LEAVING CERTIFICATE
COMPUTER SCIENCE

Fundamental Skills Development

JavaScript

JavaScript Programming

A Manual for Teachers

of

Leaving Certificate Computer Science

Table of Contents

BLANK PAGE

Manual Overview

The purpose of this manual is to provide a resource which can be used by Leaving Certificate Computer Science (LCCS) teachers to develop their own knowledge and skills of the JavaScript programming language. In doing so, we hope that teachers will gain the confidence necessary to support LCCS students as they learn to independently design and develop their own JavaScript programs.

Although the manual will serve as support material for teachers who attend the JavaScript Workshop component of our two-year CPD programme, it is envisaged that its real value will only become evident in the months and years after the workshops have taken place.

Beyond these workshops, the manual may be used as a basic reference for JavaScript programming, but more importantly, as a teaching resource that might be used to facilitate teachers in employing a constructivist pedagogic orientation towards the planning for teaching and learning of JavaScript in the LCCS classroom.

The manual itself is divided into two main sections – *core JavaScript* and *client-side JavaScript*.

Core JavaScript, as the name suggests, refers to the core aspects of the langage such as variables, datatypes, statements, expressions, sequence, selection, iteration and functions. These core constructs are covered in chapters 1-11. Chapter 12 contains a breakout activity which provides an opportunity to further develop these core JavaScript skills.

The section on client-side JavaScript introduces the two main features of JavaScript that make it *the* programming language of the web – namely document management and event handling. This section concludes with a set of tasks in breakout activity 2 designed specifically to re-inforce your client-side JavaScript programming skills.

Throughout the manual there are lots of examples and related exercises. Readers will find it helpful if they read (and try) the examples before attempting the exercises. The source code of most of the examples are available to download from the PDST GitHub repository (https://github.com/pdst-lccs/lccs-javascript) and links to sample solutions for the two main breakout activities are also available.

Conventions

To help with navigation through this manual, the following conventions are adopted:

- ✓ **Bold text** is used to highlight important new words and phrases being defined
- ✓ *Italics* are used for emphasis and also to highlight string literals
- ✓ `Courier New` font is used to denote JavaScript code such as keywords, commands and variable names

The icons illustrated below are used to highlight different types of information throughout this manual.

Space to make notes and answer questions using pen and paper.

Key point. A specific piece of information relating to some aspect of programming

Experiment. An opportunity to change code to see what happens.

Programming exercises. An opportunity for individuals/pairs to practice their JavaScript programming skills

Breakout Activities. Participants work in groups on various themed tasks

Reflection log. A space to reflect on your learning and log your thoughts.

Meet octocat! This is the GitHub integration symbol. Throughout this manual you will notice this symbol appears along with the example code. When you click on the octocat you will be directed to the source code on GitHub. Readers are recommended to copy the code from GitHub to their preferred Integrated Development Environment (IDE).

PROGRAMMER TIP
Practice! Practice! Practice!

Key tips to pass on to novice programming students.

HTML Code	JavaScript Code

1. Introduction

Background

JavaScript is considered to be *the* programming language of the web. Along with HTML and CSS, it is one of three fundamental technologies that lies behind every web page and website that you have ever visited.

HTML and CSS are responsible for the content and presentation of web pages respectively - JavaScript controls their behaviour.

Content/Structure
Paragraphs
Headings
Lists
Tables
Links (HyperText)

Presentation/Style
Font
Colour
Layout
Border
Backgrounds

Behaviour
Client-side processing
Control structures and logic
Interactive websites
Dynamic pages
Event handling
Geolocalisation
WebGL
Server-side processing (Node.js)
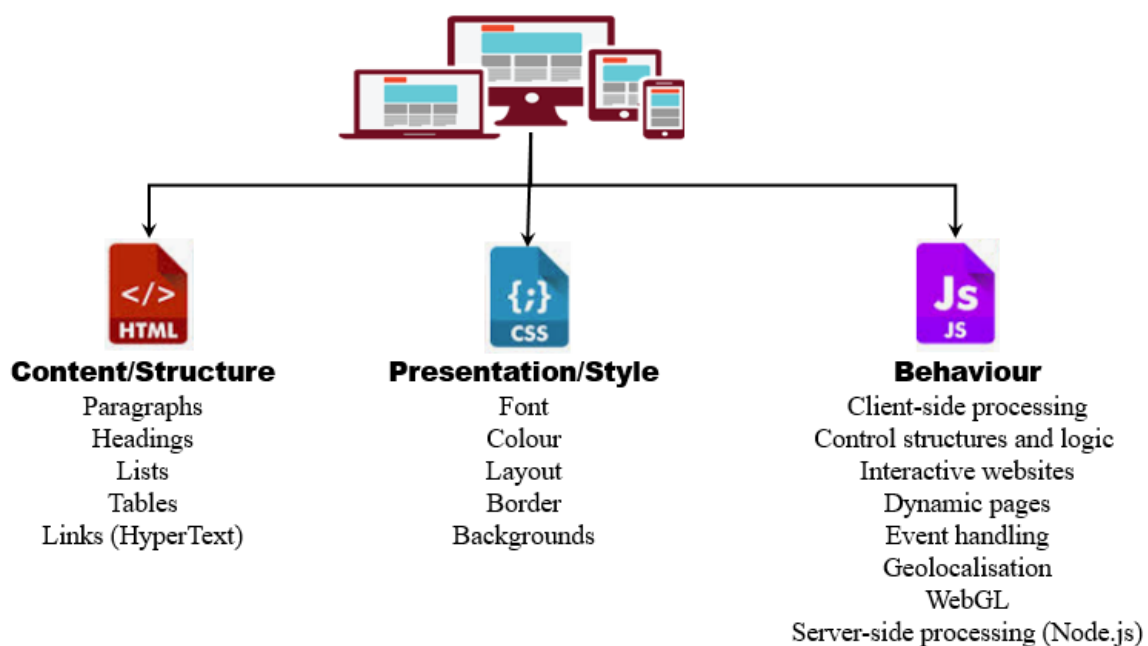
The above graphic depicts the three main types of files[1] that make up every website. These files reside on special computers called *web servers*. The main function of a web server is to 'serve up' (i.e. deliver/send) these files to clients (i.e. end-users) who request them typically through their *web browser*. Google Chrome, Firefox, Microsoft Edge, Safari and Opera are all examples of well-known browsers.

KEY POINT: Websites created with HTML and CSS are called static because their content and presentation is always the same. JavaScript can be used to add dynamic content and presentation to static websites.

---

[1] A website contains lots of other resource files especially relating to media content (e.g. audio, video, graphics etc.) but for the purpose of this discussion we are just interested in HTML, CSS and JavaScript files.

When a user visits a website – usually by entering the address (URL) of that site or by clicking on a link on some other site – a request is made from the user's browser to the site's server. The server responds by serving the requested file(s) back to the browser. At this point it is useful to think of web browsers as, not just as portals through which web pages can be accessed and displayed, but also as programs that interpret and run the HTML, CSS and JavaScript code in which these pages are written.

KEY POINT: HTML, CSS and JavaScript are all run by web browsers.

HTML and CSS are both *declarative* languages. This means that they declare to a browser what to do as opposed to how to do it. For example, the HTML `<p>` tag declares a paragraph, the `<h1>` tag declares a Level 1 heading and so on. Similarly, CSS declares a set of rules that determine the 'look and feel' of a web page. HTML and CSS are both living languages – the specification of their syntax and grammar is constantly evolving under the control of an international standards organisation called the World Wide Web Consortium or W3C for short[2].

JavaScript on the other hand is an *imperative* programming language. As such, it contains features that can be used to change a program's state and flow of control (e.g. variables, selection statements and loops). JavaScript is based on a standard defined by an organisation called the European Computer Manufacturer's Association (ECMA)[3].

The original purpose of JavaScript was to provide a means for web designers and developers to control the behaviour of their web pages. By including JavaScript code in their pages, designers and developers were enabled to implement logic that could respond in different ways to different users depending on the context. For example, JavaScript code could be used to detect and respond to invalid data being entered on a web page by a user.

In short, JavaScript enabled the development of websites with which users can interact.

---

[2] The latest living standards for HTML and CSS can be found at https://www.w3.org/TR/html52/ and https://www.w3.org/TR/css-2018/ respectively.

[3] The latest version of JavaScript (ECMAScript 2020) can be found at https://tc39.github.io/ecma262/

Client-side JavaScript vs. server-side JavaScript

For many years JavaScript was a *client-side* scripting language. This was because JavaScript programs could only be run from inside web browsers which were installed on client machines. Because of the fact that JavaScript code can run on client devices it means that a number of security restrictions had to be built into the language. The most notable of these restrictions is that JavaScript cannot read/write files to/from the client machine's disk.

Just think about it for a moment - a web developer writes a JavaScript program and includes it as part of a web page. The website is deployed into production as a set of files on a web server. At some stage, the page is requested by an end-user and the JavaScript code is run by that user's browser (which is running on the user's client device). Unless there were restrictions built-in to the language, there would be nothing to prevent web developers from writing code that could for example delete the client's entire file system! For this reason, JavaScript has no built-in file i/o capabilities – it does not allow the reading or writing of files.

KEY POINT: *Client-side JavaScript* refers to JavaScript programs that are designed to be run inside a web browser environment. It can be contrasted with *server-side JavaScript* which refers to JavaScript programs that run outside browser environments.

In recent years the JavaScript programming language has steadily evolved into a flexible and powerful general purpose language that can be used both inside and outside of web browsers. Implementations of the language that are designed to run programs outside a web browser environment are referred to as *server-side* implementations. Some notable, contemporary examples of server-side JavaScript are Node.js, Rhino, V8, and SpiderMonkey. These can all be thought of as standalone environments designed to run standalone JavaScript applications (in much the same way as any application written in any other programming language is run).

Server-side implementations of JavaScript do not have the same restrictions as apply on the client-side. As such they can include features to access the computer's file system and network resources directly (as well as many other features that are not supported by client-side JavaScript).

In this section of manual we will focus on client-side JavaScript (also called core JS).

JavaScript and Browser Wars

In the very early days of the World Wide Web Netscape Navigator overtook Mosaic as the most popular web browser. Netscape Navigator was owned by a company called Netscape.

In 1995 Netscape decided to enhance the capabilities of their browser by incorporating an interpreter for a scripting language called LiveScript. LiveScript was written by Brendan Eich and its main purpose was to allow web developers build interactive websites. LiveScript was soon renamed as JavaScript as a marketing 'ploy' designed to 'piggy back' on the name of the then new programming language called Java which was gaining rapid worldwide popularity among the software development community.

Around the same time Microsoft released Internet Explorer v1.0 and soon after in an effort to gain market share it developed its own scripting language called JScript (first released as part of IE3 in 1996.) And so, what became known as the first 'browser war' began.

By 2002 IE owned 95% of the web browser market and in 2004 Netscape essentially handed their browser code over to a new organisation called the Mozilla Foundation. The first browser war had ended but with the release of Mozilla Firefox v1.0 in 2004 the second was about to begin.

Rewind to 1997. Netscape submitted JavaScript to the European Computer Manufacturers' Association (ECMA) for standardisation. The resulting standard was called ECMAScript (or ES for short). ES5 was released in 2009, the same year Google entered the browser market with Chrome. Chrome supported HTML5 and conformed greatly with ES5 – it became an instant success.

Since 2009 browser popularity has greatly depended on the extent to which they conformed to the latest ES standard. The rise of Chrome coupled with Firefox and other browsers such as Opera and Safari eroded and eventually ended Microsoft's dominance. Chrome overtook IE as the market leader in 2012 and has remained so since. Since 2015 Microsoft's browser strategy has shifted away from IE towards its new browser, Microsoft Edge.

By 2017 Chrome had 60% of the market share and the second browser war was widely accepted as having ended.

JavaScript history timeline

Some of the main milestones in the history of JavaScript are listed below. Prior to JavaScript, the main purpose of a browser was to serve up and render/display static HTML pages.

- 1989 WWW invented by Sir Tim Berners-Lee
- 1995 Netscape release LiveScript
- 1995 LiveScript renamed to JavaScript
- 1996 Microsoft release JScript
- 1997 ES1 (ECMAScript v1.0)
- 1998 ES2
- 1999 ES3
- 2009 ES5
- 2015 ES2015 (ES6)
- 2016 ES2016 (ES7)
- 2017 ES2017 (ES8)
- 2018 ES2018 (ES9)
- ES Next – a dynamic term used to refer to the next release of ECMAScript

Since 1997 the European Computer Manufacturers' Association (ECMA) have been defining the standard for the JavaScript language. Each standard is essentially a (big) document that describes the features of the language i.e. its syntax and semantics. Browser companies take this standard and provide their own implementations. Implementations are known as JavaScript engines. These engines run the JavaScript programs written by web developers. JavaScript programmers need to be aware that their code it is not guaranteed to behave the same way in all browsers. This is because different JavaScript implementations sometimes interpret elements of the standard slightly differently - sometimes they ignore elements of the standard and sometimes they include their own features that are not part of this standard.

The names and logos of some of most popular browsers in use today are depicted below:

Chrome   Safari   Opera   Firefox   Internet Explorer   Edge

Our First Program - Hello World

In the true tradition of learning any new programming language we will start our learning journey with a JavaScript program to display the text, *Hello World!*.

```
// My first JavaScript program
console.log("Hello World!");
```

The program is fairly straightforward.

- The first line is a **comment**. Comments are ignored by the JavaScript interpreter. They are written by programmers to improve program readability. In JavaScript, comment lines begin with //. Once it sees a **double slash**, JavaScript will ignore the rest of that line.

- The second line tells the JavaScript interpreter to display the text *Hello World!* on the console. Like all programming languages, JavaScript is very fussy about syntax. We will explain more about syntax rules as we go, but for the moment it is important to know that when typing in code it must be done exactly as it appears in the listings – this includes the case, dot, opening and closing brackets, quotation marks and the semi-colon at the end of the JavaScript statement.

We now consider how to run our first program. To run any JavaScript program, we need to include it on a web/HTML page. There are two main ways to include JavaScript in a HTML document – either internally as part of the HTML or externally as a separate file. Both techniques require the use of the HTML `<script>` element and are explained on the next page.

KEY POINT: The `script` tag is HTML's way of telling a browser that it contains some JavaScript code. JavaScript code is traditionally executed in a web browser.

PROGRAMMER TIP!
`console.log` is a very simple way to display text in a browser's console window. It is used by JavaScript developers primarily for debugging purposes but can also be used as a great way to learn JavaScript.

Throughout this manual `console.log` is used a way to see what is happening in the example programs.

The HTML code shown below includes JavaScript that is written *inline* i.e. it is part of the HTML code and appears enclosed between opening and closing `<script>` tags.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello!</title>
    <meta charset="utf-8">
  </head>
  <body>
    <!- Inline JS for this web page -->
    <script>
      // My first JavaScript program
      console.log("Hello World!");
    </script>
  </body>
</html>
```

> If you click on the octocat you will be directed to the source code on our GitHub repository. From there you will be able to copy and paste the code between the opening and closing `script` tags in a simple `index.html` file such as the one shown here.

The JavaScript code is contained within the HTML file

Externally written JavaScript code is contained in a separate file (which by convention is named with `.js` extension) and is included in the HTML page using the `src` attribute of the `<script>` tag as shown here.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello!</title>
    <meta charset="utf-8">
  </head>
  <body>
    <!-- import JS file for this web page -->
    <script src="hello.js"></script>
  </body>
</html>
```

```
// My first JavaScript program
console.log("Hello World!");
```

hello.js

*The JavaScript is loaded from an external file called **hello.js***

To run (and test) your JavaScript code all you have to do is load the HTML file into any modern browser. When we load this page into the browser the message *Hello World!* Is displayed on the browser's console. To open the browser's console, use the shortcut F12 (this works on most browsers) or on a Mac COMMAND-OPTION-I. You should see something like this …

```
Hello World!
>
```

Congratulations - you have run your first JavaScript program!

2. Basic Syntax

Let's take a closer look at our first JavaScript program.

```
// My first JavaScript program
console.log("Hello World!");
```

Notes

➢ Line comments being with a double slash, `//`.

➢ `console.log` is used to display information in the browser's console.

➢ The semi-colon is used to terminate JavaScript statements.
(Although not mandatory, the use of semi-colons at the end of every statement is considered good practice.)

➢ JavaScript is **case-sensitive**. This means that JavaScript sees the code snippets shown below all differently. Try them for yourself – each snippet contains a syntax error.

```
// Syntax error (capital C)
Console.log("Hello World!");
```

```
// Syntax error (all capitals)
CONSOLE.LOG("Hello World!");
```

```
// Syntax error (capital L)
console.Log("Hello World!");
```

```
// Syntax error (missing dot)
console log("Hello World!");
```

➢ JavaScript ignores whitespace during execution. This includes blank lines.

➢ If JavaScript comes across a word (or symbol) it doesn't understand a **syntax error** will usually be displayed (normally on the browser's console)

➢ A program that contains a syntax error will not run properly. Therefore, if a programmer (i.e. you!) sees a syntax error it should be fixed immediately.

> **TEACHING TIP**
> Students should be encouraged, from an early stage, to learn how to deal with syntax errors. One way to build student confidence is to get students to fix syntactically incorrect code (and even deliberately create and fix their own syntax errors).

Experiment!
Predict what the each of the following code snippets do.
Then record the actual output. Were your predictions correct?

```javascript
// My first JavaScript program
Console.log("Hello World!");
```

```javascript
// My first JavaScript program
console.log(Hello World!);
```

```javascript
// My first JavaScript program
console.log "Hello World! ");
```

```javascript
// My first JavaScript program
console.log("Hello World!")
```

```javascript
// My first JavaScript program
// console.log("Hello World!");
```

```javascript
My first JavaScript program
console.log("Hello World!");
```

```javascript
console.log("Hello World!");
console.log("Welcome to JavaScript");
```

```javascript
/*
console.log("Hello World!");
console.log("Welcome to JavaScript");
*/
```

Features of JavaScript

In this section we provide an overview of some of the main features of JavaScript.

The JavaScript language defines many features known as **language constructs**. Variables, datatypes, operators and functions are the names of some of JavaScript's more common features. Constructs for selection (e.g. `if`, `if-else` and `switch`) and iteration (e.g. `while`, `do-while` and `for`) are also important. JavaScript programs are written by using constructs such as these in conjunction with the keywords shown on the next page.

Variables have datatypes and their values are based on the result of an expression. Expressions are evaluated by the JavaScript engine at runtime. They can be simple literals (i.e. hard-coded values such as numbers e.g. `-3`, `0`, `2.71828`, strings e.g. `"Hi Mum!"` or any of the two Boolean values, `true` and `false`). Expressions can also be arithmetic or Boolean.

Arithmetic expressions involve the use of the standard arithmetic operators such as addition (`+`), subtraction (`-`), multiplication (`*`) and division (`/`) among some others. They are usually carried out on numeric values or other arithmetic expressions, and usually result in a single numeric value being returned.

Boolean expressions are formed by using the comparison operators e.g. is equal to (`==`), strictly equal to (`===`), not equal to (`!=`), strictly not equal to (`!==`), greater than (`>`), greater than or equal to (`>=`), less than (`<=`) and less than or equal to (`<=`). Boolean expressions usually evaluate to either `true` or `false`. They can be combined into larger (more powerful and complex) Boolean expressions by using the logical operators i.e. logical NOT (`!`), logical AND (`&&`), and logical OR (`||`).

Datatypes themselves can be simple or compound. JavaScript's simple datatype are `number`, `string`, `boolean`, `null` and `undefined`. Compound datatypes are also known as objects. Before the JavaScript engine starts to execute a program it creates a special object called **the Global object**. The Global object contains a number of special properties and functions that can be used by any JavaScript program. The most notable of these are `Infinity`, `NaN`, `undefined`, `isFinite()`, `isNaN()`, `parseInt()`, `parseFloat()`, `String()`, `Number()`, `Math()`, `Boolean()`, `Array()`, `Date()` and `Object()`.

JavaScript reserved words

A **reserved word** is a word that has special meaning to JavaScript. Each word has an associated syntax and semantics (meaning) which is described in the language specification. When the JavaScript engine comes across a reserved word in a program, its behaviour is governed by the rules set out in the specification.

Programmers should use reserved words for their intended purposes only. In particular, this means that reserved words should never be used as names for variables or identifiers (i.e. names of variables and functions) in any JavaScript program. A major part in the journey of learning JavaScript (and any programming language) is becoming familiar with the meaning of its reserved words. The syntax and semantics of many of JavaScript's reserved words will be explained and exemplified throughout this section of the manual.

The full list of JavaScript reserved words is shown in the table below.

await	debugger	false	instanceof	this	void
break	default	finally	let	throw	while
case	delete	for	new	true	with
catch	do	function	null	try	yield
class	else	if	return	typeof	
const	export	import	super	undefined	
continue	extends	in	switch	var	

ECMAScript 2018 keywords

Notes:

➤ Although the words `true`, `false`, `let`, `null` and `undefined` are strictly speaking not JavaScript reserved words, it is fair to treat them as if they were. Other words that fall into this category but are not listed above include `boolean`, `byte`, `char`, `double`, `float`, `long`, and `short`.

➤ It is also fair to think of global variables and functions such as those referred to on the previous page as keywords, and as such these should never be used as identifiers either.

Flow of Control

The **flow of control** refers to the sequence in which the lines of a program are executed.

Key in (or download) the following two programs (one at a time!) and compare their outputs.

The output is …..

```
// Displays a welcome message in 5 languages
console.log("Welcome to JavaScript!");
console.log("Fáilte roimh JavaScript!");
console.log("Bienvenido a JavaScript!");
console.log("Willkommen bei JavaScript!");
console.log("Bienvenue sur JavaScript!");
```

and,

The output is …..

```
// Displays a welcome message in 5 languages
console.log("Bienvenue sur JavaScript!");
console.log("Willkommen bei JavaScript!");
console.log("Bienvenido a JavaScript!");
console.log("Fáilte roimh JavaScript!");
console.log("Welcome to JavaScript!");
```

What did you notice about the output (in relation to the programs)?

KEY POINT: Lines of code are normally executed in the same order in which they appear in a program. This is called *sequential processing*. We say that the flow of control is *sequential*.

We will see later that the JavaScript language contains features (constructs) which allow programmers to write code that executes in a non-sequential fashion. Two such features are called **selection and iteration**.

➢ **Selection** is used by programmers when they want one of possibly several blocks of code to be selected for execution. The most common selection constructs are `if` and `if-else` statements.

➢ **Iteration** (or looping) is used by programmers when they want the same block of code to be executed possibly multiple times. The most common looping constructs are `for` and `while` statements.

A note on indentation

Indentation refers to the empty space(s) at the beginning of a line of code (called leading space(s)).

One key difference in syntax between JavaScript and Python is that unlike Python, JavaScript is not fussy about indentation. The listings below are all syntactically correct and semantically equivalent (i.e. they have the same meaning to the JavaScript interpreter).

```
console.log("Welcome!");
 console.log("Fáilte!");
console.log("Bienvenido!");
 console.log("Willkommen!");
console.log("Bienvenue!");
```

```
console.log("Welcome!");
 console.log("Fáilte!");
  console.log("Bienvenido!");
   console.log("Willkommen!");
    console.log("Bienvenue!");
```

```
    console.log("Welcome!");
   console.log("Fáilte!");
  console.log("Bienvenido!");
 console.log("Willkommen!");
console.log("Bienvenue!");
```

JavaScript is not fussy about indentation. These three listings are all correct.

For the sake of clarity, we recommend using the same level of indentation for blocks of code that are logically related. By default, your JavaScript code should *not* be indented unless the indentation improves its readability.

```
console.log("Welcome!");
console.log("Fáilte!");
console.log("Bienvenido!");
console.log("Willkommen!");
console.log("Bienvenue!");
```

It is good practice not to use leading spaces unless you have to

Lines of code that belong together in a program are referred to as **code blocks**.

JavaScript uses curly braces to delimit blocks of code. The opening **curly brace** (i.e. {) marks the start of a code block and the closing curly brace (i.e. }) marks the end of a code block.

KEY POINT: JavaScript's syntax requires that every opening curly brace must have a corresponding closing curly brace.

3. Datatypes and Literals

It is important for programmers to be aware of the different types of data their programs need to deal with. For this it is necessary to understand the underlying types supported by a language. JavaScript supports both primitive (simple) and object (compound) types. Let's start by looking at JavaScript's primitive types.

Primitive Datatypes

The five common JavaScript primitive types are illustrated in the graphic below:

```
         ┌──────────────────┐
         │    JavaScript    │
         │  Primitive Types │
         └──────────────────┘

  ┌────────┐ ┌────────┐ ┌─────────┐ ┌──────┐ ┌───────────┐
  │ number │ │ string │ │ boolean │ │ null │ │ undefined │
  └────────┘ └────────┘ └─────────┘ └──────┘ └───────────┘
```
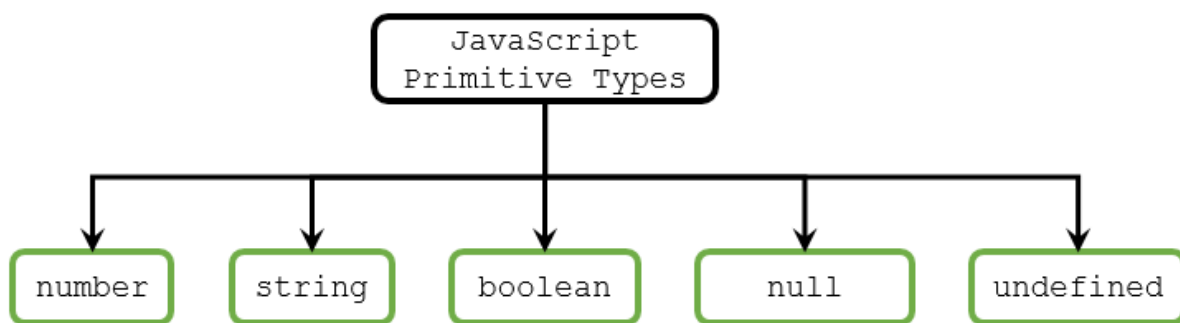
JavaScript primitive types

A **literal** is any value that can appear directly in a program. Sometimes, literals are referred to as **hard-coded values**. Literals, just like all values, have an underlying datatype. It makes sense therefore to talk about **numeric literals** or **string literals**.

Let's take a look at some examples involving JavaScript primitive datatypes and associated literals.

string

The `string` datatype is used to represent string literals. A *string literal* is any sequence of characters enclosed in quotation marks – either single or double. String literals can contain normal alphabetic characters, numeric characters and basically any Unicode symbol. See https://unicode-table.com/en/#combining-diacritical-marks for a complete list of symbols that can be used in string.

The following code demonstrates the use of string literals:

```
console.log("A string is any sequence of characters enclosed in double (or single) quotes.");
console.log("A string can contain digits e.g. 1,2,3 and symbols e.g. €, %, &, $ etc. or ...");
console.log("... any Unicode character from the Unicode table e.g. \u03B1, \u03B2, \u03B3, \u03C0");
console.log("console.log", "can handle", "multiple strings", "separated by commas");
```

The code causes the following output to be displayed on the console:

```
A string is any sequence of characters enclosed in double (or single) quotes.          hello.js:1
A string can contain digits e.g. 1,2,3 and symbols e.g. €, %, &, $ etc. or ...          hello.js:2
... any Unicode character from the Unicode table e.g. α, β, γ, π                         hello.js:3
console.log can handle multiple strings separated by commas                             hello.js:4
```

JavaScript accepts the vast majority of literal characters in a string literal. However, in order to understand a small number of special characters (e.g. tab, single quote, backslash) JavaScript requires the use of an **escape sequence**. The escape sequence identifies the special character to JavaScript. We say the character is *escaped*. The backslash character introduces an escape sequence in a string.

Escape Sequence	Meaning
\n	Newline
\t	Tab
\'	Single Quote
\"	Double Quote
\\	Backslash
\uXXXX	Unicode character

Common escape sequence characters

KEY POINT: An escape sequence is used to identify certain special characters (usually white-space or non-printable characters) that cannot be represented literally as part of a string.

Experiment!
Predict the output of each of the code snippets shown below.
Then record the actual output. Were your predictions correct?

```
console.log("\"Hi\", she said.");
```

```
console.log("A new\nline");
```

```
console.log("Gobbledy\tgook!");
```

```
console.log("Now I know my \x41 \x42 \x43");
```

number

The `number` datatype is used to represent both integers and decimals (i.e. floating point numbers).

The unary operators + and – can be used to denote the sign of any numeric literal (i.e. positive or negative). Numeric literals cannot contain commas or spaces. Examples of base 10 integer literals are $-20, 0, 12345$ and 7.

Integers literals in number systems other than base 10 can be specified by using special leading characters (i.e. characters placed before the number). For example,

- *Hexadecimal integers* are prefixed by `0x` (or `0X`) e.g. 0xFF is decimal 255
- Base 2 or *binary integers* are prefixed by 0b or 0B e.g. 0b11111111 is decimal 255
- Although not part of the official language specification many implementations support o*ctal integers* by using 0 as the prefix e.g. 0777 is decimal 255

A floating point literal must be made up of at least one digit and either a decimal point or `E` (or `e`). For example, 10E6 (i.e. $1,000,000$), `3.14E-2` (i.e. `0.0314`), `-0.000123` and `.000123` are all valid floating point literals.

Two special properties relating to number are defined by JavaScript: `Infinity` and `NaN`. These will both be discussed shortly.

boolean

The two JavaScript `boolean` literals are `true` and `false` – these should both treated as reserved words.

