repeat	Specifies whether a background image is repeated (tiled) and how it is repeated, e.g.: {background-repeat: no-repeat;}, {background-repeat: repeat-x;} or {background-repeat: repeat-y;}. By default, a background-image is repeated vertically and horizontally.
border	Defines the border for all four sides of a in one declaration. First value relates to border width, second value to border style and third value to border colour, e.g.: {border: 2px solid silver;}.

Textual Content Properties

Property Name	Description
font-family	Defines a prioritised list of font names and/or generic font family names, e.g.:

Property Name	Description
	{**font-family**: Arial, Helvetica, sans-serif;}, {**font-family**: Geneva, Arial, Helvetica, sans-serif;}, {**font-family**: Tahoma, Geneva, Helvetica, sans-serif;}, {**font-family**: Verdana, Arial, Helvetica, sans-serif;}, {**font-family**: Georgia, "Times New Roman", Times, serif;}, {**font-family**: "Times New Roman", Times, serif;}, {**font-family**: "Courier New", Courier, mono;}or {**font-family**: "Comic Sans MS", cursive, sans-serif;}.
font-size	The size of the displayed font, e.g.: {**font-size**: 12px;}.
color	Defines the text colour. The colour may be specified using one of the predefined HTML color names, hex value or RGB value, e.g.: {**color**: blue;}, {**color**: #000080;} or {**color**: rgb(0,0,128);}.
font-weight	Weight of the displayed font, e.g. {**font-weight**: bold;} or {**font-weight**: normal;}.
font-style	Defines the font style, e.g. {**font-style**: italic;} or {**font-style**: normal;}.
text-decoration	Sets the text decoration, e.g. {**text-decoration**: underline;}, {**text-decoration**: line-through;} or {**text-decoration**: none;}.
text-align	Aligns content, not only text, within an HTML element, e.g. {**text-align**: left;}, {**text-align**: center;} or {**text-align**: right;}.
text-indent	Indents the first line of a block of text, e.g.: {**text-indent**: 35px;}.
letter-spacing	Sets the spacing between individual character, e.g.: {**letter-spacing**: 2px;}. Negative values are allowed, e.g.: {**letter-spacing**: -1px;}.
word-spacing	Sets the spacing between individual words, e.g.: {**word-spacing**: 2px;}.

Property Name	Description
line-height	Sets the height of a text line, e.g.: {line-height: 20px;}.

List Properties

Property Name	Description
list-style	Defines all the list style properties in one declaration. Properties for elements: {list-style: decimal;}, {list-style: lower-alpha;}, {list-style: upper-alpha;}, {list-style: lower-roman;} or {list-style: upper-roman;}. Properties for elements: {list-style: none;}, {list-style: disc;}, {list-style: circle;} or {list-style: square;}}.
list-style-image	Defines the image used as a list style marker, e.g.: {list-style-image: url ("images/marker.gif");}.

JavaScript Operators

Operator Function

x + y	Adds x to y if both are numerical – otherwise performs concatenation
x – y	Subtracts x from y if both are numerical
x * y	Multiplies x and y
x / y	Divides x by y
x % y	Divides x by y, and returns the remainder
-x	Reverses the sign of x
x++	Adds 1 to x AFTER any associated assignment

++x	Adds 1 to x BEFORE any associated assignment
x-	Subtracts 1 from x AFTER any associated assignment
--x	Subtracts 1 from x BEFORE any associated assignment

Comparison Function

x == y	Returns true if x and y are equivalent, false otherwise
x != y	Returns true if x and y are not equivalent, false otherwise
x > y	Returns true if x is numerically greater than y, false otherwise
x >= y	Returns true if x is numerically greater than or equal to y, false otherwise
x < y	Returns true if y is numerically greater than x, false otherwise
x <= y	Returns true if y is numerically greater than or equal to x, false otherwise

Appendix 3: Permanent Storage

Note: permanent storage requires running your web app from "localhost://". This means setting up a web server on your desktop machine in order to test & run your app, not running it purely as a file "file://" . Setting up a web server is a slightly complicated process, but if you want to have a try of it, there are plenty of tutorials on the web. For Microsoft Windows, it requires downloading software. For OS X, there is a web server inbuilt, but it needs to be configured (using the command-line).

