

# Institute <sub>of</sub> Data

---

2024



# Logistics

- Zoom and Slack

Primary class communication happens via Zoom during class sessions. Slack is also used for general announcements and public/private messages between classes, please check it regularly.

- Breaks

Regular breaks are important! Each cohort has flexibility to set them at suitable times.

- Questions

Asking questions is an essential part of learning and a vital way to stay engaged. Don't wait and fall behind, ask early and often!



# Introductions

- Please share with the class:
  - Current role and background
  - Why are you here?
    - Your **objectives and expectations** of attending the course
  - Your current skill levels in:
    - **Programming**
  - Other related areas (if applicable to you):
    - Information Technology
    - Business domain knowledge
  - Your **experience completing the prerequisites**

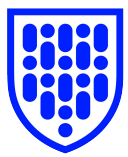


# What is a Software Engineer's job?

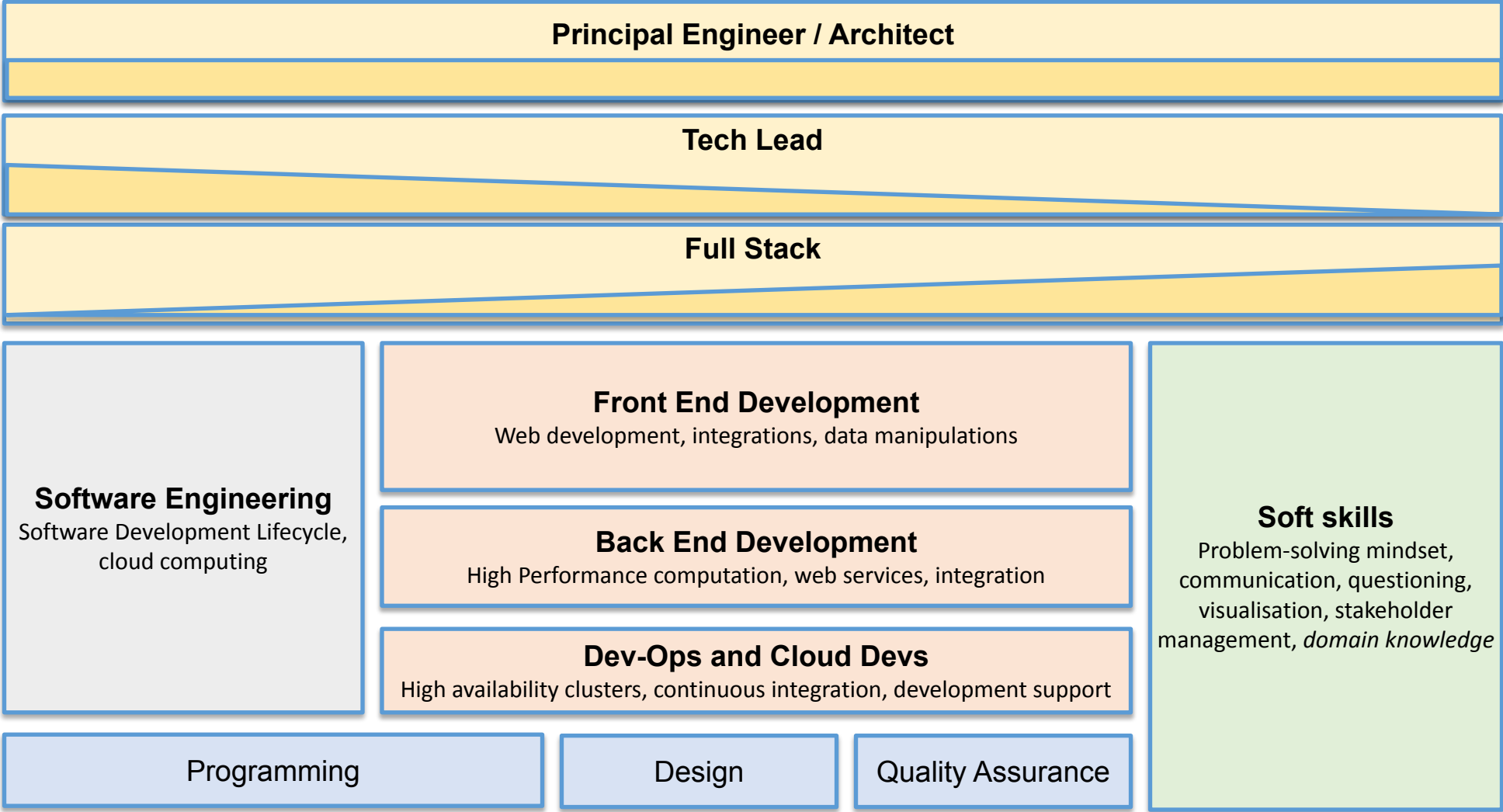
In simple terms, we look at the world around us and create digital solutions to solve problems.

Specific tasks include:

- **Identify the problems** that offer the greatest opportunities to the organisation
- **Understand** the user using a **user-centric** approach
- Work towards building the **right software** and **building it right**
- **Understand technology** and find ways to **solve old problems** in more efficient ways
- **Design applications** that satisfy user requirements and provide **delightful experiences**
- Build software understanding the **concepts and tradeoffs of performance**
- **Solve** real-world problems
- Work with a **team** to generate **impact**



# Software skills for industry





# Software Engineering Course Objectives

By the end of the Software Engineering program you will be able to:

*Help businesses to make effective decisions and track their effectiveness using the appropriate combination of the following tasks:*

- Collect **user requirements** and transform them into **software specifications**
- Design applications that are **user-centric**, helping you to build the right software
- Create **design prototypes** using industry standard applications such as Figma
- Understand the role of data and how to **design data-driven** applications
- **Develop web applications** which are highly available, well-designed and efficient.
- **Develop robust backends** to support the data driven economy
- Identify key **organisational problems** and design software to solve them
- Apply the skills you have learnt and become a **digital problem solver**



# Software Engineering skills for industry

- **Foundational skills** that are required to learn Software Engineering:
  - Programming
  - Problem solving
  - User-centric approach
  - Application Design
- **Domain expertise**
  - Backend Design and implementation
  - Frontend Development
  - High-performance computations
  - Devops and Cloud Development
  - User-centric Design



## SOFTWARE INDUSTRY ROLES

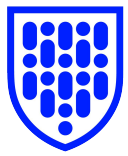
Web Developer	Junior Software Developer
Front End Developer	Back End Developer
UI Developer	JavaScript Developer
React Developer	Product Engineer
Quality Assurance Analyst	IT Consultant
Programmer Analyst	App Designer





# Software Engineering: Course Overview

<b>Foundational Skills</b> <ul style="list-style-type: none"><li>• Software Development Principles</li><li>• JavaScript Fundamentals</li></ul>	<b>Core Software Engineering Skills</b> <ul style="list-style-type: none"><li>• Introductory Front-end Development</li><li>• Intermediate and Advanced JavaScript programming</li><li>• Interactive Web Design</li><li>• Back-end and Full Stack Development</li><li>• React</li><li>• Databases</li><li>• API Development</li><li>• Software Deployment</li><li>• Software Engineering industry practices</li></ul>	<b>Applying Software Engineering in Industry</b> <ul style="list-style-type: none"><li>• Applying software engineering in different domains</li><li>• Defining a software engineering project</li><li>• Designing a software engineering project</li><li>• Delivering a software engineering project</li><li>• Overall end-to-end solution</li><li>• Presenting to stakeholders and obtaining buy-in</li><li>• Capstone project</li></ul>
<b>Interpersonal Skills</b> <p>Critical Thinking, Questioning, Documenting, Presenting, Job Outcomes</p>		
<b>Learning how to learn effectively framework</b> <p>Minimal Viable Learning (MVL), Multimodal learning, Learn-Create cycle</p>		



# Software Engineering

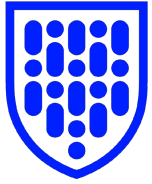
Module 1

Part 1:

---

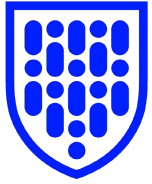
## Introduction to Programming

---



# Introduction to Programming

- Getting setup
- The basics of Unix commands
- Your Code Editor



# Software

A developer needs to know a number of software tools. We are going to be using **NodeJS** as our programming runtime environment, **VS Code** as our code editor, and **Git** as our code repository.

These are just a few of the essential tools designed to make software development easier. As you move forward in your career, you will find that a major aim is to make your life easier, and as Bill Gates says:

**“I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it.”**





# What is Node.js?

**Node.js** is an **environment** that runs **JavaScript** code.

JavaScript runs natively in a web browser, but if we want to write code that can run outside of this (eg. a back-end server application) we need a tool that can **compile** and **execute** the code.

*Node.js is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.*

It allows JavaScript developers to be full-stack software engineers who can use JS code to create both front and back end applications.



Node.js is an environment that processes your code. It runs in the background, without any graphical interface you can see.

You will only interact with it via a command prompt, when we're ready to run our code.

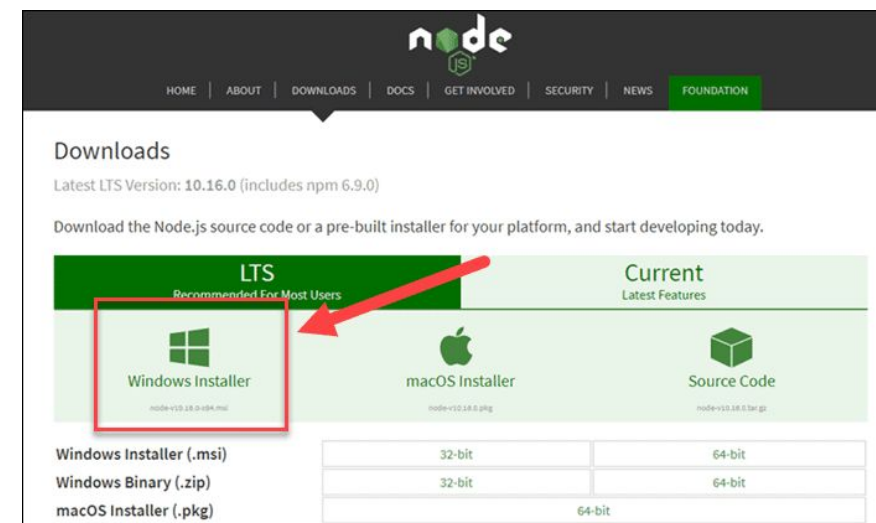
So you won't see any changes after installing it, until we start writing code.



# Installations - Node

## How to install node.js on Windows or Mac?

1. Navigate to <https://nodejs.org/en/download/>. Click the Windows/MacOS Installer button to download the latest default version.
2. Once the installer finishes downloading, launch it. Open the downloads link in your browser and click the file. Or, browse to the location where you have saved the file and double-click it to launch.
3. The system will ask if you want to run the software – click Run.
4. You will be welcomed to the Node.js Setup Wizard – click Next.





# Installations - Node (continued)

5. On the next screen, review the license agreement. Click Next if you agree to the terms and install the software.
6. The installer will prompt you for the installation location. Leave the default location, unless you have a specific need to install it somewhere else – then click Next.
7. The wizard will let you select components to include or remove from the installation. Again, unless you have a specific need, accept the defaults by clicking Next.
8. Finally, click the Install button to run the installer. When it finishes, click Finish.

## Command Terminal

On Windows, search for 'PowerShell' or 'Command Prompt' from the bottom left search bar. On Mac, use the 'Terminal' tool. These programs will let you type various commands.

## Verify your installation

Using a command terminal, enter the command **node -v**

```
PS C:\> node -v
v14.17.3
PS C:\> npm -v
6.14.13
```

If you get some numbers, it means your installation is successful.



# What is Git?

**Git** is a tool for storing and **managing files**, and **tracking changes** across versions, features and releases.

It is the leading **Source Code Management** (SCM) platform for managing projects of any size and complexity.

*Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.*

It allows developers to regularly save their changes and revert them, tracking individual revisions, experimental features and official releases across a large codebase with multiple team members safely and efficiently.



Git is an environment that manages your code. It runs in the background, without any graphical interface you can see.

You can interact directly with it using a command prompt.

So you won't see any changes after installing it, until we start managing our code.





# Installations - GIT

- You can install git for **Windows** by simply navigating to <https://git-scm.com/download/> and downloading the latest version, then running the file and finalising the setup. **Mac** users may first need to install **homebrew** and then follow the instructions from the link.
- You can verify that the installation was successful by simply opening your PowerShell and using the command `git --version`.
- You will see an output on your screen with the git version installed.

```
PS C:\> git --version
git version 2.32.0.windows.2
PS C:\>
```

- If the system says that it doesn't recognise the command git, just restart your system and it should work.
- *In your own time, read through [Chapter 1 of the Git Handbook](#) to understand more*



# GitHub

Git is a free version control system designed to handle storage and management of projects of any size. It takes some getting used to, but is simple to start with and we will continue to build up our skill set during the course.

Once you've installed Git locally, we also need to create a free cloud-based **Github** account. This will allow you to **synchronise** your **local** code repositories with a secure **remote** developer platform, making it easier to share your code, track updates, and avoid accidentally losing it.

Github provides unlimited repositories and best-in-class version control, on the world's largest developer platform. It's a vital tool for any developer, so sign up and create your free account at [github.com/signup](https://github.com/signup), then share your username with your trainers.

Welcome to GitHub!  
Let's begin the adventure

Enter your email\*

→

Continue



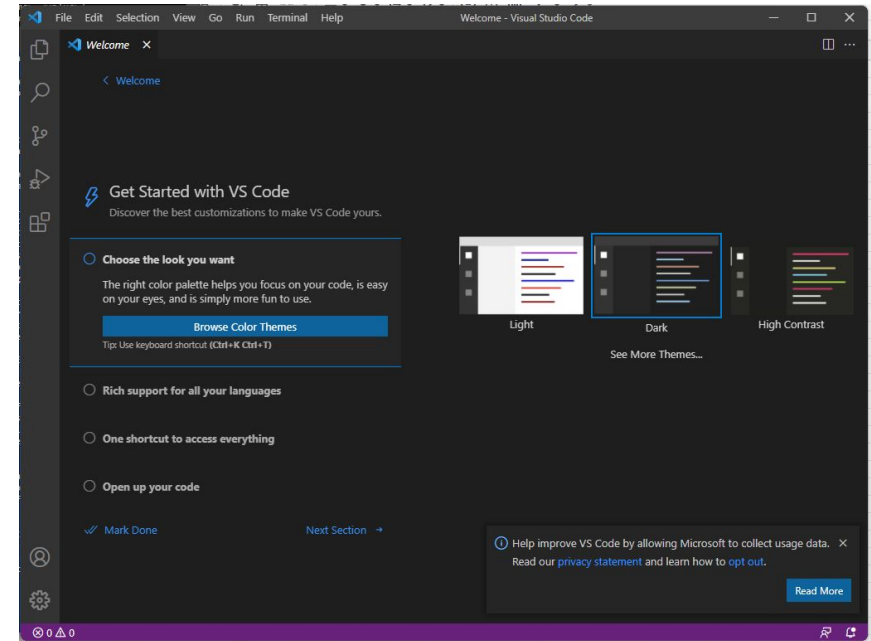
# Installations – Visual Studio

Microsoft's **Visual Studio Code** is a popular source-code editor for Windows, Linux, and macOS. Support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git are just a few of the features.

You can simply install it by going to the link below. Once you have downloaded the appropriate version for your machine, you can run the setup file and finish setup.

You can verify that VS Code has been installed by just running the VS Code application.

<https://code.visualstudio.com/download>



VS Code is a graphical interface tool that helps developers write, run and debug their code.

We will use it throughout the course, so you will have plenty of chances to get familiar with it.