JavaScript also supports a notion called '*truthy*' and '*falsey*' – these refer to any expression that evaluates to the literals, `true` and `false` respectively. The following values all evaluate to `false` (as well as `false` itself):

- The number zero (0)
- An empty string ("")
- `null`
- `undefined`
- `NaN`

All values other than those listed here evaluate to `true` and so are called 'truthy'.

null and undefined

These are two different types that share the same (empty) value. They are both 'falsey' values, but also have subtle differences that are best described by their uses:

- `null` is used to indicate the (intentional) absence of a value
- `undefined` is used to indicate the lack of definition of a value respectively. If a variable has a value of `undefined` it usually means that the variable has been declared but has no value.

In JavaScript two values are said to be equal if they share the same value. However, they are *strictly equal* to each other if and only if they both have the same value and type. Therefore, while `null` and `undefined` are equal (because they are the same value) they are not strictly equal (because they are different types). This is a perfect example of the idiosyncratic nature of JavaScript!

KEY POINT: JavaScript is a dynamically typed language which means that during their lifetime, variables can be used to store values of different underlying datatypes. You can use the `typeof` operator to inspect a variables type at any stage in a JavaScript program..

PROGRAMMER TIP!
Use the `typeof` operator to discover the datatype of a value (or expression).
Try the following:
```
console.log(typeof(true)); // boolean
console.log(typeof(12)); // number
console.log(typeof(1.2)); // number
console.log(typeof("Joe")); // string
console.log(typeof(undefined)); // undefined
console.log(typeof(null)); // object
```

Object Types (compound types)

Object-oriented programming (OOP) is a programming paradigm that involves the creation and use of objects. Objects are a type of data that are used by programmers to represent real-world things (e.g. bank account, student, bicycle). Because objects can consist of multiple values they are referred to as a compound datatype. This can be contrasted with the simple or primitive types we have just discussed.

JavaScript is an example of an object-oriented language which means that it has features to support object data structures. Object data structures can be thought of as a programmatic representation of a person, place or thing. In this sense OOP provides a means by which these 'things' can be modelled in a program. These objects can be described by their properties (using variables) and behaviour using functions (more commonly referred to as methods in the world of OOP).

Using JavaScript to create objects is beyond the scope of this manual (and LCCS) but it is important to understand what they are and how they can be used. Suffice to say for the moment that if we have an object `o` that has a property `p` and a method `m` then the dot (aka member) operator can be used to access the property and invoke the method. The syntax for this is `o.p` and `o.m()` respectively.

The illustration below[4] depicts how a real-world thing such as a bicycle could be represented (or modelled) as an object in a program.

| *A software object* | *A bicycle modelled as a software object* |
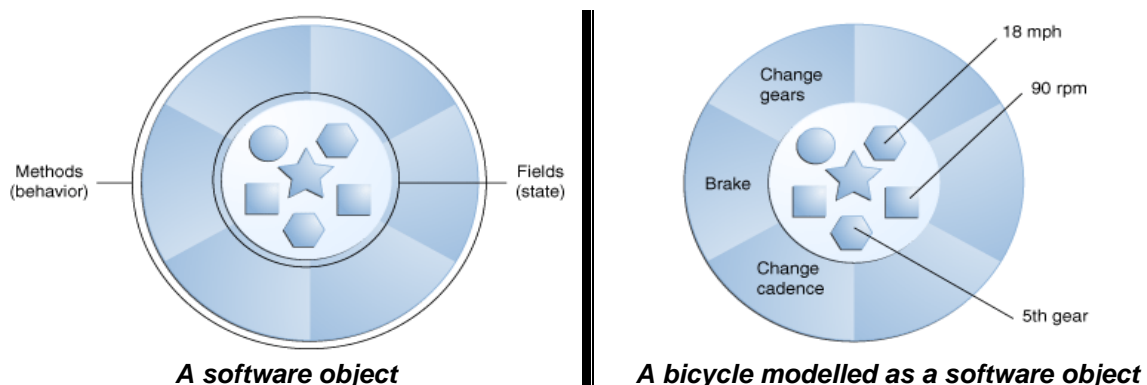
Objects have state (i.e. properties) and behaviour – state is represented as variables and behaviour is represented as methods. If an OO programmer chooses to model a bicycle as

---

[4] Source: https://docs.oracle.com/javase/tutorial/

an object they can store state such as speed, cadence and gear in the object's variables; and they can describe the bicycles behaviour using methods such as accelerate, break, change gear etc. (An online store for a bicycle shop might use an application that creates multiple runtime instances of these objects – one for each individual bicycle.)

Built-in objects

JavaScript comes with built-in support for three types of objects – browser objects, document objects and global objects. Because these objects are models they can be referred to and the Browser Object Model, the Document Object Model and the Global Object Model (or BOM, DOM and GOM for short!).

The **Browser Object Model** consists of a set of properties and methods that relate to the browser that your JavaScript program runs within. The two methods – `alert` and `prompt` – which we will be using extensively throughout this manual belong to the browser's object model. In fact, `console.log` which we have been using in the examples to display information on the browser's console belongs to the BOM.

The **Document Object Model** is feature of JavaScript that makes dynamic and interactive web pages/websites possible. The DOM is a runtime representation of a HTML document or web page. It provides a programmatic view of the HTML code behind every page. We can use the DOM to change the content and behaviour of a web page 'on the fly'. This means that we can alter, add or delete anything relating to the content/appearance of a page from within a running JavaScript program. And all this is made possible by the DOM. We will return to the DOM at a later stage in the manual.

The **Global Object Model** is important to be aware of, even to the novice JavaScript programmer. This is because it contains a number of useful properties and functions that we can make use of in even the simplest of our JavaScript programs. These properties and methods can be categorised under the following sub-headings under the Global Object Model

- global properties
- global constructor functions
- build-in objects and
- global functions

KEY POINT: When the JavaScript engine first starts it creates a single instance of its global object. The properties and functions of this object can be accessed from anywhere in a JavaScript program via a special object variable called `this`.

Some of the more commonly used elements of JavaScript's Global Object are depicted in the tree diagram below. This is followed by a brief outline of some of these properties and functions.
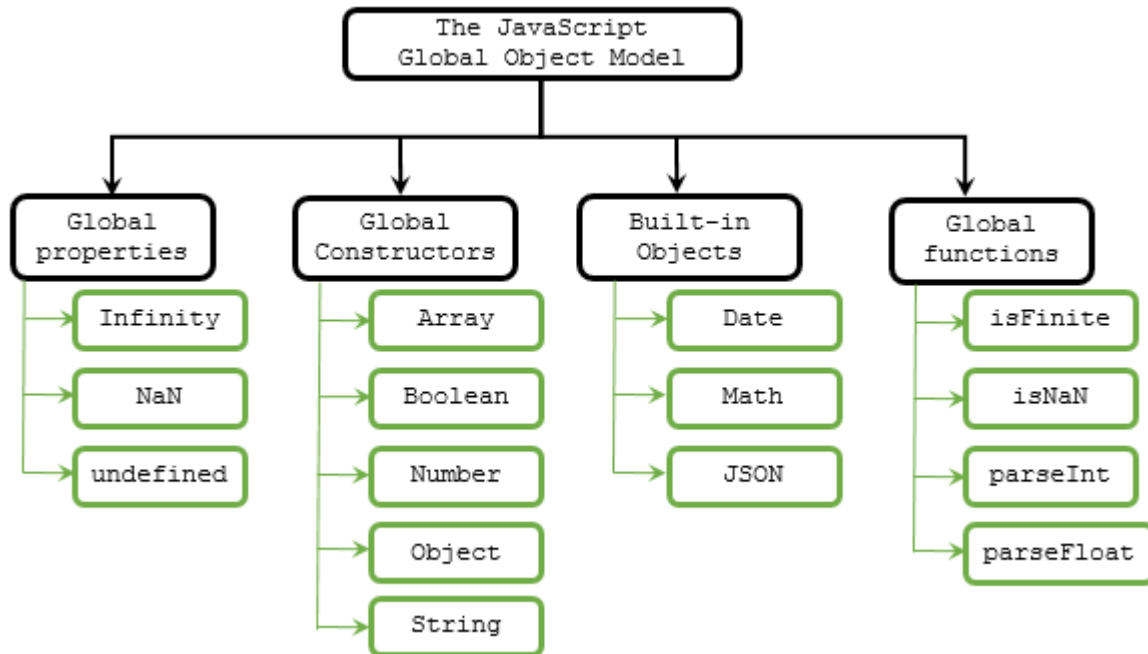
JavaScript global objects

Global Properties

`Infinity`

Internally JavaScript represents numbers (i.e. integers and decimal) as 64-bit floating point values. Integer values from -2^{53} up to 2^{53} can be represented accurately. For decimal numbers the range is $\pm 1.797693134862315E308$

The value `Infinity` is used to denote numeric values that exceed this stated maximum and `-Infinity` is used to denote numeric values that are smaller than the possible minimum. Interestingly enough, division by zero returns `Infinity`.

`NaN`

`NaN` stands for Not-A-Number. The value `NaN` denotes a non-numeric value. It is used in JavaScript to indicate that a value cannot be represented as a number (typically in situations where the value is required to be a number). For example, if you try to multiply an integer by some string literal the result will be `NaN`.

Global Constructors

These are special functions which, when called, create objects.

Global constructors are often used to represent primitive types as objects – a process called object *wrapping*. For example,

- `String()` is a constructor function that can be used to represent values of type `string`
- `Number()` is a constructor function that can be used to represent values of type `number`
- `Boolean()` is a constructor function that can be used to represent values of type `boolean`

`Array()` is the name of another constructor function. Both arrays and strings are discussed in detail elsewhere in this manual.

The following line of code demonstrates how the numeric literal `19.64738` can be wrapped using `Number()`

```
Number(19.64738)
```

By wrapping this primitive value using the global constructor for `Number`, our code can now avail of functions such as `toFixed` and `toPrecision` which are built-in as part of the definition of the `Number` object.

This is illustrated in the following line of code which displays `19.64738` using 4 significant digits i.e. `19.65`.

```
console.log(Number(19.64738).toPrecision(4));
```

The code looks more complicated than it actually is – at runtime it gets executed in three steps as follows:

1. `Number(19.64738)`

   The global constructor for `Number` is called. This call results in the creation of a `Number` object for the primitive value `19.64738`.

2. The function `toPrecision` is called on the `Number` object just created. The purpose of this call in the code shown here is to generate a string representation of the number using four significant digits.

3. The result i.e. `19.65` is displayed using `console.log`

The names and a brief descript of some other `Number` functions (methods) are shown in the table below.

Method name	Description
`x.toExponential()`	Returns a string representation of `x` in exponential notation `19.64738.toExponential()` → `1.964738e+1`

`x.toFixed(len)`	Returns a string representation of `x` with `len` digits after the decimal point
	
	`19.64738.toFixed()` → `20`
	`19.64738.toFixed(1)` → `19.6`
	`19.64738.toFixed(2)` → `19.65`
	`19.64738.toFixed(3)` → `19.647`

`x.toPrecision(len)`	Returns a string representation of `x` rounded to `len` significant digits
	
	`19.64738.toPrecision(0)` → `19.64738`
	`19.64738.toPrecision(2)` → `20`
	`19.64738.toPrecision(4)` → `19.65`
	`19.64738.toPrecision(6)` → `19.6474`

One common use for `Number()` is to convert values from different datatypes to numbers that can be worked with using the above functions. This is illustrated in the examples shown below

```
Number("123"); // convert the string literal to 123
Number("1.23"); // 1.23
Number(true); // returns 1
Number(false); // returns 0
```

If the value passed into Number cannot be converted to a type of `number` then the result will be `NaN`.

```
Number("Joe"); // NaN
```

PROGRAMMER TIP
Objects can be constructed using the `new` operator. For example, the statement below will construct a new `Number` object with a value of 123.
`new Number(123);`

Global Functions

`isNaN` **and** `isFinite`

Example uses of `isNaN` and `isFinite` are shown side by side in the table below.

isNaN	IsFinite

➢ `isNaN("fifty"); // true`
➢ `isNaN("10"); // false`
➢ `isNaN(99); // false`
➢ `isNaN(NaN); // true`
	➢ `isFinite("fifty"); // false`
	➢ `isFinite("10"); // true`
	➢ `isFinite(99); // true`
	➢ `isFinite(NaN); // false`

The difference between `isNaN` and `isFinite` is subtle. `isNaN` returns `true` if the value passed in is not of type `number` or `NaN`. Otherwise it returns `false`. `isFinite` returns `true` if its argument is a finite number. Otherwise it returns `false`.

`parseInt` **and** `parseFloat`

These two global functions are typically used to convert strings to base-10 integers and floating-point numbers respectively.

Example uses of both functions are shown side by side in the table below.

parseInt	parseFloat

➢ `parseInt("111"); // 111`
➢ `parseInt("111two"); // 111`
➢ `parseInt("111",2); // 7`
➢ `parseInt("4.7",10); // 4`
➢ `parseInt("4.7"); // 4`
➢ `parseInt("Joe"); // NaN`
➢ `parseInt(true); // NaN`
	➢ `parseFloat(0.0123E+2) // 1.23`
	➢ `parseFloat(123E-2) // 1.23`
	➢ `parseFloat("50.5") // 50.5`
	➢ `parseFloat("50point5") // 50`
	➢ `parseFloat(".5") // 0.5`
	➢ `parseFloat(fifty5) // NaN`
	➢ `parseFloat(false) // NaN`

Note that the base (radix) of the value to parse can be specified as an optional second argument to `parseInt`. Hence, `parseInt("111",2);` takes `"111"` as a base-2 number and yields a result of 7 which is the base-10 equivalent.

4. Variables and Assignments

A **variable** is a placeholder for data.

During execution, a program's data is held in temporary memory locations that are referenced by variable names. Variable names can be thought of as memory locations.

> **TEACHER TIP**
> Some students find it useful to think of a variable as a 'box' in the computer's memory (i.e. a memory location) where a value is stored. The name of the variable can be written on the outside of the box.

Variables are created by programmers so that their programs can temporarily store data which can then be used at another point in a program. The *scope of a variable* refers to the parts of a program where a variable can be legitimately used. The scope of a variable can be *local* or *global* depending on how and where it is declared within the program.

KEY POINT: A variable is a programming construct used to store (remember) data.

> **TEACHER TIP**
> The concept of a variable is fundamental to programming. When presenting new problems it can be useful to ask students to think about what variable(s) their program will require.

Reflection
Identify some data/variables that might be needed by the systems listed below

System	Suggested variable names (to store data used in the system)
ATM System	PIN, option, account number, balance, amount requested, date/time
A School Library System	
Netflix	
Snapchat	
Spotify	
Your favourite console game	

Guidelines and Rules for Naming Variables

As a general guideline, programmers should choose names for variable that are simple and meaningful. A meaningful name is one that tells something about what the variable is used for. The use of meaningful variable names makes programs more readable and understandable to fellow programmers.

When choosing a name for a variable it can be helpful to think of a noun that describes the purpose of the variable.

Since JavaScript programs are written using the Unicode character set it follows that any character can be used in a variable name. The following exceptions apply:

➢ The first character in a variable name must be a letter, an underscore (_), or a dollar sign ($). Letters and digits can follow.

➢ Spaces, dashes and dots cannot be used as part of a variable name.

➢ A variable name cannot be the same as any of the JavaScript keywords or reserved words (e.g. `if`, `else`, `function` etc.)

In JavaScript it is considered standard practice to separate interior words in multi-word variable names using *camel case* e.g. `firstName`, `highScore`, and `payRate`. The use of an underscore to separate individual words written in lower case is also considered acceptable e.g. `first_name`, `high_score` and `pay_rate`.

The following are examples of valid (legal) variable names: `guess`, `_randomNum`, `$userName`, `x_pos`, `höhe`, and `x1`.

The following are examples of invalid (illegal) variable names: `1x`, `function`, `pay-rate`, and `student.name`

If the JavaScript engine comes across a name it does not understand it will display a syntax error on the console.

Declaring Variables

To declare a variable means to introduce a variable to a program for the first time i.e. to let JavaScript know here is a new storage location for data.

The `let` keyword is used to declare a variable in JavaScript.

The code snippet shown below illustrates the use of `let` to declare three variables – `firstNumber`, `secondNumber` and `sum`.

```
let firstNumber = 1;
let secondNumber = 2;
let sum = firstNumber + secondNumber;
console.log("%d + %d = %d", firstNumber, secondNumber, sum);
```

When this program is run the console output will look something like this:

```
1 + 2 = 3
>
```

Variables can also be declared using the `var` keyword but as we will soon see we will *not* be recommending the use of `var`. The snippet below demonstrates how to declare variables using `var`. The output displayed is identical to that of the previous snippet.

```
var firstNumber = 1;
var secondNumber = 2;
var sum = firstNumber + secondNumber;
console.log("%d + %d = %d", firstNumber, secondNumber, sum);
```

So, `let` and `var` are JavaScript keywords used to declare variables. Another closely related keyword which we will explain soon is `const`.

When a variable is declared for the first time it is recommended to assign it some initial value. This is done using an assignment statement.

PROGRAMMER TIP
It is a good habit to initialise a variable as part of its declaration.

Assignments

The process of setting the value of a variable is called *assignment*. We say *a variable is assigned a value* or, *a value is assigned to a variable*

The general form of an assignment statement is

```
<variable-name> = <expression>;
```

The name of the variable appears on the left hand side and an expression appears on the right hand side. **The '=' symbol in the middle is the JavaScript *assignment operator*.**

KEY POINT: Although the symbols used to denote the JavaScript assignment operator and a mathematical equation are identical, they should not be confused as they have completely different meanings.

The use of '=' indicates an *assignment statement*. **When the JavaScript engine comes across an assignment statement it evaluates the expression on the right hand side first**. The result of this evaluation is then stored in the variable named on the left hand side.

The right-hand-side expression can be any combination of:

- a *literal value* such as a string (e.g. "*Welcome*") or a number (e.g. 7, 3.14)
- other variable(s)
- a call to a function which itself returns a values
- any combination of literal values, variables, and/or functions (i.e. any valid JavaScript expression)

TEACHER TIP
Assignments can be modelled 'unplugged' using physical boxes for variables with values written on separate pieces of paper.

The use of `let` vs. `var`

There are some subtle differences between the `let` and `var` – two of these differences can be explained in terms of scope and hoisting.

Scope

When a variable is declared using the `var` keyword its scope will be global or local depending on whether it was declared outside or inside a function. Variables declared inside a function using `var` are only visible within that function; otherwise they are visible to the entire script in which they are declared.

The scope of a variable declared using the `let` keyword is narrower. This is because its scope is confined to the block of code within which it is declared. (Recall that curly braces are used to mark the start and end of JavaScript code blocks.) This gives programmers much more control over the sections of code which can use their '`let`' variables. For example, is possible to declare a '`let`' variable with a scope that does not extend outside a `while` loop or the body of an `if` statement.

Hoisting

Before a script is executed, the browser (as part of the page load process), scans the entire code for variables declared using the `var` keyword. As it does so it builds up a list of '`var`' variables. Each time the browser comes across a new '`var`' variable it adds it to this list.

Because of this pre-processing step, the JavaScript engine knows the names of all '`var`' variables in advance of running a program. It treats '`var`' variables as though they are declared at the top of the program. The net effect of this phenomena which is called **hoisting** is that '`var`' variables can appear (anywhere) in a script before they are declared.

KEY POINT: Declaring a variable using the `var` keyword anywhere in the code is equivalent to declaring it at the top. This is called *hoisting*.

The following two code snippets are executed in the same way by the JavaScript engine.

```
month = 1;
var month;
console.log("The month is", month);
```

```
var month;
month = 1;
console.log("The month is", month);
```

These two snippets are equivalent.
The use of `var` means `month` is hoisted before the code runs.

Variables declared using the `let` keyword are *not hoisted* by JavaScript. Such variables must be declared before they are used (which is a very wise step in any case!).

For example, the following code would result in an error because and attempt is made to assign 1 to `month` before it is declared.

```
// The use of let is good practice because it forces programmers to ...
// ... declare variables before they are used.
// This code generates a syntax error.
month = 1;
let month;
console.log("The month is", month); // Error
```

PROGRAMMER TIP!
Use `let` in preference to `var` to declare variables.

Further Reading

See https://www.geeksforgeeks.org/difference-between-var-and-let-in-javascript/ for more information on the difference between `let` and `var`.

An explanation of why the name "**let**" was chosen can be found here.
https://stackoverflow.com/questions/37916940/why-was-the-name-let-chosen-for-block-scoped-variable-declarations-in-javascri

KEY POINT: A variable is said to be declared when it has been made known to the program. The recommended way to declare a variable is to use the `let` keyword.

Constants

The syntax for declaring a constant is similar to that for declaring a variable. The `const` keyword is used as opposed to `let` (or `var`).

For example, the line below declares a constant called `PI` and initialises it to 3.14.

```
const PI = 3.14;
console.log("PI:", PI); //3.14
```

Many programmers conventionally denote constants using upper case letters (with individual words being separated by underscore if the constant contains more than a single word.)

Constants must be initialised as part of their declaration. (We will soon see that this is not the case for variables.) Thus, the following code would generate an error.

```
const PI; // SyntaxError: Missing initializer in const declaration
```

As you might expect the value of a constant cannot be changed once it has been declared. This is illustrated in the line below which attempts to change the value of the previously declared constant, `PI`.

```
PI = 3.14159; // TypeError: Assignment to constant variable.
```

Other points worth noting about constants are:
- The scope rules for constants are the same as those for '`let`' variables. Therefore, `const` identifiers do not hoist to the top of the block (unlike `var` identifiers).
- The name of a constant cannot be the same as the name of a variable that has already been defined within the same scope.
  Therefore, the code shown below would generate an error saying: *Identifier 'corkToDublin' has already been declared*

```
let corkToDublin;
const corkToDublin = 258.5; // SyntaxError: …
console.log("Distance:", corkToDublin);
```

Undefined variables

An undefined variable is one that has been declared but not initialised. One of the many subtle differences between JavaScript and Python is that variables in JavaScript can be declared without initialisation. (Python requires that variables are initialised as part of their declaration.)

The code snippet shown below declares a variable called `area` without any initialisation. (The identifier could have been created in the same way using `var` instead of `let`)

```
let area;
console.log("The area is", area); // The area is undefined
```

When this happens JavaScript automatically assigns an initial value of `undefined` to the variable in question. (`undefined` is actually a JavaScript global variable.)

KEY POINT: An `undefined` variable is one that has been declared but not (yet) assigned a value.

Undefined variables can be a source of problems and are therefore considered bad practice. For example, if you try to add a number to an undefined value – as illustrated in the code snippet below - the result will be `NaN`. `NaN` stands for *Not a Number*. As already mentioned it is one of JavaScript's predefined global variables.

```
let count;
count = count+1;
console.log("Count:", count); // Count: NaN
```

For this reason, among others, it is generally considered to be poor programming practice not to initialise a variable as part of its deceleration.

TEACHER TIP
Students should be encouraged to initialise a variable as part of its declaration.

Multiple declarations

Undefined variables can be considered acceptable if they are initialised together using a technique that is sometimes called *assignment chaining*. Assignments can be 'chained' together in order to assign a single value to several variables.

```
let x, y, z; // declare 3 (undefined) variables
x = y = z = 0;
console.log("x:", x, "y:", y, "z:", z); // x: 0 y: 0 z: 0
```

This program generates the output shown below:

```
    x: 0 y: 0 z: 0
>
```

The following code demonstrates how to declare and initialise several variables in a single statement. Note that the values of `h` and `l` are both undefined after the lines are executed. You should note that this is considered poor practice.

```
let length=10, breadth=15;
let l, b=20;
console.log("length:", length, "breadth:", breadth);
console.log("l:", l, "b:", b);
```

When this program is run the following output is displayed:

```
    length: 10 breadth: 15
    l: undefined b: 20
> |
```

It is also possible to declare a variable more than once using `var`. For example, the code snippet below (another fine example of bad practice!) is legal. However, this is not possible with `let` – another good reason to use `let` instead of `var`!

```
var distance;
console.log("Distance:", distance); // Distance: undefined
var distance=10;
console.log("Distance:", distance); // Distance: 10
```

When the previous program is run it generates the following output:

```
Distance: undefined
Distance: 10
>
```

KEY POINT: The use of `let` safeguards against multiple declarations of the same variable in the same block of code.

When initialising multiple variables in a single statement care must be taken to ensure that the required values are defined in advance. For example, the statement, `let x=0, y=x;` is legal because `x` gets its value before `y` is initialised. The line, `let x, y=x;` is also legal but it results in two undefined variables - `x` and `y`. However, the line `let x=y=0;` results in an error because `y` is undeclared.

The code below makes the point that care must be taken not to inadvertently overwrite any variables during initialisation. (The initial value of `y` is overwritten by the value in `z`!)

```
let y = 0;
let z = 1;
let x = y = z;
console.log("x:", x, "y:", y, "z:", z); // x: 1 y: 1 z: 1
```

Undeclared variables

An undeclared variable is a variable that is defined with a value without using any of the three keywords - `let`, `var`, or `const`.

For example, the following code defines and displays the contents of an undeclared variable called `n`.

```
n = 7;
console.log("The value of n is", n);
```

While the above code is legal it is not recommended.
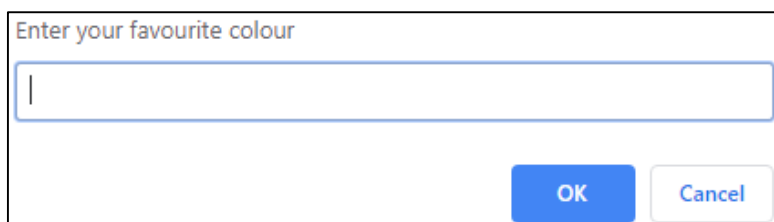
The JavaScript engine generates a warning when it comes across undeclared variables in a program. This is because as they can lead to unexpected program behaviour.

User Input

The simplest way to capture data from the end-user into a JavaScript program is to use the `prompt` command. `prompt` is a browser API which, when called, displays a pop window with an entry field that can be used to enter data. An example of such use is shown here:

```
let favColour = prompt("Enter your favourite colour");
```

The above line would cause the screen shown here to popup in the user's browser. The string *Enter your favourite colour* is displayed as a prompt to the end user on the popup.

If the user clicks the `OK` button the value entered by the user is returned as a string and assigned to the variable `favColour`. Prompt returns `null` if the user clicks the `Cancel` button (even if there is some text in the entry field).

`prompt` returns whatever value is entered as a string. This means that if you want to capture numeric data you will need to convert it in your program. One common way to do this is to wrap the call to prompt inside the `Number` object as shown here.

```
let age = Number(prompt("Tell me your age"));
```

In this case whatever value the user enters is converted to a number and stored in the variable `age`. This means that `age` can be used in arithmetic operations such as subtraction as shown below.

```
let age = prompt("Enter your age and I will tell you the year you were born");
let currentYear = new Date().getFullYear(); // Get the current year, yyyy
console.log("You were born in", currentYear - age);
```

The use of prompt is intrusive for the end user - later we will examine how to capture data from a form on web page using DOM and event handling but for the moment we will use prompt.

JS Programming Exercises – Variables

1. Write a line of code to do each of the following:
   a) declare an integer variable called `result`, initialised to zero
   b) assign the value 50 to `result`
   c) display the value of `result`
   d) assign the value 80 to `result`
   e) display the value of `result`

2. Write a line of code to do each of the following:
   a) declare a variable called `firstName`, initialised to your own first name.
   b) declare a variable called `lastName`, initialised to your own surname
   c) display the contents of the variables on the output console.

3. Read carefully the following block of code and answer the questions which follow:

```
let a=10;
let b=5;
let temp=a;
a=b;
b=temp;
```

   a) How many variables are there? (What are their names?)
   b) What are their initial values?
   c) What are the values of `a`, `b` and `temp` at the end of the program?
   d) Explain – in one sentence - what the program does?
   e) What is the purpose of the variable `temp`?

4. List some variables that might be used in the following systems.
   a) ATM System
   b) Airline Reservation System
   c) College Application System
   d) Amazon/Facebook
   e) Calculator Application

5. Describe what do you think the following program does?

```
let colour =  prompt ("Please enter your favourite colour");
console.log("Your favourite colour is", colour);
```

6. Key in the following code and run it. What does it do?

```
let firstName = prompt("What is your name?");
let colour = prompt("What is your favourite colour?");
console.log("Hi", firstName, "Your favourite colour is", colour);
```

7. Modify the code in the previous question so that it asks the user for their surname as well as their first name.

8. Complete the following:
a) Write a line of code that asks a user how many brothers they have. Store the value entered in a variable called `brothers`.
b) Now write a line of code that asks a user how many sisters they have. Store the value entered in a variable called `sisters`
c) Finally, write a line that uses the values entered in parts a) and b) to output a message like, *You have 2 brothers and 3 sisters*

5. Arithmetic Operators and Expressions

In this section we will examine the valid operations for different primitive datatypes.

Arithmetic Operations

JavaScript supports the basic binary arithmetic operations of addition (+), subtraction (-), multiplication (*), division (/), remainder (%, aka, modulus), and exponentiation (**, aka power).

All operators work with two operands which can be numeric literals or variables and evaluate to a single numeric value.

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Remainder	x % y
**	Power	x ** y

Common arithmetic operators

Key in (or download) the following code and record its output.