JSON

There are times when we want to take a group of data (e.g. an Object or an Array) and collapse it into a single String, usually because data on disk systems or over networks are stored as Strings. In very special circumstances, we may even collapse the data into a raw byte array, though that is too advanced for this unit.

The process of collapsing an in-memory object into a storable stream of data, ready to save somewhere, is often called **data serialisation**.

Whenever we convert a complex data object into a String, we need an encoding. An encoding, much like HTML is, and CSS is, is simply a specification for the rules of grammar that the text file will follow. We need a set of rules to encode, so that we can later decode it successfully, and convert it back to in-memory objects. Examples of encoding are CSV (comma-separated value files), XML, and JSON.

JSON is one such standard, in fact, the most common standard of data formatting for JavaScript. The JSON format is in fact extremely similar to the JavaScript syntax for describing objects. Take a look at this example of a JSON file:

```
{
	"name": "Henry",
	"score": 10,
	"finished": true
}
```

This syntax is extremely similar to JavaScript's syntax for defining an object, which is why it's so popular in JavaScript programs. The only real difference is that all keys **MUST** be in double quotation marks. In JavaScript, this is optional. Aside from that, everything else is mostly the same.

Now, if you imagine for a moment, the above code is an object in memory, with 3 keys (or "fields") of data stored within that one object. 3 pieces of information contained in one location. When we serialise this and convert it to JSON format, this gets collapsed into a String, essentially it is taking those 3 key:value pairs as part of an object, and encoding them into lines of **text**, which on a file system look like this:

```
"{\n\"name\":\"Henry\",\"score\":10,\n\"finished\":true\n}"
```

In a text editor, any new-line character "\n", is rendered as an actual line break. As a pure file however, a line break is simply a character. Everything in a String is a character. And everything in a text file is a character as well (as a text file is simply one really long String, it's nothing else). As a side note, because this data is in a String, anytime there is a quotation mark for a String data type inside, the quotation character is "escaped", meaning it is converted to \", as a " mark by itself would mean the end of the entire String, not the end of a String encoded within that text.

There are only two commands (functions) in JavaScript to do with JSON: JSON.stringify(), and JSON.parse().

```
// myData is my actual object in memory
var myData = { name: "Henry" };

// myString is now a String-representation (encoding) of that data
var myString = JSON.stringify(myData);

// let's display it
alert(myString);

// now let's convert the String back to an in-memory JavaScript
object
myData = JSON.parse(myString);
```

JSON can be used to store data and can be fetched to be stored in localStorage. A JSON file can be read using the JavaScript parse() function.

Example

```
<!DOCTYPE html>
<html>
<body>
<h2>Fetching from JSON String</h2>
<p id="demo"></p>
<script>
var jsonString = '[{"name" : "Nicole", "age" : "25"},{"name" :
"Aaron", "age" : "20"}]';
var obj = JSON.parse(jsonString);
document.getElementById("demo").innerHTML = obj[0].name + ", " +
obj[0].age;
</script>
</body>
</html>
```