# Basic Linux Commands

- Linux-based operating systems are common and widely used for developing and deploying software applications
- It is essential to know basic Linux commands as a software engineer, as it is more likely that you will find a remote server running without a GUI than with one.
- In this session we will learn essential basic Linux commands that are frequently used when operating Linux based systems. There are MANY more which can be referenced as needed, but these basics will help with some of the most common tasks.



This Photo by Unknown Author is  
licensed under [CC BY](#)



# Command Line Interfaces

You are probably used to using programs or apps with a **Graphical User Interface** (GUI). This includes things like Microsoft Office, any web browser, image editing programs, even basic ones like Notepad. Many software development environments, particularly in production, do not have access to these, or require tools (eg. Node.js and Git) that do not use a GUI.

Instead they require using a **command line interface** (CLI). This is sometimes referred to as a **shell**, a **terminal**, or a **console** - these terms are mostly interchangeable.

## What is a Shell?

A shell is a **program** that lets you interact with your computer's operating system using **text** commands. It acts as a middleman between you and the system, interpreting and executing your commands.

Common shells include **Bash** on Unix-like systems, **Command Prompt** or **PowerShell** on Windows, and **Terminal** (using Bash, Zsh, or other shells) on macOS.

A **terminal** provides a user interface.

A **console** can refer to physical hardware or a text-based interface.

A **shell** is a command-line interpreter that processes commands.

The **command line** is the method of entering textual commands into the shell.



# File system navigation

In order to navigate around folders and create new folders, we use various commands in our command terminal. Let's see a few of them by using **PowerShell**. You can access this by searching for it on Windows, or also use Git Bash (from the Start menu) or the Terminal (Powershell) that comes with VS Code.

First, it is important to be aware of the **current location** our commands are being run from - also called our **working directory**. The command prompt (where your cursor is) will usually indicate this for you, eg:

```
PS D:\IOD\SE> |
```

*(**folder** and **directory** are just different names for the same thing)*

All commands entered at the above prompt will be run from the **SE** folder, which is located in the **IOD** folder on **D** drive. Yours will probably be different - this is fine! Any changes to the current directory or creation of new directories will happen from here.



# File system navigation commands

- To create a folder (**m**ake a **d**irectory):

**mkdir** <foldername>

```
PS C:\> mkdir newfolder
```

- To **c**hange **d**irectory:

**cd** <foldername>

```
PS C:\> cd newfolder
PS C:\newfolder>
```

- To change up to the parent folder, use double dots:

**cd** ..

```
PS C:\newfolder> cd ..
PS C:\>
```

- To **l**ist files in a directory:

**ls**

```
C:\> ls
```

- To **p**rint the current folder and path (the **w**orking **d**irectory):

**pwd**

```
PS C:\Users> pwd
Path
----
C:\Users
```

- To delete (or **r**emove) a file or folder:

**rm** <filename>

```
PS C:\> rm test.txt
```



# File system utilities

These are just a few basic commands - there are MANY others, and they can be combined to perform complex shell scripting tasks. Although this is outside the scope of the course, the more you know, the better you will be able to control your applications across any server.

Many commands also have **options**, that modify their behaviour slightly. These are usually specified using a dash - (or minus/hyphen), followed by various letters.

For example, if you try to remove a folder which is not empty, it may fail or ask for confirmation. We can instead do **rm -rf** to specify **recursive** behaviour with the **r** option, and **forced** removal with the **f** option, so that the folder with all its contents is removed. *Be careful with this one!*

## Dots and slashes

A single dot **.** when navigating refers to the current path, or folder.

Double dots **..** refers to the folder immediately above the current folder in the directory tree.

A forward slash **/** is a separator between folder and file names.

So **./test.txt** refers to a file called test.txt in the *current* folder.

**../test.txt** refers to a file called test.txt in the *parent* folder.

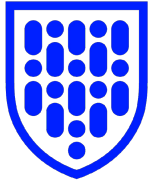




# File system utilities

Some tips:

- When using commands that reference a file path, most terminals will let you press **Tab** to auto-complete the folder or file name.  
**Try it:** type **cd** followed by a space, and then press **Tab**. It should auto-complete the first matching folder - if you press Tab again it will cycle through available options
- Folders or files with **spaces** in the name need to be **quoted**, since a space has meaning for all of these types of commands - it can be best to avoid spaces in your file/folder naming
- Different operating systems may implement commands slightly differently
- On Linux-based systems, you may need certain permissions to execute commands. Prefixing your command with **sudo** will temporarily grant **super-user** access to **do** the command, and is a common trick for running server-based commands.

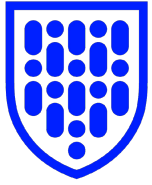


# Lab Exercise 1

Practice using the commands you learnt to solve the following.

- Create a new folder
- Create another folder inside the first one
- Print the contents of the first folder
- Change directory to the second folder, and print the current path
- Change directory back to the original starting place
- Delete the first folder

*Extension: Research the **cat** and **find** commands, and experiment!*

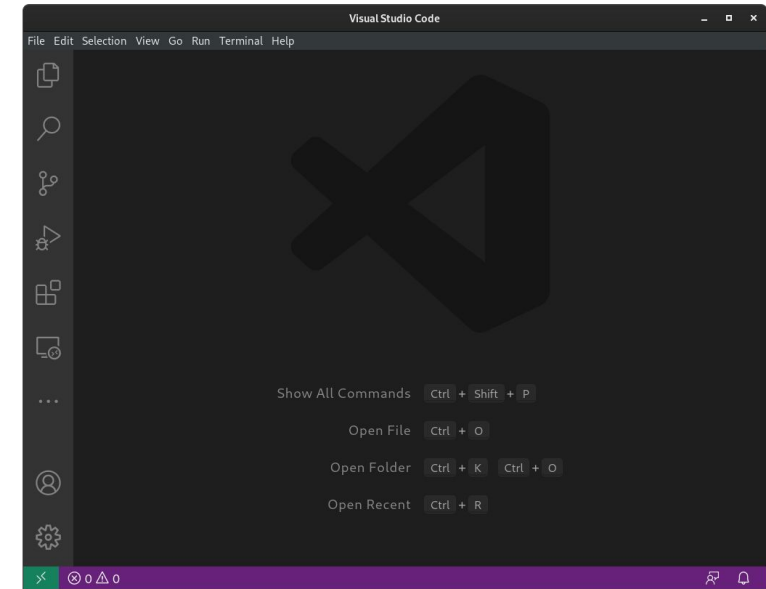


# Your Code Editor – VS CODE

Visual Studio Code is a source-code editor (not an IDE) made by Microsoft for Windows, Linux and macOS.

Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git.


The key strengths of VS Code are the performance and the community. It is one of the most versatile environments for starters.



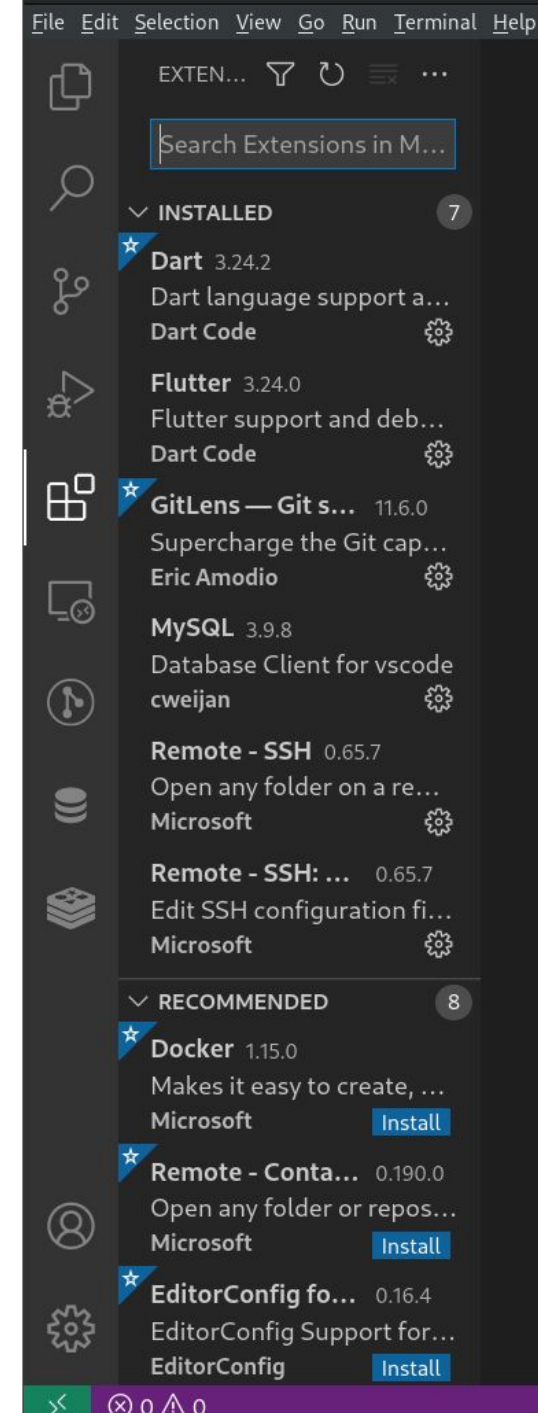


# VS Code : Extensions

Extensions are what make VS Code extremely powerful and pleasant to use. A vast community of developers has created thousands of extensions to make a programmer's life easier, from colour themes, to automatic code snippets, to auto-formatting, to almost everything. Once you become competent, you will be able to create the dev environment of your dreams!

Open the extension tab by clicking on  or press Ctrl-Shift-X

Also check out <https://docs.emmet.io/cheat-sheet/> for some time saving shortcuts.

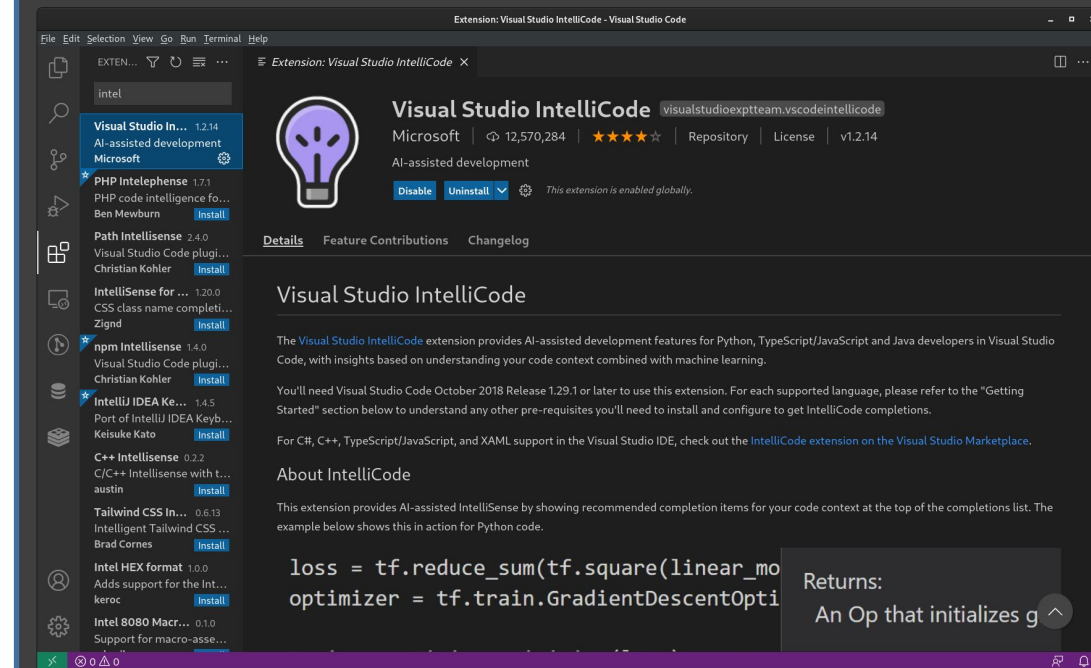




# VS Code : Extensions

Here is a list of some simple and effective extensions you can try even now.

- Intellisense
- Remote – SSH
- Prettier
- Gitlens
- JavaScript Code Snippets

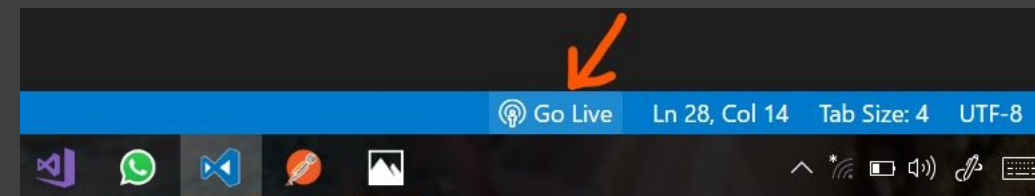
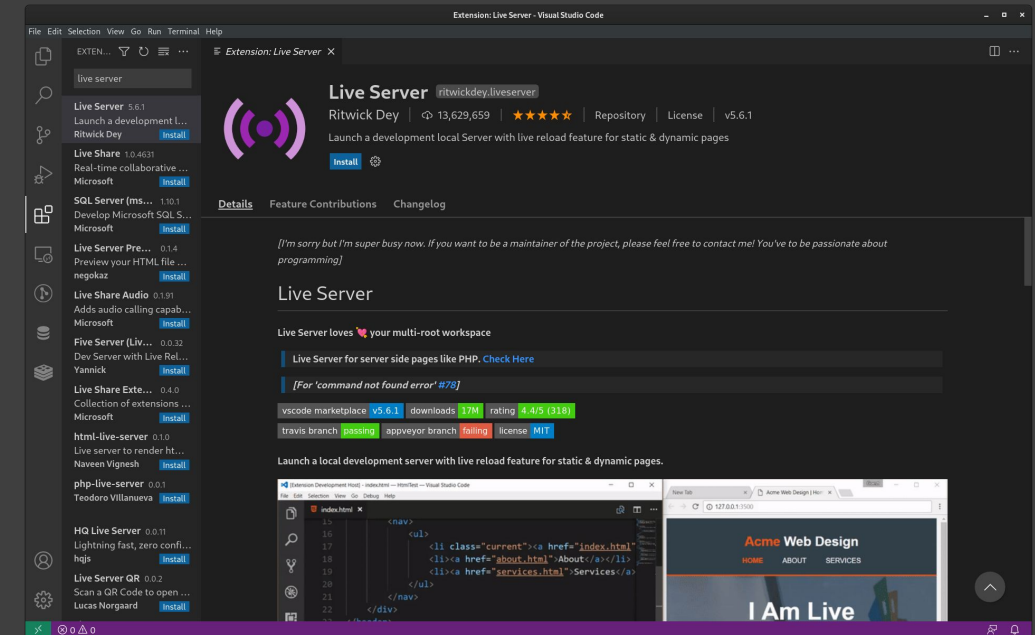


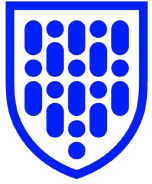


# VS Code : Live Server

Live Server is one of the great extensions available in VS Code, which enables you to run your HTML code on a local server with auto-reload.

- Install **Live Server** from the Extensions tab
- Run **Live Server** by clicking on Go Live in the status bar of VS Code



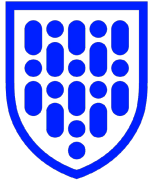


## Lab Exercise 2

- Create a folder for storing all IOD related work
- Open VS Code, then use Open Folder to open the above folder. Create a new folder for **Module1**. Inside this folder, create a new file, call it **helloWorld.html** and copy this code into that file.

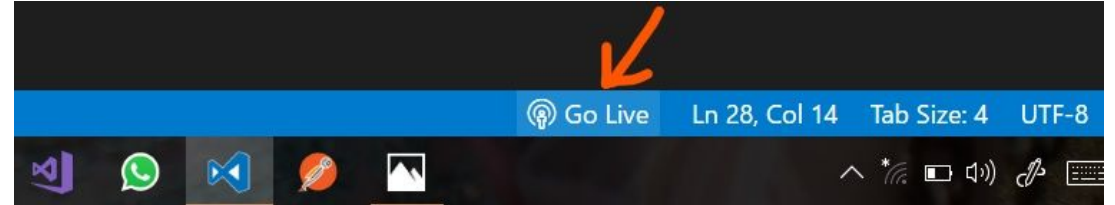
```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World!!!</h1>
  </body>
</html>
```

- Save the file



## Lab Exercise 2 continued

- Find the **Go Live** button (should be on the bottom right corner of the VSCode window)



- Press “Go Live”, if there is a prompt window, choose “Yes”. Then, we are live! Try making changes and saving them.