JavaScript Code	Use the space below to record the output
`let x = 2, y = 3, z=6;` `console.log("Sum:", x, "+", y, "=", x+y);`	
`console.log("Difference:", z, "-", y, "=", z-y);`	
`console.log("Product:", y, "*", z, "=", y*z);`	
`console.log("Quotient 1:", y, "/", z, "=", y/z);`	
`console.log("Quotient 2:", z, "/", x, "=", z/x);`	
`console.log("Remainder 1:", z, "%", x, "=", z%x);`	
`console.log("Remainder 2:", x, "%", z, "=", x%z);`	
`console.log("Divide into zero:", 0, "/", x, "=", 0/x);`	
`console.log("Divide by zero:", x, "/", 0, "=", x/0);`	
`console.log("Zero into zero:", 0, "/", 0, "=", 0/0);`	
`console.log("Remainder and zero 1:", 0, "%", x, "=", 0%x);`	
`console.log("Remainder and zero 2:", x, "%", 0, "=", x%0);`	
`console.log("Remainder and zero 3:", 0, "%", 0, "=", 0%0);`	
`console.log("Power 1:", x, "**", y, "=", x**y);`	
`console.log("Power 2:", x, "**", -y, "=", x**-y);`	

Increment and Decrement Operators

JavaScript supports increment and decrement operators - these are $++$ and $--$ respectively. Both operators are *unary* meaning that they require just one operand. Unary increment adds one to its operand and unary decrement subtracts one. In both cases the operand must be a variable (i.e. it cannot be a literal value). This is because **the resultant value is stored in the variable**.

The program below demonstrates the use of the unary increment operator:

```
// Unary arithmetic - increment and decrement
let x = 7;
let y = x;
console.log("x before post-increment", x); // 7
console.log("Post-increment:", x++); // 7
console.log("x after post-increment", x); //8

console.log("y before pre-increment", y); // 7
console.log("Pre-increment:", ++y); // 8
console.log("y after pre-increment", y); // 8
```

```
x before post-increment 7
Post-increment: 7
x after post-increment 8
y before pre-increment 7
Pre-increment: 8
y after pre-increment 8
```

| *JavaScript Code* | *Output* |

KEY POINT: If the operator is used before the variable (e.g. `++a`) the new value is returned *after* the operation. This is called *prefix notation.*

If the operator is used after the variable (e.g. `b--`) the value of the variable is returned *before* the operation is carried out. This is called *postfix notation*.

Based on the above, what output do you think would be generated by the following code?

```
// Unary decrement
let a = 2;


console.log("a before post-decrement", a);
console.log("Post-decrement:", a--);
console.log("a after post-decrement", a);
console.log("Pre-decrement:", --a);
console.log("a after pre-decrement", a);
```

OUTPUT

Compound Assignment Operators

Arithmetic operators and assignments can be combined together using JavaScript's compound assignment operators. These operators are a shorthand syntactic construct that can be used when the value of a variable is being re-computed in terms of itself.

The most common compound assignment operators are +=, −=, *= and /=. A simple example of their use shown below.

`let score = 100;`	Initialise a variable called `score` to 100
`score += 50; // 150`	Add 50 to `score` – new value is 150
`score -= 10; // 140`	Subtract 10 from `score` – new value is 140
`score /= 2; // 70`	Divide `score` by 2 – new value is 70
`score *= 5; // 350`	Multiply `score` by 5 – new value is 350

KEY POINT: In a compound assignment an operation is carried out on a variable and the result is assigned to that same variable.

Compound assignment operators take two operands – the left operand must be the name of a variable and the right operand must be an expression. The operation is then applied to the two operands and the resulting value is stored in the left operand.

The table below summarises JavaScript's main compound assignment operators:

Operator	Name	Example	Same as
+=	Addition and Assignment	x += y	x = x + y
-=	Subtraction and Assignment	x -= y	x = x - y
*=	Multiplication and Assignment	x *= y	x = x * y
/=	Division and Assignment	x /= y	x = x / y
%=	Remainder and Assignment	x %= y	x = x % y
**=	Exponentiation and Assignment (not recommended)	x **= y	x = x ** y

Common Compound Assignment Operators

Type Conversions

We already know that the addition operator (+) returns the sum of two numeric values. This works fine when the datatype of both operands is `number` as show in the examples here.

`1 + 2` returns 3

`1 + −2` returns −1 and

`1.234 + 4.321` returns 5.555

But what happens when the datatype of either or both operands is not a `number`? Let's look at a few examples.

Expression	Result	Comment
`"Joe" + "Blogs"`	"JoeBlogs"	The addition of two strings results in a new string which is made up of the first string followed the second. This is known as *concatenation*.
`1 + "Bloggs"`	"1Blogs"	The addition of a numeric value and a string results in a new string. The numeric value is converted to a string and the two values are concatenated
`1 + "2"`	"12"	Again the first operand is converted to a string.
`1 + 2 + "Goals"`	"3Goals"	The operation is carried out from left-to-right.
`1 + (2 + "Goals")`	"12Goals"	The brackets take precedence

KEY POINT: In JavaScript the + operator will perform arithmetic addition only when *both* operands can be converted to a numeric value. The default behaviour of the + operator is string **concatenation**.

For all operations other than addition (i.e. −, ∗, / etc.) JavaScript will try to convert any string operands to numbers. (The reason for this is because strings do not support these other arithmetic operations.) This is demonstrated in the following expressions:

➢ `10 - "2"` evaluates to 8

➢ `"10" * "50"` evaluates to 500

➢ `"100" / 50` evaluates to 2

*The strings shown here are referred to as **numeric strings** (because they can be converted to numbers). The conversion is said to be **implicit** because it happens automatically.*

If neither operand is a numeric string JavaScript will return the value `NaN` (Not a Number). The expressions below all evaluate to `NaN`.

➢ `"fifty" - "ten"`
➢ `5 * "hotel"`
➢ `99 / "true"`

`NaN` is a unique 'quirk' of JavaScript. Not only is it a primitive type but it is also a value. Furthermore, it is a value which is not equal to itself (or any other value)! Consequently, the following expressions all return `false`

➢ `"fifty" == NaN // false`
➢ `"5" == NaN // false`
➢ `5 == NaN // false`
➢ `NaN == NaN // false`

The implication is that when we want to find out whether an expression is a number or not we cannot compare it to `NaN`. But fear not! In order to determine whether a value is a number or not JavaScript provides a global function called `isNaN`. This function returns `true` if the argument passed to it is not a number (i.e. cannot be converted to a number); `false` otherwise.

An alternative way to determine whether a value is a number is to use the `isFinite` global function.

Example uses of `isNaN` and `isFinite` are shown side by side in the table below. The difference is subtle but important.

➢ `isNaN("fifty") // true`	➢ `isFinite("fifty") // false`
➢ `isNaN("10") // false`	➢ `isFinite("10") // true`
➢ `isNaN(99) // false`	➢ `isFinite(99) // true`
➢ `isNaN(NaN) // true`	➢ `isFinite(NaN) // false`
isNaN	***isFinite***

Arithmetic and Booleans

When Boolean values are used in arithmetic expression JavaScript converts `true` and `false` to numeric values 1 and 0 respectively. The excerpt below demonstrates how + behaves when either or both of the operands are Boolean. (The output of each line is displayed in the comment.)

```
console.log(true + 1); // 2
console.log(false + 1); // 1
console.log(true + false); // 1
console.log(true + "false"); // true/false
```

Subtraction, multiplication and division all behave in a similar manner.

```
console.log(true - false); // 1
console.log(99 * true); // 99
console.log(true / 1); // 1
console.log(false / 1); // 0
console.log(true / false); // Infinity
```

Note in the last example - division by `false` is the same as division by zero. In JavaScript division by zero always results in the value `Infinity`.

The `Math` global object[5]

Math is a global JavaScript object. This means that it is automatically available to all JavaScript programs.

Some of the more common operations supported by `Math` are described in the table below.

Method	Description	Examples	Result
`Math.round(x);`	Returns `x` rounded up or down to the nearest integer	`Math.round(9.7);`	10
		`Math.round(9.3);`	9
`Math.ceil(x);`	Returns the nearest integer greater than or equal to `x`	`Math.ceil(9.7);`	10
		`Math.ceil(9.3);`	10
`Math.floor(x);`	Returns the nearest integer less than or equal to `x`	`Math.floor(9.7);`	9
		`Math.floor(9.3);`	9
`Math.pow(x, y);`	Returns `x` raised to the power of `y`.	`Math.pow(2,5);`	32
		`Math.pow(5,2);`	25
`Math.sqrt(x);`	Returns the positive square root of `x`	`Math.sqrt(25);`	5
		`Math.sqrt(-25);`	NaN
`Math.cbrt(x);`	Returns the cube root of `x`	`Math.cbrt(64);`	4
		`Math.cbrt(-64);`	-4
`Math.abs(x);`	Returns the absolute value of `x`	`Math.abs(25);`	25
		`Math.abs(-25);`	25
`Math.max(x, …)`	Returns the maximum of a list of 1 or more numbers	`Math.max(1,-2,-1);`	1
`Math.min(x, …)`	Returns the minimum of a list of 1 or more numbers	`Math.min(1,-2,-1);`	-2

In general, a `Math` method will return `NaN` if the argument(s) provided are invalid.

The `Math` object also supports a number of constants. The most notable of these is `Math.PI` which is defined as 3.141592653589793.

---

[5] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math for more details

Random Numbers

The `Math` library also contain a function called `random` which returns a pseudo-random floating-point number between 0 (inclusive) and 1 (exclusive).

> **KEY POINT:** to generate a random positive integer between `min` and `max` inclusive use the following line of JavaScript code.
>
> `Math.floor(Math.random() * (max - min +1)) + min;`

Some example uses of `Math.random` are given in the table below:

Example	Description
`Math.floor(Math.random() * 10);`	Returns an integer r such that: $0 \leq r < 10$
`Math.floor(Math.random() * 11);`	Returns an integer r such that: $0 \leq r \leq 10$
`Math.floor(Math.random() * 10) + 1;`	Returns an integer r such that: $1 \leq r \leq 10$

Write a JavaScript statement to generate a random number, `r` between two integers `x` and `y` such that:

(i) $x < r < y$

_____

(ii) $x \leq r \leq y$

_____

Outline three possible scenarios where the use of random numbers could be useful.

1.

_____

2.

_____

3.

_____

Operator Precedence

Operator precedence refers to the order in which operators are applied when an expression is being evaluated.

Expressions with only one type of operation are usually evaluated from left to right. However, when an expression contains more than one kind of operation JavaScript uses its precedence rules to determine what order to evaluate the expression. These precedence rules describe the relative importance of the JavaScript operators in relation to one another.

The precedence of the more commonly used JavaScript operators is illustrated below[6].

Operator	Name
++, --	Postfix Increment/Decremenet
++, --, !, +, -	**Prefix Increment/Decremenet, Logical NOT, Unary Plus, Unary Negation**
**	**Exponentiation**
*, /, %	Multiplication, Division, Remainder
+, -	Addition, Subtraction
<, <=, >, >=	Less Than, Less Than Or Equal To, Greater Than, Greater Than Or Equal To
==, !=	Equality, Inequality,
===, !==	Strict Equality, Strict Inequality
&&	Logical AND
||	Logical OR
=, *=, /=, %=, +=, -=	**Aassignments (simple and compound)**
,	Comma

Precedence of Common JavaScript Operators

Notes:

1) Operators with higher precedence appear higher up in the table than operators with lower precedence. This is why, in the absence of brackets, Multiplication and Division are always carried out before Addition and Subtraction. Therefore, $2 + 3 \times 4 \rightarrow 14$ (and not 20) and $10 - 8 \div 2 + 3 \rightarrow 9$ (and not 4).

2) Operators that appear on the same row have the same level of precedence. These operators are usually evaluated from left-to-right. For example, Multiplication and Division have the same level of precedence. Therefore, $8 \times 4 \div 2 \rightarrow 16$ whereas, $8 \div 4 \times 2 \rightarrow 4$. Similarly, $8 + 4 - 2 \rightarrow 10$ and $8 - 4 + 2 \rightarrow 6$

---

[6] For a more complete reference on operator precedence browse to:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

3) The operators on the lines highlighted in bold are evaluated from right-to-left. For example, $2 ** 3 ** 2 \rightarrow 512$ (i.e. 2^9 and not 8^2 as might be expected). Notice also that *assignments are evaluated from right-to-left.* This explains why the expression on the right hand side of an assignment operator is evaluated first (i.e. before the result is assigned).

4) As a final note it is worth pointing out that although grouping (i.e. brackets), the `new` operator, the dot operator (also known as the member operator because it is used to access object properties and invoke their methods) and function calls – none of which are shown in the precedence table on the previous page - all share the same level of precedence which is higher than that of all the operators shown.

5) Brackets have the highest level of precedence when it comes to evaluating JavaScript expressions. So, any expression inside brackets will always be evaluated first. For example, $(2 + 3) \times 4 \rightarrow 20$

Use the space below to record any points of note in relation to operator precedence

Programming Exercises (Arithmetic Operators)
Try the following.

1. Given that `x==2`, and `y==6` what do the JavaScript expressions below evaluate to.

a) x+y

b) x-y

c) 5*y

d) 5*x+y

e) 3*(x+y)

f) x*y+4

g) y/x+1

h) y/x+y

i) (x*y)+(y/x)

j) y%x

2. Write JavaScript expressions to evaluate each of the following:

a) 74 multiplied by 64

b) 81 divided by 10

c) 81 divided into 100

d) 25π

e) $(7-1) \div (4-2)$

f) $\frac{10 \times 50}{5}$

g) $10 \times \frac{50}{5}$

h) 7^2

i) $\sqrt{7}$

j) $2\sqrt{7}$

In all cases once you have your expression written you should use it in an assignment statement to store your answer in an appropriately named variable and then display the value of that variable in a meaningful message on the console. So for example the answer for part a) above, 74 multiplied by 64 would be:

```
let ans = 74 * 64;
console.log("74+64=", ans);
```

3. Write a line of code that adds the numbers 62 and 47 and displays the result.

4. Write code to do the following:
   a) add the numbers 62 and 47 and store the answer in a variable
   b) display the contents of the variable

5. The arithmetic mean of two numbers is calculated by adding them and dividing the result by 2. Write the code to calculate and display the arithmetic mean of 62 and 47.

6. Given the following code fragment that initialises three variables - `x`, `y` and `z` - write a line of code that computes and displays their arithmetic mean.

```
let x=27;
let y=15;
let z=18;
```

7. Predict the output following code:

```
let x=3*4;
let y=10/2;
let z=6-1;
let sum=x+y+z;
console.log(x, y, z, sum);
```

8. What (if anything) is wrong with the following? (You can assume `x`, `y` and `z` have already been initialised.)

```
a) let a=x+y;          k) let a=8*(-2);
b) let 3=x+y;          l) let b=(8)(2);
c) let 10/2=5;         m) let c=8*(+2);
d) let sum=x+y+z;      n) let c=4*a;
e) let sum=xy+z;       o) let c=4a;
```

9. The area of a rectangle can be computed by multiplying its width by its height. Given the following two lines of code to initialise the variables `width` and `height`, write a third line that computes and displays the rectangle's area.

```
let width=7;
let height=5;
```

10. The perimeter of a rectangle can be computed by adding twice the `width` to twice its `height`. Given the following two lines of code which initialise these two variables, write a third line that computes and displays the rectangle's perimeter.

```
let width=7;
let height=5;
```

11. Given the following formula to convert Fahrenheit (f) to Centigrade (c) write a program to convert 100°F into °C and display the result.

$$c = (f - 32) \times \frac{5}{9}$$

12. Now modify the program you just wrote to prompt the user to enter any Celsius value and display its Fahrenheit equivalent.

13. Read the following code carefully (the values are in cents) and answer the questions that follow:

```
let fifties=4;
let twenties=5;
let tens =7;
let total=fifties*.5 + twenties*.2 +tens*.1;
console.log(total);
```

   a) How many variables does the program use?
   b) What is the purpose of each variable?
   c) What does the code do?

14. Modify the above program to calculate the number of euro given that I have a five euro note, 22 fifty cent coins, 17 twenty cent coins, 25 ten cents and 13, two cent pieces.

15. Modify the above program (again) to prompt the user to enter the various values and display the total amount.

16. The length of the circumference of a circle is given by the formula $l = 2\pi r$. Write a program that prompts a user to input a value for the length of the radius (r), and then calculates and displays the length of the circumference.

17. Write a program to calculate the area of a circle. (Note: $A = \pi r^2$)

18. Two points in a plane are specified using the coordinates $(x1, y1)$ and $(x2, y2)$. Write a program that uses the formula below to calculate the slope of a line through two points entered by the user.

$$slope = \frac{y_2 - y_1}{x_2 - x_1}$$

19. Write a program that accepts two points and then determines and displays the distance between them

6. Boolean Operators and Expressions

Boolean expressions are expressions which can evaluate to either `true` of `false`. Simple Boolean expressions can be programmed using JavaScript's comparison operators and compound Boolean expressions can be programmed using JavaScript's logical operators. Both types of operators are now discussed.

Comparison Operators

As the name suggests comparison operators are used to compare two values. Values can be compared for equality or for difference. The result of a comparison will always be either `true` or `false`.

Comparison operators are important because they allow programmers to construct conditions. Conditions are the basis of control structures such as loops and selection statements which are used by programmers to control the runtime execution of their programs. They are the basic building blocks of programs.

The table below lists JavaScript's comparison operators[7].

Operator	Name	Description
==	Equality	Returns `true` if both operands have the same value (after conversion if necessary)
===	Strict equality	Returns `true` if both operands have the same value and type
!=	Inequality	Returns `true` if both operands have different values (after conversion if necessary)
!==	Strict inequality	Returns `true` if both operands have different values and/or types (with no conversion)
>	Greater than	Returns `true` if the left operand is > the right operand
>=	Greater than or equal to	Returns `true` if the left operand is >= the right operand
<	Less than	Returns `true` if the left operand is < the right operand
<=	Less than or equal to	Returns `true` if the left operand is <= the right operand

JavaScript comparison operators

When values being compared are of the same datatypes the behaviour of the comparison operators is relatively straightforward – numbers are compared on magnitude and strings are

---

[7] The full reference for comparison operators can be found by browsing to:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

compared alphabetically i.e. on a character by character basis using the Unicode value of each character. Some examples of comparisons involving the same types are now given:

```
// Comparisons – same datatypes
console.log(3 < 20); // true
console.log(7 != 7); // false
console.log(5 >= 7); // false
console.log("cat" == "dog"); // false
console.log("man" != "woman"); // true
console.log("man" < "woman"); // true
console.log("man" <= "men"); // true
```

When the datatypes are different JavaScript converts the values to numbers before comparing them. For example, if one operand was a string and the other was a number JavaScript will attempt to convert the string to a number before the comparison. If the conversion is successful, the two numbers are compared in the normal way; if the conversion is unsuccessful it will return NaN which always results in false when compared with anything. This is demonstrated in the following examples:

```
// Comparisons - strings and numbers
console.log(3 < "20"); // true ("20" converted to 20)
console.log("3" <= "20"); // false (comparing two strings: Unicodes are 3:51, 2:50)
console.log("5" == 5); // true ("5" converted to 5)
console.log("5" === 5); // false (because they are not strictly equal)
console.log("five" == 5); // false ("five" is NaN)
console.log("shark" > "5"); // true (comparing two strings: Unicodes are s:115, 5:53)
console.log("shark" > 5); // false ("shark" is NaN)
```

When Boolean values are being compared to numbers, true converts to 1 and false converts to 0. This is exemplified in the example code below.

```
// Comparisons - boolean and numbers
console.log(true == 1); // true
console.log(true === 1); // false (because they are not strictly equal)
console.log(99 != true); // true
console.log(false != 0); // false
```

Note: Two variables are said to be strictly equal (===) to each other if the values they contain are both equal *and* of the same datatype. If the values are the same but the datatypes are different then the variables are 'just' equal (==) to one another.

JS Test your understanding (comparison operators)
Try the following.

1. What do these expressions evaluate to? (`true` or `false`)

a) `7 == 7`

b) `7 != 7`

c) `7 >= 6`

d) `2 < 3`

e) `3 < 2`

2. The following conditions all evaluate to `false`. Change the operator so that they would evaluate to `true`. (You can have more than one answer for each part if you wish.)

a) `4 < 3`

b) `8 >= 9`

c) `5 == 4`

d) `true != true`

e) `99 == false`

f) `8 < "10"`

g) `"8" != 8`

h) `"8" == 8`

i) `"8" === 8`

3. Assuming $x = 1$, and $y = 0$ to what do the conditions below evaluate?

a) $x > 5$

b) $5 > x$

c) $y <= 0$

d) $0 == y$

e) $x == y$

f) $x != y$

g) $x > y$

h) $y > x$

i) $x <= y$

j) $x >= y$

Logical Operators

We know from the previous section that comparison operators can be used to construct simple conditions. In this section we look at how these simple conditions can be connected using JavaScript's logical operators – shown in the table below - to form new conditions called compound conditions.

Operator	Name	Syntax
!	Logical NOT	!expr
&&	Logical AND	expr1 && expr2
||	Logical OR	expr1 || expr2

JavaScript Logical Operators

KEY POINT: A compound condition is made up of two or more simple conditions. Conditions can be connected using JavaScript's logical operators, `!` (not), `&&` (and), and `||` (or).

As can be seen from the syntax column in the above table, the three logical operators – `!` (logical NOT), `&&` (logical AND), and `||` (logical OR) – all operate on operands which are Boolean expressions[8] (shown for example as `expr1` in the above table).

Logical NOT is a unary operator which means that it requires just one operand to work. Both logical AND, and logical OR are binary operators meaning that they require two input operands. For the sake of simplicity, it is safe to think of the value returned by the logical operators as being either `true` or `false`.[9]

The rules for how these simple Boolean expressions are combined are expressed in **truth tables**. The truth tables for logical NOT, logical AND, and logical OR are now presented.

Each truth table lists all the possible inputs in the left column(s) and the corresponding outputs in the rightmost column.

---

[8] The operands can, in fact, be of any datatype - not just Boolean. This is because non Boolean operands are converted to truthy or falsey values as part of their evaluation.
[9] Non-Boolean return values can always be converted to Boolean primitives.

NOT

The `!` (logical NOT) operator is used to invert a single Boolean value. The truth table for NOT is shown below. `A` is the input and `!A` is the output. The first row in the table shows that whenever `A` is `true`, `!A` is `false`.

A	!A
false	true
true	false

Truth table for logical NOT

NOT
For an input of `true` the output is `false` and for an input of `false` the output is `true`.

AND

The truth table for `&&` (logical AND) is shown below. The first two columns show the inputs A and B; the third column is the output A && B. The first row in the truth table shows that whenever both inputs A and B are false then the output A && B is also false. The second row shows that if A is false and B is true then A && B is false and so on.

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

Truth table for logical AND

AND
In order for the output to be `true` both inputs must be `true`

OR

The truth table for `||` (logical OR) is shown below. The first two columns show the inputs A and B; the third column is the output A || B. The first row in the truth table shows that whenever both inputs A and B are false then the output A || B is also false. The second row shows that if A is false and B is true then A || B is true and so on.

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

Truth table for logical OR

OR
In order for the output to be `true` either input must be `true`

Examples

1. The table below shows how logical operators are applied to the simple Boolean expressions shown in the leftmost column called (i.e. Condition)

Condition	Result	Explanation
!(2 == 5)	true	2==5 is false not false is true
!(2 < 5)	false	2<5 is true not true is false
(5 > 2) && (5 > 4)	true	5>2 is true 5>4 is true true and true is true
(5 > 2) && (5 > 7)	false	5>2 is true 5>7 is false true and false is false
(3 > 2) || (3 > 4)	true	3>2 is true 3>4 is false true or false is true

2. For the purpose of this example assume the variable valid is true and the variable finished is false.

Condition	Result	Explanation
!valid	false	not true is false
!finished	true	not false is true
finished && valid	false	false and true is false
finished || valid	true	false or true is true
finished || !valid	false	false or not true false or false is false

KEY POINT: A truth table is a convenient way of showing the rules for combining Boolean expressions

3. Let's say we had a variable called `age` and we wanted to program a computer to determine whether a person was a teenager or not. We know a teenager is a person whose age is between 13 and 19 inclusive so we will need to tell the computer to compare `age` to these two values. To do this we ask three questions:

   o is the value of `age` greater than or equal to 13?
   o is `age` less than or equal to 19?
   o is `age` greater than or equal to 13 *AND* is `age` less than or equal to 19?

The first two questions can be expressed in JavaScript using comparison operators as follows

- `age >= 13`
- `age <= 19`

We form our compound condition by using the logical and operator, **&&** to combine the two simple conditions as follows:

- `((age >= 13) && (age <= 19))`

PROGRAMMER TIP!
The formation of Boolean expressions using the comparison and logical operators is a key programming skill that needs to be practiced and can be honed over time.

In order to evaluate the above compound expression, the following four possibilities need to be considered:

1) `age >= 13` is `false` and `age <= 19` is `false`. There is no possible age that could be both less than 13 and greater than 19. So, the only logical conclusion is that the person is not a teenager.

2) `age >= 13` is `false` but `age <= 19` is `true`. (For example, the age could be 12.) In this case the overall expression evaluates to `false` and the computer can conclude that the person is not a teenager.

3) `age >= 13` is `true` but `age <= 19` is `false`. (For example, the `age` could be 21.) In this case the overall expression evaluates to `false` and the computer can conclude that the person is not a teenager.

4) `age >= 13` is `true` and `age <= 19` is `true`. (For example, the `age` could be 16.) In this case the overall expression evaluates to `true` and the computer can conclude that the person is a teenager.

The four possibilities are summarised in the table below. The first two columns contain the *inputs,* the *output* is displayed in the third column. As can be seen there is only one situation that yields an output of `true` (highlighted). This occurs when the age is between 13 and 19 inclusive. In all other cases the outputs are `false`.

age >=13	age <=19	age >=13 && age <=19
false	false	false
false	true	false
true	false	false
true	true	true

The output column on the right indicates whether a person is a teenager or not

TEACHER TIP!
It can be useful to make an analogy between truth tables and the arithmetic tables we learned at primary school. Take the '5 times tables' as an example. The table lists all outputs generated by multiplying 5 by every integer from 0 to 12 (i.e. the inputs). The operator used here is of course is multiplication (×).

In Boolean algebra the operators can be logical NOT, logical AND and logical OR. Their respective truth tables list the outputs generated by applying these operators. The only permissible values in Boolean algebra are `true` and `false`. Hence all inputs are `true` or `false` and all outputs are `true` or `false`.

KEY POINT: Truth tables can be used as an aid to evaluating Boolean expressions. By listing all the inputs on the left hand side we can then use the standard tables for NOT, AND, and OR to look up the results and record them in the output column.

4. Construct the truth table for `A AND NOT B` i.e. `A && !B`

Solution

In the truth tables shown below 1 is used to represent a value of `true` and 0 a value of `false`

STEP 1: We start off by constructing a truth table with all the possible combinations of inputs.

A	*B*
0	0
0	1
1	0
1	1

STEP 2: We add the first output column for `NOT B`

A	*B*	!*B*
0	0	1
0	1	0
1	0	1
1	1	0

STEP 3: The `NOT B` column just created is *ANDed* with *A* to give our desired output column `A AND NOT B`

A	*B*	!*B*	*A*&&!*B*
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

5. Construct the truth table for `NOT A OR B` i.e., `!A || B`

Solution

As was the case in the previous Example 1 is used to represent a value of `true` and 0 a value of `false` in the truth tables shown below.

STEP 1: We start off by constructing a truth table with all the possible combinations of inputs.

A	*B*
0	0
0	1
1	0
1	1

STEP 2: We now add the first output column `NOT A`

A	*B*	*!A*
0	0	1
0	1	1
1	0	0
1	1	0

STEP 3: The `NOT A` column just created is *ORed* with *B* to give our desired output column `NOT A OR B`

A	*B*	*!A*	*!A‖B*
0	0	1	1
0	1	1	1
1	0	0	0
1	1	0	1

Boolean Logic

Boolean logic is a branch of Mathematics in which there are only two values – `true` and `false`. It was invented by a mathematician named George Boole 1815-1864 who was the first professor of Mathematics at University College Cork (UCC). In recent times Boole has become known all over the world as *the forefather of the information age* - it is no exaggeration to state the Boolean Logic forms the basis for all digital electronic technology and all software systems.

Boolean logic is useful because it provides a rigour for dealing with statements known as propositions. Propositions are assertions that are either `true` or `false`. The following are all examples of propositions:

- Three is an odd number
- $1 + 1 = 3$
- The time is 14:00 hours
- Cork is the largest county in Ireland
- The snail is moving at a speed of less than 1 metre per hour
- The car is travelling at a speed greater than 120km per hour
- The plane is at an altitude of between 31,000 and 38,000 feet
- The object is on the screen (e.g. game character, shape, animation, sprite etc.)

Propositions such as those listed above can be evaluated by humans without too much difficulty. If we have the information that is needed, we can decide in an instant whether the proposition is `true` or `false`. Although we may not be aware of it at the time we usually evaluate propositions by comparing two values. Consider the assertion that the time is 14:00 hours – we compare the current time with $14{:}00$ and if they are the same the proposition is `true`; otherwise it is `false`.

The power of Boolean logic is that it provides a framework for taking propositions such as those listed above and writing them as Boolean expressions which can be included in our programs (and then evaluated by the hardware at runtime). Boolean expressions are the principle means by which programmers can infuse logic into their code and for this reason they are a critical part of all computer programs. The ability to formulate Boolean expressions is a vital skill which is part and parcel of the art of computer programming.

JS *Test your understanding of logical operators*
Try the following.

1. Evaluate each of the conditions shown in the table below given the variables:

   $x = 1$, $y = 0$ and $z = -1$. (Results are `true` or `false`.)