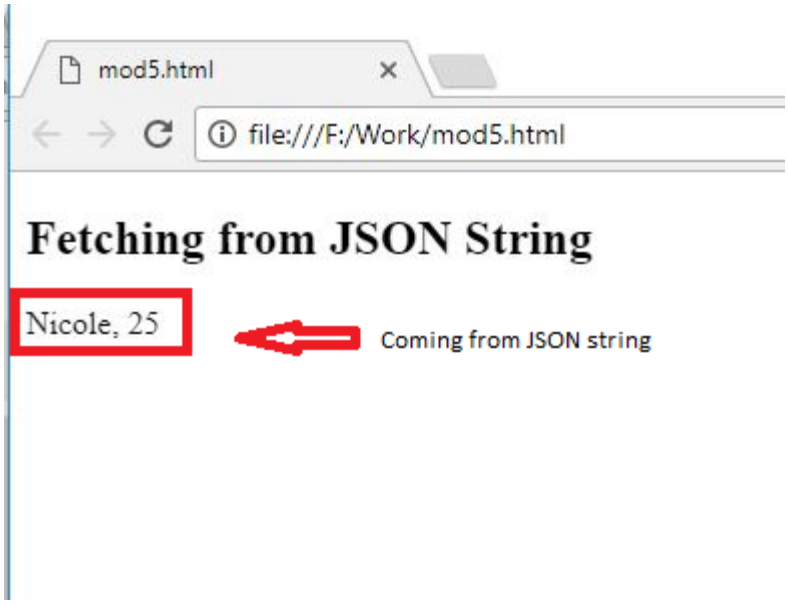


Fig: Resulting webpage

Here JSON string is defined as an array of objects. So, when parsed it will retrieve each rows of the array and can be accessed using indexes.

LocalStorage

Important: LocalStorage is only available when running your web app through a local web server, or on a desktop / mobile if compiled (through Electron / Cordova). As we'll only be using web browser in this unit, if you do want to test LocalStorage, you have to set up a local "Web Server" on your computer. There are free (open-source) ones out there, it just depends on your operating system. Setting them up takes a bit of command-line usage, so it's outside the scope of this unit.

But what it means, in short, is that you cannot access LocalStorage if you are accessing your index.html file through the "file:// …" URL. It only works for "http://" , e.g. "http://localhost/~YourComputerUsername/project2/index.html" (this only works if you've set up Web Server software on your local computer. It won't work otherwise).

HTML web storage provides two objects for storing data on the client's disk drive:

- *window.localStorage* - stores data with no expiration date
- *window.sessionStorage* - stores data for one session (data is lost when the browser tab is closed)

The *localStorage* object stores the data with no expiration date. The data will not be deleted when the browser is closed, and will be available the next day, week, or year. You cannot specify the file location, as you have no access to the file system. But within this localStorage

object, you can store information (usually up to a maximum of 5mb, which is pretty small actually, but that's to prevent many websites from using up users' hard drive space with irrelevant files).

Syntax:

To create:

```
localStorage.setItem("<key>",<value>);
```

To access:

```
localStorage.getItem("<key>");
```

Example:

```
// Store the value to a localstorage
localStorage.setItem("name", "John");
// Retrieve the content
document.getElementById("result").innerHTML =
localStorage.getItem("name");
```

Appendix 4: Cordova (compiling to mobile devices)

Overview

Cordova is a tool for building hybrid mobile applications using HTML, CSS and JavaScript. Cordova wraps your HTML/JavaScript app into a native container, a bare-basic empty app with a viewer to your web app). Cordova also provides a platform for accessing device functions (like camera, GPS, etc) of devices through JavaScript.

Setting up and using Cordova the first time is pretty complicated, and requires using the command-line. So only attempt this is you're quite confident.

You'll also need development environment(s) (called SDKs) for the platform you wish to emulate / compile to. Note that, while Android can be compiled & emulated on any computer (using Android Studio), iOS applications can only be compiled & emulated on Mac computers (using XCode).

Cordova Features

Following are some important features of Cordova platform:

Command Line Interface (Cordova CLI)

This tool can be used for starting projects, building processes for different platforms, installing plugins and lot of other useful things that make the development process easier.

Cordova Core Components

Cordova offers a set of core components that every mobile application needs. These components will be used for creating base of the app so we can spend more time to implement our own logic.

Cordova Plugins

Cordova offers API that will be used for implementing native mobile functions to our JavaScript app.