**Hello World!!!**





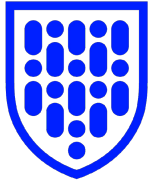
# Software Engineering

Module 1  
Part 2:

---

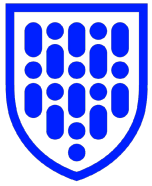
## Web Development 101

---



# Agenda: Module 1 Part 2

- Definition
- HTML Structure
- HTML Basics
- HTML Attributes
- HTML Styles ( Introduction to CSS )
- Lab – HTML Practice.



# Definition - What is HTML

HTML is the standard markup language for creating Web pages.

For an HTML file, the extension should be `.html`. It stands for **H**yper**T**ext **M**arkup **L**anguage, and it forms the foundation of all web pages. It is an “interpreted” language, as it follows a structure that is then read by the machine and “**rendered**” in your browser.

## How to edit HTML ?

Any text editor can work with HTML. Even Notepad/text editor. We will be using Visual Studio Code to do so.

*This section will give a brief introduction to HTML, but we will return to it in later modules.*





# HTML Structure

HTML uses the concept of **Tags** – these tags can be nested into one another, and together, they form the **document**. All tags (except for a few exceptions) are **opened** and **closed**, meaning that inner content is limited to within that tag. Here we have some examples.

The `<!DOCTYPE html>` declaration defines that this is an HTML5 document

The `<html>` tag is the root element of an HTML page

The `<head>` tag contains meta information about the page

The `<title>` tag specifies a title for the HTML page (which is shown in the browser's title bar or in the page's tab)

The `<body>` tag defines the document's body, and is a container for all the visible contents, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.

The `<div>` tag defines a division or a section in an HTML document.

The `<h1>` tag defines a large heading

The `<p>` tag defines a paragraph

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Page Title</title>
</head>
<body>
  <div>
    <h1>This is a heading</h1>
    <p>This is a paragraph</p>
  </div>
</body>
</html>
```



# HTML - Elements

These tags delimit what we call a **HTML element**, which is everything from the start tag to the end tag. Elements are also defined by their **class**, which is a styling property we will explore later.

`<tagname>` Content here `</tagname>`  
start tag      inner content      end tag  
————— HTML ELEMENT —————

```
<div>Content goes here...</div>
```

We can also build our elements in a nested way, for example, if we wanted to build an area with two areas within it, we would do something like the right:

The major advantage is the communication cost. The browser can create visually complex documents from a few bytes of code, instead of large Word documents or hi-res images which take longer to download.

```
<div>
  <div class="left">
    This is the left side
  </div>
  <div class="right">
    This is the right side
  </div>
</div>
```

This is the left side

This is the right side



# HTML - Headings

**Heading** elements, with other primitives, are some of the most basic building blocks. They are elements which have some predefined styles. Like any other element, they can be restyled manually.

Remember, HTML was first created in the 90s. Back then, communication costs were extreme. For this reason we had premade styles, so that people could create documents without transferring styles as well.

```
<h1>This is heading 1</h1>  
<h2>This is heading 2</h2>  
<h3>This is heading 3</h3>  
<h4>This is heading 4</h4>  
<h5>This is heading 5</h5>  
<h6>This is heading 6</h6>
```

**This is heading 1**

**This is heading 2**

**This is heading 3**

**This is heading 4**

**This is heading 5**

**This is heading 6**



# HTML - Hyperlinks

If the pages are hyper, then links should be too! Hyperlinks are links to other pages, not just online, but anywhere that can be mapped through the network. This is normally a page URL, for example, [www.google.com](https://www.google.com). The `<a>` tag, also referred to as the “anchor” tag, is an **html primitive**, meaning it has a default, predefined behavior. For example, most browsers will make it [blue and underlined](#).

We create an anchor tag in the following way.

```
<a href="https://google.com">Click here!</a>
```

The `<a></a>` tag is the **element**.

The href **attribute** indicates the link's destination.

The default behaviour of an anchor tag includes a **click event**, so clicking it will take you to your destination.





# HTML - Images

Remember, HTML is a language to make web pages superlight. Adding images would break that model, so what we do instead is create a tag to load an image.

Use `<img>` to add an image.

The `<img>` tag has two required attributes:

**src** - Specifies the path to the image

**alt** - Specifies an alternate text for the image

```

```

## Class Exercise

- Use the example provided to create your own page.
- Try to add a second image
- Research and experiment on how to make both of them the same size.

```
<html>
<body>
<h2>Image Example</h2>

</body>
</html>
```







# HTML - Tables

When dealing with the “interpretation” part of the browser, you may find that behavior is not always consistent, and things may not appear as you planned, but there is one structure which is always risk-free, the **Table**. The Table tag allows you to create a tabular row/column structure which is fixed and highly customisable. If you need consistency for grid-based data, this is the way to go. Experiment using the table.

The **<table>** tag defines an HTML table.

- Each table row is defined with a **<tr>** tag.
- Each table header cell is defined with a **<th>** tag.
- Each table data/cell is defined with a **<td>** tag.
- By default, the text in **<th>** elements are bold and centered.

Tables are great for fixed size data grids on larger screens, but not so good at adapting for small screens.

```
<table border="1">
  <tr>
    <th>Row 1 column 1</th>
    <th>Row 1 column 2</th>
    <th>Row 1 column 3</th>
  </tr>
  <tr>
    <td>Row 2 column 1</td>
    <td>Row 2 column 2</td>
    <td>Row 2 column 3</td>
  </tr>
</table>
```

Row 1 column 1	Row 1 column 2	Row 1 column 3
Row 2 column 1	Row 2 column 2	Row 2 column 3



# HTML - Forms

The HTML **form** is another important primitive, but using it requires that you understand more complex concepts, such as internet methods (HTTP methods) to fully understand and appreciate. For now, you can take for granted the following:

- The Form takes the data from the **input** fields within it and stores it into a network package.
- Upon clicking the **submit** button, it will send the data to the url described in the **action** attribute.

This functionality has evolved over time, but it is still as used today as when it was created in the 90s.

In later modules we will explore more types of form fields such as checkboxes, radio buttons, drop-downs and more.

```
<!DOCTYPE html>
<html>
<body>
  <h2>HTML Forms</h2>
  <!-- If you click the "Submit" button, the form-data
        will be sent to a page called "/action_page" -->
  <form action="/action_page">
    <div>
      <label for="firstName">First name:</label><br>
      <input type="text" id="firstName"
            name="firstName" value="Joel">
    </div>
    <div>
      <label for="lastName">Last name:</label><br>
      <input type="text" id="lastName"
            name="lastName" value="Doe">
    </div>
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

## HTML Forms

First name:

Last name:



# HTML - Attributes

All HTML elements can have **attributes**, which provide additional information about elements.

For example, `<img>` tags have “src” and “alt” attributes.

The **src** attribute specifies the path to the image to be displayed.

**alt** specifies an alternate text for an image, if the image for some reason cannot be displayed. HTML elements can have many more attributes, some of which are common to all elements (such as **id** and **class**), and others which differ depending on the element being used.

`<img>` tags also have “**width**” and “**height**” which can set the picture size, and many more. On top of this you can also create your own attributes.

To find more about HTML elements premade attributes and how to use them you can visit the MDN [HTML elements reference](#).



MDN Web Docs is an open-source project documenting Web technologies, including CSS, HTML, JavaScript. They provide an extensive set of learning resources for beginning developers and a reliable reference for all developers.

MDN's mission is to provide a blueprint for a better internet and empower a new generation of developers and content creators to build it.



# HTML - Practice

You have now learnt a number of tags. Let's start putting them together into a new document.

- Within your Module1 folder (see Exercise 2), create a new html file and call it **index.html**
- Copy or retype the code here on the right, or create something similar.
- Click the Go Live button in the bottom right hand corner to launch the file in the browser.

Files with the special name 'index' are auto-loaded. Using this naming convention indicates that your file should be loaded by default when the folder is accessed.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>page title</title>
<meta charset="utf-8">
</head>
<body>
<header>
  <h1>Header</h1>
</header>
<section>
  <nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">Articles</a></li>
    </ul>
  </nav>
  <article>
    <h2>Article heading</h2>
    <p>Paragraph...</p>
    <a href="https://www.google.com/">More information...</a>
  </article>
  <aside>
    <h2>Related</h2>
    <ul>
      <li><a href="#">Article 1</a></li>
      <li><a href="#">Article 2</a></li>
    </ul>
  </aside>
</section>
<footer>
  <p>@copyright 2050 by nobody. all rights reversed</p>
</footer>
</body>
</html>
```



# HTML - Practice

If everything went according to plan, your page should look like the one here on the right. Quite boring! That's because we added no styling whatsoever.

But imagine, this is what web pages looked like in the early 90s. Common HTML elements are still rendered by browsers with these built-in basic styles that reflect their purpose.

Typically we want to override these defaults with our own custom styling.

**Next we will add some styles to our pages.**

## Header

- [Home](#)
- [Articles](#)

## Article heading

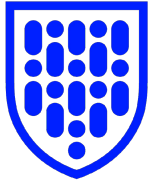
Paragraph...

[More information...](#)

## Related

- [Article 1](#)
- [Article 2](#)

@copyright 2050 by nobody. all rights reversed



# Cascading Style Sheets - CSS

- Definition
- Example of using CSS
- Lab – HTML+CSS (Inline/Internal/External) practice



# CSS

CSS gives our web pages the look and feel, the colours, the structure, the behavior, even animations. Most of the aspects related to anything visual about our page comes from CSS.

CSS is added to HTML elements in a number of ways, and most of the time, they define things such as the colour. For example, the following indicates that this specific div will have a red background colour.

```
<div style="background-color:red"></div>
```

CSS can be added to HTML documents in 3 ways:

**Inline** - by using the **style attribute** inside HTML elements

**Internal** - by using a **<style>** tag in the **<head>** section

**External** - by using a **<link>** element to link to an external CSS file





# CSS - Inline

One way, the quick and dirty way, is to add CSS directly to the styled HTML element, in what we call **inline styles**. This is a practice which can be good if you are making one-off changes in small projects, but it is in no way a recommended practice. Although the end result is the same, it is simply unfeasible to manage a project in this way, as it requires changing the HTML files directly, in multiple places, for every style change, making them unreadable and unmanageable.

**Here are some examples.**

Using the style attribute to set colour inside HTML elements:

## Class Exercise

Start practicing by adding some “color” to your **index.html** file using inline styles.

```
<p>I am normal</p>  
<p style="color:red;">I am red</p>  
<p style="color:green;">I am blue</p>
```

I am normal

I am red

I am blue

Colours can be specified in a few ways:

- using predefined colour names such as red, blue, pink, black, etc
- using hexadecimal colour codes such as #FFFFFF or #333333
- using RGB notation such as rgb(200,0,0)

Try a few options - we will look more closely at these later





# CSS – Internal and classes

The second way is to create separate CSS rules and keep these **style definitions** inside a reserved space - the `<style>` tag in the `<head>` section of your `.html` file.

CSS rules are associated with one or more HTML elements. An element can have many rules associated with it.

CSS **classes** can be defined once initially and then applied to any number of elements. We will come back to classes later. We can also change the **global classes** of the elements, like **H1**. In this way, we can keep our work cleaner, and we have a single place where we can change and define all our styling.

This is one step closer to recommended practice, but still, not something you should use except for personal or temporary work.

*Which HTML element/s does each of these CSS rules target?*

```
<!-- cssdemo.html -->
<!DOCTYPE html>
<html>
  <head>
    <style>
      body { background-color: black; color:
white; }
      h1   { color: blue; }
      p    { color: red; }
      .title { font-family: sans-serif; }
    </style>
  </head>
  <body>
    <h1 class="title">This is a heading</h1>
    <p>This is a paragraph.</p>
    <h2 class="title">Smaller Heading</h2>
  </body>
</html>
```

**This is a heading**

**This is a paragraph.**

**Smaller Heading**



# CSS – External File

The last way is the recommended method. We store all of our CSS into a separate `.css` file, one or many. This allows us to be very flexible, and make changes reliably and consistently. A normal size project may have tens of CSS and HTML files. By separating them, you can easily share the CSS around and link it from each relevant HTML file.

Also, we explore the important concept of **modularity**, by reusing the CSS anywhere we need it.

To do this, we need to link the separate HTML and CSS files, and we do this by using the `<link>` tag. This references the CSS file from the HTML file, as for the example on the right.

**Use `<link />` tags to add a CSS reference to an HTML document.**

```
<!-- cssdemo.html -->
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet"
href="cssdemo.css"/>
  </head>
<body>
  <h1 class="title">This is a
heading</h1>
  <p>This is a paragraph.</p>
  <h2 class="title">Smaller Heading</h2>
</body>
</html>

/* cssdemo.css file */

body { background-color: black; color:
white; }
h1  { color: blue; }
p   { color: red; }
.title { font-family: sans-serif; }
```



# CSS – Class Exercise

Let's create a CSS file, named **styles.css**, and put it in the same folder where we created index.html before.

Our styles.css looks something like this (see right).

Inside the index.html **<head></head>** tags we add a **<link />** to import CSS.

```
<meta charset="utf-8">
<title>page title</title>
<link rel="stylesheet" href="styles.css">
</head>
```

*Play around with the styles, especially with the colour, width and paddings, and watch the changes.*

There are hundreds of attributes you can modify, and we will revisit these in later modules.

```
* {
  box-sizing: border-box;
  margin: 0;
}

body {
  font-family: sans-serif;
}

header {
  background-color: blue;
  color: white;
  text-align: center;
  padding: 20px;
}

nav {
  float: left;
  background-color: gold;
  width: 30%;
  height: 300px;
  padding: 20px;
}

article {
  background-color: whitesmoke;
  float: left;
  width: 70%;
  padding: 20px;
}

aside {
  clear: both;
}

footer {
  background: #222;
  color: #eee;
  text-align: center;
  padding: 0.5em;
  font-size: 0.9em;
}
```



# CSS – Conclusion

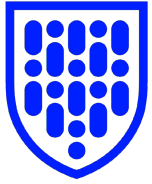
Let's look at our page now. Different right? It is not spectacular, but it is a big improvement.

Making things look attractive will come from:

- Learning design principles
- Experience
- Research (Try to look for inspiration on other websites or even Pinterest)

Next, we will introduce JavaScript and we will use it to make some changes to our page.



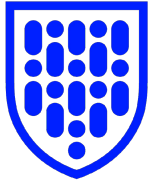


# Introduction to JavaScript

- Definition
- JavaScript to interact with HTML/CSS
- Lab – HTML+CSS + JavaScript Practice.

Note – we will refer to JavaScript with either the full term or just the shortcut **JS**





# Definition - JavaScript

JavaScript is the programming language of the Web, used both on client-side and server-side.

For **client-side**: HTML and CSS give structure and style to web pages displayed by the browser, and JavaScript gives web pages interactive elements that engage a user.

For the **server-side**: it's just like C# or Java, but the syntax is JavaScript. We still need an interpreter for the code, which is where NodeJS comes in.





# JavaScript – Script tag

The HTML **<script>** tag is used to define a client-side script (JavaScript). When we refer to client side, we always mean what is available to the user through the web browser.

The **<script>** element either contains script statements, or it points to an external script file through the **src** attribute, similar to how we link images or external CSS files.

In this example, we use JavaScript to execute a simple **alert** action. JavaScript (**JS**) is executed in the order it is read. In this case, the browser will read the HTML content from top to bottom. When it finds a JS part, it will execute it.

In this case, it will execute this alert *after* loading the HTML. *Try it!*

```
<!-- jshello.html -->

<!DOCTYPE html>
<html>
<body>
<h2>Use JavaScript to Interact</h2>
<!-- This example will activate an alert : -->
<p id="content"></p>

<script>
alert('Hello from JavaScript!')
</script>
</body>
</html>
```



# JavaScript – Change the document elements

In this example, we demonstrate how we can use JS to manipulate CSS to **change** how things look and feel. For example, we could make a link change colour after it is clicked, or make an image zoom in while the user is hovering on it with the mouse. JS is extremely versatile.

In this example, we also meet the concept of **id**. So far, our elements have all been generic. If we need to identify one specific element, we can give it a unique id, so we can recognise and target it with JS or CSS, and apply specific changes.

In this case, we want to change the colour of the element which has an id of “**demo**”. The `getElementById` function you see goes through the HTML **document** searching for an element of id **demo** and when it finds it, it changes the **style colour** to **red**.