Condition	Result
`!(x==y)`	
`!(x==y+z)`	
`!(y==x+z)`	
`(x>y) && (y>z)`	
`(z<x) && (y<x)`	
`(x<y) || (y>z)`	
`(y!=x+z) || (y>z)`	

2. Evaluate each of the conditions shown in the table below given the variables:

   `valid` is `false` and `finished` is `true` (Results are `true` or `false`.)

Condition	Result
`!finished`	
`!valid`	
`!finished && !valid`	
`!finished || valid`	
`valid || finished`	

3. Evaluate the following expressions:
   a) true or not true

   b) false and (true or not true)

   c) (true or false) and true

   d) false or true or not false

   e) not false and not true or not false

4. Complete the truth table below to find `NOT A AND B`

A	B	!A	!A && B
0	0		
0	1		
1	0		
1	1		

5. Complete the truth table below to find `A OR NOT B`

A	B	!B	A || !B
0	0		
0	1		
1	0		
1	1		

6. Complete the truth table below to find `NOT (A AND B)`

A	B	A && B	!(A && B)
0	0		
0	1		
1	0		
1	1		

7. Complete the truth table below to find `NOT (A OR B)`

A	B	A || B	!(A || B)
0	0		
0	1		
1	0		
1	1		

.

7. Selection Statements (conditionals)

Selection statements – also known as decision statements and conditionals - are written by programmers to build alternative execution pathways into their programs.

The idea of selection is depicted in the illustration below.
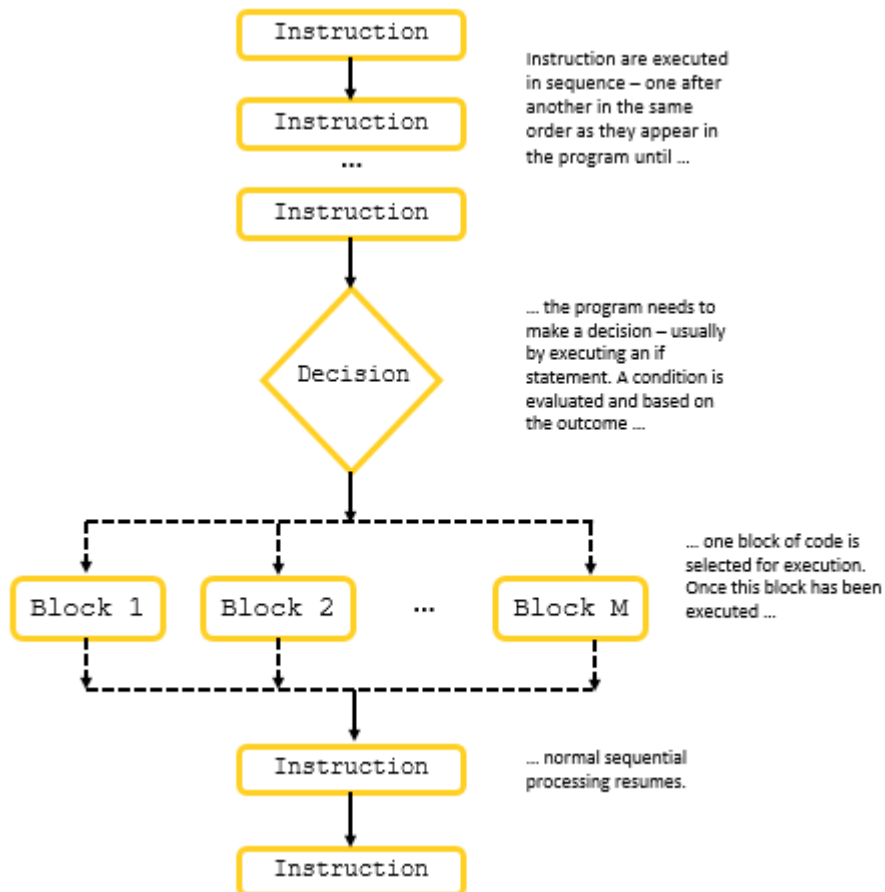
Selection statements are a branching mechanism whereby, based to the outcome of a decision, a specific block of code is selected for execution and other blocks of code are skipped. The decision is typically programmed as a Boolean expression.

JavaScript, like most programming languages, support selection by including `if` statements as part of its syntax. In this section we will explore the syntax and semantics of the JavaScript `if` statement and its variants (i.e. `else`, and `else-if`) along with the `switch` statement. We will also that a quick look at JavaScript's only *ternary operator* and how it can be used to perform selection.

The `if` statement

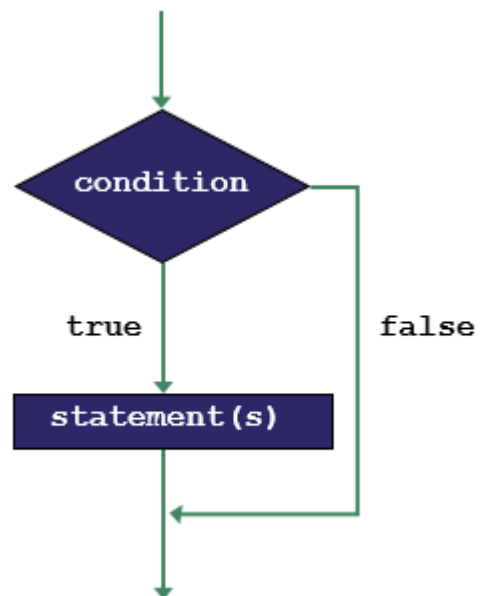The syntax of the JavaScript `if` statement is as follows:

```
if (condition) {
    statement(s)
} // end if
// program continues from here
```


The flow diagram to the right illustrates the runtime execution of an `if` statement.

The first thing that happens when an `if` statement is executed is the condition is evaluated and the result is converted to `true` or `false`.

If the condition evaluates to `true` then the statement(s) inside the `if` statement will be executed. The statement(s) inside the `if` statement are called the *body* of the `if` statement.

If the condition evaluates to `false`, the body of the `if` statement will be skipped and execution will continue from the next line after the `if` statement.

The flow diagram shown above depicts two different execution paths – in one path the statements(s) in the body of the `if` statement are executed, and in the other they are bypassed.

The condition is by far the most important part of the if statement. Recall from earlier that the values `null`, `undefined`, `NaN`, empty string ("") and the number zero all evaluate to `false`. All other values evaluate to `true`.

KEY POINT: The statement(s) that make(s) up the body of the `if` statement are only executed if the condition evaluates to `true`.

Let's look at some examples. (Try them out for yourself!)

Example 1

```
let age = prompt("Enter your age");

if (age >= 18) {
    console.log("You are an adult");
}

console.log("Thank you. Goodbye.");
```

The condition here is `age >= 18`. If the user enters 18 or any value greater than 18 the condition will evaluate to `true` and the message *You are an adult* will be displayed. If the user enters any value that is less than 18 the condition will evaluate to false and the conditional code will not be executed.

KEY POINT: Programs can behave differently each time they are run. This runtime behaviour depends on the data provided to the program and the conditions programmed by the programmer.

The last line of code in the example is outside the `if` statement and will therefore always be executed regardless of the outcome of the condition. The program always displays the message *Thank you. Goodbye.*

So, in summary, depending on the input the possible outputs are:

You are an adult
Thank you. Goodbye.   *or just*   Thank you. Goodbye.

Note that the two curly braces - opening, `{` and closing, `}` - are only needed when the condition's body contains more than one line of code. So technically, they are not needed in this example because there is only one line of code in the body of the if-statement.

TEACHER TIP!
It is a common student misconception to think that an `if` statement triggers whenever its condition becomes `true`. Students should therefore be encouraged to realise that `if` statements are executed as part of the normal flow of control. One way of doing this is to get them to run example programs multiple times. The input data should be deliberately chosen so as to trigger all possible execution paths.

Example 2

In this example the user is prompted to enter the current year. If the value entered is the same as the year set on the computer, the program will display the messages:

You are correct
Well done!

```javascript
let date = new Date();
let computerYear = date.getFullYear();
let userYear = prompt("Enter the current year");

if (userYear == computerYear) {
    console.log("You are correct");
    console.log("Well done!");
}

console.log("The current year is", computerYear);
```

Again, the last line is executed unconditionally. Try this example with a year other than the current year (e.g. enter 2020 and see what happens).

Note that the curly braces are needed in this example because that condition's body has more than one statement.

PROGRAMMER TIP!
It is considered good practice to use the curly braces – even if there is only one statement pertaining to the conditional expression.

Example 3

In this example, the condition `hourlyPay < minimumWage` always evaluates to `true` and therefore the program will always generate the same output.

```javascript
let hourlyPay = 5;
const minimumWage = 10;

if ( hourlyPay < minimumWage ){
    console.log("The hourly rate of pay is below the minimum wage.");
}

console.log("Have a nice day!");
```

Experiment!
What output would the above program generate if `hourlyPay` was set to 15? What about 10? Modify the program so that the user is prompted to enter a value for the hourly pay rate.

JS Programming Exercises
Try the following.

1. Study the code below carefully and predict its output in the space provided.

```
let x = 3;
let y = 2;

if (x == y) {
  console.log(x, "is equal to", y);
}

if (x != y) {
  console.log(x, "is not equal to", y);
}

if (x >= y) {
  console.log(x, "is greater than or equal to", y);
}

if (x > y) {
  console.log(x, "is greater than", y);
}

if (x <= y) {
  console.log(x, "is less than or equal to", y);
}

if (x < y) {
  console.log(x, "is less than", y);
}
```

What output would be generated if the initial values of x and y were set using the values shown below:

a) x=2 and y=2

b) x=2 and y=3

c) x="Jim" and y="Jam"

2. Write a program to accept a single number and display the word NEGATIVE if the number is less than zero

3. Write a program to accept a single number and display the word BOILING if the number is 100 or greater.

4. What's wrong with the following? Suggest *two* solutions to the problem.

```
// Prompt user to enter a value 0 - 100
let result = prompt("Enter student result");
if (mark == 100) {
  console.log("Full marks - well done");
  console.log("Perfect score!");
}
console.log("The result was", result);
```

The `if-else` statement

An `else` clause is always used in conjunction with an `if` statement. It is used to provide an alternative execution path in situations when the `if` condition evaluates to `false`.

The syntax of the JavaScript `if-else` statement is as follows:

```
if (condition) {
    statement(s)
} // end if
else {
    statement(s)
} // end else
// program continues from here
```
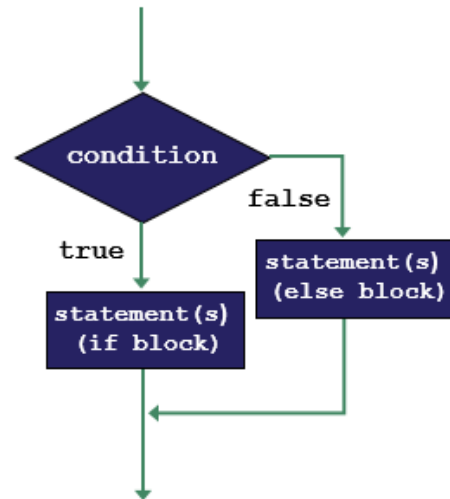

The flow diagram illustrates how the `if-else` statement works.

The condition is evaluated and depending on the outcome either the statements inside the `if` block or the statements inside the `else` block are executed.

➢ If the condition evaluates to `true` the statements inside the `if` block are executed.

➢ If the condition evaluates to `false` the statements inside the `else` block are executed

KEY POINT: Only one set of statements will always be executed either the `if` block or the `else` block. Never both!

Once the selected block of code has been run, execution continues from the next line following the `if-else` statement.

TEACHER TIP

A common student misconception is that both branches of an `if-else` statement are *always* executed. Teachers should encourage students to use test data that will activate both branches in separate runs of the program.

Let's look at some examples.

Example 1

```
// if-else statement
let num1 = prompt("Enter a number");
let num2 = prompt("Enter another number");

if ( Number(num1) >= Number(num2) ){
   console.log(num1, "is greater than or equal to", num2);
}
else {
   console.log(num1, "is less than", num2);
}

console.log("Program execution continues from here");
```

JavaScript evaluates the condition, `Number(num1) >= Number(num2)`, and, depending on the outcome either the `if` block or the `else` block (never both) is executed.

For an input of 5 and 7 the output generated would be:

```
5 is less than 7
Program execution continues from here
```

For an input of 12 and 7 the output generated would be:

```
12 is greater than or equal to 7
Program execution continues from here
```

The last line is not part of the `if-else` statement and so it is always executed.

Experiment!
What output would the above program generate if the two numbers entered were the same? Does the program work with negative numbers? What would happen if the condition was just `num1 >= num2`**? Make the change and try 12 and 7 as inputs. Can you explain the output? Modify the program so that it uses < in the condition (instead of >=)**

Example 2

This example compares the date entered by a user to the computer's internal date.

```
let date = new Date();
let computerYear = date.getFullYear();
let userYear = prompt("Enter the current year");

if (userYear != computerYear) {
   console.log("Incorrect answer");
   console.log("The current year is", computerYear);
} else {
   console.log("You are correct");
   console.log("Well done!");
}
```

Use the space below to list any differences in logic between this example and Example 2 in the previous section

Example 3

The condition used in this example - num1 % 2 == 0 – demonstrates how a program can determine whether a number is even or odd.

```
let num1 = prompt("Enter a number");

if ( num1 % 2 == 0 ) {
   console.log(num1, "is even");
}
else {
   console.log(num1, "is odd");
}
```

The program performs a '*divisibility by 2 test*' and displays the result.

What output would this example program display if the numbers entered were 10 and 5?

a) 10 _____

b) 5 _____

Use the program to see if it works for negative integers. What about zero?

Challenge!
Modify this program so that it reads a second number – num2 – and then displays whether or not num1 is evenly divisible by num2.

1. Study the two programs below carefully and answer the questions that follow. (You can assume that 50 is a pass mark.)

```javascript
let mark = prompt("Enter student mark");
if (mark >= 50) {
  console.log("Well done - you passed!");
  console.log("Some options now are ....");
  console.log("1. Get a job");
  console.log("2. Do an apprenticeship");
  console.log("3. Go to college");
  console.log("4. Travel the world!");
} else {
  console.log("Hard luck - unsuccessful!");
  console.log("Some options now are ....");
  console.log("1. Get a job");
  console.log("2. Repeat");
  console.log("3. Social Welfare");
  console.log("4. Travel the world!");
}
```
Program 1

```javascript
let mark = prompt("Enter student mark");
if (mark <= 50) {
  console.log("Hard luck - unsuccessful!");
  console.log("Some options now are ....");
  console.log("1. Get a job");
  console.log("2. Repeat");
  console.log("3. Social Welfare");
} else {
  console.log("Well done - you passed!");
  console.log("Some options now are ....");
  console.log("1. Get a job");
  console.log("2. Do an apprenticeship");
  console.log("3. Go to college");
}
console.log("4. Travel the world!");
```
Program 2

Are the programs logically equivalent? Explain any difference.

Which program do you think is better? Why?

2. Write a program that prompts the user to enter a year and display the word PAST if it is before the current year and FUTURE if the year is greater than the current year.

3. Write a program that prompts the user to enter a single number and display the word POSITIVE if the number is greater than zero and NEGATIVE if the number is less than zero. What happens when you enter zero itself?

4. Study the program below carefully and answer the questions that follow.

```
// Generate two random numbers between 1 and 10 incl.
let n1 = Math.floor(Math.random() * 10) + 1;
let n2 = Math.floor(Math.random() * 10) + 1;

let message = "What is "+n1+" + "+n2;
let userAnswer = prompt(message);

if ( userAnswer == n1+n2 ) {
  console.log("Correct - well done!");
} else {
  console.log("Sorry - incorrect answer!");
  console.log("The right answer is", n1+n2);
}
```

a) Describe what the program does.

b) What change would need to be made so that following line gets executed regardless of what the user enters?
```
console.log("The right answer is", n1+n2);
```

c) The arithmetic expression `n1+n2` appears twice in the above program. Are both instances of this same expression *always* executed at runtime? Explain.

d) Without changing the logic of the program, suggest a change so that the expression `n1+n2` only occurs once in the program.

e) Let us say the condition was changed to `userAnswer != n1+n2`. What other changes would need to be made so that the logic of the program remained unaltered.

5. Write a program that generates two random numbers and then asks the user to enter their product. If the user is right the program should display *Correct*. Otherwise, the program should display *Incorrect*.

The else-if statement

The else-if statement provide the necessary logic to cater for situations where there are more than two alternative possibilities.

The syntax of the *if...else-if* statement is as follows.

```
if (condition 1) {
    statements(s)
} else if (condition 2) {
    statements(s)
...
} else if (condition N) {
    statements(s)
} else {
    statements(s)
}
```

Starts with an `if`

Each `else-if` section must have a condition

One `else-if` section for each alternative

The `else` at the end is optional – it doesn't have a condition

The logic flowchart is depicted below.
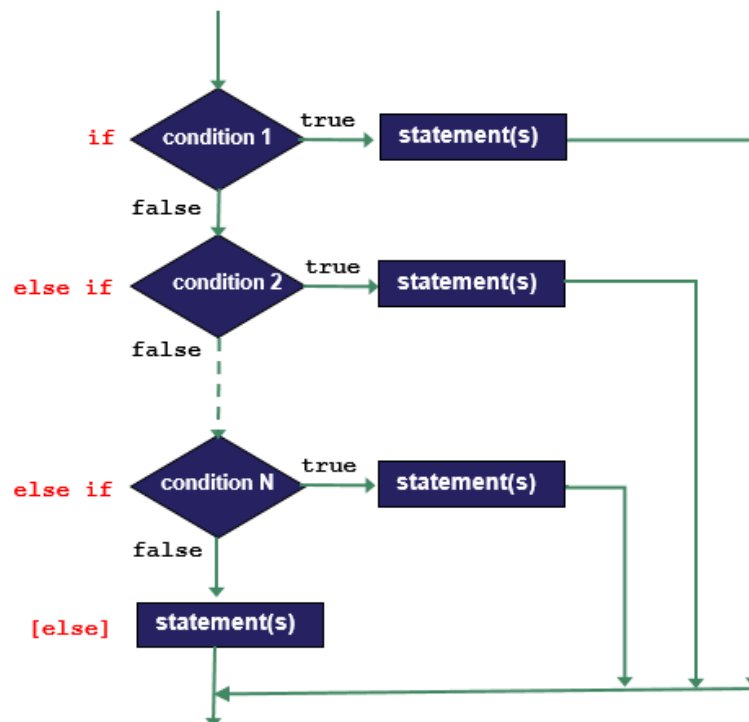
The `else-if` statement works by testing each condition in sequence until it finds one that evaluates to `true` whereupon the associated statement(s) are executed and the statement ends. If all the conditions evaluate to `false` then the optional `else` statements at the end are executed (if they exist).

Let's take a look at some examples.

Example 1

This program prompts the user to enter an integer and, depending on its sign, displays one of the following three messages:

The number is positive
The number is negative
Neither positive nor negative so the number must be zero!

```javascript
let number = prompt("Enter any integer:");

if (number > 0) {
  console.log("The number is positive");
} else if (number < 0) {
  console.log("The number is negative");
} else {
  console.log("Neither positive nor negative so the number must be zero!");
}
```

The third possibility is catered for by the `else` statement at the end. The three code listings shown below are all logically equivalent even though the code is arranged in slightly different orders. Try them for yourself! (Just change the example above to look like each of these.)

```javascript
let number = prompt("Enter any integer:");

if (number == 0) {
  console.log("Neither positive nor negative so the number must be zero!");
} else if (number < 0) {
  console.log("The number is negative");
} else {
  console.log("The number is positive");
}
```

```javascript
let number = prompt("Enter any integer:");

if (number < 0) {
  console.log("The number is negative");
} else if (number == 0) {
  console.log("Neither positive nor negative so the number must be zero!");
} else {
  console.log("The number is positive");
}
```

```javascript
let number = prompt("Enter any integer:");

if (number > 0) {
  console.log("The number is positive");
} else if (number == 0) {
  console.log("Neither positive nor negative so the number must be zero!");
} else {
  console.log("The number is negative");
}
```

Example 2.

In this example the program compares two numbers entered by the user and then displays an appropriate message stating how the numbers relate to each other.

```javascript
let num1 = prompt("Enter a number");
let num2 = prompt("Enter another number");

if ( Number(num1) > Number(num2) ){
   console.log(num1, "is greater than", num2);
}
else if ( Number(num1) < Number(num2) ){
   console.log(num1, "is less than", num2);
}
else {
   console.log(num1, "is equal to", num2);
}

console.log("Program execution continues from here");
```

The table below illustrates various outputs for three different runs of the program.

Inputs		Outputs
num1	**num2**	
12	12	12 is equal to 12 Program execution continues from here
32	16	32 is greater than 16 Program execution continues from here
1000	1	1000 is greater than 1 Program execution continues from here

Each row of the table shows what the output would be for the given inputs. The inputs shown only trigger two of the three possible scenarios catered for in the code (the less than block is never executed). Only one of the three blocks of code is executed on each run.

Experiment!
What output would the program generate if the value entered for `num1` was 1 and `num2` was 1000. Does the program work for negative numbers? If the first condition was changed to `Number(num1) == Number(num2)` *what other changes would need to be made? What would happen if you removed the call to* `Number` *from the program (there are 4 occurrences)?*

KEY POINT: Every time a program is run it can behave differently depending on the data it is working with during that run.

Example 3.

The example below uses multiple `else-if` statements to display the capital city of a country name entered by the user.

```javascript
let country = prompt("Enter a country and I will tell you its capital");

if (country == "Ireland") {
   console.log("Dublin");
} else if (country == "Scotland") {
   console.log("Edinburgh");
} else if (country == "England") {
   console.log("London");
} else if (country == "Wales") {
   console.log("Cardiff");
} else if (country == "France") {
   console.log("Paris");
}
```

a) The code is programmed to work for Ireland, Scotland, England, Wales and France. What happens if you enter the name of some other country?

b) Modify the program so that it displays the name of the country's continent (i.e. Europe) on a separate line underneath the capital. Can this be done by adding one line?

c) Now extend the program so that it can deal with USA (Washington), Japan (Tokyo) and Australia (Canberra). (Don't forget to test for each new country.) Does the problem you identified in Part a) still exist? Does the program display the correct continent name?

d) Extend the program again – this time add the following code at the end. What difference does this code make? Why does the `else` not have a condition attached?

```javascript
else {
   console.log("Unknown country. Sorry.");
}
```

JS

Programming Exercises
Try the following.

1. Write a program that prompts a user to enter a day number and then displays the corresponding day name as per the following flowchart.
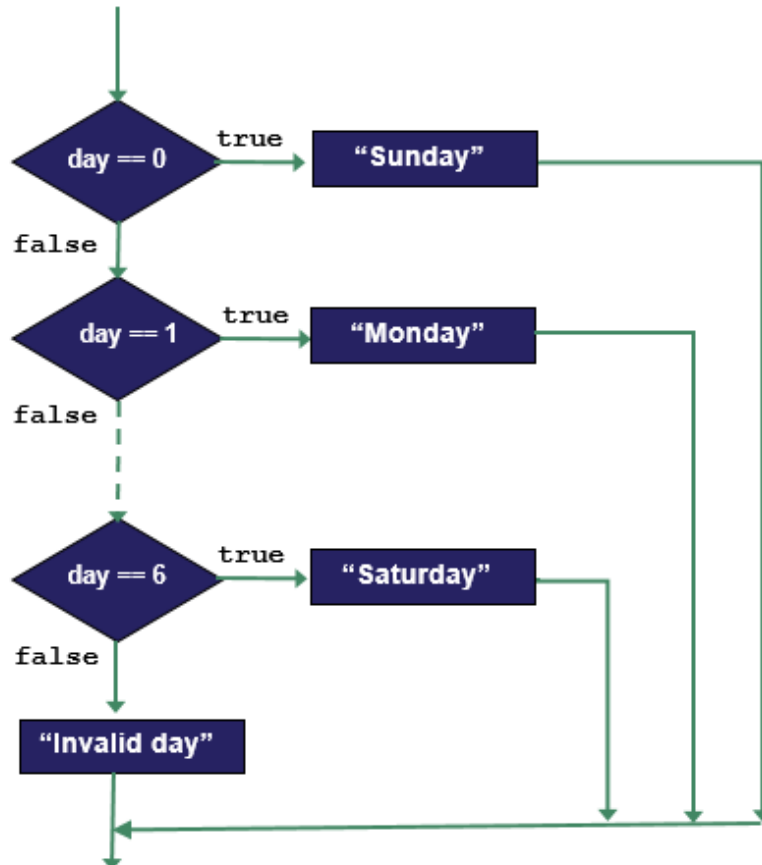
2. The following two lines of code can be used to retrieve the weekday number for the computer on which they are run (0 for Sunday, 1 for Monday and so on)

```
let date = new Date();
let dayNo = date.getDay();
```

Modify your answer to question 1 so that instead of prompting the user to enter a day number the program uses these two lines.

3. A certain Computer Science teacher gives five-point quizzes that are graded on the scale 5-A, 4-B, 3-C, 2-D, 1-F, 0-NG. Write a program that accepts a quiz score as an input and displays the corresponding grade as output.

4. The intention of the program below is to display a student grade based on a percentage mark entered by the user.

The table on the right illustrates how marks are mapped to grades.

Study the code carefully and answer the questions which follow. (This initial version just deals with higher level grades, H1-H7.)

Mark (%)	Grade	
90 – 100	H1	O1
80 – 89	H2	O2
70 – 79	H3	O3
60 – 69	H4	O4
50 – 59	H5	O5
40 – 49	H6	O6
30 – 39	H7	O7
0 – 29	H8	O8

```javascript
let mark = Number(prompt("Enter percentage mark (0-100):"));

if (mark >= 0) {
   console.log("H8");
} else if (mark >= 30) {
   console.log("H7");
} else if (mark >= 40) {
   console.log("H6");
} else if (mark >= 50) {
   console.log("H5");
} else if (mark >= 60) {
   console.log("H4");
} else if (mark >= 70) {
   console.log("H3");
} else if (mark >= 80) {
   console.log("H2");
} else if (mark >= 90) {
   console.log("H1");
}
```

a) Why will the program not do what it is intended to do?
b) Suggest and implement solution to the problem.
c) Modify the (fixed) code from the previous question so that it stores the grade in a variable (call it `grade`) and just one statement at the end to display its value.
d) Think about how the program could be modified so that it can cater for ordinary level grades as well as higher level grades.
   (One suggested solution to this problem is shown on the next page.)

One possible solution to the last problem on the previous page can be broken into three steps as follows:

1) prompt the user to enter either an 'H' or an 'O' to indicate higher or ordinary. This letter is stored in a variable called `gradeLetter`.

2) set a variable `gradeLevel` to some value ranging from 1 to 8 depending on the `mark` entered by the user

3) Display the grade as a combination of the `gradeLetter` and the `gradeLevel`.

The code is as follows.

```
let gradeLetter = prompt("Enter a letter: H = Higher, O = Ordinary");
let gradeLevel;
let mark = Number(prompt("Enter percentage mark (0-100):"));

if (mark >= 90) {
   gradeLevel = 1;
} else if (mark >= 80) {
   gradeLevel = 2;
} else if (mark >= 70) {
   gradeLevel = 3;
} else if (mark >= 60) {
   gradeLevel = 4;
} else if (mark >= 50) {
   gradeLevel = 5;
} else if (mark >= 40) {
   gradeLevel = 6;
} else if (mark >= 30) {
   gradeLevel = 7;
} else if (mark >= 0) {
   gradeLevel = 8;
}

console.log("Final Grade: ", gradeLetter + gradeLevel);
```

An alternative solution would be to use a nested `if` statement which is now explained.

Nested if-statements
A nested `if` statement as its name suggests is an `if` statement within an `if` -statement.

The syntax of a simple nested `if` is shown below. The first `if` statement is referred to as the *outer if* and the nested `if` is referred to as the *inner* `if`.

```
if (condition) {
   if (condition) {
      statement(s)
   } // end inner if
} // end outer if
```

Although there is no limit to the number of `if` statements that can be nested, more than three levels of nesting are rarely seen in practice. (After this point code can become difficult to follow and there is usually a clearer alternative.). Three levels of nesting would look like this.

```
if (condition1) {
   if (condition2) {
      if (condition3) {
         statement(s)
      } // end if 3
   } // end if 2
} // end if 1
```

The inner `if` condition will only be executed if the preceding `if` condition(s) evaluate(s) to `true`.

Nesting can also occur inside the `else` clause - the syntax is as follows:

```
if (condition) {
   statement(s)
} // end if
else {
   if (condition) {
      statement(s)
   } // end if
   else {
      statement(s)
   } // end else
} // end else
```

PROGRAMMER TIP!
Care should be taken when nesting `else` statements as improper use of braces can lead to *dangling else problems*. Every `else` is taken to correspond to its closest preceding `if`.