Installing Cordova

Cordova installation guide and the necessary resources can be found in the website - https://cordova.apache.org/

Following steps can be used to create Cordova CLI

1. Download and install Node.js. On installation, you should be able to invoke node and npm on your command line.
2. Install the cordova module using npm utility of Node.js. The cordova module will automatically be downloaded by the npm utility.

   In Windows:
   ```
   C:\>npm install -g cordova
   ```

   In Mac
   ```
   $ sudo npm install -g cordova
   ```
3. Following installation, you should be able to run cordova on the command line with no arguments and it should print help text.

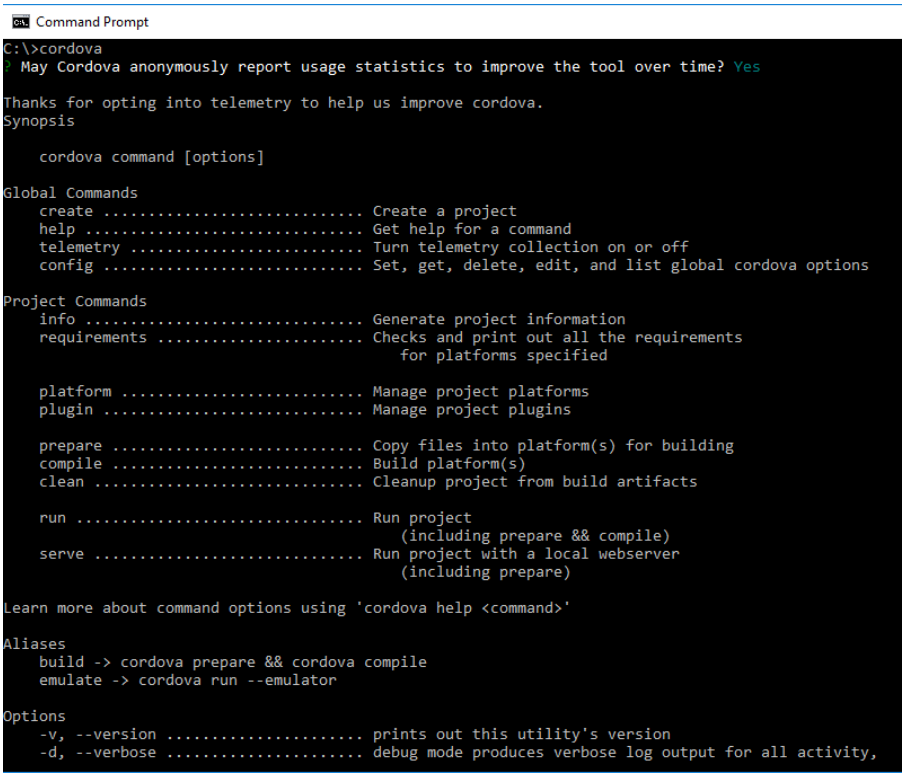


Fig: Command prompt of showing successful installation Cordova

Cordova: Installing SDKs

For Cordova to work, it needs the framework for the device you are targeting: either Android or iOS. Cordova doesn't make these platform-specific tools, as they are provided by the platform developers (Google, who make the Android platform, and Apple, who make the iOS platform).

Android SDK

For Android platform, you need to have Android SDK installed on your machine. Following is the list of software's you will need before you start your Android application programming.

- Java JDK8 or later version
- Android Studio

You can download the latest version of Java JDK from Oracle's Java site – Java SE Downloads. You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Finally set PATH and JAVA_HOME environment variables to refer to the directory that contains java and javac.

The Android Studio can be downloaded for free from the following website - https://developer.android.com/studio/index.html and follow the instructions when asked during the installation process. Note that you may be required to mention JDK7 path or later version in android studio installer so make sure you have installed the JDK before installing Android studio.

XCode

For iOS platform, you need to have a Mac, and then install the XCode software on your machine. You can find XCode in the Mac App Store.