```
<!-- jsdemo.html -->

<!DOCTYPE html>
<html>
<body>
<!-- this demo will change the text colour to
red. -->
<h1>JavaScript Change style Demo</h1>
<p id="demo">JavaScript can change the style
of an HTML element.</p>

<script>
  document.getElementById("demo").style.color
  = "red";
</script>

</body>
</html>
```

## JavaScript Change style Demo

JavaScript can change the style of an HTML element.





# JavaScript – functions

Now we can create a button to activate a **function**. A function is a piece of code which gets executed only when it is called. In this case, the function is called by the click of a button we created. The button itself has an **onclick** attribute. This is a special attribute, which is for adding behaviour to specific **events**, in this case, a click.

Our function **helloWorld** will pop up a message when the button is clicked. The button itself, is another HTML primitive. It is a special element with premade behaviors, in this case, behaving like a button.

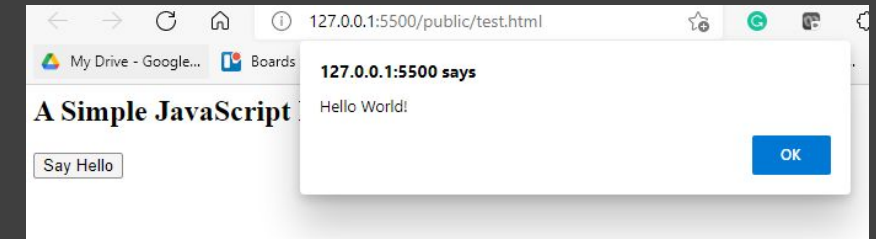
## Class Exercise

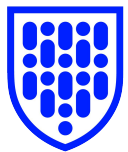
Copy and execute the code. With a reference to the previous slide, try to change the colour of the button when clicked!

```
<!-- jsfunction.html -->

<html>
<body>
<h2>A Simple JavaScript Function</h2>
<!-- This example button calls a function
which displays an alert: -->
<p id="demo"></p>
<button onclick="helloWorld()">Say Hello</button>

<script>
function helloWorld(){
    alert('Hello World!')
}
</script>
</body>
</html>
```





# JavaScript Exercise

We can now use JavaScript in the `<script>` tag to manipulate some of the page content from our previous **index.html** file.

First we change our text in the `<h1>` tag to “Hello Javascript”.

Then we add a background colour to the `<aside></aside>` tag.

We also need to add “id” attributes to the tags we would like to change.

Then we use JavaScript to get the tag by the id and modify the background color or text.



## Class Exercise

Format the page to resemble a very simple online shop.

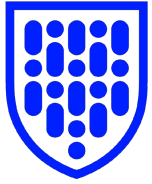
Add an image where the article is, you can use any image online.

Create a new section that will contain a form with the address to ship the item to.

Add a button with the text “Order Now” that generates an alert saying, “Your order will be shipped soon.”

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>page title</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="styles.css" />
</head>
<body>
  <header><h1 id="header">Header</h1></header>
  <section>
    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">Articles</a></li>
      </ul>
    </nav>
    <article>
      <h2>Article heading</h2>
      <p>Paragraph...</p>
      <a href="https://www.google.com/">More
information...</a>
    </article>
    <aside id="aside">
      <h2>Related</h2>
      <ul>
        <li><a href="#">Article 1</a></li>
        <li><a href="#">Article 2</a></li>
      </ul>
    </aside>
  </section>
  <footer>
    <p>@copyright 2050 by nobody. all rights
reversed</p>
  </footer>
  <script>
    document.getElementById("header").innerText =
'Hello JavaScript';

document.getElementById("aside").style.backgroundColor =
'green';
  </script>
</body>
</html>
```



## Lab Exercise 3

Using this code as a starter:

- Add a 'Change Me' button under each heading
- Style your buttons with CSS
- Add a script tag with two JS functions, one for each button
- Clicking each button should change the background colour and the heading text for that column.

*Extension: add text fields under the headings, and replace the heading text with the value of the text fields when the buttons are clicked!*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
content="width=device-width, initial-scale=1.0">
  <title>Module 1 Exercise 3</title>
  <style>
    body { font-family: sans-serif; margin: 0; }
    .container { display: flex; min-height:
100vh; align-items: stretch; }
    .column { background: goldenrod; display:
flex; flex-direction: column; justify-content:
center; align-items: center; flex: 1; }
  </style>
</head>
<body>
  <div class="container">
    <div class="column" id="column1">
      <h1 id="heading1">Hello</h1>
    </div>
    <div class="column" id="column2">
      <h1 id="heading2">World</h1>
    </div>
  </div>
</body>
</html>
```



# Software Engineering

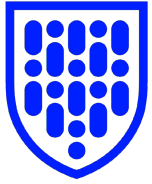
Module 1

Part 3:

---

## Managing Code

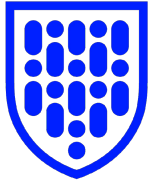
---



# Source Control

Keeping track of your work and collaborating is the most important part of being a good developer.

We are going to learn **GIT** to achieve this. Git, and in our case, GitHub, are applications used to coordinate work among developers. It is by far the most common way to work in a team on software development.



# GIT and Source Control

Steps to using GIT effectively:

- Create a new repository (repo)
- Add a file and commit
- Push changes
- Clone a repo
- Branching : checking out to different branch
- Update and merge
- Miscellaneous Commands : log and tag
- Gitk -- built in git GUI



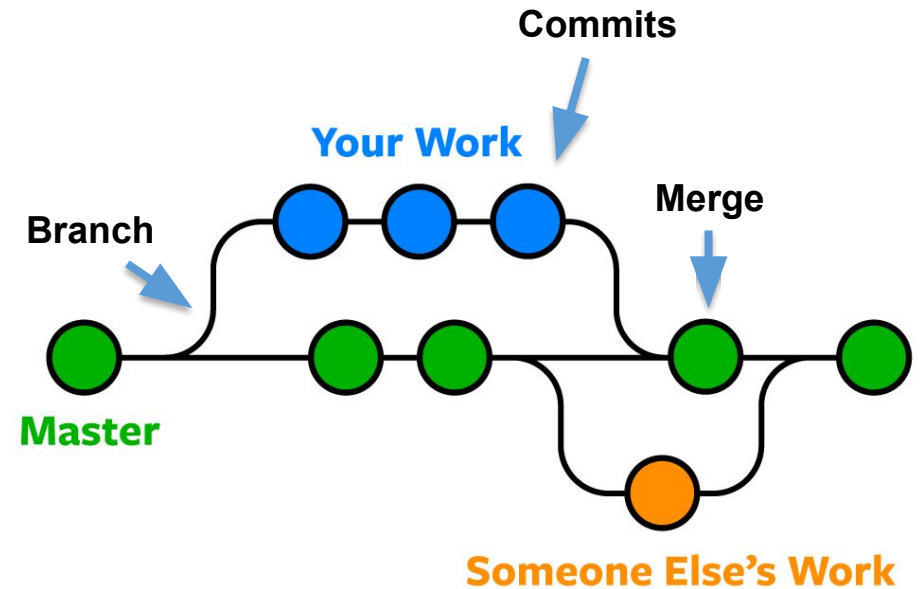


# Basics of Git

Understanding git is not the simplest thing in the world, as you need to understand the concept of local, remote, staging and many more terms.

Git allows us to track our work through time. We start from a **master (main)** timeline and instead of developing on it, we branch into our own custom timeline/s. This creates a parallel version of the code. We write our code here, and along this timeline, we **commit** our work continuously.

When we are confident we have done our work, we make a final **commit** and then **merge** it back into the master timeline. We then **push** the code so that it becomes available for everyone else on the team.



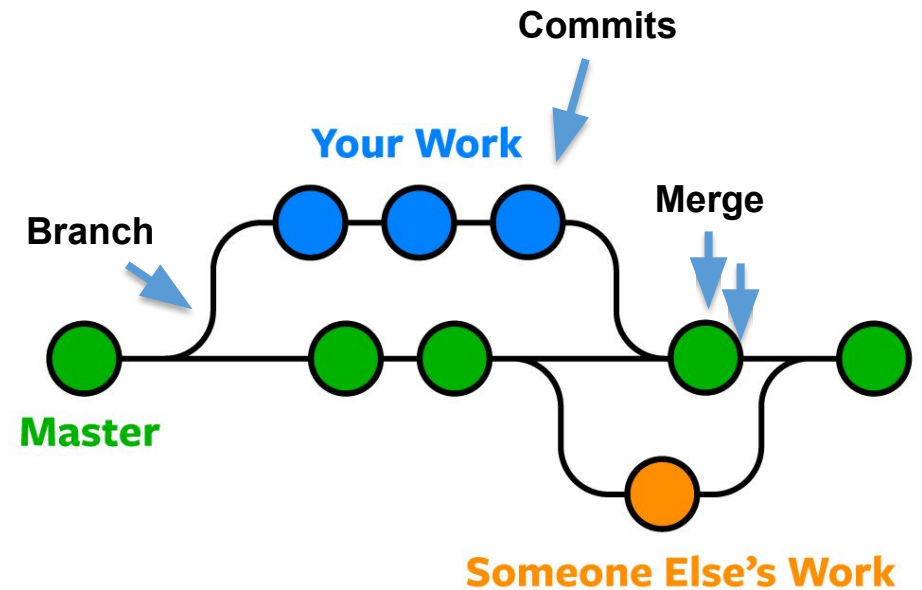


# Basics of Git

By doing this, we can ensure that our development work will not affect other people until it is tested and finalised. Good developers commit their work frequently. You can always go back to a previous moment in time, resetting your code to that specific moment.

There is a fair amount to work through if this is your first time using Git. Don't rush it, every step is needed.

We will use Git for the entire course, so you will have plenty of time to become familiar with it.







# Git – local configuration

**Previously we created a GitHub account, this will come in handy now.**

Git is extremely big, and important. Remember, your work and your business' work may be here, so you want to make sure that it is always protected. Run the following commands to configure your account details.

Open the command prompt (the Terminal in VS Code will do).

Run these commands to configure your username and email, making sure to replace the quoted values with your own Github account details:

```
git config --global user.name "Your Username"  
git config --global user.email "youremail@email.com"
```



# Git – Creating a repo online

A repo, or a repository, is a synchronised version of your code. We call **remote** the one which is on GitHub, and we call **local** the one which is on your machine. We use GitHub to store our code, share it with the world, and also, to share it with our teams. We **synchronise** through pulls and pushes. When we request code from the remote, we call it a **pull**. When instead, we upload code from our local machine, we call it a **push**.

Expert users will know how to manage code straight from their machines, but for us, we will begin by creating the repository online.

Navigate to [GitHub repository](#) and create a repo. Give it a name, and follow the prompt to initialise it and add a README file.

One created, you will be able to clone the online repository locally.

Make sure you 'Add a README file' at this stage - this is good practice and will make the cloning process go more smoothly.



# Git – cloning the repo

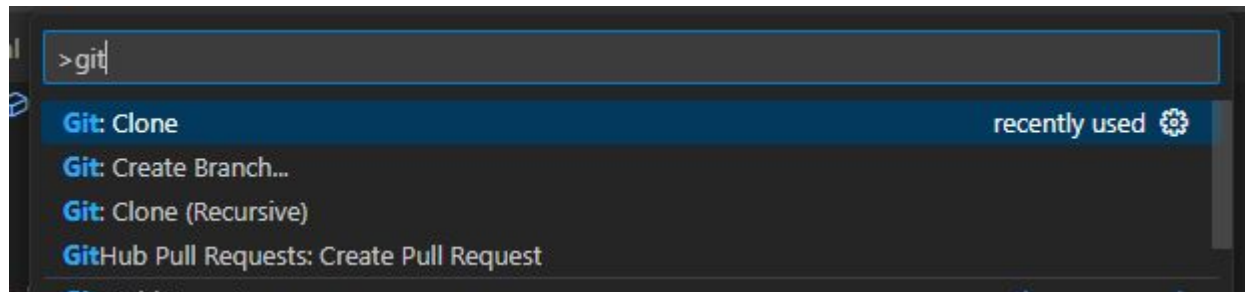
The first time we make a local copy of the repo, we call the action a **clone**, as we are cloning the remote code into the local environment.

Open a terminal and use the command:

```
git clone <url of the remote repo> (make sure you paste your actual repo URL)
```

And this will make a perfect local copy.

You can also use VSCode for this: use the shortcut **Ctrl-Shift-P**, then type 'git' to find the Git: Clone command:



Git is the window to the soul of every developer. If you apply for a job, you can expect people to look up your git repository.

Click **Git: Clone** then paste in the link for your new repository. VSCode will ask where to store your remote repo on your local computer (find your IOD directory), then ask if you want to open a new window - choose this option.



# Git – solo source control

Go into the folder you just created after the cloning.

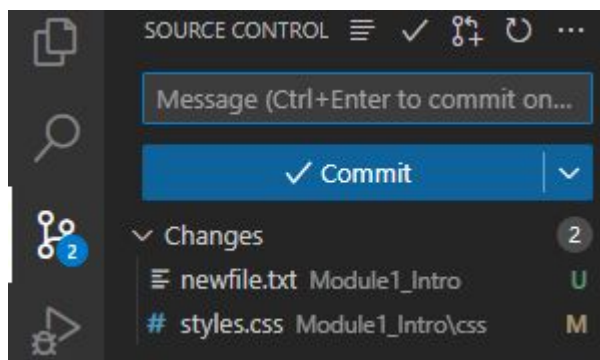
If you haven't created a README at creation, create one now, if you did create it, just open it. Make some changes, and save the file.

Run the command `git status`, it will tell you that the file has been modified. If you created it, it will say it is not tracked.

Run the command `git add .` This will add all the files to current commit, and will track them.

Run the command `git commit -m "write something here"`. This will create a set moment in time. At this moment, your repos are now out of sync. Your local is ahead.

You can also use the Source Control icon in VSCode:



This area tracks your changed files and allows you to commit them all together when you're ready.

Remote

Local

Master  
Time 0

Master  
Time 0

Master  
Time 1

As you can see, we can really refer to GIT as a **timeline management system**. Once we committed Master Time 1, we can go back to Time 0 if needed.



# Git – solo source control

Because your local and remote repos are out of sync, if someone else were to clone the remote repo, we would be in trouble. Also, if our machine were to die, we would be in trouble.

So the next step is to push the code back to the remote.

Once we are happy with the commits, we just need to run the command **git push**. If this is the first time you run it, you will get prompt to update the remote, you can just copy and paste message to fix it.

**After pushing, the remote and local will be back in sync.**

Remote

Local

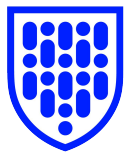
Master  
Time 0

Master  
Time 0

Master  
Time 1

Master  
Time 1

**This system is the backbone of every project, you will learn with experience to use many more commands, but for now, let's keep it simple.**

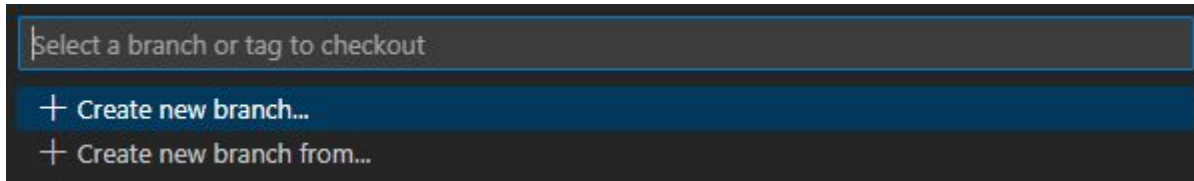


# Git – branch source control

Create a new branch called feat1. Use the command `git branch feat1` or via VSCode - click the 'main' branch in the bottom leftcorner:



Create a new branch called feat1:

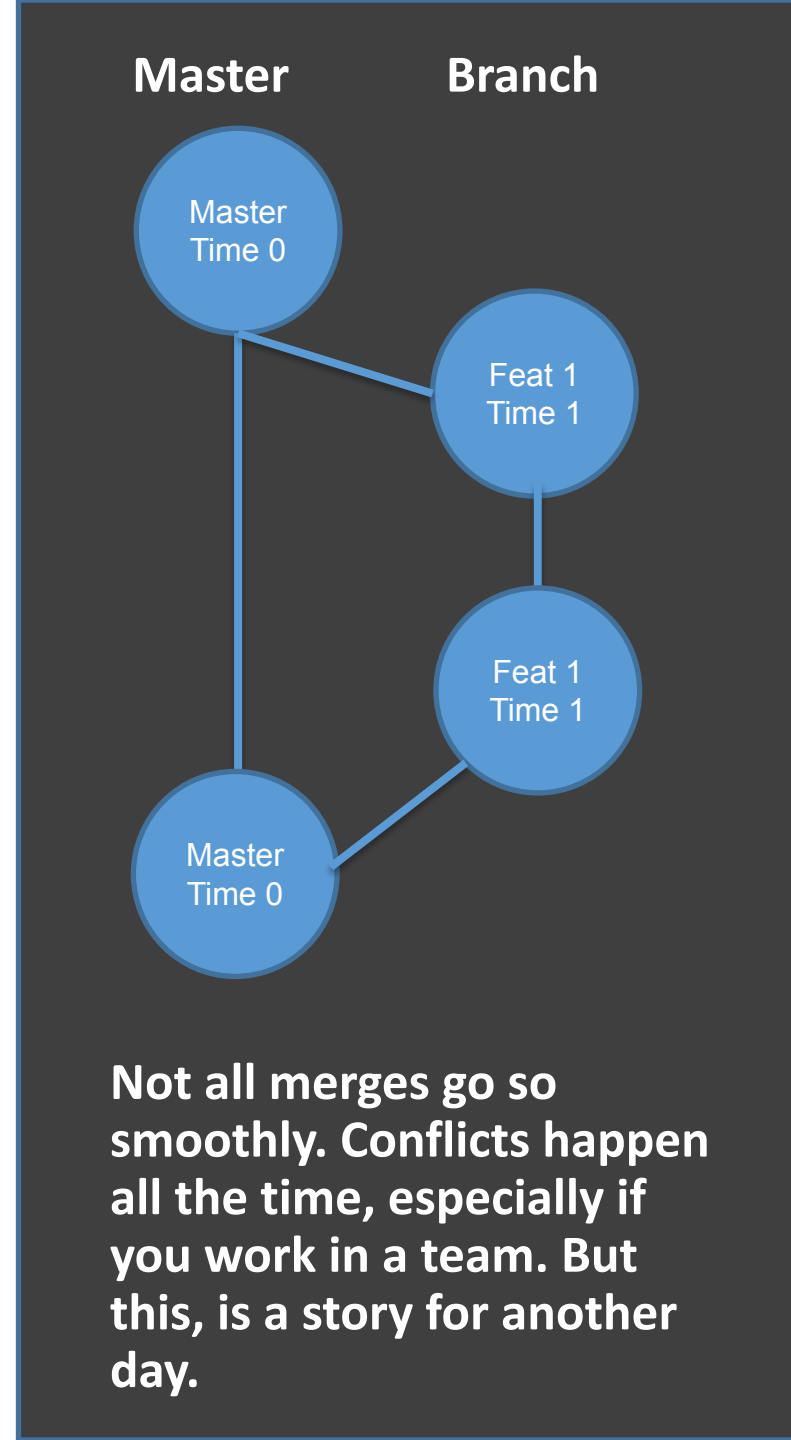


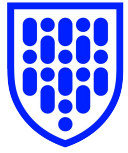
In your branch, do the **add** and **commit** routine again, like you did in the previous one.

Now you have created another point in time. Let's assume you have finished your feature, and you are ready to put it back into main.

Switch into the main branch, and then run the command `git merge feat1`, this will merge the code together. Congratulations, you have completed your first **feature branch**.

Remember, this is still just local, you now push the main back into the remote, so you can share it with your team.



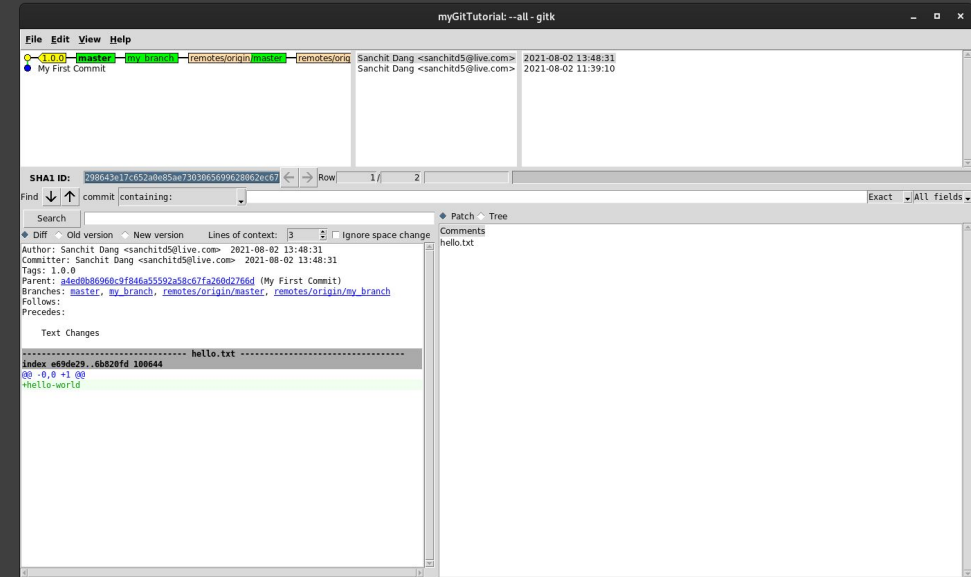


# Git – GUI ?

You can do things using the help of VS Code and other GUI based applications, like the GitHub app, or Source tree, or the many available in the market. Knowing how to do things through the console line is also important, as many times, you will not have access to a GUI.

Final step: explore the commands **git log** and **git tag**. They are very useful even at the early stages, especially when things go wrong.

Try GitK, it is a graphical model, an alternative to **git log**. You can run it by executing the command `gitk -all`



Later we will also explore GitHub pages. As of 2024, they are extremely popular among young and veteran developers, to showcase your work online.



# Software Engineering

Module 1

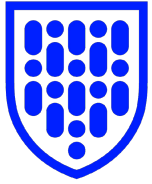
Part 4:

---

## Intro to NodeJS

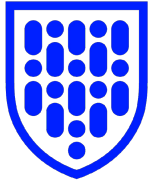
---





# Introduction to NodeJS

- Definition
- Usage
- Application Program Interface (API)
- Benefit
- Example of big projects that use Nodejs
- Lab – Node practice.

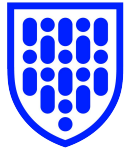


## Definition : NodeJS

Node.js is a **runtime environment** for executing JavaScript code outside of a browser. It literally changed the way we develop because it took down a huge barrier in web development.

Prior to this, developers would be divided into **frontend** (HTML and JS) and **backend**, (Java / .Net). NodeJS allowed for the use of JS in both front and backend, making it simple and fast to become “**Full Stack Developers**”.

It hit 1.2 billion downloads in 2018 (Medium), used by at least 20 million sites (W3Tech).



# Popular Node Backend Frameworks

A major benefit of Node.js is the large library of packages and frameworks that have been already written and are available for use to speed up development. We will use several of these throughout the course, including:

**Express.js** – rapid server-side programming for building RESTful APIs.

**Socket.io** – building real-time apps and establishing bidirectional communication between web clients and servers.

**dotenv** – securely stores configuration details, making it simple to switch between development and production environments without changing code

Other good and popular frameworks can be found here:

[Node.js examples](#)



# Benefits of using NodeJS

- Great for prototyping and agile.
- Highly scalable and fast to build.
- Uses JavaScript - front-end developers can easily pick it up, to become full-stack developers.
- Cleaner code base from front to back (both use JavaScript).
- Large open-source libs and supported by many services.
- Most programming paradigms are shifting to JS, for example Tensorflow, one of the most important AI frameworks, is moving from Python to NodeJS.

Many large companies make use of NodeJS, as its versatility makes it one of the most flexible programming languages for the industry.

Here is just a short list of some of them:

- LinkedIn
- Netflix
- NASA
- PayPal



# Write some Node

First of all, from Part 1, check that you have Node running. You can do this by opening the command prompt, or PowerShell, or terminal,

```
PS C:\Users> node -v  
v14.16.1
```

and run the command **node -version**. If this goes well, you should see some numbers.

Create a new folder in your **Module1** directory and call it **backend** or similar

```
console.log('Hello World')
```

Using VS Code, create a file inside this new folder called **helloworld.js**

Insert the following content and save it.

It is time to test your first helloworld program.

The simplest way is to run it using the following command

**node helloworld.js** in the terminal, from where the file is stored. If everything goes well, you should be able to see the following.

```
PS C:\Users\ [redacted] \JS Practice\Course Writing\nodejs> node .\helloworld.js  
Hello World !
```

**To change the code, after changing, save it, close the current process by doing CTRL-C or Command-C twice . You have to do this manually every time, unfortunately, until you start using packages.**



# Functions

We encountered functions before, but now we need to spend more time on them.

A function is a piece of reusable code with the following features: it has a **name**, it has a **signature** and it has a **body**. The signature and body can be empty, but it requires a name.

A function is a block of code stored in memory, therefore we need to initiate, or call it, to make it run. Calling this function will simply output to the console **“Hello”**

```
// this is the structure of a function
// function ThisIsTheName(Signature) {
//     Body
// }

function sayHello() {
    console.log('Hello')
}
```

The signature is the arguments that the function will use. Arguments are INPUTS, and are passed into the body.

Within the body, we have the core logic.

```
function sum(a, b) {
    console.log(a + b)
}
```

This means that the function is expecting two inputs, and it will output to the console, the sum of these two.

A function must be called, and we would call it like this:

```
sum(5, 1)
```

This will output 6 to the console.



# Functions - returns

Understanding the **return** in a function is not always easy. Return means “return something to whoever called it”. For example, let’s assume the **sum** function again.

```
function sum(a, b){  
    console.log(a + b)  
}
```

This function does not return anything. It just does its job of printing a calculated value to the console, and whoever called it has no notion of whether or not the function has completed or if it was successful.

Normally we will run a function to “obtain” some value back, not just to do an isolated job. The returned value can then be used in any number of ways and is more versatile.

Let’s slightly modify the body of our previous function.

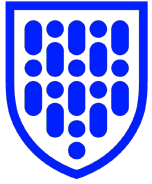
```
function sum(a, b){  
    return a + b  
}
```

In this way, the function does not output anything, but it will instead, return the value to the caller. Let the requestor handle it!

```
console.log(sum(5,1))
```

Now we modify the caller. In this way, the caller decides what to do with the result of the function.

In this case, the caller will output it to the console, not the function anymore. The function has no part in this, it is just a **service**.

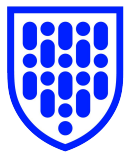


## Exercise 4

Use the learnings from this lesson to design some very simple functions. Call them with different values.

1. Create 4 functions for the 4 main mathematical operations (-, +, /, \*). Return the calculated value and then output it to the screen.
2. Create a function that takes the **name** of a person as an argument, and prints out “Hello <name>” to the console.  
**Hint:** search online on how to concatenate two strings.





# Software Engineering

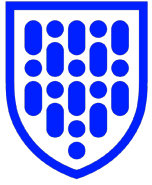
Module 1

Part 5:

---

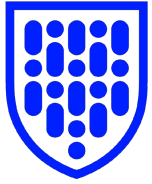
## Package Managers

---



# Introduction to NodeJS

- Definition
- Usage
- yarn
- npm website
- Pracs: using a package.



## Definition : Packages

When we create applications, we rarely build all of it. In fact, we mostly rely on pre-existing packages. A developer's main job is to almost “fill in the gaps”. Building complex applications comes from experience, and that experience translates into architectural patterns and packages. For example, we do not need to make our own sentiment analysis functionalities. Other people did it before, and we can just borrow it.



# Node Package Manager

npm (originally short for Node Package Manager) is a package manager for the JavaScript runtime environment Node.js maintained by npm, Inc.

It consists of a command line client, also called npm, and an online database of public and paid-for private packages, called the npm registry.

The registry is accessed via the client, and the available packages can be browsed and searched via the npm website.

Written entirely in JavaScript and was developed by Isaac Z. Schlueter. Released in 2010.

Over 10 million users in 2018.

Open-source.

While it is important to learn to write complex code, we also need to learn to leverage what is already out there.

Go on the npm website, and try to get a feeling for it. Putting them together to form an application is an intricate puzzle.

## Class Exercise

Find three packages on NPM that you could use to build some application, to add features such as encryption, logging, AI support, parsing content from files, or sending emails.



# Usage

- NPM can manage packages that are local dependencies of a particular project, as well as globally-installed JavaScript tools.
- All the dependencies of a project are defined through the **package.json** file.
- It allows users to consume and distribute JavaScript modules that are available in the registry.
- We can use packages published by others.
- We can publish packages for others to use.

The **package.json** file is used to keep track of packages we use locally. The part called dependencies, are all the packages downloaded and managed through NPM. It is not abnormal to have 90% of an application made up of packages, and not our own code.

```
{
  "name": "deakin-crowds",
  "version": "0.0.0",
  "dependencies": {
    "body-parser": "^1.19.0",
    "cf-deployment-tracker-client": "*",
    "commander": "^2.6.0",
    "express": "^4.14.1",
    "http-post": "^0.1.1",
    "http-proxy": "^1.8.1",
    "mongodb": "^3.6.6",
    "socket.io": "^2.3.0"
  },
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/IBM-Bluemix/bluemix-hello-node.git"
  }
}
```



# NPM Packages

Use the folder you used in the previous part. Within it, run the command **npm init**. This will initialise the folder to be npm usable. A number of questions will come up, you can skip through them, or fill them up. They will simply populate your **package.json** file.

Next, run the following command: **npm install -g nodemon**. The **-g** indicates a global package, meaning it can be accessed by any process on the machine.

Try to run the code again, but this time, use **nodemon helloworld.js**. Once it started, try to modify the file and simply save. Something magical will happen, your application will restart without having to stop it manually. This is one very common package that makes your life easier. This is a global package. Normal packages are only for the project at hand, and use no **-g** in their installation.

When you run **npm install <name of package>** the new package will automatically be added to the **package.json** file.

Why do we need it ?  
Remember, 90% of the application could be packages, then why save them? We don't, we only install them when we need to run the application.

For example, there are no packages on GIT, who ever clones a project, will run the command **npm install**, this will read the **package.json** and install all of the dependency packages.



# NPM Packages

Packages you download which are specific to your application, and not global, need to be first installed, and then “required”, meaning they need to be made available to the application.

This is done by “requiring” it at the top. This is done to save memory and make the application more efficient, as you may actually get a large package, but only use a small part of it. Try the following code.