Solution to grade problem using nesting

The program below shows how nesting can be used to cater for both Ordinary and Higher
level grades in mapping a percentage mark to a final grade.

```
let gradeLetter = prompt("Enter a letter: H = Higher, O = Ordinary");
let mark = Number(prompt("Enter percentage mark (0-100):"));
let grade;

if (gradeLetter == "H") {
  if (mark >= 90) {
    grade = "H1";
  } else if (mark >= 80) {
    grade = "H2";
  } else if (mark >= 70) {
    grade = "H3";
  } else if (mark >= 60) {
    grade = "H4";
  } else if (mark >= 50) {
    grade = "H5";
  } else if (mark >= 40) {
    grade = "H6";
  } else if (mark >= 30) {
    grade = "H7";
  } else if (mark >= 0) {
    grade = "H8";
  }
} // end if gradeLetter is "H"
else if (gradeLetter == "O") {
  if (mark >= 90) {
    grade = "O1";
  } else if (mark >= 80) {
    grade = "O2";
  } else if (mark >= 70) {
    grade = "O3";
  } else if (mark >= 60) {
    grade = "O4";
  } else if (mark >= 50) {
    grade = "O5";
  } else if (mark >= 40) {
    grade = "O6";
  } else if (mark >= 30) {
    grade = "O7";
  } else if (mark >= 0) {
    grade = "O8";
  }
} // end if gradeLetter is "O"
```

In the above program, the outer `if` is used to select the code for higher or the code for
ordinary. The selection (i.e. decision) is based on the condition `gradeLetter == "H"`. If this
evaluates to `false` the program will then test the condition `gradeLetter == "O"` (specified in
the `else` part of the outer `if`).

Experiment!
**What would happen if the user entered a letter other than 'H' or 'O'? What if
the user entered a mark over 100 or negative mark?**

The code below implements the same logic as the program on the previous page. This solution also uses nesting but here the if statements are nested in a slightly different order

```
// Alternative nesting to achieve the same result
let gradeLetter = prompt("Enter a letter: H = Higher, O = Ordinary");
let mark = Number(prompt("Enter percentage mark (0-100):"));
let grade;

if (mark >= 90) {
   if (gradeLetter == "H") {
     grade = "H1";
   } else if (gradeLetter == "O") {
     grade = "O1";
   }
} else if (mark >= 80) {
   if (gradeLetter == "H") {
     grade = "H2";
   } else if (gradeLetter == "O") {
     grade = "O2";
   }
} else if (mark >= 70) {
   if (gradeLetter == "H") {
     grade = "H3";
   } else if (gradeLetter == "O") {
     grade = "O3";
   }
} else if (mark >= 50) {
   if (gradeLetter == "H") {
     grade = "H5";
   } else if (gradeLetter == "O") {
     grade = "O5";
   }
} else if (mark >= 40) {
   if (gradeLetter == "H") {
     grade = "H6";
   } else if (gradeLetter == "O") {
     grade = "O6";
   }
} else if (mark >= 0) {
   if (gradeLetter == "H") {
     grade = "H8";
   } else if (gradeLetter == "O") {
     grade = "O8";
   }
}
```

Study the above code carefully. Make sure all the opening and closing braces match up.

Evaluate the above program. What is missing? Do you think the approach taken in this solution is any better than the one on the previous page? Why? Why not?

Finding the maximum of three numbers

The program below determines and displays the largest of three numbers entered by the user.

```
// max of 3
let x1 = Number(prompt("Please enter 1st number: "));
let x2 = Number(prompt("Please enter 2nd number: "));
let x3 = Number(prompt("Please enter 3rd number: "));
let max;

if ((x1 >= x2) && (x1 >= x3)) {
    max = x1;
} else if ((x2 >= x1) && (x2 >= x3)) {
    max = x2;
} else {
    max = x3;
}

console.log("The largest number you entered was", max);
```

Note the use of the logical AND operator, `&&`. The strategy used in this solution is to compare each value to all the other values. This is explained as follows:

- The first condition tests whether `x1` is greater than or equal to both `x2` and `x3`. If the test evaluates to `true` then we can conclude that `x1` must be the largest of the 3 numbers and so we save it in the variable `max`.

- If the first test fails the program moves on to test the second condition. This condition asks whether `x2` is greater than or equal to both `x1` and `x3`. If the answer is yes then we can save `x2` in the variable `max`.

- If the second test fails the only possibility that remains is that `x3` is the largest of the three numbers and so we set `max` accordingly.

> *Key in the above program and test it. Does it work? What is the minimum number of tests you would need to run in order to be sure it works?*

> *Now evaluate the program. How many comparisons need to be made – look at the best and worst cases. What if we wanted to find the largest of four numbers? What about five?*

The flowchart diagrams below illustrate two alternative strategies that can be used to solve the 'max. of three' problem. **Implement and evaluate both solutions**.

Solution A

The strategy taken here is to use a decision tree (implementation involves the use of nesting).
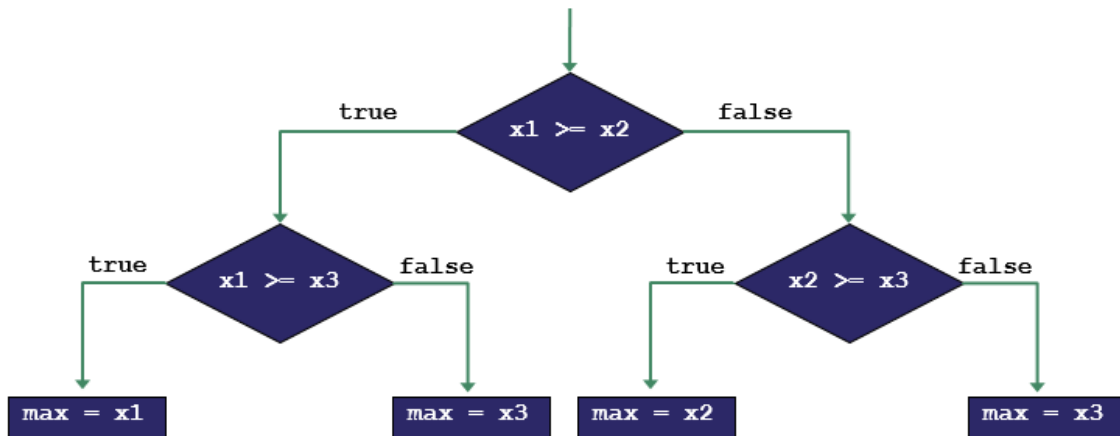
Solution B

This is a more linear (sequential) strategy. It starts by assigning `x1` to `max` and then proceeds to compare each number to `max`. As it does so the value of `max` gets changed to any number found to be larger.
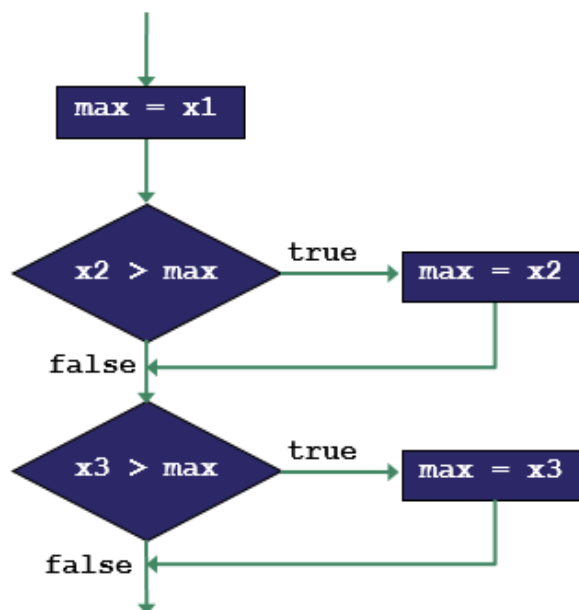
Compare the two strategies shown on the previous page. Which do you think is better and why? How does each solution scale?

Reflection!
What challenged you? What extended your thinking? How could this be used in the LCCS classroom?

The ternary operator

The ternary operator is a shorthand way of writing an `if-else` statement. It is included here for completeness.

The main use of the ternary operator is to assign a value to a variable based on the output of some simple condition. Let's say we wanted to determine the larger of two numbers and store the result in a variable called `max`. We could write:

```
if (n1 > n2) {
  max = n1;
} else {
  max = n2
}
```

The same logic can be achieved with the ternary operator as follows.

```
let max = (n1 > n2) ? n1 : n2;
```

The operator has three parts (hence the name ternary) – delimited by a question mark and colon. The syntax of the ternary operator is as follows:

```
condition ? expression1 : expression2
```

The condition is evaluated first. If the result is `true` the first expression is evaluated and returned. Otherwise, if the condition evaluates to `false` the second expression is evaluated and returned.

The example below assigns `true` to a variable called `isEven` if some integer represented by `num` is even; otherwise `isEven` is assigned the value `false`.

```
let isEven = (num % 2 == 0) ? true : false;
```

The `switch` Statement

A `switch` statement is another code selection mechanism. The body of a `switch` statement is made up of a number of separate *case clauses*. Each `case` clause comprises a value and an associated block of code.

The statement works by testing an expression (e.g. a literal value or a variable) for equality against each `case` value. If a match is found the statements that make up the corresponding block of code are selected for execution.

The syntax of the `switch` statement is shown on the left below and the semantics are explained on the right.

```
switch(expression) {

    case value1:
        statement(s)
        break;

    case value2:
        statement(s)
        break;

    ...

    default:
        statement(s)
} // end switch
```

Starts with the `switch` keyword followed by some expression enclosed in brackets

The result of the expression is compared to each value following the `case` keyword

If a match is found, the associated statement(s) are executed until the `break` statement is reached

The `break` statement causes the flow of control to 'jump' to the first line of code following the end of the `switch` statement

There can be any number of `case` clauses

The statement(s) in the (optional) `default` block are executed if no match is found in any of the `case` clauses

Notes:

1) The result of the expression is compared to the value for a `case` using the strict equality operator (===). The values must therefore match without any type conversion.

2) When the value being switched on is equal to a `case`, the statements following that `case` will execute until a `break` statement is reached.

3) When a `break` statement is reached, the `switch` terminates, and the flow of control jumps to the next line following the switch statement.

4) There can be any number of `case` statements within a `switch`. Each `case` is followed by the value to be compared to and a colon.

5) Not every `case` needs to contain a `break`. If no `break` appears, the flow of control will *fall through* to subsequent cases until a `break` is reached. (It is generally considered good programming practice to include a `break` statement at the end of each set of case statements.)

6) A `switch` statement can have an optional `default` case, which must appear at the end of the `switch`. The `default` case can be used for performing a task when none of the matches evaluate to `true`. Since `default` is usually left at the end of the `switch` statement it usually doesn't need to include a `break` statement. (It does no harm however to include a `break` statement inside the `default`.)

KEY POINT: A `switch` statement is an alternative to the multiway `else-if` construct described earlier in this section and is typically used when the number of `else-if` branches start to exceed four or five.

The following two programs are logically equivalent. The program on the left makes repeated use of `else-if` statements and the program on the right makes use of a `switch` statement to do the same thing (i.e. display the word for any integer from 1-4.)

```
let x = prompt("Enter a number from 1-4")

if (x == 1) {
    console.log("One");
} else if (x == 2) {
    console.log("Two");
} else if (x == 3) {
    console.log("Three");
} else if (x == 4) {
    console.log("Four");
} else {
    console.log("Sorry. Unknown value.");
}
```

```
let x = prompt("Enter a number from 1-4")

switch (x) {
  case 1:
      console.log("One");
      break;
  case 2:
      console.log("Two");
      break;
  case 3:
      console.log("Three");
      break;
  case 4:
      console.log("Four");
      break;
  default:
      console.log("Sorry. Unknown value.");
      break;
}
```

JS

Programming challenge!
Implement the following program so that it uses a `switch` statement

```javascript
let country = prompt("Enter a country and I will tell you its capital");

if (country == "Ireland") {
   console.log("Dublin");
} else if (country == "Scotland") {
   console.log("Edinburgh");
} else if (country == "England") {
   console.log("London");
} else if (country == "Wales") {
   console.log("Cardiff");
} else if (country = "France") {
   console.log("Paris");
}
```

Experiment!
Key in the following code and enter some values. See if you can figure out what the code does. Describe exactly what it does and how it does it. Can you find any bugs and, if so, can you suggest any solutions?

```javascript
let num = prompt("Enter a number");
let rem = num % 10;
let suffix;

switch (rem) {
   case 1:
      suffix = "st";
      break;
   case 2:
      suffix = "nd";
      break;
   case 3:
      suffix = "rd";
      break;
   default:
      suffix = "th";
} // end switch

console.log("Output:", num+suffix);
```

Programming Exercises – Selection Statements

1. Write a program that prompts a user to enter a month number and then display the number of days in that month using the name of the month in the output message. (Assume February has 28 days.) For example, if the user enters 3, the program should display the message – *March has 31 days*.

2. Write a program that prompts a user to enter two separate integers - a day number and a month. The program should output the message *The dd/mm are a valid combination* if the combination is valid, and *The dd/mm combination is invalid* otherwise. Examples of invalid combinations are 32, 1 (January has only 31 days) and 5, 13 (there are only 12 months).

3. Write a program that prompts a user to enter a year and display whether the year entered was (or will be) a leap year or not. A year is defined to be leap if it is exactly divisible by 4 except when it is also exactly divisible by 100. Years that are exactly divisible by both 4 and 100 are leap only if they are also divisible by 400. So,

   ✓ 1992, 2020 and 2104 are leap years because they are divisible by 4 (and not by 100).
   ✓ 1800, 1900 and 2200 are not leap because they are exactly divisible by 4 and 100 but are not further divisible by 400.
   ✓ 1600, 2000 and 2400 are leap because they are exactly divisible by 4 and 100 and 400.

   There are many coding solutions to determine whether a year is a leap year or not. The pseudo-code for one such solution is provided here. See if you can implement it.

   ```
   if (year is not divisible by 4) → (Not Leap)
   else if (year is not divisible by 100) → (Leap Year)
   else if (year is not divisible by 400) → (Not Leap)
   else (Leap Year)
   ```

   There are lots of other ways to express the same logic – see if you can come up with some of your own!

Reflect on all the exercises in this section.

Suggest how you could adapt any of the exercises for use in your own LCCS classroom.

8. Iteration Statements (loops)

Iteration statements are commonly referred to as loops. They enable programmers to write code that will be repeatedly executed at runtime.

The idea of iteration is shown below - the green line depicts the execution path of the loop. The red line shows what happens when the loop is not executed.
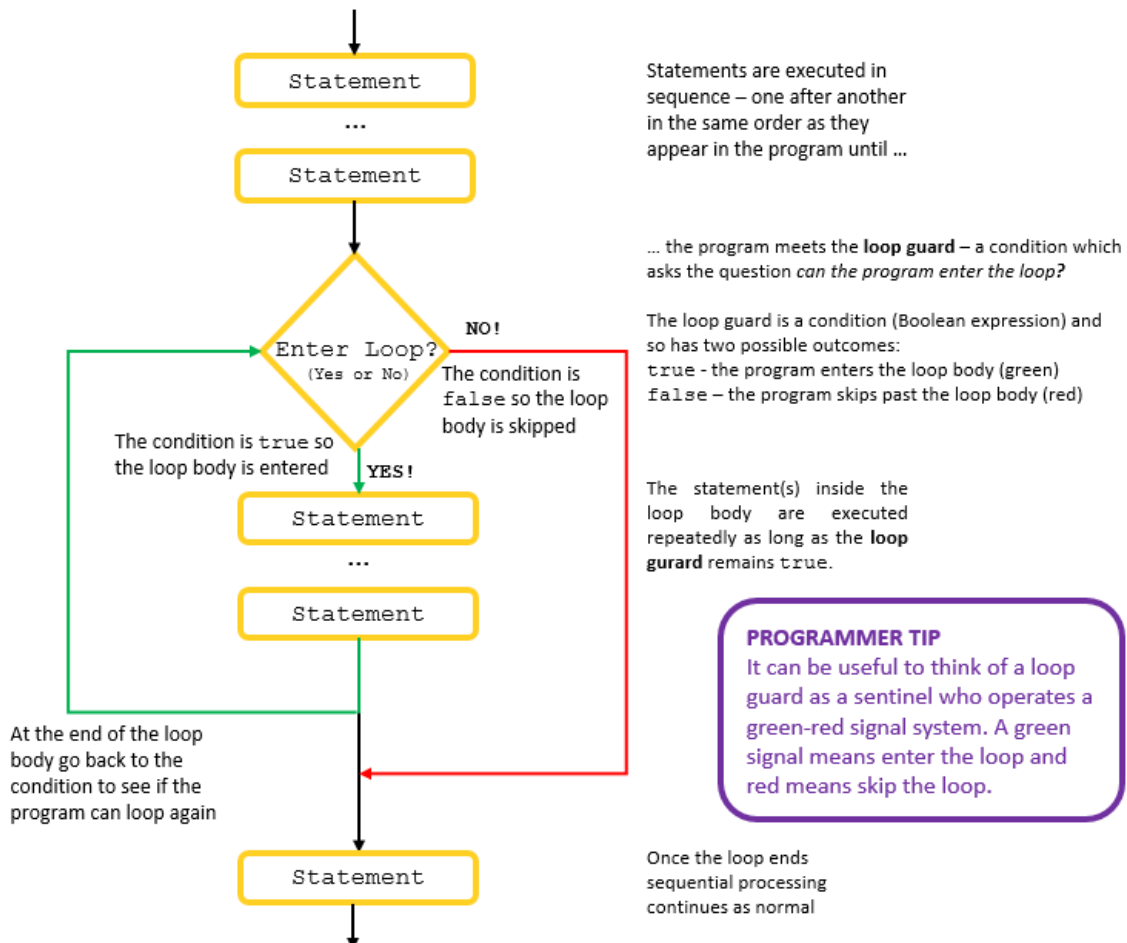
A loop is made up of two key components – a **loop guard** and a **loop body**.

- A *loop guard* is a condition used to determine whether the loop body should be executed or not. If the loop guard evaluates to `true` the loop body gets executed; otherwise it doesn't.
- A *loop body* is simply a block of code that gets executed over and over again. Every time a loop body is executed it is known as an *iteration*. The loop guard is re-tested at the end of every iteration and loop body is executed as long as the result of this test is `true`. Once the loop guard condition evaluates to `false` the loop said to *terminate* and processing continues at the next line after the loop body.

Loops are useful because they save programmers from having to copy-and-paste potentially many lines of code in their programs. For example, let's say a programmer wanted to display the string *Hello World* 100 times on the output console. Without loops they would have to write 100 lines of code – one line per each line of output – as follows

```
console.log("Hello World"); // 1st time                    Hello World
console.log("Hello World"); // 2nd time                    Hello World
console.log("Hello World"); // 3rd time                    Hello World
console.log("Hello World"); // 4th time                    Hello World
///                                                        ...
console.log("Hello World"); // 56th time. (Yawn!)          Hello World
console.log("Hello World"); // 57th time. (Yawn! Zzz!)     Hello World
///                                                        ...
console.log("Hello World"); // 99th time. (zzzZZ!)         Hello World
console.log("Hello World"); // 100th time (At last!)       Hello World
```

A 100 line program to display the string Hello World 100 times   ***Program Output***

KEY POINT: A loop is a programming construct that allows the same block of code to be executed multiple times

The repetition in the above code is obvious. The diagram below illustrates how the repetitive nature of this code can be exploited by loops.
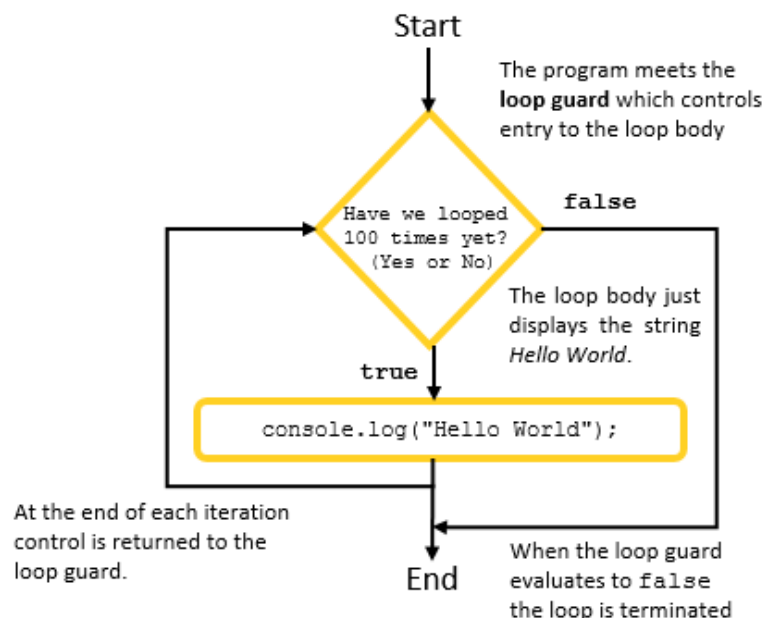
JavaScript supports a number of different types of loops – `while, do-while` and `for`. The syntax and semantics of each are now considered in turn.

The `while` loop

The syntax of the `while` loop is as follows:

```
while (condition) {
    statement(s)
}
```

> The condition is the loop guard and the statements make up the loop body

The code below demonstrates the use of a `while` loop to display *Hello World* 100 times.

```
let counter = 0;
while (counter < 100) {
  console.log("Hello World");
  counter++; // Add 1 to counter – eventually it will reach 100
}
```

That's it – just five lines (as opposed to 100) - simple!

The semantics of while loops can be explained using the flowchart to the right.

The key point is that loop body statements are executed each time the loop guard condition evaluates to true

In our *Hello World* example, the loop guard is `counter < 100`. The two statements inside the loop body are repeatedly executed as long as counter remains less than 100. Notice that the counter is incremented at the end of each iteration ensuring that the loop will eventually end.
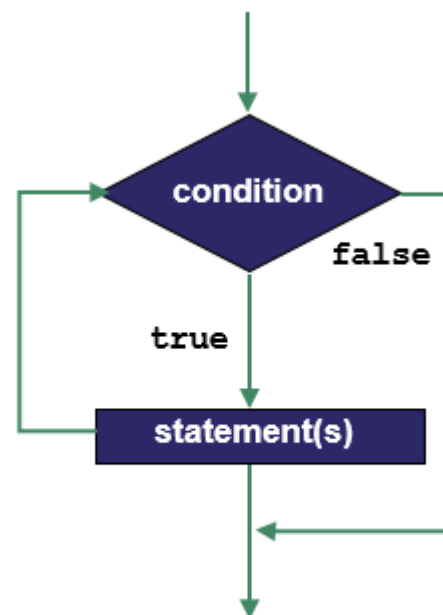
TEACHER TIP
A common student misconception is to think that a while loop's condition is being constantly evaluated and the loop exits the instant it becomes `false` inside the loop body.

Let's look at some examples.

Example 1 (times tables)

In this example, we will develop a program to display the 7 times tables (up to 12).

Think about it – we need to display all the lines from $7 \times 1 = 7$ up to $7 \times 12 = 84$. The desired output is shown on the right.

In all of these output lines the first integer is constant i.e. 7. The second integer is variable i.e. it varies from 1 up to 12. This could be our loop counter – let's call it `counter`.

The value on the right hand side of the equals sign is calculated by multiplying 7 by `counter`.

Each output line is generated by the statement:
`console.log("7 x", counter, "=", 7*counter);`

```
7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
7 x 11 = 77
7 x 12 = 84
```

The full program to display the 7 times table is now given:

```
let counter = 0;
while (counter <= 12) {
   console.log("7 x", counter, "=", 7*counter);
   counter++; // Add 1 to counter
```

The program shown below is a slight enhancement – it asks the user what times tables they wish to have displayed.

```
let counter = 0;
let tables = prompt("What times tables do you require?");
while (counter <= 12) {
   console.log(tables, "x", counter, "=", tables*counter);
   counter++; // Increment counter
}
```

Key it in and try it out for yourself!

Example 2 (class average)

Let's say we wanted to write a program that prompted the user to enter five integers and then display their arithmetic mean.

One solution could be achieved as follows.

```javascript
let x1 = Number(prompt("Enter integer value 1"));
let x2 = Number(prompt("Enter integer value 2"));
let x3 = Number(prompt("Enter integer value 3"));
let x4 = Number(prompt("Enter integer value 4"));
let x5 = Number(prompt("Enter integer value 5"));
let total = x1+x2+x3+x4+x5;

console.log("Mean value:", total/5);
```

PROGRAMMER TIP
Recognising repeated patterns in code can act as a trigger to ask the question – could a loop be used here instead?

This all seems to be a bit repetitive and verbose – these can often be sure signs that a loop might be a better solution.

Let's explore a solution to this problem that uses a `while` loop.

The loop solution will need to keep a running total of all the values entered. Once the final value has been entered the mean can be calculated, simply by dividing the total by five. We use pseudo-code as an initial step in the development of our solution.

```
Initialise the total to zero
Initialise a counter to 0

Loop 5 times (while the counter is less than 5)
   Prompt the user to enter an integer value
   Update the total with the value just entered (total+=value)

Display the total divided by 50
```

Challenge!
Can you translate the pseudo-code above into a JavaScript program?

KEY POINT: When the number of loop iterations is known before a program is run, a counter variable can be used in the loop guard. This is called **counter controlled repetition**.

Example 3 (sentinels)

In the previous two examples we used a counter to control the number of loop repetitions. This was possible because we knew the number of times we wanted the loop body to be executed in advance of running the program. However, there are many situations where the programmer does not know the number of iterations required in advance. Consider for example the problem of adding an unknown number of numbers.

The challenge here is to develop a loop guard that allows the loop to be executed a variable number of times.

```
Initialise the running total to zero
Prompt the user to enter the first number

Loop as long as there are more numbers to add
   Add the number just entered to the running total
   Prompt the user to enter another number

Display the total
```

What exactly does '*as long as there are more numbers to add*' mean? To answer this question, we need to understand *sentinels*.

KEY POINT: A **sentinel** is a value used in a loop guard when the number of iterations is not known before the program is run. The sentinel value is decided upon by the programmer and used to form the condition that will terminate a loop.

The shown code below is a JavaScript implementation of the pseudo-code shown above. The sentinel is the condition `number != 0` which means that the loop runs as long as the value in number is not equal to zero. An unknown number of numbers can be entered.

```
let runningTotal = 0;
let number = Number(prompt("Enter a number (0 to end):"));

while (number != 0) {
   runningTotal += number;
   number = Number(prompt("Enter a number (0 to end):"));
}

console.log("Total:", runningTotal);
```

The only way the loop can end is by the user entering a value of zero for `number`. This would cause the condition `number != 0` to evaluate to `false` which in turn would cause the loop to end.

Loop counters and sentinels

It is not uncommon for counters to be used in sentinel controlled loops. The main purpose of such counters is to keep a track of the number of iterations that took place.

For example, if we wished to calculate the average of an unknown number of values – each entered by the user – we would need to divide by the counter once all the values had been added. This is illustrated in the program below.

```
let counter = 0;
let runningTotal = 0;
let number = prompt("Enter a number ('stop' to end):");

while (number != 'stop') {
   counter++; // Increment counter
   runningTotal += Number(number);
   number = prompt("Enter a number ('stop' to end):");
}

if (counter == 0) // avoid dividing by zero
   console.log("No mean value as there were no values entered");
else
   console.log("Mean value:", runningTotal/counter);
```

Note the use of 'stop' as a sentinel value i.e. the loop continues until the user enters 'stop'.

In the above code the variable `counter` is initialised to zero and then incremented on every iteration of the loop. In effect the `counter` is being used to record of the number of integer values that are entered by the user. When the loop finally comes to an end the value of `counter` could be any integer value greater than or equal to zero. For all positive integers the mean value is calculated as `runningTotal/counter` and displayed. If `counter` is zero, the user gets a message to say there is *No mean value as there were no values entered*.

The general pattern for a sentinel loop can be expressed in pseudo-code as follows

```
prompt the user to enter the first value
while value is not the sentinel:
   process the value
   prompt the user to enter the next value
```

JS

Programming Exercises (choosing sentinels)
Translate the pseudo-code given in the questions below into JavaScript.
The main challenge in each question will be to choose a reliable sentinel.

1. The purpose of this program is to calculate and display the product of a variable list of positive numbers entered by the user.

```
Initialise product to one
Prompt the user to enter the first number
while number is not equal to sentinel value
    calculate the product (i.e. product=product*number)
    prompt the user to enter the next number
Display the total product of all the numbers entered
```

2. This program repeatedly reads a positive integer from the user and displays its square root.

```
prompt the user to enter the positive number
while number is not the sentinel
    root = Math.sqrt(number);
    display the root
    prompt the user to enter another positive number
```

3. This program selects a random number between 1 and 10 and keeps asks its user to enter a guess until the correct number has been guessed.

```
generate a random number between 1 and 10
read the first guess from the user
while the random number is not the same as the guess:
    read the next guess
```

Suggest how the program could be modified to:
a) allow a maximum of three guesses
b) end when the user types the word 'stop'

PROGRAMMER TIP
When choosing a sentinel value programmers need to be careful not to use a value that might be 'of interest' to the loop. For example, if the loop body is dealing with numbers greater than zero, then the sentinel could be either zero or any negative number or any word such as 'stop'.

Example 4 (validating data)

One common use for loops and sentinels is to validate date i.e. make sure that the data entered by a user is valid. Exactly what is meant by valid must be decided by the development team as part of the design process. For example, if you were entering someone's age into a system what would the valid values be? (Any numeric value between 0 and 120 might be considered reasonable.) What about a snapchat handle or an email address?

The general pattern is to keep prompting the user to enter the value in question until it is valid. This pattern is shown in the following pseudo-code. The loop ensures that by the time the last line is reached the program has a valid value to process.

```
prompt user to enter a value
while the value is not valid
    [display error message] // optional
    prompt user to enter a value

process value
```

The example below shows how to validate a yes/no type of response. The loop guaranteed that by the time the program ends the user will have entered either *Y* or *N*. All other values are 'trapped' by the loop.

```
// Validate a yes/no response
let response = prompt("Do you wish to continue (Y/N)");
while (response != "Y" && response != "N") {
    response = prompt("Do you wish to continue (Y/N)");
}

// response is valid
console.log("Thank you. You entered a valid response ... ");
```

This next example keeps looping until the user enters an integer between 1 and 12 inclusive. It could be used to validate a month number.

```
// Validate a month number
let month = Number(prompt("Enter a month number (1-12)"));
while (isNaN(month) || (month < 1 || month > 12)) {
    month = Number(prompt("Enter a month number (1-12)"));
}

console.log("Thank you. You entered a valid month number ... ");
```

The `do-while` loop

A `do-while` loop is similar to a `while` loop, except that a `do-while` loop is guaranteed to execute at least once.