Creating and Emulating the Mobile App

You can create and emulate a mobile app using Cordova. For this you need a working directory and go to this to create your cordova project.

Open the cmd or terminal in the work directory and do the following:

```
cordova create project1 com.introtoapps.app1 app1
```

This will create a project named *project1* as the directory with *com.introtoapps.app1* as your reverse domain and *app1* as the title of the app. This will create the default project structure in the *project1* directory as below:

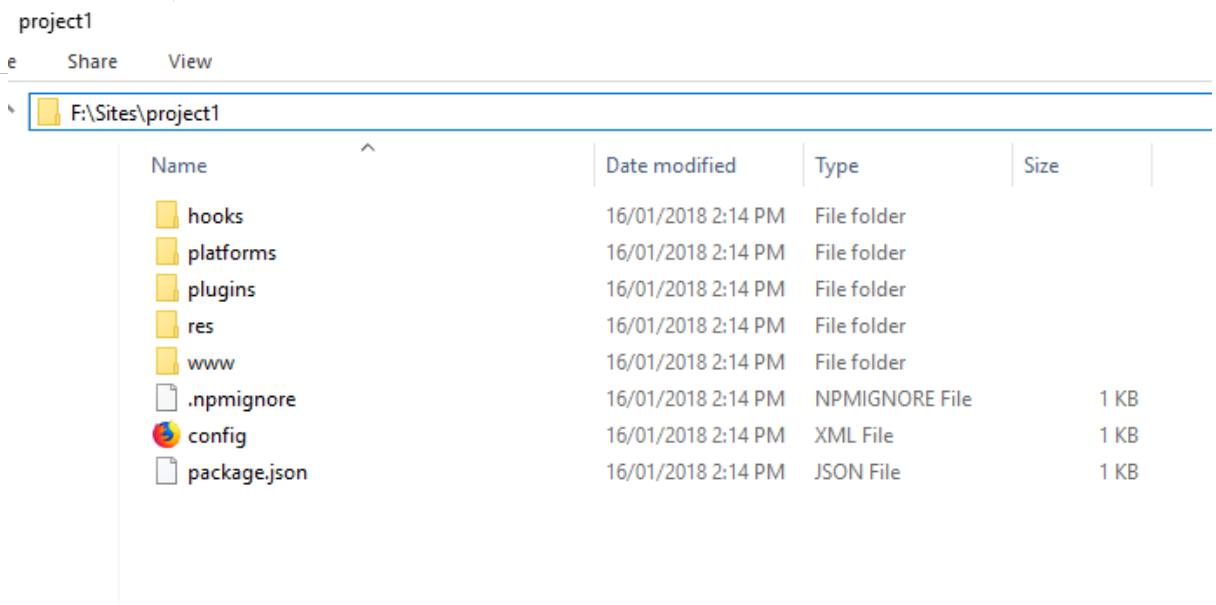


Fig: Project directory after the project has been added by Cordova

All the html files (index. html) and other project related files will be usually in *project1/ www* folder. The index.html will be the homepage for your web application.

Adding Platforms

To create the app in different platform, you need to add the platform to your project. The following is how it can be done.

Go to your project folder (here project1) in terminal (Mac) or cmd (Windows) and perform the following:

```
cordova platform add ios
cordova platform add android
```

Note that you may need to install suitable SDKs for each platform for to be able to run/emulate the app in the respective platform. Check the requirements for each platform by typing the following:

```
cordova requirements
```

Refer to appendix for more information on adding SDKs for each platforms

Build and Emulate App

To build and emulate the app, the following commands can be used:

```
cordova build
```

This will build the app for all platforms added to the project. After the successful building of the app, you can emulate the app by the following command:

```
cordova emulate android
cordova emulate ios
```

The app will be initialized and the default emulator will show the following screen (for android):

Fig: Cordova emulated mobile app (initial screen)