```
const Sentiment = require('sentiment');
const sentiment = new Sentiment();
const result = sentiment.analyze('Cats are stupid.');
```

`console.dir(result);` // Score: -2, Comparative: -0.666

## Class Exercise

Try to make a function to return the sentiment score of any sentence!

Go on npm and look for the Sentiment package, you should get this.

One key aspect is the weekly downloads, normally good packages will have more than 10K.

To use this package, simply copy the install command **npm i sentiment** and run it in your terminal in the project folder.

The screenshot shows the npm package page for 'sentiment'. At the top, there's a search bar and navigation links for 'Sign Up' and 'Sign In'. The package name 'sentiment' is displayed with its version '5.0.2', 'Public' status, and 'Published 2 years ago'. Below this, there are links for 'Readme', 'Explore', 'Dependencies' (0), 'Dependents' (112), and 'Versions' (24). The main section describes 'sentiment' as an 'AFINN-based sentiment analysis for Node.js'. It includes a 'PASSED' badge from Greenkeeper and a 'Move to Snyk' link. A description states it uses the 'AFINN-165 wordlist' and 'Emoji Sentiment Ranking' for sentiment analysis. A list of features includes performance benchmarks, the ability to append/overwrite wordlist pairs, easy support for new languages, and custom strategies for negation/emphasis. On the right, there's an 'Install' section with the command 'npm i sentiment', a 'Repository' link to 'github.com/thisandagain/sen...', a 'Homepage' link, and a 'Weekly Downloads' graph showing 15,051 downloads. A 'Table of contents' lists 'Installation', 'Usage example', and 'Adding new languages'. At the bottom right, it shows 'Version 5.0.2' with 'MIT' license, 'Unpacked Size 457 kB', and 'Total Files 47'.



# Software Engineering

Module 1

Part 6:

---

## Debugging

---





# Debugging demystified

Debugging is the art of finding bugs in your software. There are a number of ways to go about doing this, and every developer should know at least the basics.

Writing good code comes from paying attention to the code itself. Errors happen because we have the logic wrong, forgot to require something, or we simply made a typo.

Remember, computers are machines that do exactly what they're told, they don't act randomly. Unfortunately all the errors are our fault and our responsibility.

## Types of debugging

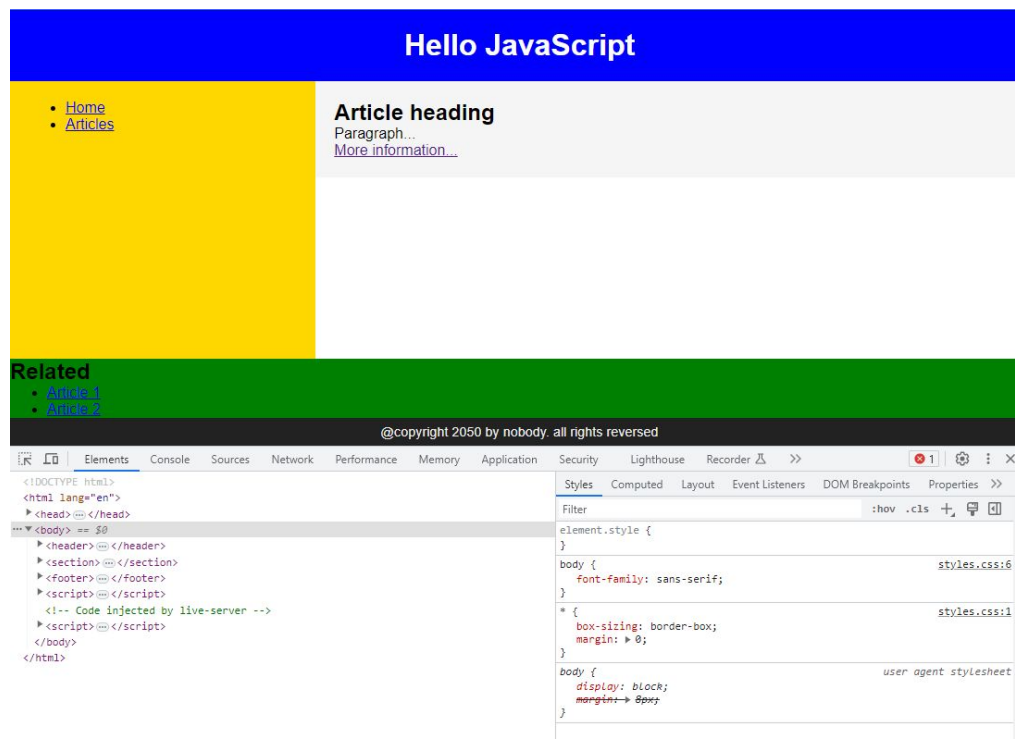
- Console debugging
- IDE level debugging
- Browser level debugging



# Browser Inspector

If you are using Chrome, or Edge, or Safari, or Mozilla, you should have access to the inspector mode – or developer mode.

You can use this to “inspect” the page. You can do this by pressing **ctrl + shift + i** or simply right-clicking on the browser anywhere, and clicking **inspect**.



## Class Exercise

Learning to use the inspector window is vital to finding problems, and making corrections on the fly.

Explore this window and find the tab “**console**”

Everytime you use the **console.log** command, messages will be displayed here.

Spend some time finding more information about the Network tab and the Elements tab. Try clicking on items on a page, they will be highlighted in the rendering window.

*Try to change the values!*



# Debugging demystified

Basic debugging can be done using console logs. There are various logging levels available such as

**console.debug( )** : outputs a message to the console with the log level debug

**console.info( )** : is used to output message which provide information, such as “server started”

**console.warn( )**: used to display warnings

**console.error()** : is used to output an error message. You may use string substitution and additional arguments with this method.

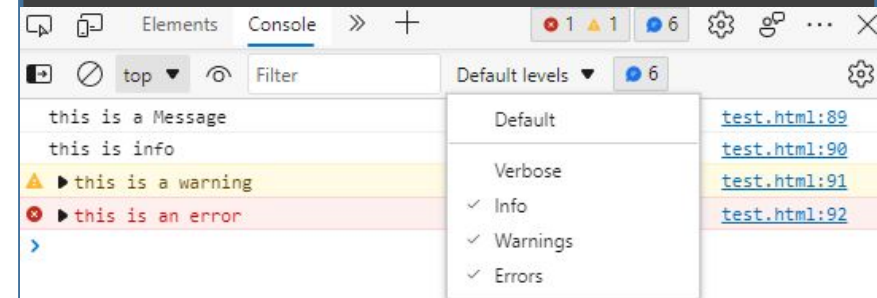
These will help you find mistakes faster.

## Class Exercise

Include the following in the script of your index.html page and run it using live server.

```
console.log('this is a Message')
console.info('this is info')
console.warn('this is a warning')
console.error('this is an error');
```

Open the inspector and look at the messages. You can see there is also a level you can customise.



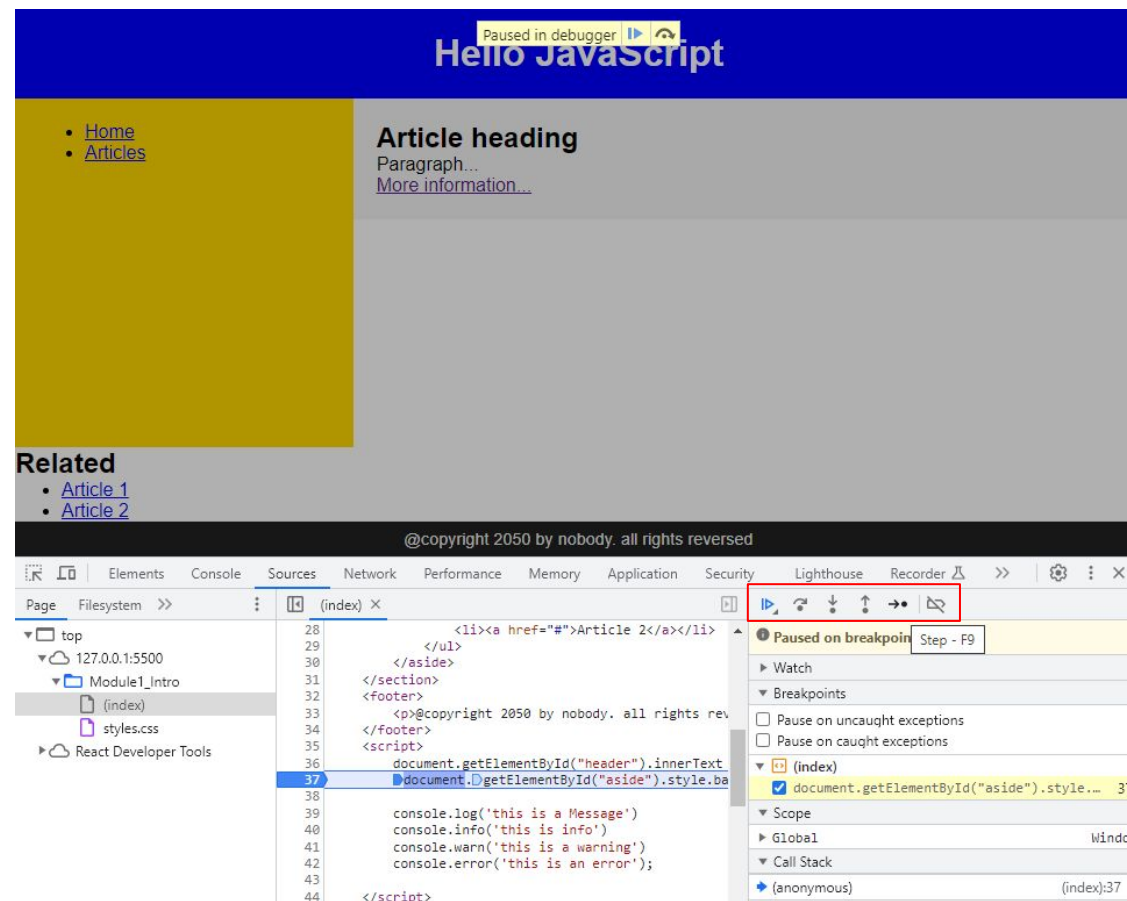


# Breakpoints

Breakpoints allow us to halt the code execution at a particular point and step through each subsequent statement one by one, while monitoring the logic and variable values.

After setting a breakpoint, when you reload the page, the renderer will stop at exactly that moment. We can then use the console to verify that it has not passed the breakpoint and manually control each step of the code.

Open the inspector window, then click the **Sources** tab and load your code by selecting it from the left panel. Click the line number to set a breakpoint there, and then refresh the page. The script execution will halt when it hits your breakpoint.



At the top, there are several icons to control the script: **Resume** and **Step**. Resume will continue with no more stops. Step will instead advance the code by one line.



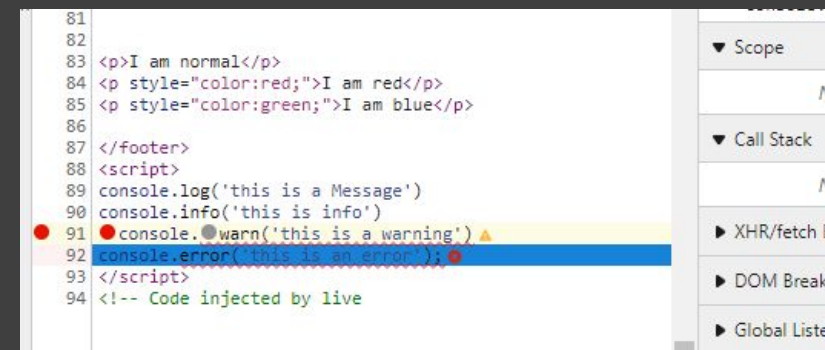
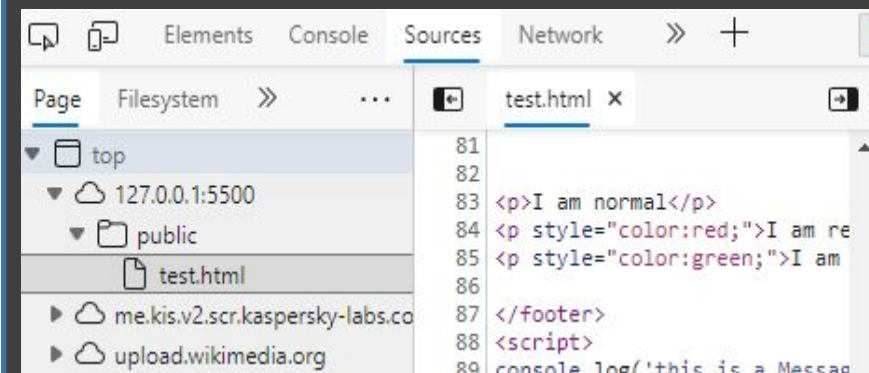
# Breakpoints

Run any HTML page that uses some JS in the browser using Live Server and open the inspector window.

In the “Sources” tab (or Debugger on Firefox), you will see a tree structure with your page listed - click on this to load the HTML source code for your page.

Scroll down the to find the JS section, and click just left on any line where the line numbers are. This will add a breakpoint - different browsers will use different colours to indicate it.

Reload the page, and then use the control icons to step through each line of code individually!



This technique is the most common way of debugging applications.

On the right you have a column with all the machine state. Don't worry, it is too early for you to understand that part yet, but we will get to it!



# Software Engineering

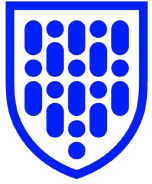
Module 1

Part 7:

---

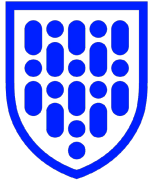
## Variables and Data Structures

---



# Agenda: Module 1 Part 7

- Understanding Declarations (var vs let vs const)
- Lists (array)
- JSON structures
- JSON Manipulation in ES6



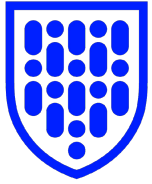
# Software Engineering

When we think of engineering, we think of building cars or buildings or bridges, and less about something as virtual as software. But software is as intricate as any other discipline out there.

Designing the wrong software, or designing the software wrong, can lead to catastrophic consequences.

The same way you can learn to lay bricks to make a house, you can learn programming to make simple applications, but can you make a skyscraper you are sure is not going to fall? That's the role of a software engineer!





# Basics of Declaration

Let's begin with our bricks, **variables**, and how they differ. In JavaScript, the process of creating a variable is called **declaration**. There are three different alternatives, and each will reserve a named place in memory for storing some value.