The syntax of a `do-while` loop is:

```
do {
    statement(s)
} while(condition);
```

> Notice the semi-colon at the end of the `while`

> The statement(s) in the loop body are run at least once

The use of curly braces is recommended even though technically they are needed only when there is more than one statement in the loop body. Notice that the opening curly brace appears directly *after* the `do` keyword and the closing curly brace appears directly *before* the `while` keyword.

The above syntax is illustrated in the flow diagram shown below.
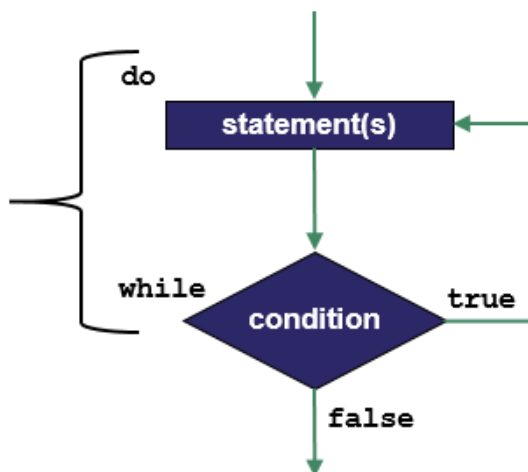
Notice that the loop guard condition appears at the end of the loop body, so the statement(s) in the loop body are guaranteed to execute once before the condition reached.

If the loop guard condition evaluates to `true`, the flow of control jumps back up to the first statement in the loop body, for another iteration. This process repeats until the loop guard condition evaluates to `false`.

The choice between using a `while` and `do-while` can often be a matter of personal taste to the programmer. The logic of a `do-while` can always be achieved with a `while` loop but the reverse isn't always the case. For this reason, the `while` construct is considered more flexible and probably used more often than `do-while`.

The unique selling point of `do-while` is that the statements in a loop body are guaranteed to be executed at least once. This is exploited in the example snippet below which validates a yes/no response entered by the user. The loop continues as long as the response is neither Y nor N.

```
// Validate a yes/no response (do-while version)
let response;
do {
    response = prompt("Do you wish to continue (Y/N)");
} while (response != "Y" && response != "N");

// response is valid
console.log("Thank you. You entered a valid response ... ");
```

Challenge!
Modify the two code listings below so that they use a `do-while` loop instead of a `while` loop.

```
// Validate a month number
let month = Number(prompt("Enter a month number (1-12)"));
while (isNaN(month) || (month < 1 || month > 12)) {
   month = Number(prompt("Enter a month number (1-12)"));
}

console.log("Thank you. You entered a valid month number ... ");
```

```
let runningTotal = 0;
let number = Number(prompt("Enter a number (0 to end):"));

while (number != 0) {
   runningTotal += number;
   number = Number(prompt("Enter a number (0 to end):"));
}

console.log("Total:", runningTotal);
```

Reflection!
What is your preference – `do-while` or `while`?

JS *Programming Exercises (`while` loops)*

1. Read each of the following code snippets carefully and predict the output that would be generated by each of the while loops that contain. In each case you should key in and run the code to compare your predictions with the actual output.

Loop 1	Predicted Output	Actual Output
``let count = 1;
while (count <= 5) {
console.log(count);
count++;
}```		

Loop 2	Predicted Output	Actual Output
``let count = 1;
while (count <= 5) {
count++;
console.log(count);
}```		

Loop 3	Predicted Output	Actual Output
``let count = 1;
while (count > 5) {
console.log(count);
count++;
}```		

Loop 4	Predicted Output	Actual Output
``let count = 0;
while (count < 10) {
console.log(count);
count += 2;
}```		

Loop 5	Predicted Output	Actual Output
``let count = 5;
while (count > 0) {
console.log(count);
count--;
}```		

Loop 6	Predicted Output	Actual Output
``let count = 5;
while (count > 0) {
count--;
console.log(count);
}```		

Loop 7	Predicted Output	Actual Output
``let count = 5;
while (count >= 0) {
console.log(count);
count--;
}```		

Loop 8	Predicted Output	Actual Output
``let count = 1;
while (count++ <= 5) {
console.log(count);
}```		

Loop 9	Predicted Output	Actual Output
``let count = 1;
while (++count <= 5) {
console.log(count);
}```		

2. Write a program that prompts the user to enter two integers and then displays all the integers from the lower integer up to and including the higher. So, for example if the user entered 3 and 7, the output generated would be 3 4 5 6 7 (with each integer being displayed on a separate line.
An extra challenge would be to display the integers on a single line.)

3. Write programs that generate the output depicted below:
a) The addition table for 4
b) A Celsius to Fahrenheit lookup table for all Celsius values between zero and 100 in steps of 10. The formula is $F = \frac{9}{5}C + 32$.

a)
```
4 + 0 = 4
4 + 1 = 5
4 + 2 = 6
4 + 3 = 7
4 + 4 = 8
4 + 5 = 9
4 + 6 = 10
4 + 7 = 11
4 + 8 = 12
4 + 9 = 13
4 + 10 = 14
4 + 11 = 15
```

b)
```
C     F
0     32.0
10    50.0
20    68.0
30    86.0
40    104.0
50    122.0
60    140.0
70    158.0
80    176.0
90    194.0
100   212.0
```

4. Write a program that sums all the numbers from `low` to `high` where `low` and `high` are two integers entered by the end-user. For example, if the end-user entered 8 and 13 the program would compute and display the result of $8 + 9 + 10 + 11 + 12 + 13$.

5. The factorial of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Write a program that prompts a user to enter a number and then computes and displays its factorial

6. Suggest (and implement) a possible validation rule for the following data values:
   a) A percentage mark
   b) A grade in the Leaving Certificate
   c) A CAO course code
   d) A Twitter handle
   e) An email address
   f) An Eircode
   g) Any telephone number
   h) An Irish vehicle registration number

The `for` loop

The syntax of `for` loops is shown below.

```
for(initialisation; condition; step){
    statement(s)
}
```

The `for` statement contains three parts each separated by semi-colons - initialisation, condition and step. The statement(s) make up the loop body.

The semantics of `for` loops can be explained using the following annotated flowchart.
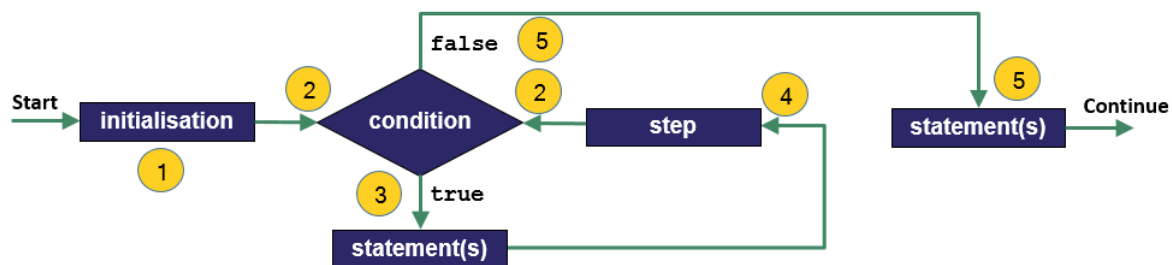
1. The loop variable is initialised at the start

2. The condition is evaluated on each loop iteration

3. The statements that make up the loop body are executed every time the condition evaluates to `true`

4. When the loop body ends the loop variable is stepped (usually an increment or decrement operation)

5. Once the condition evaluates to `false` the loop ends and processing continues at the next statement after the loop.

Let's explain this using an example. The code below uses a `for` loop to display the string *Hello World* three times.

```
for (let counter = 0; counter < 3; counter++) {
    console.log("Hello World");
}
```

```
Hello World
Hello World
Hello World
>
```

| *A simple `for` loop* | *The output* |

When the JavaScript engine runs the above code, it starts in the initialisation section (1). The variable `counter` is declared and initialised to zero. The condition `counter < 3` is then evaluated (2). The first time the loop is executed this condition will evaluate to `true` and so the loop body is executed (3). The loop body displays the string *Hello World* on the console and processing continues at (4) where the value of `counter` is incremented (`counter++`). After this step the condition is re-evaluated, and, based on the outcome either the

statements in the loop body will be executed again or the loop will terminate in which case the next statement to be executed will be the first statement after the loop body (5).

This description can be expressed using code as follows. Both this, and the code snippet shown on the previous page are logically equivalent. They both generate the same output as shown here on the right.

```
Hello World
Hello World
Hello World
>
```

```javascript
// Initialisation
let counter = 0;

// Condition – 1st iteration
if (counter < 3)
  console.log("Hello World");
counter++; // step 1

// Condition – 2nd iteration
if (counter < 3)
  console.log("Hello World");
counter++; // step 2

// Condition – 3rd iteration
if (counter < 3)
  console.log("Hello World");
counter++; // step 3

// At this stage the value of counter will be 3
```

It is worth pointing out that the variable in the initialisation section is referred to as the *loop variable*. The name of the loop variable in the above example is `counter`. The condition in a `for` loop is typically some Boolean expression involving the loop variable. At the end of each loop iteration the value of the loop variable is normally stepped – typically by incrementing/decrementing it. The condition should be programmed so that it will eventually evaluate to `false` at which point the loop will end.

KEY POINT: In a `for` loop the condition is *re-evaluated* every time after the step stage. If the condition evaluates to `true`, the loop body is executed. If the condition evaluates to `false` the loop terminates and processing continues at the next statement that appears in the code after the end of the loop body.

Examples - `for` loops and `while` loops

Another way to explain `for` loops is in terms of `while` loops. Even though the syntax of `while` and `for` (shown below) is different, the two loops are logically equivalent.

```
initialisation;                for(initialisation; condition; step) {
while (condition) {                 statement(s)
   statement(s)                }
   step
}
```

Syntax of `while` loop   **Syntax of `for` loop**

Each pair of code snippets in the examples below do the same thing – the `while` loop implementation is shown on the left and the logically equivalent `for` loop implementation is shown on the right. Study each example carefully and use the space provided to record your notes as you do so.

Example 1 - display the integers from 1 to 10.

```
let count = 1;
while (count <= 10) {
   console.log(count);
   count++;
}
```

```
for (let count = 1; count <= 10; count++) {
   console.log(count);
}
```

Notes:

Example 2 - display the sequence 0, 2, 4, 6, ... 100

```
let count = 0;
while (count <= 100) {
  if (count % 2 == 0)
    console.log(count);
  count++;
}
```

```
for (let count = 0; count <= 100; count+=5) {
  if (count % 2 == 0)
     console.log(count);
}
```

Notes:

Example 3 - display every 5th integer between 0 and 100 and its square

```javascript
let count = 0;
while (count <= 100) {
  if (count % 5 == 0)
     console.log(count,
Math.pow(count,2));
  count+=5;
}
```

```javascript
for (let count = 0; count <= 100; count+=5) {
  if (count % 5 == 0)
     console.log(count, Math.pow(count,2));
}
```

Notes:

_____

_____

_____

TEACHER TIP

Sometimes students ask when should one loop construct be used in preference over another. The simple answer is that it doesn't really matter – student programmers should use whichever they feel more comfortable with. The following more detailed suggestion might be also help.

Use a `for` loop in preference to a `while` loop when the number of iterations are known in advance of running the program. Counter contolled situations such as this lend themselves well to the use of a `for` loop. A `for` loop is also particularly useful for iterating over individual elements of any objects such as the characters in a string or elements of an array.

The `while` loop is generally regarded as a more flexible loop construct and therefore should be used in all other situations.

Example 4 - count the number of multiples of 3 between 0 and 100

```javascript
let i=0, multiplesOf3 = 0;
while (i <= 100) {
  if (i % 3 == 0)
    multiplesOf3++;
  i++;
}
console.log(multiplesOf3,
"found");
```

```javascript
let i, multiplesOf3;
for (i = 0, multiplesOf3 = 0; i <= 100; i++) {
  if (i % 3 == 0)
     multiplesOf3++;
}
console.log(multiplesOf3, "found");
```

Notes:

_____

_____

_____

Example 5 - calculate 5! (i.e. 5x4x3x2x1) and then display the result

```
let fact = 1;
let n = 5;
let number = n;
while (number > 0) {
  fact = fact * number;
  number--;
}
console.log(n+"! =", fact);
```

```
let fact = 1;
let n = 5;
for (let number = n; number > 0; number--) {
  fact = fact * number;
}
console.log(n+"! =", fact);
```

Notes:

_____

_____

_____

Infinite Loops

If the loop guard always returns `true`, the loop body will continue to be executed forever. Such loops are called **infinite loops**. There are occasions when programmers deliberately program their loops to run forever. However, when they are not programmed intentionally infinite loops cause a running program to 'hang' (which is a major inconvenience for the end-user). When writing loops programmers should take care to safeguard against unintentional infinite loops. In most cases the loop body should have some statement that will eventually render the loop guard `false` thereby causing the loop to terminate.

```
count = 0
while (count <= 10) {
   console.log("Infinite Loop");
}
```

```
for (; ;) {
   console.log("Infinite Loop");
}
```

Two example infinite loops. The loop guard condition never becomes `false`

KEY POINT: An **infinite loop** is one that never ends. They occur in code where the loop guard condition always evaluates to `true`.

PROGRAMMER TIP
Care should be taken not to write code that inadvertently causes infinite loops. Loop guards need to be designed with caution and testing needs to be carried out thoroughly.

PDST
Professional Development | An tSeirbhís um Fhorbairt
Service for Teachers | Ghairmiúil do Mhúinteoirí

HTML CSS JS

JS *Programming Exercises (`for` loops)*

1. Read each of the following code snippets carefully and predict the output that would be generated by each of the `for` loops shown. In each case you should key in and run the code and then compare your predictions with the actual output.

Loop 1	Predicted Output	Actual Output
```for (let count = 0; count <= 5; count++) { console.log(count); }```		

Loop 2	Predicted Output	Actual Output
```for (let count = 1; count < 5; count++) { console.log(count); }```		

Loop 3	Predicted Output	Actual Output
```for (let count = 0; count <= 100; count+=10) { console.log(count); }```		

Loop 4	Predicted Output	Actual Output
```for (let count = 5; count > 0; count--) { console.log(count); }```		

Loop 5	Predicted Output	Actual Output
```for (let count = 10; count >= 0;) { console.log(count); count -= 2; }```		

2. Compare the two programs below in terms of what they do and how they do it

a) Use the space provided to record the output that would be displayed by each program if the user entered a value of 3 for start and 7 for end

OUTPUT for 3 and 7

```
let start = Number(prompt("Enter a number to start from"));
let end = Number(prompt("Enter a number to end at"));

for (; start <= end; start++) {
   console.log(start);
}
```

b) Why does the above for loop not have any initialisation expression?

c) Can you suggest an initialisation expression to use in the for loop?

d) Experiment! Can the same logic be achieved without start++ in the for loop?

```
let start = Number(prompt("Enter a number to start from"));
let end = Number(prompt("Enter a number to end at"));
let outStr = "";

while (start <= end) {
   outStr = outStr + start + ", ";
   start++;
}
console.log(outStr);
```

OUTPUT (for 3 and 7) :

e) What is the purpose of the variable outStr in the above program?

f) Modify the program so that there is no trailing comma displayed at the end of the output (i.e. commas should only appear *between* the values displayed).

The `break` and `continue` statements

We now turn our attention to two keywords that relate to loops - `break` and `continue`.

break

A `break` statement can only be used inside any loop or `switch` statement. When it is executed it forces the loop or `switch` statement in which it is used to stop. Processing is transferred to the first line after the end of the loop or `switch`.

The following example illustrates the use of `break`. The output is shown to the right.

```
let someValue = 7;
while (someValue > 0) {
  someValue--; // decrement
  if (someValue == 3)
    break;
  console.log(someValue);
}
```
```
6
5
4


Program Output
```

The most common use for `break` is to exit a `switch` statements once a `case` has been processed. The `break` statement is also used by experienced programmers as a means to exit from loops that are guarded by conditions that always evaluate to `true` (i.e. infinite loops).

continue

The `continue` statement can only be used inside any of the loop structures. It causes the loop to skip one iteration by immediately jumping to the next iteration of the loop. The flow of control is altered as follows:

− in `while` and `do-while` loops, the `continue` statement causes the flow of control to jump immediately to the loop guard condition
− in a `for` loop, the flow of control immediately jumps to the step expression (i.e. the third part of the for statement)

The following example illustrates the use of `continue`. The output is shown to the right.

```
let someValue = 6;
while (someValue > 0) {
  someValue--; // decrement
  if (someValue == 3)
    continue;
  console.log(someValue);
}
```
```
5
4

2

1

0
```

Nested loops

A nested loop is a loop inside another loop.

The example shown below contains two for loops. The first `for` loop is called the *outer loop* and the second `for` loop is referred to as the *inner or nested loop*. The inner loop is nested inside the outer loop.

```
for (let row = 1; row<5; row++) {
  for (let col = 1; col<=3; col++) {
      console.log(row, col);
  }
}
```

The outer loop iterates over the variable `i` and the inner loop iterates over the variable `j`.

The inner loop is executed once for each iteration of the outer loop.

In this example, the outer loop is executed 4 times. As is the case with all loops, the loop body is repeatedly executed. In this case the loop body happens to be another loop – the *inner loop*. The inner loop does 3 iterations – its loop body displays the loop variables for both loops on the console i.e. `row` and `col`. The output is shown here to the right.

As can be see the value of `row` is displayed before the value of `col`. The outer loop variable moves slower than the inner loop variable. Between the two loops there are a total of 15 iterations.

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
```

The reason that programmers use nested loops are no different to those for using any other loop i.e. to repeat a block of code. The block of code to be repeated just happens to be a loop.

Take for example the `for` loop shown here which display the 7 times tables.

```
for (let count=0; count<=12; count++) {
   console.log("7 x", count, "=", 7*count);
}
```

As can be seen, the code loops through all the integers from 0 to 12 multiplying each by 7 as it does so.

Now let's say we wanted to display all the multiplication tables from 1 up to 10 (and not just the 7 times tables). The way to do this would be with a nested loop.

We take our '7 times' loop and wrap it inside another loop that iterates 10 times. Of course we don't want to display the 7 times tables 10 times so we use a variable called `tables` to indicate which 'times tables' are to be displayed. The solution is as follows.

```
for (let tables=1; tables<=10; tables++) {
   for (let count=0; count<=12; count++) {
      console.log(tables,"x", count, "=", tables*count);
   } // inner loop

   console.log("\n\n");
} // outer loop
```

The outer loop iterates through the tables from 1 to 10 and the inner loop iterates through the values from 1 to 12 for each table. In effect we are wrapping the code to display a single multiplication table inside a loop that steps through ten tables.

PROGRAMMER TIP
To create the code for nested loops it is sometimes useful to decompose the task into separate steps. First create the loop that will eventually become the nested loop. Once this has been tested ask how many times does this inner loop need to be executed. Now write an outer loop with that number of iterations and include the inner loop as its loop body.

Nested loops are commonly used for iterating over two dimensional (2D) arrays (i.e. the inner loop does the horizontal processing and the outer or slower loop does the vertical processing). 2D arrays are a data structure used to represent any data that can be organised into rows and columns e.g. matrices, battleship, chessboard, Sudoku etc.). They are used in the implementation of many board games.

A nested loop can also contain a loop – in this case we would have a loop within a loop within a loop (e.g. loop 3 inside loop 2 which is inside loop1, the outer loop). In theory, loops can be nested to any depth - the example below contains four levels of nesting. This example is a little extreme –in practice two levels of nesting is most common.

Example

The following program is an advanced illustration of the use of nested loops to find and display all the 4 digit happy numbers. A number is said to be a 'happy' if the sum of the first two digits is the same as the sum of the last two digits. Examples of happy numbers are 1111, 5005 and 8439 - can you think of others?

```
// Happy numbers e.g. 8439 because 8+4 == 3+9
for (let a = 0; a<10; a++)
  for (let b = 0; b<10; b++)
    for (let c = 0; c<10; c++)
      for (let d = 0; d<10; d++)
        if (a+b == c+d)
          console.log("Happy Number:", a, b, c, d);
```

PROGRAMMER TIPS

The following steps should be taken into consideration when designing loops:

1. The first step is to recognise situations where a loop could be used. This takes experience which comes with practice. The possibility that a block of code will be executed more than once is a sure sign that a loop will be needed. (The block of code will be the loop body.)

2. Decide on a loop guard - the following might be of some help in this regard. Under what condition(s) do you want to execute the loop body? Do you want the loop executed a fixed number of times? If so you need to decide how many times and use a counter in your loop guard. If your loop doesn't have a fixed number of repetitions then you might consider using a Boolean variable (or flag) as a sentinel.

3. Make sure that your loop body contains code that will cause your loop guard to eventually evaluate to `false`. This may be as simple as adding one to a counter.

Exercise (Example of a parson's problem)

Move the three lines given on the right hand side into the correct places (A, B, and C) to the code block shown so that it will display the factorial of the first five natural numbers.

```
let factorial;
A
   factorial = 1;
      B
      factorial = factorial * number;
   } // end inner
   C
} // end outer
```

```
for (let number = count; number > 0; number--) {
```

```
console.log(count, "! =", factorial);
```

```
for (let count = 1; count <= 5; count++) {
```

Hint: The code shown below calculates and displays 6!

```
let factorial = 1;
for (let number = 6; number > 0; number--) {
   factorial = factorial * number;
} // end for
console.log(6, "! =", factorial);
```

Swimming involves repetitive actions.
Explain how the concept of nested loops could be used to direct a 'bot' to swim 10 lengths of a pool.

JS Programming Exercises - Loops

1. Write a JavaScript loop that displays the integers from 1 up to 10 inclusive.

2. Write a JavaScript loop that displays the integers from 10 to 20 inclusive.

3. Write a JavaScript loop that displays the integers from 10 down to 1 inclusive.

4. Write a loop that displays every 10th integer between zero and 100 inclusive i.e. 0, 10, 20, 30 etc.

5. Write a JavaScript loop that calculates the sum of the first n natural numbers.

6. The reciprocal of a number x is denoted by $\frac{1}{x}$. For example, the reciprocal of 5 is $\frac{1}{5}$. Write a program to calculate and display the sum of the reciprocals of the first 10 natural numbers i.e.

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \frac{1}{10}$$

7. Write a program that generates two random numbers (between 1 and 10) and displays their sum. Extend the program so that it does the this three times.

8. Write a program that generates two random numbers (say between 1 and 10) and asks the user to enter their sum. The program should continue until the user enters the correct answer.

9. Write a program to iterate over all the numbers between 1 and 100. Every time it comes across a multiple of 5 it prints 'fizz', for every multiple of 10 it prints 'buzz', and for every other number it just prints the number.

10. Write a program that repeatedly prompts a user to enter a number. If the user enters an even number the program should display *Even*; otherwise, the program should display the message *Odd*. The program should end when the user enters the word 'stop'.

11. Write a program that keeps a count of the number of even numbers entered by a user until the word 'stop' is entered.

12. Write a program that simulates the rolling of a die (Hint: use a random number between 1 and 6.) The program should continue until a six is 'rolled'. The program should display the number of rolls it took to return a 6.

13. Extend the previous program to count the number of throws it takes to get 'snake eyes' Snake eyes mean two ones in a row with one die or double ones if using two dice. Implement with one die first i.e. a single roll at a time. Then two.

14. A Fibonacci sequence is a sequence of numbers where each successive number is the sum of the previous two. Thus, the first 7 numbers in the Fibonacci sequence are 1, 1, 2, 3, 5, 8, 13. Write a program to that computes the nth Fibonacci number. For example, if the user entered 6, the program would output 8 (as 8 is the 6th Fibonacci number)

15. The CAO awards points to students based on their achievements in the Leaving Certificate examination. The table below illustrates the mapping from student marks to CAO points for higher and ordinary level. counting their best six subjects only.

% Bands	Higher		Ordinary	
90 – 100	H1	100	O1	56
80 – 89	H2	88	O2	46
70 – 79	H3	77	O3	37
60 – 69	H4	66	O4	28
50 – 59	H5	56	O5	20
40 – 49	H6	46	O6	12
30 – 39	H7	37	O7	0
0 – 29	H8	0	O8	0

Write a program that asks a user to enter six results. For each result enter a code ('H' to indicate the result is higher level; 'O' for ordinary level) followed by the actual percentage. The program should then determine the relevant points for the percentage entered and keep a running total of the points to date. Once the last result has been entered the program should display the points total accumulated. (Bonus points and six best subjects are ignored.)

Reflect on all the exercises in this section.

Suggest how you could adapt any of the exercises for use in your own LCCS classroom.

9. Strings

Recall that a string is any sequence of characters enclosed in quotation marks. (The quotations can be single or double but must be consistent.)

Strings are very flexible datatypes. They can be used to represent anything from a person's name, or phone number to large amounts of text such as the contents a web page, a newsfeed or even a manual such as this.

The table below contains some example string literals along with a brief description of each.

Example string literal	Brief description
"11 The Laurels, Dublin 24"	Strings can contain a mix of letters and numbers (and any Unicode character)
"0861234567"	A string can be made up of entirely of digits
"+353-(0)86-1234567"	An international phone number
"970-0-393-63499"	A string used to store an ISBN
"https://glitch.com/edit/#!/pdst-wkshp-day2?path=string.js:8:0"	A string used to store a URL
'Game of Thrones'	Strings can be delimited inside single quotes (as well as double quotes
'St. Patrick\'s Day, March 17'	The single quote at the end of *Patrick's* needs to be escaped. Otherwise, JavaScript would see this as the closing quote to mark the end of the string.
"St. Patrick's Day, March 17"	The escape sequence is not needed here because the string is enclosed inside double quotes.
"A string can span \ multiple lines of JS code \ using the \\ character!"	Strings can span several lines of a JavaScript program. This string resolves to: *A string can span multiple lines of JS code using the \ character!*
"A string can span \n multiple lines \n using the \\n character!"	Strings can also contain multiple line. This string resolves to: *A string can span* *multiple lines* *using the \n character!*

String indexing

Individual characters can be accessed using an index (in the same way as an index is used to access the elements of an array). The index of the first character in every string is zero and the index of the last character in a string of length n is $n - 1$.

Let's consider the string `s` declared and initialised as follows:

```
let s = "Hello World!";
```

The diagram below depicts `s` with the index of every character displayed underneath.

Every character in a string can be identified by a position known as an index.

From the diagram it should be evident the expression `s[0]` would return the string "H"; `s[1]` would return "e" and so on until `s[11]` which would return the string "!". Note that negative indices are not supported and a value of `undefined` is returned if the index used is out of range.

Furthermore, it should be noted that strings in JavaScript are *read-only*. This means that once a JavaScript string has been created its value cannot be changed.

Let's say we wanted to change the string `s` from *Hello World!* to *Howdy World*! – we might proceed as follows:

```
s[1] = "o";
s[2] = "w";
s[3] = "d";
s[4] = "y";
console.log(s); // Hello World!
```

Even though the code doesn't result in any errors it doesn't work and the string `s` will remain unchanged.

KEY POINT: Individual string elements can be accessed using the index operation but they cannot be changed (because strings are immutable).

Primitive Strings vs. Strings as Objects

JavaScript strings can be created either as primitive strings or as objects. The code below demonstrates different ways strings can be created.

```
let s1 = "Joe";
let s2 = "Joe";
let s3 = s2;
let s4 = new String("Joe");
let s5 = new String("Joe");
let s6 = s5;
```

In the above code the variables `s1`, `s2` and `s3` are all primitive strings. Their datatype is `string`. Primitive strings are created simply by assigning a string literal to a variable. They can also be created by assigning another (pre-existing) string to a variable (as in the case of `s3` above)

Variables `s4`, `s5` and `s6` are all of type `object`. The strings `s4` and `s5` are both created explicitly as objects by applying the `new` operator to `String` which is a built-in JavaScript object. (Using the terminology of other object-oriented languages such as C++ and Java, `String` can be thought of as a *constructor/wrapper* for the *static/global* class `String`.). `s6` is simply a reference to the string `s5`.

The diagram below provides a visual of how above variables might be represented in memory following their declaration. (The actual representation will depend on the implementation of JavaScript and therefore may vary from browser to browser.)

`s1, s2, s3` are all identical – their value is the string *Joe*.

`s4` and `s5` are two separate string objects.

`s5` and `s6` both refer to the same string object.
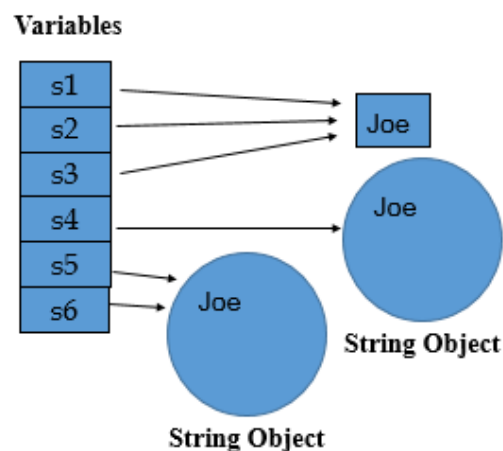
On the previous page we learned that JavaScript strings can be created either as primitive strings or as objects. **One important feature of JavaScript strings however, is that regardless of how they are created, all strings behave as objects.**

This means that the methods and properties defined for built-in string objects can also be used by string primitives.

For example, let's say we had a primitive string `s1` declared as shown in the code below. Because `s1` can also be treated as an object we can access its `length` property as demonstrated in the code.

```
let s1 = "A quick brown fox";
let len = s1.length;
console.log("The length of s1 is", len);
console.log("The last character of s1 is", s1[len-1]);
```

The code results in the following output being displayed on the console.

```
The length of s1 is 17
The last character of s1 is x
```

Similarly, we can call any string method on any string primitive. The name of one such method is `includes`. (We will look at more string methods soon.)

The `includes` method returns `true` if some specified string is included within a string; `false` otherwise. If `s1` is a primitive string with the value *A quick brown fox* we can write:

```
s1.includes("fox") → true because fox is included as part of s1
s1.includes("dog") → false because dog is not included as part of s1
```

Each time a method is called on a primitive string the JavaScript engine converts the string to an object internally. The method is then invoked on this internal string object (and upon completion the internal object ceases to exist).

PROGRAMMER TIP!
Declare strings as primitives as opposed to using the new operartor.

Comparing Strings: Literals vs. Object References

The fact that all strings behave as objects can be a source of confusion especially when it comes to comparing them.

Two strings can be compared for equality using the equality operator (==) or the strict equality operator (===). Tests for equality always return either `true` or `false`.

Care needs to be taken to understand the difference between comparing string literals and string objects. Take for example the following – the output is shown as comments.

```javascript
let s1 = "Joe";
let s2 = "Joe";
let s3 = s2;
let s4 = new String("Joe");
let s5 = new String("Joe");
let s6 = s5;

console.log(s1 == s2); // true
console.log(s1 == s3); // true
console.log(s1 == s4); // true
console.log(s4 == s5); // false
console.log(s5 == s6); // true

console.log(s1 === s2); // true
console.log(s1 === s3); // true
console.log(s1 === s4); // false
console.log(s4 === s5); // false
console.log(s5 === s6); // true
```

Note `s1` and `s4` are equal because they both contain the same value. However, the test for strict equality returns `false` because the underlying datatypes of both variables are different. The datatype of `s1` is `string` and the datatype of `s4` is `object`.

> ***Use the space below to reflect on other results of the comparisons in the above code. For example, why does `s4 === s5` yield `false` and yet `s5 === s6` return `true`?***

Strings can also be compared for inequality by using not-equal-to, less-than and greater than operators. The table illustrates the results of some such string comparisons.

Test (comparison)	Result	Comment
"apple" < "banana"	true	The letter 'a' comes before 'b'. The string "apple" is therefore less than the string "banana" and the test returns true.
"apple" > "banana"	false	The string "apple" is not greater than the string "banana"
"apple" < "aardvark"	false	Since the first letter in both strings is the same (i.e. 'a') the test continues from the second letter
"apple" > "apple tart"	false	Both strings are identical up to the 'e'. The second strings is longer than the first and is therefore considered greater.
"Apple" != "apple"	true	Since the strings are not identical the test for inequality returns true
"Apple" > "apple"	false	The code for upper case 'A' is less than lower case 'a'.
"Zebra" <= "giraffe"	true	The Unicode values for *all* upper case letters are less than the Unicode values for the corresponding lower case letters.

The comparison operations work by comparing the strings on a character-by-character basis in what is called *lexicographical order*.

Experiment!
Use the code below to investigate the method `localeCompare`
(Reference: https://www.w3schools.com/jsref/jsref_localecompare.asp)

```
let apple = "apple";
let banana = "banana";
console.log(apple.localeCompare(banana));
console.log(banana.localeCompare(apple));
console.log(banana.localeCompare(banana));
```

Describe what the method does, how to call it and what its return values are.

_____

_____

_____

_____

_____

String Methods

Because JavaScript automatically treats primitive strings in the same way as if they were String objects, it is possible to call any String object method on a string primitive.

This is illustrated in the program below which demonstrates the use of a selection of string object methods being applied to two primitive strings – `pangram` and `toungeTwister`.

```javascript
// Declare two strings to work with
let pangram = "Pack my box with five dozen liquor jugs";
let toungeTwister = "Sally sells seashells by the sea shore";

// charAt and charCodeAt
console.log("The Unicode representation for", pangram.charAt(), "is", pangram.charCodeAt());
console.log("The Unicode representation for", pangram.charAt(1), "is", pangram.charCodeAt(1));

// toUpperCase, toLowerCase and concat
console.log("toUpperCase:", pangram.toUpperCase());
console.log("toLowerCase:", pangram.toLowerCase());

// concat
let lowerCaseStr = toungeTwister.toLowerCase();
let upperCaseStr = toungeTwister.toUpperCase();
console.log(lowerCaseStr.concat(upperCaseStr));

// indexOf
let index = toungeTwister.indexOf("ells");
console.log("indexOf first \'ells\' is:", index);
console.log("indexOf of next \'ells\' is:",  toungeTwister.indexOf("ells", index+1));

// lastIndexOf
let lastIndex = toungeTwister.lastIndexOf("ells");
console.log("lastIndexOf \'ells\' is:", lastIndex);
console.log("2nd lastIndexOf of \'ells\' is:", toungeTwister.lastIndexOf("ells",lastIndex-1));

// slice
console.log("slice 1:", pangram.slice(5));
console.log("slice 2:", pangram.slice(5, 11));

// replace
console.log("replace 1:", pangram.replace("box", "bag"));
console.log("replace 2:", toungeTwister.replace("sells", "sold"));

// split
console.log("split 1:", pangram.split());
console.log("split 2:", toungeTwister.split(" "));
```

The output generated by this program is shown on the next page.

KEY POINT: All JavaScript strings behave as objects.

```
The Unicode representation for P is 80
The Unicode representation for a is 97
toUpperCase: PACK MY BOX WITH FIVE DOZEN LIQUOR JUGS
toLowerCase: pack my box with five dozen liquor jugs
sally sells seashells by the sea shoreSALLY SELLS SEASHELLS BY THE SEA SHORE
indexOf first 'ells' is: 7
indexOf of next 'ells' is: 17
lastIndexOf 'ells' is: 17
2nd lastIndexOf of 'ells' is: 7
slice 1: my box with five dozen liquor jugs
slice 2: my box
replace 1: Pack my bag with five dozen liquor jugs
replace 2: Sally sold seashells by the sea shore
split 1: ▶ ["Pack my box with five dozen liquor jugs"]
split 2: ▶ (7) ["Sally", "sells", "seashells", "by", "the", "sea", "shore"]
```

It should be noted that while certain methods e.g. `concat`, `toLowerCase`, `toUpperCase`, `trim` etc. appear to change the value of the string on which they act, this is in fact not the case. What actually happens is that such methods actually create and return a new string leaving the original string unchanged. Of course, the reason for this is because strings are immutable.

A brief description of each of the methods used in this example is provided on the next page.

Use the space below to record your understanding of string methods

Method name	Description
`strA.concat(strB)`	Returns a new string made up of the characters of `strA` followed by the characters of `strB`.
`str.charAt(index)`	Returns a new string made up of the character at the specified `index` in `str` (or an empty string if `index` is out of bounds)
`str.charCodeAt(index)`	Returns the Unicode code of the character at the specified index in `str` (or `NaN` if `index` is out of bounds)
`str.toUpperCase()`	Returns a new string with all the characters of `str` converted to upper case
`str.toLowerCase()`	Returns a new string with all the characters of `str` converted to lower case
`str.indexOf(item [,fromIndex])`	Returns the index of the first occurrence of the value specified by `item` in `str`. Unless `fromIndex` is specified the search starts at index zero. If `item` is not found the method returns `-1`
`str.lastIndexOf(item [,fromIndex])`	Starting from the end (or at `fromIndex`) and working backwards, this method returns the index of the first occurrence of the value specified by `item` in `str`. If `item` is not found the method returns `-1`
`str.slice([i1, [i2])`	Returns a new string made up of the characters of `str` from `i1` up to but not including `i2`. If `i1` is not specified it is taken to be zero; if `i2` is not specified it is taken to be `str.length`. The contents of the original string are unchanged.
`str.replace(old, new)`	Replaces all occurrences of `old` in `str` with `new`.
`str.split([separator])`	Returns an array of strings split at the point denoted by the `separator`
`str.trim()`	Creates a new string based on `str` with leading and trailing whitespaces removed. Note `trimStart()` removes only leading whitespaces and `trimEnd()` removes only trailing whitespaces

Browse to https://www.w3schools.com/jsref/jsref_obj_string.asp for a more complete reference to the JavaScript String object.

Traversing Strings

Example 1. The following program counts and displays the number of vowels in a string entered by the user. The program uses a `for` loop to traverse every character in the string. This operation is called **string traversal**.

```
let inString = prompt("Enter a string:");
let vowels = 0;
let ch;

for(let i = 0; i < inString.length; i ++) {

  // Extract the next character (from position i) ...
  // ... and convert it to upper case
  ch = inString.charAt(i).toUpperCase();

  if (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U')
    vowels ++;
}

console.log("The number of vowels found was", vowels);
```

Key in the program, try it out and answer the following questions

What is the purpose of the `for` loop?

Why is the method `toUpperCase` used?

Without using the method `charAt` what other technique could have been used to access the individual characters of `inString`?

Challenges!
Based on the above, write programs to ….

- count and display the number of non-vowel characters in a string
- count and display the number of consonants characters in a string
- count and display the number of upper-case characters in a string
- count and display the number of words in a string
- calculate and display the average word length of the words in a string

<u>Example 2.</u> The following program prompts the user to enter a string and then displays the string in reverse order. Each character is displayed on a separate line.

``let str = prompt("Enter a string:");

for(let i = str.length-1; i >= 0; i--)
{
  console.log(str[i]);
}``` | If the user entered *Joe* the output would be:

e
o
J |

Compare the `for` loop in this program to the `for` loop in the example on the previous page.

Look at the initial value for `i` in both programs. What's the difference?

_____

_____

Look at the loop guard (the terminating condition) in both programs. What's the difference?

_____

_____

Explain why the loop step is `i--` in this program but `i++` in the previous example?

_____

_____

Explain the purpose of the variable outStr in the code below. What does the program do?

```
let inStr = prompt("Enter a string:");
let outStr = "";

for(let i = inStr.length-1; i >= 0; i--) {
  outStr += inStr[i];
}

console.log(outStr);
```

_____

_____

_____

Example 3. This example program prompts the user to enter a string and then displays a message stating whether the string is a palindrome or not.

A **palindrome** is a word or phrase that read the same in both directions. Examples of single word palindromes are NAVAN, MADAM, RACECAR and EYE.

```javascript
let s = prompt("Enter a string:");
let isPalindrom = true;

// Traverse the string comparing each char
for (let i=0, j = s.length-1; i < s.length; i++, j--) {
  if (s[i] != s[j]) {
    isPalindrom = false;
    break;
  }
}
// Display the result
if (isPalindrom)
  console.log(s, "is a palindrome!");
else
  console.log(s, "is not a palindrome!");
```

Key in the program, try it out and answer the following questions

Explain how the program works?

_____

_____

_____

_____

_____

What is the purpose of the variable `isPalindrome`? If the initial value of this variable was set to `false` (on line 2) what changes would need to be made to maintain the correctness of the program?

_____

_____

Does the program work for every palindrome? Test it with the following and identify any issues.
Navan? RACEcar? MADAM I'M ADAM? DON'T NOD? never odd or even? Murder for a jar of red rum?

_____

_____

_____

_____

Can you design and develop 'fixes' to any of the 'bugs' you identified?

_____

JS Programming Exercises - Strings

1. State whether a string would be appropriate for the following types of data:

    a) The eircode of your school

    b) Your date of birth

    c) The price of a product

    d) A product code

    e) A PPSN

    f) Your most recent social media post

2. Write a program that prompts a user to input their first name (e.g. *Joe*) followed by their surname (e.g. *Blogs*) and then print a message along the lines:
   Hello Joe Blogs. Welcome to my crazy world!

3. Answer the following questions in relation to the string *JavaScript*

    a) What is the length of the string?

    b) Which character occurs at the zeroth index position?

    c) What character occurs at index position four?

    d) At what index position does the character 't' occur?

    e) What would be an appropriate variable name be to store this string?

4. What output does the following program display?

```
let uprCaseLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
let lwrCaseLetters = "abcdefghijklmnopqrstuvwxyz"
let letters = uprCaseLetters+lwrCaseLetters

console.log(uprCaseLetters.toLowerCase());
console.log(lwrCaseLetters.toLowerCase());

console.log(uprCaseLetters.slice(0,5));
console.log(uprCaseLetters.slice(20));
console.log(lwrCaseLetters.slice(1,6));

console.log(letters.slice(26,52));
console.log(letters.indexOf("a"));
console.log(letters.lastIndexOf("A"));
console.log(letters.replace("abc", "123"));

console.log(uprCaseLetters == lwrCaseLetters);
console.log(uprCaseLetters == lwrCaseLetters.toUpperCase());
```

5. Write a small program to extract (slice) the following strings from the string:
   The quick brown fox jumps over the lazy dog

   a) "quick"
   b) "fox"
   c) "The"
   d) "The quick brown fox"
   e) "jumps over the lazy dog"

6. Write a program that asks a user to input their first name (e.g. *Arnold*) followed by their surname (e.g. *Schwarzenegger*) and then outputs the initial of the forename followed by the first seven characters of the surname (e.g. *ASchwarz*).
   Test your program using *Joe Blogs* as the name.

7. An acronym is a series of letters used as an abbreviation for some phrase or name. It is usually formed by combining the initial letter of each word in the phrase name. Examples include TLA (Three Letter Acronym), IBM (International Business Machines) and LOL (Laugh Out Loud). Write a program that generates an acronym from a phrase entered by the end-user.

8. Write a program that encodes English language sentences into 'Pigs Latin'. Many variations of 'Pigs Latin' exist – attempt to implement the following two:
   a) Insert 'eg' at the end of every vowel in every word of the sentence so that an input of '*She sat under the table*' would become '*Sheeg saegt uegndeegr theeg taegble*'
   b) Move the first letter of every word in the input sentence to the end of that word and the add on 'ay'. In this way an input of '*He switched on the computer*' would become '*ehay witchedsay noay hetay omputercay*'

Reflect on all the exercises in this section.

Suggest how you could adapt any of the exercises for use in your own LCCS classroom.

10. Arrays

An array is a collection of zero or more values that can be accessed using a single variable. Each individual value in an array is called an *element* and each element exists at a particular position known as an *index*. An array index is a zero-based positional offset that can be used to address the individual elements of an array.

KEY POINT: Arrays are JavaScript objects. As such they have associated properties (e.g. `length`) and methods (e.g. `concat`).

Arrays are useful because they provide us with a means of grouping multiple values into a single variable. Without arrays we would need a separate variable for each value.

Let's say we wanted to store the ages of six students. We could use six variables (e.g. `age1`, `age2` etc.) or we could simply declare an array called `ages` as follows:

```
let ages = [18, 16, 18, 17, 19, 17];
```

This tells the JavaScript engine to allocate space for six integers and store them together under the name `ages`. The memory representation for `ages` is depicted below. Notice that the index of each element is shown directly below the element itself.
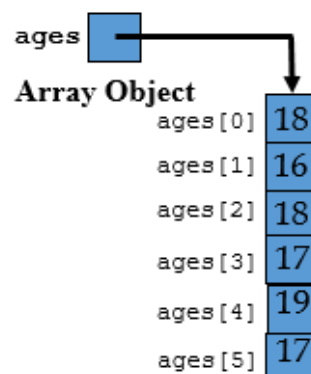

An array object called `ages` with six elements

Individual elements of an array can be referenced using the array name followed immediately by an index enclosed in square brackets.

For example, `ages[0]` refers to the first element of the array, `ages[1]` the second and so on. The last element of our example array is referenced using `ages[5]`. This is depicted here to the right.

While it is both important and necessary to understand the syntax for creating and manipulating arrays the real art of programming involves recognising situations that require the use of arrays in a program.

Take for example the following program which simulates one million dice rolls and keeps count of the number of times each side of a die appears.

```javascript
// This sample program motivates a use for arrays

// Let's say we wanted to count the number of times each side of a dice is rolled
// The program uses six separate variables to store the counts ...
// ... 'ones' stores the number of times 1 is rolled, ...
// ... 'twos' stores the number of times 2 is rolled etc.
// These variables are declared and initialised as follows

let ones = 0, twos = 0, threes = 0, fours = 0, fives = 0, sixes = 0;

// Simulate a million rolls
for (let i = 1; i <= 1000000; i++ ) {
  // Generate a random number between 1 and 6
  let roll = Math.floor(Math.random() * 6) + 1

  if (roll == 1)
    ones++;
  else if (roll == 2)
    twos++;
  else if (roll == 3)
    threes++;
  else if (roll == 4)
    fours++;
  else if (roll == 5)
    fives++;
  else if (roll == 6)
    sixes++;
} // end for

// Display the frequencies on the console
console.log("Ones:\t%d", ones);
console.log("Twos:\t%d", twos);
console.log("Threes:\t%d", threes);
console.log("Fours:\t%d", fours);
console.log("Fives:\t%d", fives);
console.log("Sixes:\t%d", sixes);
```

The program simulates a roll by generating a random number between 1 and 6 and repeats this process a million times.

A separate variable is used to keep track of each side of the die i.e. `ones` stores the number of times a 1 is rolled, `twos` stores the number of times a 2 is rolled and so on.

The solution is considered to be awkward because of the number of variables needed. (Imagine we wanted to track how often each number 'came up' in the National Lottery. We would need 47 separate variables!)

A better solution would be to use arrays as shown in the following listing.

Code	Sample Output

<pre>// Declare an array called 'counts'
let counts = [0,0,0,0,0,0,0];
let roll;
 	
for (let i = 1; i <= 1000000; i++) {
  // Generate a random number between 1 and 6
  roll = Math.floor(Math.random() * 6) + 1;
  counts[roll]++; // <-- This is the MAGIC!!
} // end for
 	
// Display the output
console.log("Face\tFrequency\n");
for (let i = 1; i <= 6; i++)
  console.log("%d\t%d\n", i, counts[i]);</pre>
	Face Frequency
	1 166605
	2 166853
	3 166752
	4 166568
	5 166083
	6 167139

KEY POINT: Arrays are the data structure of choice to represent groups of related values in a single variable.

In this example the array `counts` is used to store a list of frequencies. Arrays could also be used to store lists of other types of numbers (e.g. ages, salaries, sales figures, heights etc.), or names, phone numbers, books, days of the week, months of the year, dates and so on. Virtually any group of objects you can think of can be represented using arrays.

List five 'real world things' that could be represented using arrays.
(e.g. can you think of any list of items stored on your phone, or in a game?)

1. _____

2. _____

3. _____

4. _____

5. _____

Example Program – A Sentence Generator

The Python program shown below makes use of four different arrays - `article`, `noun`, `verb` and `preposition` – to generate a random sentence.

The sentence is created by selecting a word at random from each array in the order:

        article, noun, verb, preposition, article, noun.

The program concatenates the words (separated by spaces) to form the final sentence.

```python
import random

articles = ['the', 'a', 'one', 'some', 'any']
nouns = ['boy', 'girl', 'dog', 'town', 'car']
verbs = ['drove', 'jumped', 'ran', 'walked', 'skipped']
prepositions = ['to', 'from', 'over', 'under', 'on']

sizeOfLists = len(articles)-1

wordIndex = random.randint(0, sizeOfLists)
word1 = articles[wordIndex]

wordIndex = random.randint(0, sizeOfLists)
word2 = nouns[wordIndex]

wordIndex = random.randint(0, sizeOfLists)
word3 = verbs[wordIndex]

wordIndex = random.randint(0, sizeOfLists)
word4 = prepositions[wordIndex]

wordIndex = random.randint(0, sizeOfLists)
word5 = articles[wordIndex]

wordIndex = random.randint(0, sizeOfLists)
word6 = nouns[wordIndex]

sentence = word1+' '+word2+' '+word3+' '+word4+' '+word5+' '+word6
print(sentence)
```

Sample output (sometimes the sentences generated make no sense!)

- a dog ran under a girl
- any car ran to some boy
- a town ran under any town
- some dog jumped on a car

Challenge!
Translate the above Python program to JavaScript.

.

Changing the contents of an array

Arrays are both mutable and dynamic – this means that existing elements can be changed and new elements can be added. Array elements can also be destroyed using the `delete` operator. Consider the example shown:

```
let items = ["Bread", "Milk", "Tea"];
items[0] = "Sliced Pan"; // modify an element
delete items[2]; // delete an element
items[3] = "Butter"; // add a new element
items[4] = "Jam"; // add another new element

console.log(items); // display the array contents
```

When the above code is run the following output is displayed on the console:

▶ (5) ["Sliced Pan", "Milk", empty, "Butter", "Jam"]

Notes:

✓ The first element of `items` has been changed from *Bread* to *Sliced Pan*

✓ The third element (i.e. element at index position 2) has been removed from the array.

✓ Two new elements have been added to the end of the `items` array (*Butter* and *Jam*)

✓ The original array had a length of 3. After the code has been run the length of the array is 5.

When an element is deleted from an array the array size does not change. Rather the value is simply removed from the array but the 'slot' is still part of the array. In the above example the value *Tea* is removed from `items` – memory for `items[2]` is cleared but remains allocated. Arrays such as this that contain empty slots are called *sparse arrays*.

KEY POINT: The length of any array is the number of elements it contains.
For any array `a`, the expression `a[a.length-1]` always returns the last element.

Array length

You can find out how many elements are in an array by using the array's `length` property[10]. The dot operator is used to access the `length` property (just as it is used to access all object properties).

For example, let's say we wanted to find the length of the array `items` declared as follows:

	This array has three elements:
`let items = ["Bread", "Milk", "Tea"];`	`items[0]` → *Bread*
	`items[1]` → *Milk*
	`items[2]` →*Tea*

The following line displays the length of `items` on the console.

```
console.log(items.length); // displays 3
```

By this stage it should be evident that the index of the last element in an array is always the length of the array minus 1. This is to compensate for the fact that the index of the first element is zero. Thus, the expression `items[items.length-1]` would return the string *Tea* in this example.

KEY POINTS:
- ✓ The first element in every array has an index of zero.
- ✓ The last element in every array, `a` has an index of, `a.length` – 1.
- ✓ Therefore valid indices range from 0 up to array length minus 1

Experiment!
Devise a situation to test what would happen when you try to access an array using an index that is out of range.

---

[10] In JavaScript, all arrays are treated as objects and as such they have associated properties and methods.

Array Methods[11]

Like all objects, arrays have methods - here's a list of some of the more common ones.

Method name	Description
`arrA.concat(arrB)`	Returns a new array made up of the elements of `arrA` followed by the elements of `arrB`.
`arrA.indexOf(item)`	Returns the index of the first occurrence of the value specified by `item` in `arrA`. If the item is not found the method returns `-1`
`arrA.lastIndexOf(item)`	Starting from the end, returns the index of the first occurrence of the value specified by `item` in `arrA`. If `item` is not found the method returns `-1`
`arrA.join([separator])`	Returns all the elements of the array joined together as a string. The default value of the optional separator is a comma.
`arrA.push(items)`	Appends one or more elements (as specified by `items`) to the end of `arrA` and returns the new length of the array.
`arrA.pop()`	Removes the last element of `arrA`. Returns the element removed or `undefined` if the array was empty
`arrA.shift()`	Removes the first element of `arrA`. Returns the element removed or `undefined` if the array was empty
`arrA.unshift(items)`	Inserts one or more elements (as specified by items) to the start of `arrA` and returns the new length of the array.
`arrA.sort()`	Sorts the elements of array in place and returns the sorted array (in alphabetical order)
`arrA.reverse()`	Sorts the elements of array in place and returns the sorted array (in alphabetical order)
`arrA.slice([i1], [i2])`	Returns a new array made up of the elements of `arrA` from `i1` up to but not including `i2`. If `i1` is not specified it is taken to be zero; if `i2` is not specified it is taken to be `arrA.length`. The contents of the original array are unchanged.
`arrA.splice(i, [n, [items]])`	Adds/replaces/remove elements from an array *in place*. `i` is the starting index, `n` is the number of elements to remove and `items` are the new elements. Returns a new array with any removed elements. (If no elements are removed an empty array is returned.)

---

[11] Browse to either of these sites for a more complete reference to array methods:
https://www.w3schools.com/jsref/jsref_obj_array.asp
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

The code below demonstrates the use of some of the methods just described.

push **and** *pop*

Elements can be added to and removed from an array using the methods `push` and `pop`.

- ✓ `push` adds one or more elements to the end of an array. It increases the length of the array by the number of values that are added.

- ✓ `pop` removes the last element from an array. It reduces the length of an array by 1.

Try the following:

```
let items = ["Bread", "Milk", "Tea"];
items.push("Butter", "Jam"); // add Butter and Jam
items.pop(); // remove Jam

console.log(items); // display the array
```

The resulting output is:

`["Bread", "Milk", "Tea", "Butter"]`

Explain why 'Jam' is not included in the output of the array `items` shown above.
How might 'Tea' have been removed before adding 'Butter' and 'Jam'?

KEY POINT: Methods are invoked using the dot operator. To invoke a method `m` on an array `a` we write `a.m()`

TEACHER TIP
A common student misconception is that a method
can only be invoked once (on each object or in total).

shift **and** *unshift*

Two other array methods that are closely related to `push` and `pop` are `shift` and `unshift`.

```
let items = ["Bread", "Milk", "Tea"];

items.shift(); // ??
console.log(items); // display the array

items.unshift(); // ??
console.log(items); // display the array
```

Experiment! Key in and run the code above. Based on the output describe what the two methods ***shift*** *and* ***unshift*** *do?*

slice, concat, sort, reverse **and** *join*

These methods are demonstrated in the short program.

```
let weekDays = ['Mon', 'Tue', 'Wed', 'Thur', 'Fri'];
let weekendDays = ['Sat', 'Sun'];

console.log(weekDays.slice(2));
console.log(weekDays.slice(1,3));
console.log(weekDays.concat(weekendDays););
console.log(weekendDays.concat(weekDays););
console.log(weekDays.sort());
console.log(weekendDays.reverse());
console.log(weekendDays.join(' and '));
```

The output generated by each of the `console.log` statements is shown below. You should read through the code carefully and try to understand how this output is arrived at.

```
["Wed","Thur","Fri"]
["Tue", "Wed"]
["Mon","Tue","Wed","Thur","Fri","Sat","Sun"]
["Sat","Sun","Mon","Tue","Wed","Thur","Fri"]
["Fri", "Mon", "Thur", "Tue", "Wed"]
["Sun", "Sat"]
Sat and Sun
```

Predict the output generated by the code shown below.

_____

_____

_____

_____

_____

_____

_____

```javascript
let weekDays = ['Mon', 'Tue', 'Wed', 'Thur', 'Fri'];
let weekendDays = ['Sat', 'Sun'];

let daysOfWeek = weekendDays.concat(weekDays);

console.log(daysOfWeek.join());
console.log(daysOfWeek.slice(2));
console.log(daysOfWeek.slice(2,5));

console.log(weekDays.concat("Weekend"));
console.log(weekendDays.sort());
console.log(weekDays.reverse());
```

splice

This handy method can be used to insert/remove/replace elements from the array it acts on.

For example, let's say we had an array called `weekDays` with the values:

`['Mon', 'Wed', 'Thur']`

`weekDays.splice(1, 0, "Tuesday");`

This line inserts 'Tuesday' at index position 1. The use of zero as the second argument tells JavaScript not to remove any elements. All elements to the right of position 1 (including 'Wed') are pushed down the array before the insertion. This results in weekDays being changed to the following: `['Mon', 'Tuesday' 'Wed', 'Thur']`.

`weekDays.splice(1, 2);`

This line tells JavaScript to remove two elements from the array called `weekDays` starting at index position 1. The array is changed in place to become `['Mon']` and a new array containing the deleted elements is returned i.e. `['Wed', 'Thur']`.

Array Processing (traversing arrays)

Consider the code shown below.

```
let daysInMonths = [31,28,31,30,31,30,31,31,30,31,30,31];

for (let i=0; i < daysInMonths.length; i++)
  console.log("Month %d has %d days",  (i+1), daysInMonths[i]);
```

The code displays the output shown here to the right.

The `for` loop iterates over every element in the array using an index variable `i`.

The index is initialised to zero and is incremented on each loop iteration. The loop continues as long as `i` is less than 12 which is the length of the array. This is fine since the index of the last element in the array is 11.

```
Month 1 has 31 days
Month 2 has 28 days
Month 3 has 31 days
Month 4 has 30 days
Month 5 has 31 days
Month 6 has 30 days
Month 7 has 31 days
Month 8 has 31 days
Month 9 has 30 days
Month 10 has 31 days
Month 11 has 30 days
Month 12 has 31 days
```

The loop body displays a single line of output on each iteration.

Because the program visits each element of the array once this type of program is known as an *array traversal*. Array traversals are quite a common pattern in programming. Here are some more examples.

KEY POINT: An array traversal is a programming pattern that involves iterating over each element in an array.

1. This program traverses two arrays simultaneously

```
let daysInMonths = [31,28,31,30,31,30,31,31,30,31,30,31];
let months = ["Jan","Feb","Mar","Apr","May", "June","July","Aug","Sept","Oct","Nov","Dec"];

for (let i=0; i < daysInMonths.length; i++)
  console.log("%s has %d days\n",  months[i], daysInMonths[i]);
```

Predict the output of the code.