```
var carName;  
let carName;  
const carName = "constant";
```

After being declared, **var** and **let** variables initially have no value, which technically we refer to as being **undefined**.



# Understanding Var

In the world before ES6, we would have only used **var** as our preferred variable declaration keyword. This can lead to unexpected and difficult to find bugs, however. So to understand things, let's understand **var** better.

Scope of **var**:

- The term "scope" refers to the area in which these variables can be used. Either **globally** scoped or function/**locally** scoped var declarations exist.
- When a var variable is declared outside of a function, its scope is **global**. Any variable declared with **var** outside of a function block is available for use across the entire script.
- When a variable is declared within a function, it is **function** scoped. This indicates it's only available to be used and referenced within that function.



# Understanding Let

- For variable declaration, **let** is currently the preferred method. It's a step forwards from var declarations because **let** variables have unambiguous scope.
- **Let** is block scoped. A block is a section of code that is bounded by {}. Curly braces are the home of a block. Anything enclosed in curly brackets is considered a block.
- As a result, a variable declared with **let** can only be used within that block. Allow me to illustrate this with an example:

```
let helloText = "say Hi";
let check = 4;
if (check > 3) {
    let hello = "say Hello instead";
    console.log(hello); // "say Hello instead"
}
console.log(hello) // hello is not defined
```



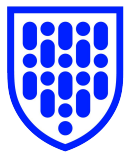
# Understanding Const

- Constant values are maintained for variables declared with the **const** keyword. **Let** declarations and **const** declarations have some similarities.
- **const** declarations are block scoped. **Const** declarations, like **let** declarations, are only accessible within the block in which they were declared.
- **const** cannot be updated or re-declared. This signifies that a variable declared with **const** has the same value throughout its scope. It can't be changed or declared again. So, if we declare a variable with **const**, we can neither do this:

```
const helloText = "say Hi";  
helloText = "say Hello instead"; // error: Assignment to constant variable
```

- Nor this:

```
const helloText = "say Hi";  
const helloText = "say Hello instead"; // error: Identifier 'greeting' has  
already been declared
```



# Major Differences

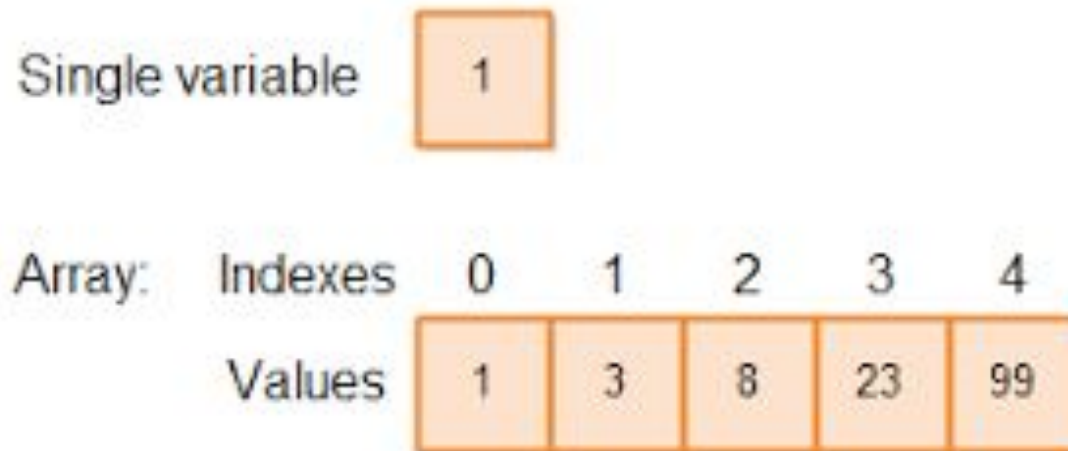
So just in case you missed the differences, here they are:

- **Let** and **const** declarations are block scoped, while **var** declarations are globally or function scoped.
- Within the scope of the variable, **var** variables can be changed and re-declared; **let** variables can be updated but not re-declared; and **const** variables cannot be updated or re-declared.
- They've all been lifted to the apex of their abilities. However, whereas *undefined* is used to initialise **var** and **let** variables, **const** variables are not.
- **Const** must be initialised upon declaration, but **var** and **let** can be declared without being initialised - in this case they will be *undefined*.



# Lists

When we talk about lists in JavaScript we are majorly referencing arrays. Arrays are a form of ordered lists when it comes to JavaScript. Each value is referred to as an element, and it is identified by an index.





# Lists (continued)

A JavaScript array has the following characteristics:

- First, an array can include values of many different types. You can have an array that stores numeric, string, and boolean data, for example.
- Second, the length of an array is auto-growing and dynamically sized. To put it another way, you don't have to declare the array size in advance.

## Creating a JavaScript array:

```
let emptyScores = new Array();  
// or  
let scores = ["A+", 95, "C-", 55, 83, 71, "B+"];
```



# Lists (continued)

## Accessing JavaScript array elements

JavaScript arrays are indexed from **ZERO**. To put it another way, an array's first element starts at index **0**, the second member at index 1, and so on.

We specify an index in square brackets [] to access an element in an array:

```
let mountains = ['Everest', 'Fuji', 'Nanga Parbat'];  
console.log(mountains[0]); // 'Everest'  
console.log(mountains[1]); // 'Fuji'  
console.log(mountains[2]); // 'Nanga Parbat'
```

We can also use square brackets to access and replace a value at a certain index:

```
mountains[1] = 'Kilimanjaro';  
console.log(mountains); // [ 'Everest', 'Kilimanjaro', 'Nanga Parbat' ]
```





# Lists (continued)

## Basic operations on arrays - adding elements

The **push()** method is used to add an element to the **end** of an array.

```
let seas = ['Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea'];  
console.log(seas); // [ 'Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea' ]  
seas.push('Red Sea');  
console.log(seas); // [ 'Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea', 'Red Sea' ]
```

The **unshift()** method is used to add an element to the **beginning** of an array.

```
seas.unshift('Adriatic Sea');  
console.log(seas); // [ 'Adriatic Sea', 'Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea', 'Red Sea' ]
```



# Lists (continued)

## Basic operations on arrays - removing elements

The **pop()** method is used to remove an element from the **end** of an array.

```
let rivers = ['Mississippi', 'Amazon', 'Nile'];  
let lastRiver = rivers.pop();  
console.log(lastRiver); // Nile  
console.log(rivers); // [ 'Mississippi', 'Amazon' ]
```

The **shift()** method is used to remove an element from the **beginning** of an array.

```
let firstRiver = rivers.shift();  
console.log(firstRiver); // Mississippi  
console.log(rivers); // [ 'Amazon' ]
```



# Lists (continued)

## Basic operations on arrays

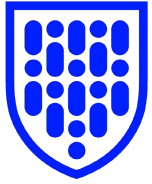
The **indexOf()** function is used to find the **index** of an element.

```
let volcanoes = [ 'Mount Vesuvius', 'Mount Etna', 'Mount Fuji' ];  
let fujiIndex = volcanoes.indexOf('Mount Fuji');  
console.log(fujiIndex); // 2 (indexes start at 0)
```

The **length** property is used to find the number of elements in an array.

```
let numVolcanoes = volcanoes.length;  
console.log(numVolcanoes); // 3
```

There are many more functions for working with arrays, but we will revisit them in later modules.



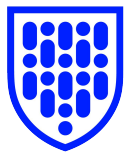
## Exercise 5

- For understanding more about arrays, try creating an array that has 5 elements.
- Replace the value of the element at position 1 and 4.
- Add a new element to the beginning of the array
- Remove the element at the end of the array
- Print the array to the console.



# Basics of JSON

- **JSON** stands for **JavaScript Object Notation** and is used as a data exchange format, as well as to store JS object variables. JSON is derived from the JavaScript programming language, but it may be used by a wide range of languages, including Python, Ruby, PHP, and Java. JSON is pronounced similarly like the name “Jason”.
- When used alone, JSON utilises the **.json** extension. It can also occur **inside quotes** as a JSON string or as an **object** assigned to a variable when defined in another file type (such as .html). The .json format is simple to use when sending data from a web server to a client or browser.
- JSON is an excellent alternative to XML (the previous data exchange standard) since it is more legible and requires less formatting and bandwidth.



# Creating your own JSON

- A JSON object is a data format that uses curly brackets to represent key-value pairs. When working with JSON, you'll commonly see JSON objects in .json files, but they can also exist in the context of a JS script as a JSON object or string.
- A JSON object looks something like this:

```
{  
  "first_name" : "Sammy",  
  "last_name"  : "Shark",  
  "age"        : 25,  
  "followers"  : 987  
}
```

- Although this is a very small example, and JSON can be many lines long, it demonstrates how the format is requires two curly braces (or curly brackets) on either end, and comma-separated key-value pairs filling the space in between.



# Displaying JSON data

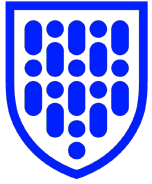
In JavaScript, JSON data is generally retrieved using dot notation. Let's look at the JSON object to see how this works. *Try creating and manipulating a new user object to represent yourself!*

```
// objects in javascript contain keys (or properties) with corresponding values
const user = {
  "first_name" : "Sammy",
  "last_name" : "Shark",
  "age" : 25,
  "followers" : 987
}

// we can access properties with dot notation
console.log(user.first_name); // Sammy
console.log(user.age); // 25

// or with array style square bracket syntax
console.log(user["last_name"]); // Shark

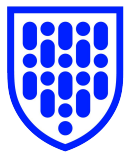
user.followers = 988; // we can also assign new values to object properties
user.location = 'Pacific Ocean'; // or create new properties
```



## Exercise 6

- Try creating a json object variable for a **book**. The book should have a title, description, author and number of pages
- Try printing these object property values in your console individually and via the whole book object
- Update the description of the book.





# Software Engineering

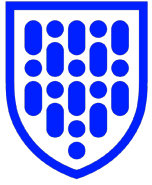
Module 1

Part 8:

---

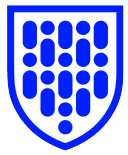
## Software Requirements Specifications (SRS)

---



# Agenda: Module 1 Part 8

- Understanding SRS
- Design Specifications
- What Makes an SRS
- Exercise

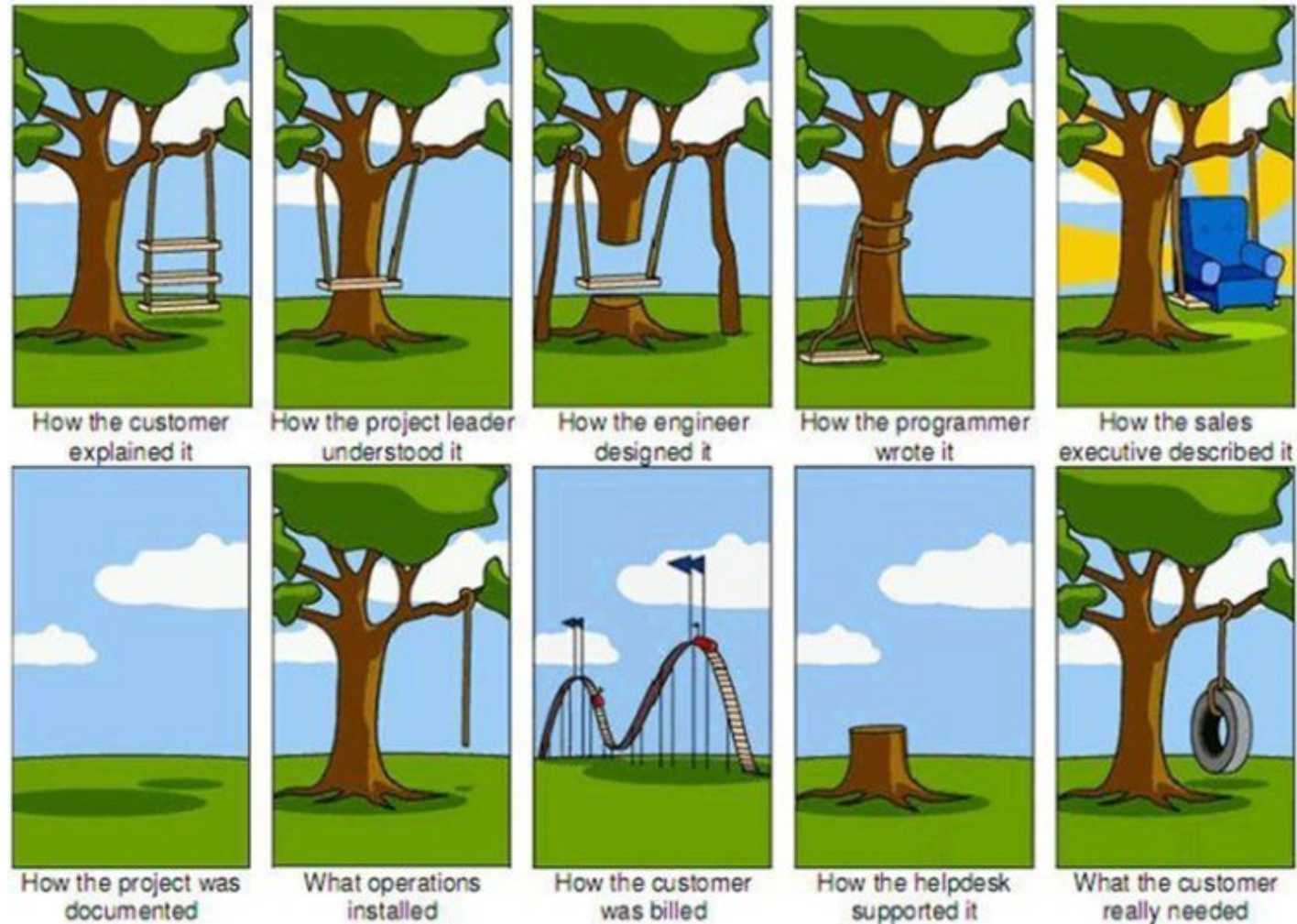


# Understanding SRS

Before we build the software right,  
we need to build the right software.

Every stakeholder will always look at  
the project from their perspective, so  
we need a way to define and deliver  
the project in a systematic way.

Failure to do this effectively makes it  
almost impossible to deliver a  
successful project.

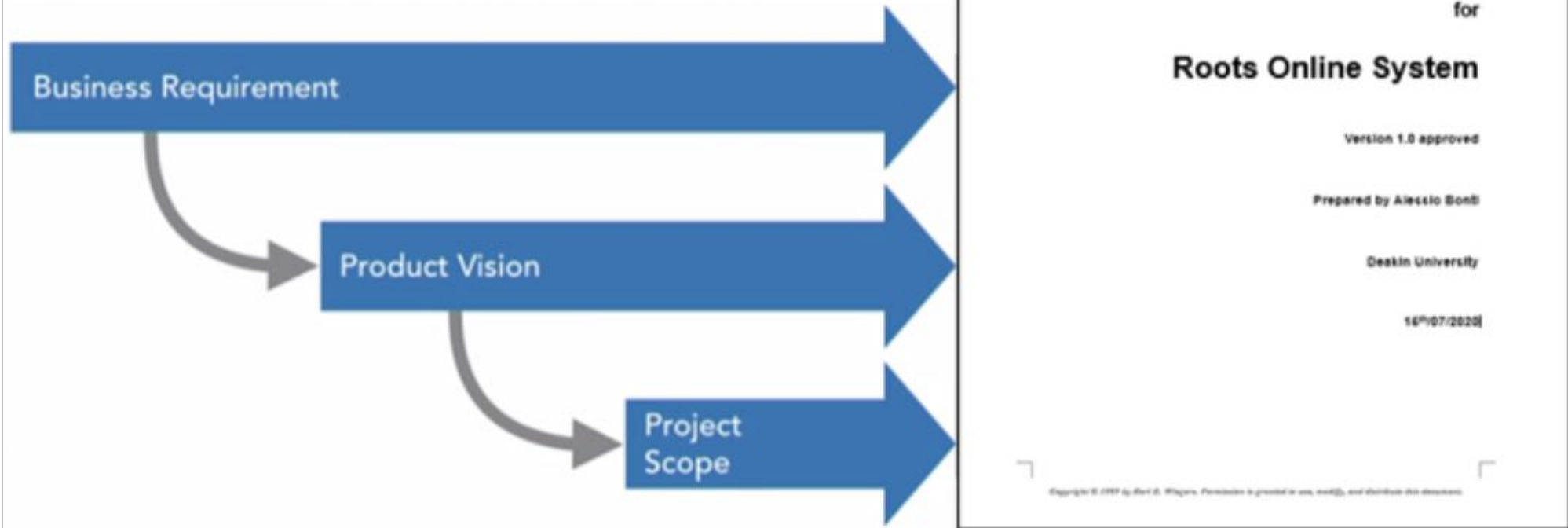




# Understanding SRS (continued)

Business requirements drive the product vision, which then define the project scope. Therefore we create an SRS document to capture each of these accurately.

You can find one good template here: [Software Requirement Specification Template example](#)






# Design Specifications

## User Stories

- A user story is a clear way to understand what the software is required to do.
- Large applications are generally made up of many user stories, each story contributes to one key aspect.
- Analysing a story reveals many requirements which may or may not have discovered in other stories before.

## USER STORY : SIGN UP

As a user, I want to be able to create my account on Roots using either my social account (Facebook or Google) or just my email, and make sure it is validated before I login for the first time.



First Name

Last Name

Username

</>

Email

✉

Password

☐ I agree to the [terms and conditions](#)

Sign Me Up



# Design Specifications

## The Use Case

- Once we know what is required to satisfy the user, we can now try to design the use case (see example), which represents the flow of interactions.
- The simplest way is to follow the natural flow of things, from top to bottom, following what may call the “User journey”.
- Do not skip steps. Keep them separated, this way you can validate one by one.

Primary actor : User

Secondary Actor : Roots signup backend.

Description : Every user is required to signup in order to log into the system. The user is required to create an account, which can be done either through a sign-up form, or through a social account.

### Basic Flow :

1. User lands on home page.
2. User clicks on sign up button
3. User is redirected to sign up form.
4. User submit all the details for the signup.
5. The account is created on the Roots system.
6. An email is sent to the user to validate.
7. The user clicks on the link in the email and is now able to log into Roots

### Alternative Flow :

4. User clicks on FB or Google SignUp
5. Is redirected to Fb for validation
6. User can now log into Roots





# Design Specifications

## Back to the SRS document

An SRS may be written by

- **Procurer**
  - Is a call for proposals
  - Must be general enough to yield a good selection of bids and specific enough to exclude unreasonable ones.
- **Bidders**
  - Is a proposal to implement a system
  - Must be specific enough to demonstrate feasibility and competence
- **Developers**
  - Reflects the developer's understanding of the customer needs
  - Forms the basis of evaluation of contractual performance.



In short, it is the key metric to understand if you and your company understand the project and can deliver it.



# What Makes An SRS?

## 1. Introduction

- What is this software about? Purpose, audience, readings.

## 2. Overall Description

- What is the software supposed to do?

## 3. External interfaces.

- How does the software interact with people, the system's hardware, other hardware, and other software?
- What assumptions can be made about these external entities?

## 4. System Features

- What are the key features of the software, e.g. login, user management?

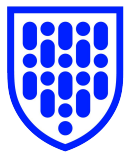
## 5. Non-functional requirements

- What is the speed, availability, response time, recovery time of various software functions, and so on?
- What are the portability, correctness, maintainability, security, and other considerations?

## 6. Other Requirements

- Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) and so on?





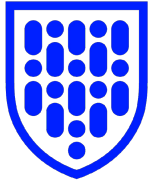
# Software Engineering

Module 1  
Part 9:

---

## Testing

---



# Testing

- Testing as Validation
- Types of testing
- Unit Testing
- Integration Testing
- Test first programming



# What is testing?

Testing is an example of a more general process called *validation*. The purpose of validation is to uncover problems in a program and thereby increase your confidence in the program's correctness. Validation includes:

- **Formal Reasoning**
- **Code Review**
- **Testing**



# Formal Reasoning

**Formal reasoning** about a program, is usually called *verification*.

Verification constructs a formal proof that a program is correct.

Verification is tedious to do by hand, and automated tool support for verification is still an active area of research. Nevertheless, small, crucial pieces of a program may be formally verified, such as the scheduler in an operating system, or the bytecode interpreter in a virtual machine, or the filesystem in an operating system.

It is not a common practice for most practical applications, however.

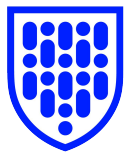


# Code Reviews

Having another skilled developer carefully read your code, and reason informally about it, can be a good way to uncover bugs. It's much like having somebody else proofread an essay you have written - a fresh pair of eyes can often see any issues much more easily and quickly.

Important factors for a good code review:

1. Consistency
2. Simplicity
3. Design patterns
4. Code level implications
5. Quality of test and documentation



# Software Testing

Software testing is an organisational process within software development in which business-critical software is verified for correctness, quality, and performance.

Software testing is used to ensure that expected business systems and product features behave correctly as expected.

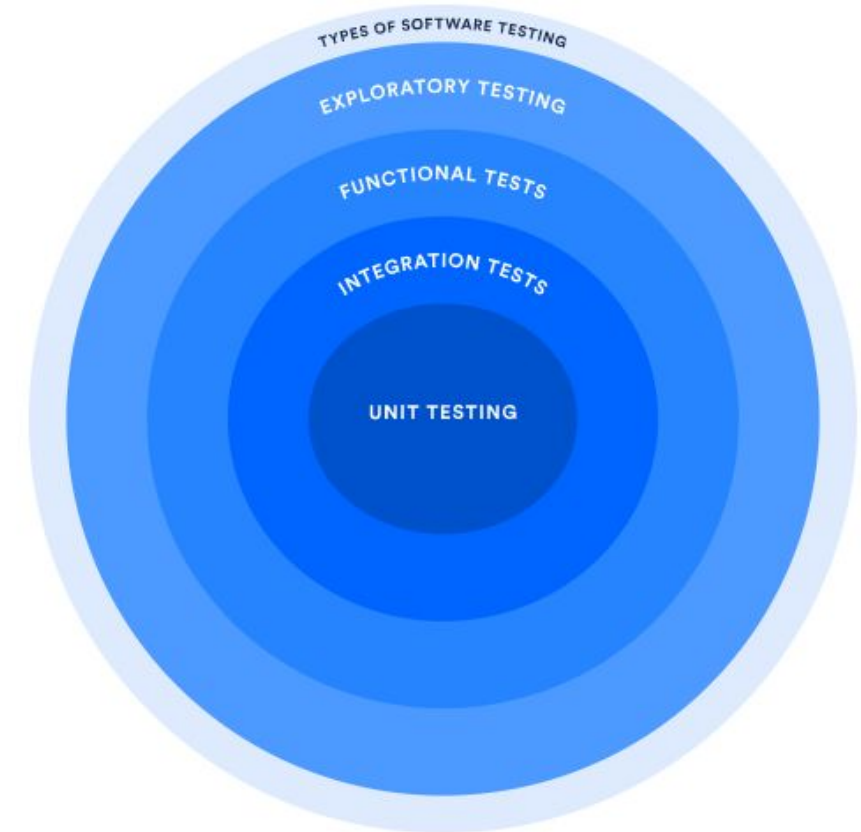
Software testing may either be a manual or an automated process.

- **Manual** software testing is led by a team or individual who will manually operate a software product and ensure it behaves as expected.
- **Automated** software testing is composed of many different tools which have varying capabilities, ranging from isolated code correctness checks to simulating a full human-driven manual testing experience.



# Software Testing

- There are several fundamental levels within software testing, each examining the software functionality from a unique vantage point within the development process. Let's take a look at each type of testing in turn and examine its practical use.
- We go from closest to the functionality (**unit testing**) to closer to the user (**exploratory testing**)





# Unit Testing

Unit testing is the practice of instrumenting input and output correctness checks for individual units of code.

The measurement unit, in this case, is the smallest piece of code that can be logically isolated in a system. In most programming languages, that is a function, a subroutine, a method or property.

Take the example of the function

```
sum(a, b) {  
    return a+b  
}
```

We can say with confidence that this is one complete unit.





# Unit Testing

During unit testing, production code functions are executed in a test environment with simulated input.

The output of the function is then compared against expected output for that input.

If the output matches the expected, the test passes. If not, it is a failure. Unit tests are a great way to validate derived data functions.

Imagine the previous function, the basic test would be

```
if(sum(2,4)!=6{  
    throw error  
})
```

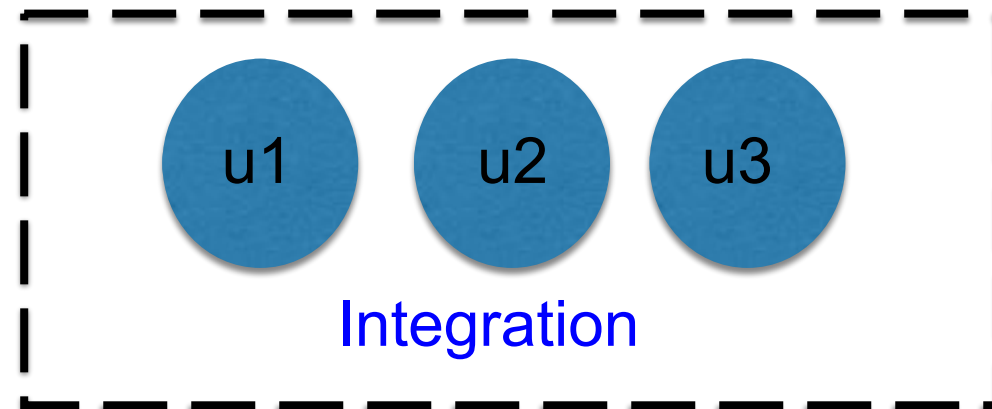
Testing is about what goes bad, not what goes right.



# Integration Testing

When a software test case covers more than one unit, it is considered an integration test. When developing a software test case, the lines between unit tests can quickly evolve into integration tests.

Oftentimes a unit test may be developed that operates against a third party code dependency. The dependency itself will not need to be tested, and the integration to it will be mocked or faked.





# Functional Testing

Test cases that simulate a full user-level experience are called functional tests or end-to-end tests. End-to-end tests use tools that simulate real human user behavior.

Common steps in an end-to-end test:

**Click this button**

**Read this text**

**Submit this form**

Because of the full experience execution context, end-to-end tests verify correctness across all the layers of a software stack.





# Test first programming

**Test early and often.** Don't leave testing until the end, when you have a big pile of unvalidated code. Leaving testing until the end only makes debugging longer and more painful, because bugs may be anywhere in your code. It's far more pleasant to test your code as you develop it.

In test-first-programming, you write tests before you even write any code. The development of a single function proceeds in this order:

- Write a specification for the function.
- Write tests that exercise the specification.
- Write the actual code. Once your code passes the tests you wrote, you're done.



# Unit Testing

The **specification** describes the input and output behavior of the function. It gives the **types** of the parameters and any additional **constraints** on them (e.g. `sqrt`'s parameter must be nonnegative). It also gives the type of the **return** value and how the return value relates to the inputs.

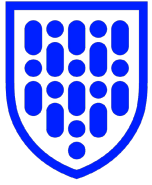
In code, the specification consists of the **method signature** and the **comment** above it that describes what it does. Writing good comments is an essential part of being a good developer and writing good code.

Specification example  
//this function returns the sum of two numbers  
**function** **sum**(a,b){  
}

Create the Test  
**if**(**sum**(2,4)≠6){**error**}

Write the Code  
**function** **sum**(a,b){  
  **return** a+b  
}

Writing tests first is a good way to understand the specification. The specification can be buggy, too — incorrect, incomplete, ambiguous, missing corner cases. Trying to write tests can uncover these problems early, before you've wasted time writing an implementation of a buggy spec.



## Exercise 7

Using the functions you created for **Exercise 4**:

- Write a specification comment for each function
- Write at least 3 unit tests for each function
- In the unit tests, try to include both expected and non-typical test values (such as zero or negative numbers)

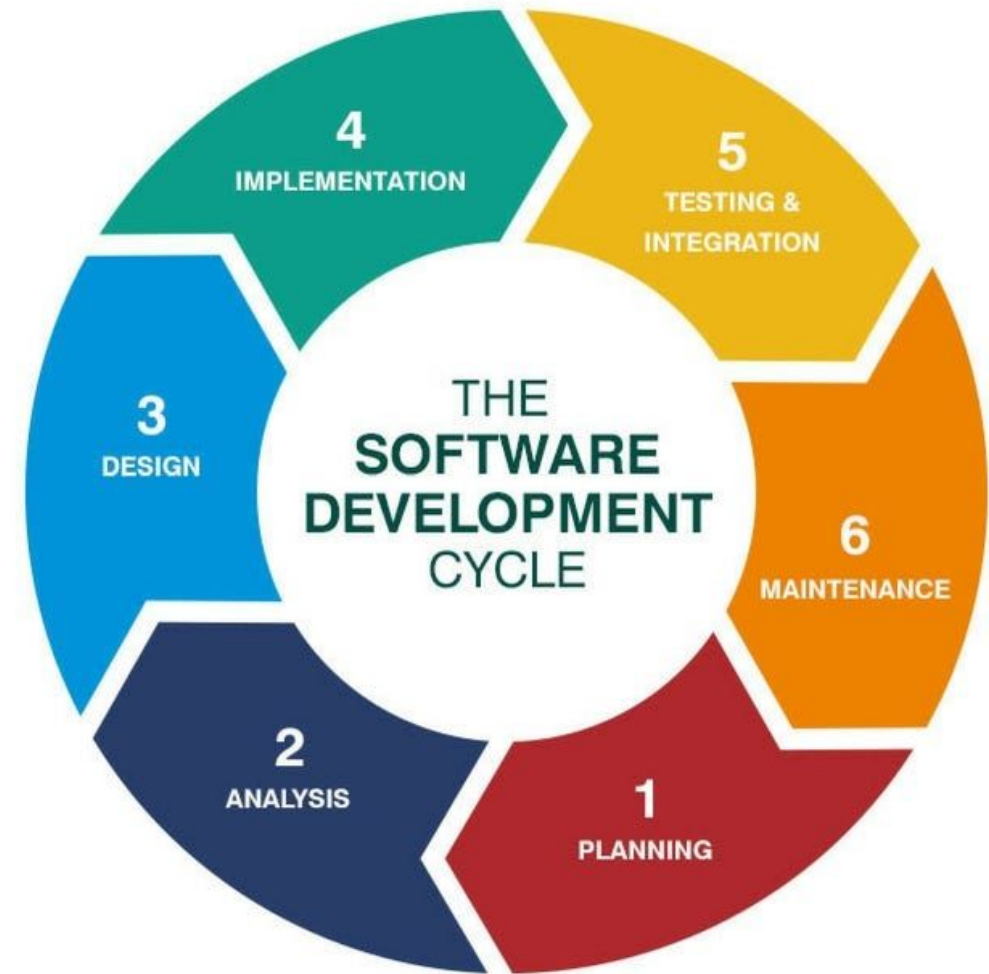


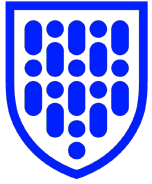
# Software Development Life-Cycle

All software projects, either formally or informally, go through similar stages. We have discussed a high-level overview of each of these and what they may look like.

It is important to understand and follow each stage of the process to ensure a successful outcome. The larger and more complex the project, the more important it will be to document and do each stage well. We will revisit each stage in later modules as we practice and expand on what each involves.

Even for your own personal projects, it is good practice to follow this process to give yourself the best chance of success.





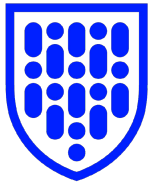
## Exercise 8 - Dice Project

You have been hired by **devsInc** to create a landing page for their new project. They want to support people playing tabletop games from home and require a Dice Generator.

### Features:

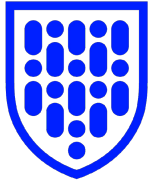
The user should be able to use different dice, such as a D6 or a D10 (number of faces).





## Exercise 8 (continued)

- Define on paper, all the user stories (functionalities) you will need for this page.
- Make a simple storyboard of the user using the system.
- Create a git repository for the project
- Clone the project locally
- Work on the project creating 2 branches, one for the UI, and one for rolling the dice, committing and merging when completed.
- Create the roll functionality by passing an argument, in order to reutilise the same function multiple times
- Write unit tests for the roll dice functionality.
- When complete, push the content to your git repository.



## Exercise 9 – your git page

Learning to code is just one step, to make people know that you know how to code will make a difference. Artists have Pinterest, makers have Etsy, we have GitHub, that is where our name will appear, and the first web site most people will google when they'll want to hire us. GitHub offers the possibility to create your own page, so let's make use of it!

- Simply follow the instructions here to create your first page, follow this [link](#) and use what you have learned to make your portfolio site! Maybe it will not look great today, but one day it will!
- Here are some ideas if you need them.

[30+ Free HTML Portfolio Website Templates – Bashooka](#)