_____

_____

_____

2. This program computes the arithmetic mean of the elements of an array

```
let ages = [18, 16, 18, 17, 19, 17];
let total = 0;

for (let i=0; i < ages.length; i++)
  total = total + ages[i];

console.log("The mean age is %d", total/ages.length);
```

How could the assignment `total = total + ages[i];` *be written more succinctly?*

Is the calculation correct in your opinion?

3. This program traverses an array to find the maximum value it contains

```
let ages = [18, 16, 18, 17, 19, 17];
let max = 0;

for (let i=0; i < ages.length; i++) {
  if (ages[i] > max)
    max = ages[i];
}

console.log("The maximum age is %d",  max);
```

Explain how the above program works.

Are the curly braces necessary? Explain.

Copying arrays and array references

Let us say we declared and initialised two integer arrays – `arr1` and `arr2` - as follows.

```
let arr1 = [1, 3, 5, 7, 9];
let arr2 = [];
```

It may be useful to think of the two arrays looking like the following in memory. The array of odd numbers shown on the left is `arr1`, and the empty array shown on the right is `arr2`.
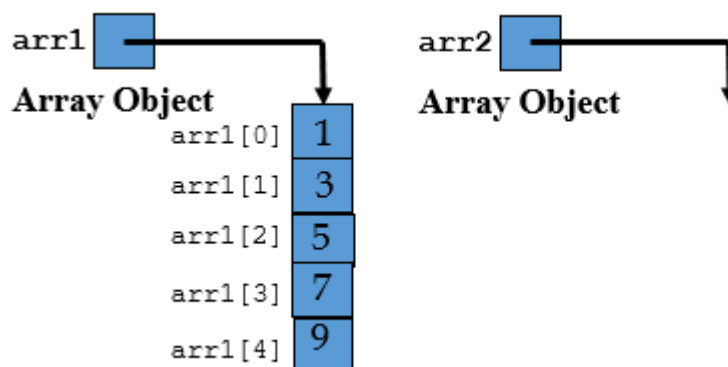
The code below copies the contents of `arr1` into `arr2`.

```
for (let index = 0; index < arr1.length; index++) {
    arr2[index] = arr1[index];
}
```

Each element of the `arr1` is copied one-by-one into the corresponding position to `arr2`. The resulting arrays are depicted below:
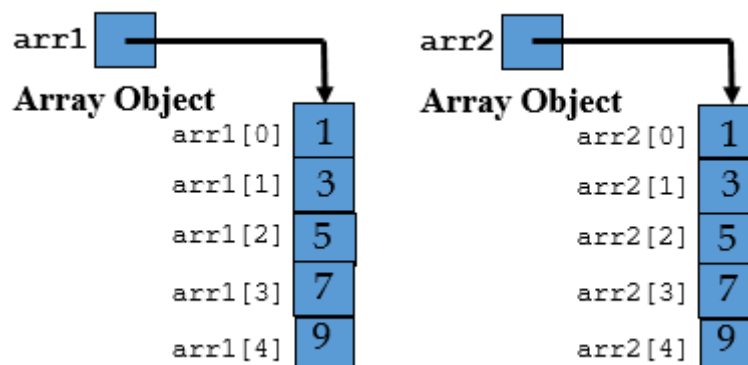
It is important to note from the previous example that `arr2` is a *copy* of `arr1`. The two arrays are distinct data structures and as such their contents can be changed independently of one another.

This can be contrasted with the situation highlighted on Line 2 in the code below in which a *reference* of `arr1` is assigned to the variable `arr2`. The output is shown on the right.


```
let arr1 = [1, 3, 5, 7, 9];
let arr2 = arr1;

console.log("BEFORE TRAVERSAL");
console.log("Array 1:", arr1);
console.log("Array 2:", arr2);

for (let i=0; i < arr1.length; i++) {
  arr1[i]++;
}

console.log("AFTER TRAVERSAL");
console.log("Array 1:", arr1);
console.log("Array 2:", arr2);
```

BEFORE TRAVERSAL
Array 1: ▶ (5) [1, 3, 5, 7, 9]
Array 2: ▶ (5) [1, 3, 5, 7, 9]
AFTER TRAVERSAL
Array 1: ▶ (5) [2, 4, 6, 8, 10]
Array 2: ▶ (5) [2, 4, 6, 8, 10]

Code   *Output*

The effect of Line 2 is depicted in the graphic below.
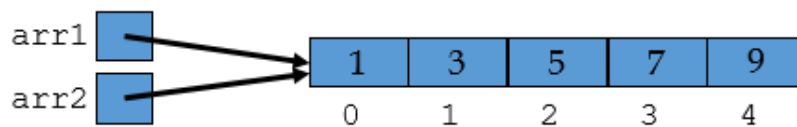
arr1 and arr2 both refer to the same array

Any changes that are made to `arr1` are also made to `arr2`. The `for` loop traverses over `arr1` incrementing every element along the way. The state of the two array objects after the `for` loop is depicted below.
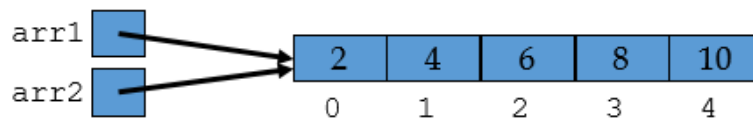
arr2 is a reference to arr1 and vice versa

Explain why the output of the above program would remain the same if the loop traversed `arr2` (as opposed to `arr1`).

Passing arrays into functions

When an array is passed as an argument to a function, it is the reference to the original array that is passed, not a copy of the array. Therefore, any changes that are made to such arrays inside the function survive after the function terminates.

This is demonstrated in the code below which defines a function called `incrementValues`. This function adds 1 to each element of the array passed into it. The function is called on the second last line of the code listing.

```javascript
function incrementValues(arrayParam) {
  console.log("incrementValues() called");

  for (let i=0; i < arrayParam.length; i++) {
    arrayParam[i]++;
  }

  console.log("incrementValues() ends");
} // end incrementValues()

let argArray = [1, 3, 5, 7, 9];
console.log("Array before call:", argArray);
incrementValues(argArray);
console.log("Array after call:", argArray);
```

The code results in the following output.

```
Array before call: ▶ (5) [1, 3, 5, 7, 9]
incrementValues() called
incrementValues() ends
Array after call: ▶ (5) [2, 4, 6, 8, 10]
```

As can be seen the value of `argArray` has been changed by the function.

KEY POINT: Arrays are passed by reference to functions.

PROGRAMMER TIP!
One way of avoiding the need for global variables is to wrap them as array objects

Programming Exercises - Arrays

1 Write a line of code to initialise an array as follows:

```
['spring', 'summer', 'autumn', 'winter']
```

2 Write a line of code to initialise an array with the days of the week (i.e. 'Sunday' through to 'Saturday'). Call your array `weekDays`. Now write code to display the elements:

a) `weekdays[0]`

b) `weekdays[5]`

c) `weekdays[weekdays.length-1]`

4. Write a line of code to initialise an array with the names of the twelve months of the year. Write a second line of code to initialise an array with the number of days in each month (assume 28 days for February)

Now implement the following:

- prompt the user to enter the name of a month.

- look up the index of the month entered from the `months` array.

- access the element at this index from the `days` array.

- display the number of days in the month.

For example, if the user entered *March* the program should display the output message *March has 31 days*.

5. Write a program to generate and store 100 randomly generated integers between 1 and 10 in an array.

6. Write a program to read in five separate values from an end user and store them in an array.

7. Write a program to find the first 10 prime numbers and store them in an array.

8. Write a program that adds the contents of two arrays and store the results in a third array. For example, if the two input arrays were initialised as follows:

```
let arrOfEvens = [2, 4, 6, 8, 10]
let arrOfOdds = [1, 3, 5, 7, 9]
```

The output array would contain:

```
[3, 7, 11, 15, 19]
```

9. Write a program to find the largest value in the array shown below.

```
[18, 23, 16, 18, 23, 21, 15, 16, 23, 21]
```

Modify your program so that it displays the number of times the maximum value occurs.

10. Write a program to find the arithmetic mean, median and mode of the array shown below.

```
[18, 23, 16, 18, 23, 21, 15, 16, 23, 21]
```

11. Given an array `daysOfWeek` declared as follows:

```
let daysOfWeek = ['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat'];
```

a) write a line of code to remove the first and last elements using shift and pop respectively. The array should end up looking like this:

```
['Mon', 'Tue', 'Wed', 'Thur', 'Fri'];
```

b) write a line of code to add the elements 'Sat' and 'Sun' to the end. (*Hint*: use `push`.)

12. Given an array `suits` declared as follows:

```
let suits = ['Hearts', 'Diamonds', 'Spades'];
```

Use the splice method to insert the value 'Clubs' between 'Diamonds' and 'Spades' so that the array is changed to: `['Hearts', 'Diamonds', 'Clubs', 'Spades'];`

13. The Python program shown here simulates the selection of a random card from a deck of cards. (i.e. pick a card, any card).

```python
1  # Program to pick a card from a pack
2  import random
3
4  suits=['Hearts','Diamonds','Spades','Clubs']
5  faces=['A','1','2','3','4','5','6','7','8','9','J','Q','K',]
6
7  # pick a random number between 0 and 3 (inclusive)
8  r1=random.randint(0,3)
9  suit=suits[r1]
10
11 # pick a random number between 0 and 12 (inclusive)
12 r2=random.randint(0,12)
13 face=faces[r2]
14
15 card=face+' of '+suit
16 print(card)
```

   a) Implement the program in JavaScript.
   b) Extend your program to deal five cards. (*Hint:* you will need to build up a new array – call it `hand` – and each time a random card is generated append it to `hand`.)
   c) Test your program to make sure that the same card cannot be dealt more than once.

14. Design and write a JavaScript program to determine whether or not a sentence entered by the user is a pangram. A *pangram* is a sentence that uses every letter of the alphabet at least once.

   Hint: You could use an array to represent the frequency of each letter i.e. the number of a's, b's etc.? If, after processing the sentence, there is no letter with a frequency of zero the sentence must be a pangram.

TEACHER TIP
Explore http://www.fun-with-words.com/index.html - a website dedicated to amusing quirks, peculiarities, and oddities of the English language - over 500 pages of word puzzles, games, amazing lists, and fun facts.

Reflect on all the exercises in this section.

Suggest how you could adapt any of the exercises for use in your own LCCS classroom.

11. Functions

A function is a group of statements designed to carry out a specific task. Functions are the building blocks of programs. They are important because they enable programmers to store useful functionality which can be invoked in a single line of code – the function call.

Function Syntax – defining and calling functions

A function needs only to be defined once, but it can be called multiple times. Each time a function is called the statements that make up the function are executed. This means that functions can save programmers from having to repeat the same lines of code every time they need a specific task carried out. Therefore, functions can be used to avoid duplication of code.

The general syntax for defining a new function is as follows:

```
function <function-name>([parameters]){

    statement(s)

}
```

> The statement(s) make up the function body. They are run when the function is called.

The first line in the function definition is important because it contains the necessary information required by other programmers to use the function. It is referred to as the *function signature*. The signature comprises the name of the function and an optional list of parameters that can be passed into the function when it is called.

The name of the function is chosen by the programmer. The rules for naming functions are the same as those for naming variables.

KEY POINT: A function is a short piece of re-usable code that carries out a specific task when it is called.

Once a function has been defined it just takes one line of code to use it. This is the function call. The general form of a function call look like this.

```
function([arguments]);
```

> A function can be called by simply using its name followed by opening and closing parentheses.

The beauty of functions lies in their ease of use. Let's look at some examples.

Example 1

The code below draws a square box on the console. The output is shown to the right.

```
console.log("+------+");
console.log("|      |");
console.log("|      |");
console.log("+------+");
```

```
+------+
|      |
|      |
+------+
```

Now let's say we wanted our program to draw two square boxes – we would have to duplicate the code as follows.

```
console.log("+------+");
console.log("|      |");
console.log("|      |");
console.log("+------+");
console.log("+------+");
console.log("|      |");
console.log("|      |");
console.log("+------+");
```

```
+------+
|      |
|      |
+------+
+------+
|      |
|      |
+------+
```

Every time we want to draw a square box we need to duplicate the code – this can get messy. A better solution is to package up the code into a function as shown below.

```
function drawBox() {
  console.log("+------+");
  console.log("|      |");
  console.log("|      |");
  console.log("+------+");
}
```

The code to draw the box is packaged into a function called drawBox

KEY POINT: Functions provide a means for programmers to package up and store functionality for use in other places in the program.

PROGRAMMER TIP
The first line of a function definition is referred to as the function *signature* or *prototype*.

When the program in the last example is run you will notice that nothing appears to happen. In particular, the box is no longer displayed. This is because the function hasn't been called – yet!

To call the function we just need a single statement - `drawBox();`

```
function drawBox() {
  console.log("+------+");
  console.log("|      |");
  console.log("|      |");
  console.log("+------+");
}

drawBox();
```

When the function is called JavaScript executes the statements in the function body i.e. it draws a box

The semantics of a function call are explained as follows. When a function is called, the flow of control jumps to the first line of the function and execution continues from that point to the last line of the function. As soon as the last line of the function has been executed the flow of control jumps back to the point from which the call to the function was initially made.

Once a function has been defined it can be called *anywhere* in the code. (In JavaScript a function call can appear in the code before its definition.)

```
drawBox();
drawBox();
drawBox();
drawBox();
drawBox();
```

The code shown here calls the function drawBox five times. The result is five boxes are displayed underneath one another on the console. Each box is drawn by using a single line of code: drawBox();

KEY POINT: A *function* call causes the code inside the function body to be executed.

PROGRAMMER TIP
Functions make it possible to develop solutions piece-by-piece. Large scale software systems are developed by breaking big problems down into smaller problems. Each function is written to do a specific task. Such systems are said to be *modular*

Example 2

The example below shows a definition of a function called `displayRhyme` - the program output is displayed on the right.

```javascript
function displayRhyme() {
  console.log("Jack loves to do his homework");
  console.log("He never misses a day");
  console.log("He even loves the men in white");
  console.log("Who are taking him away");
}

displayRhyme(); // call the function displayRhyme
console.log(""); // display a blank line
displayRhyme();// call the function displayRhyme
```

```
Jack loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Jack loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

Notice that the rhyme appears *twice* in the output. This is because the function that displays the rhyme is called *twice* in the code.

When the function is called for the first time the flow of control is switched to the function and the code is executed in sequence until the last line of the function body is reached. At this point JavaScript returns to the point in the code where the call to the function was made. The next line to be executed is `console.log("");` - once this line has been executed, JavaScript executes the last line of the program which is the second call to `displayRhyme`.

Notice in the above code that the same name i.e. *Jack* is always displayed in our rhyme. This is because *Jack* is hard-coded into the function.

Wouldn't it be nice if we could display the rhyme using different names? If we could only tell the function what name to display.

Mmm.
What if we wanted
to use a different
name e.g. James?
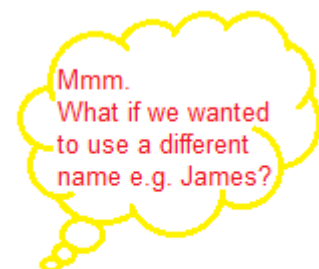
...or Joe?
...or Peter?

This can be done using parameters and arguments.

PROGRAMMER TIP
When you want to pass information into a function use parameter(s)

Parameters and arguments

A parameter is a special kind of variable which appears as part of the function signature and can be used inside the function body. Parameters allow programmers to pass information into a function.

Let's add a parameter to our previous example so that it can display the rhyme about anybody – not just Jack!

```
function displayRhyme(personName) {
  console.log(personName, "loves to do his homework");
  console.log("He never misses a day");
  console.log("He even loves the men in white");
  console.log("Who are taking him away");
}

displayRhyme("James");
```

```
James loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

The name James (and not Jack!) is now included in the rhyme

Notice from the code:

- the identifier `personName` (highlighted in red) between brackets in the function signature on the first line. This is an example of a function parameter. A parameter is a special kind of variable that is initialised when the function is called.
- the text *James* (highlighted in green) between brackets is used as part of the function call on the last line? This is a function argument. Arguments are passed into functions.

In the above example the string *James* is passed as an argument into the function `displayRhyme`. The value is received into the function by the parameter `personName`.

KEY POINT: A function parameter is a variable which gets it value from the argument passed in. When a function is called the value of the argument is assigned to the parameter.

PROGRAMMER TIP
Parameters are named by the programmer. They are identifiers and as such, their names must adhere to the same JavaScript rules for naming variables and functions.

The advantage of using parameters and arguments is that they make functions much more flexible. The runtime behaviour of a function can be altered by passing different arguments into it. This point is demonstrated below where the three calls on the left hand side result in the output displayed on the right.

```
function displayRhyme(personName) {
  console.log(personName, "loves to do his homework");
  console.log("He never misses a day");
  console.log("He even loves the men in white");
  console.log("Who are taking him away");
}

displayRhyme("James");
console.log("");
displayRhyme("Joe");
console.log("");
displayRhyme("Fred");
console.log("");
```

```
James loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Joe loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Fred loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

The parameter is `personName` and the three arguments are *James*, *Joe* and *Fred*. At runtime, the parameter is assigned to each argument and its value is used in the output.

Experiment! Key in and run the code above. Once you are familiar with what it does, change the code so that the output displayed will appear as shown below.

```
Pat loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Peter loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Paul loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

Use the space below to list your parameter(s) and argument(s)

_____

_____

_____

_____

KEY POINT: In general, you should always pass the same number of arguments into the function as the number of parameters that are specified in the function signature.

Multiple parameters/arguments

A function can have more than just one parameter. Take a look at the following:

```
function displayLyrics(line1, line2) {
  console.log(line1);
  console.log(line2);
}

displayLyrics("I read the news today", "Oh boy");
```

This function has two parameters – `line1` and `line2`. It takes two arguments

When `displayLyrics` is called the arguments *"I read the news today"*, and *"Oh boy"* are passed in and received by two parameters `line1` and `line2` respectively. Notice the use of a comma to separate parameters (and arguments) from one another?

The program causes the following output to be displayed on the output console:

```
I read the news today
Oh boy
```

Parameters are received into a function in the same order as the arguments provided. Therefore, if the arguments were switched around like this;

```
displayLyrics("Oh boy", "I read the news today");
```

the function would cause the text below to be displayed.

```
Oh boy
I read the news today
```

Experiment! Predict the output of the code below. Now key the code in and run it. Was your prediction correct? What did you notice?

```
function displayLines(line3, line1, line2) {
  console.log(line1);
  console.log(line2);
  console.log(line3);
}

displayLines ("One", "Two", "Three");
```

Study the two function definitions shown below and answer the question that follows.

```
function displayGreeting1(msg) {          function displayGreeting2(msg1, msg2) {
  console.log(msg);                          console.log(msg1);
                                             console.log(msg2);
}                                         }
        displayGreeting1                          displayGreeting2
```

State what you would expect to happen when each of the code blocks displayed on the left hand side below is run.

Code Block	Expected output
a) `displayGreeting1("Good evening, Dave");`	
b) `let str = "Good evening, Dave";` `displayGreeting1(str);`	
c) `let name = "Dave";` `displayGreeting1("Good evening", name);`	
d) `displayGreeting2("Good evening", "Dave");`	
e) `let name = "Dave";` `displayGreeting2("Good evening", name);`	
f) `let str = "Good evening, Dave";` `displayGreeting2(str);`	
g) `let sum = 2+3;` `displayGreeting1(sum);`	
h) `let sum = "2+3";` `displayGreeting1(sum);`	
i) `let sum = "2+3";` `displayGreeting2(sum, 2+3);`	
j) `displayGreeting2("2+3", "equals", "5");`	

Describe what (if anything) is wrong with each of the following pieces of code and in each case outline a solution (if appropriate).

a)

```
function displayMessage() {
  console.log(msg);
}

displayMessage("I am Sam");
```

b)

```
function displayMessage(msg) {
  console.log(msg);
}

displayMessage();
```

c)

```
function displayMessage(msg) {
  console.log(msg);
}

displayMessage("I am", "Sam");
```

d)

```
function displayMessage(msg) {
  console.log(message);
}

displayMessage("I am Sam");
```

e)

```
function displayMessage(msg) {
  console.log(msg);
}

displayMessage(I am Sam);
```

f)

```
function displayMessage(msg1, msg2) {
  console.log(msg1);
}

displayMessage("I am", "Sam");
```

Return Values

Functions can be thought of as 'little machines' that accept input(s) and sometimes generate an output. These 'function machines' are sometimes referred to as 'black boxes' – so called because programmers who use them don't really care too much about what goes on inside them. The programmers only concern is that the function does the job it is designed to do. This black box view of functions is depicted below.

The 'black box' view of a function

We already know that arguments and parameters are used to pass data into functions. But how does a function pass any data it generates back to its caller as an output? The answer is – *return values.*

Consider a scenario where we wanted to convert an amount in euros and generate a return value which represents the equivalent amount in US dollars. The JavaScript code shown below defines a function called `convert` to do the job. The function receives two parameters – `euroAmount` and `rate`. The arguments passed into the function are `amount` and 1.13 respectively. (The program reads the amount from the end-used and uses a hard-coded conversion rate of €1 = \$1.13).

```javascript
// A function to convert from € to $
function convert(euroAmount, rate) {
  let dollar = amount * rate;
  return dollar; // return the converted amount
}

// Prompt the user to enter the Euro amount
let amount = Number(prompt("Enter amount (€)"));

// Call convert and save the returned value in dollarAmount
let dollarAmount = convert(amount, 1.13);

// Display the answer
console.log("€"+amount+" is worth $"+dollarAmount);
```

The function returns dollar. This value is then assigned to the variable dollarAmount

The return value of the function is dollar

The inputs and output are depicted using our 'black box' model as follows

KEY POINT: The return value of a function can be saved for further processing by making the function call part of an assignment statement.

Example

The function below converts miles to kilometres (based on $1\ mile\ =\ 1.6km$). The input to the function is the parameter `miles`. The return value of the function is `kms`. This is the function output.

```
function miles2kms(miles) {
  let kms = miles * 1.6;
  return kms;
}
```

The return value of the function is `kms`

The code below calls the function passing in 50 as an argument. (The objective is to find out the number of kilometres in 50 miles.)

```
let kimometers = miles2kms(50);
console.log("There are "+kimometers+"kms in 50 miles");
```

The return value is assigned to the variable kilometers

The program displays the following line on the output console.

`There are 80kms in 50 miles`

The output of the function is `kms`. This value is assigned to the variable `kilometres`. Once a function ends its variables and parameters are all destroyed.

*Experiment! The function shown below (**kms2miles**) accepts a value in kilometres as input and then outputs the equivalent in miles (based on $1\ km = 0.62miles$)*

Write a line of code to call this function to convert 80 kilometres into miles and save the output in a variable called `miles`. Now write a second line to display the converted value in a meaningful message. (Can you combine the two lines into one?)

```
function kms2miles(kms) {
  let miles = kms * 0.62;
  return miles;
}
```

Now write another line of code to call the function we defined earlier (`miles2kms`). Use `miles` (returned by your call to `kms2miles`) as the argument. What is your result?

KEY POINT: A function is an abstraction for the task it performs.

Take a look at the two function definitions for add below and answer the questions that follow.

```
function add1(n1, n2) {
  return n1+n2;
}
```

```
function add2(n1, n2) {
  let sum = n1+n2;
  console.log(n1+"+"+n2+"="+sum);
}
```

Which function is better in your opinion? Why? Write a line of code to call `add1` – save your answer in a variable e.g. `answer`. Now try to do the same for `add2`. What problem do you encounter?

Describe what (if anything) is wrong with each of the following pieces of code and in each case outline a solution (if appropriate).

a)

```
function add(n1, n2) {
  let sum = n1+n2;
  return sum;
}

let result = add(8, 3);
console.log(sum);
```

b)

```
function add(n1, n2) {
  let sum = n1+n2;
  return;
}

let result = add(8, 3);
console.log(result);
```

c)

```
function add(n1, n2) {
  let sum = n1+n2;
  return sum;
}

let result = add(8, 3);
```

d)

```
function add(n1, n2) {
  return(n1+n2);
}

console.log(add(8, 3));
```

e)

```
function add(n1, n2) {
  let sum = n1+n2;
  return sum;
}

let result = add(8, add(2,1));
console.log(result);
```

Boolean Functions

A Boolean function is a function that returns either `true` or `false`. They are usually used as an abstraction for some type of test. For example, we could write a Boolean function to determine whether a given number is prime or not.

By convention the name of a Boolean function starts with the prefix **is.** By using this convention, a Boolean function to determine whether or not a given year is a leap year could be called `isLeap`. Similarly, a function to test the evenness or oddness of a number could be called `isEven` and `isOdd` respectively. An example implementation of these two functions and their use to display all even/odd numbers between 0 and 100 inclusive is shown below. (The test for evenness is based on a 'divide by 2' remainder operation.)

```
// A function to determine 'evenness'
function isEven(number) {
  if (number % 2 === 0)
    return true;
  else
    return false;
}

for (let n=0; n<=100; n++) {
  if (isEven(n))
    console.log(n);
}
```
isEven

```
// A function to determine 'oddness'
function isOdd(number) {
  if (number % 2 !== 0)
    return true;
  else
    return false;
}

for (let n=0; n<=100; n++) {
  if (isOdd(n))
    console.log(n);
}
```
isOdd

JS

Challenge!
Implement the following Boolean functions. Test your code using data entered via `prompt`.

`isLessThan(a, b)`	A function that returns `true` if `a` is less than `b`. `false` otherwise.
`isGtREqual2(a, b)`	A function that returns `true` if `a` is greater than or equal to `b`. `false` otherwise.
`isSpeeding(speed)`	A function that returns `true` if the value of `speed` is greater than 120. `false` otherwise.
`isTeenager(age)`	A function that returns `true` if the value of `age` is between 13 and 19 inclusive. `false` otherwise.
`isGoodMusic(artist)`	A function that returns `true` if the value of `artist` is a member of an array that contains the names of all your favourite music artists. `false` otherwise.

Encapsulating code in functions

The process of taking code and putting it into functions is called *encapsulation*.

Recall from earlier the problem to determine the maximum of three numbers. The code - shown again here for convenience - determines and displays the largest of three numbers entered by the user.

```javascript
// max of 3
let x1 = Number(prompt("Please enter 1st number: "));
let x2 = Number(prompt("Please enter 2nd number: "));
let x3 = Number(prompt("Please enter 3rd number: "));
let max;

if ((x1 >= x2) && (x1 >= x3)) {
    max = x1;
} else if ((x2 >= x1) && (x2 >= x3)) {
    max = x2;
} else {
    max = x3;
}

console.log("The largest number you entered was", max);
```

> The three numbers are entered

> The maximum is determined by comparing each number to the other two

> The maximum is displayed

Let's try to encapsulate this code into a function – for this we will need to decide on a name, possible parameter(s) and a possible return value.

The following three questions are useful to ask when attempting to encapsulate any code using functions:

1) What does the code do? This would be a good name for the function.
    In this example, the essence of the code is to find the maximum of three numbers so we will call our function `maxOf3`.

2) What are the inputs? The function should have a parameter for each input.
    In our example, the inputs are the three numbers – we will have one parameter for each number – `x1`, `x2` and `x3`.

3) What is the output? This will be the return value of the function
    We only have one output i.e. the maximum of the three numbers.

KEY POINT: The ability to be able to identify function parameters and, if necessary, a return value is a key programmer skill.

We can see from the code below that the code to determine the largest of three numbers has been encapsulated in the function `maxOf3` (highlighted in a darker colour).

Notice that the first three lines used to read the numbers from the end user, and the final line to display the result are not part of the function. This separation of input and output from the logic of a function is typical when it comes to code encapsulation.

```javascript
// Read the three numbers from the end user
let x1 = Number(prompt("Please enter 1st number: "));
let x2 = Number(prompt("Please enter 2nd number: "));
let x3 = Number(prompt("Please enter 3rd number: "));

function maxOf3(x1, x2, x3) {
  let max;
  if ((x1 >= x2) && (x1 >= x3)) {
      max = x1;
  } else if ((x2 >= x1) && (x2 >= x3)) {
      max = x2;
  } else {
      max = x3;
  }

  return max;
}

// Display the output
console.log("The maximum number is", maxOf3(x1, x2, x3));
```

The code is very similar to the listing shown on the previous page. However, since the functionality to find the maximum of three numbers has been captured inside a function it means that it could be used anywhere else in the wider system.

Finally, it is worth pointing out for clarity that the names of the arguments and the names of the parameters do not have to be the same (as shown above).

PROGRAMMER TIP
Avoid duplicating blocks of code. Code duplication is symptomatic of 'poor program design' and can be avoided using encapsulation.

If you find that you need to re-use a number of lines of code to do something specific it is a sure indication that the code should be encapsulated inside a function.

JS Programming Exercises - Functions

1. The two short programs shown beside each other here calculate and display 5! (i.e. 5 factorial which is $5 \times 4 \times 3 \times 2 \times 1$) The both employ the same method which is to iterate through the integers from 5 down to 1 keeping a running total of the products as they do so. The only difference is the program on the left uses a `for` loop and the program on the right uses a `while` loop.

   Encapsulate the code from either implementation into a function called `factorial`. The function should accept a single parameter and return its factorial.

```javascript
// calculate n! (e.g. 5x4x3x2x1)
let factorial = 1;
let n = 5;
for (let number = n; number > 0; number--) {
  factorial = factorial * number;
}

console.log(n+"! =", factorial);
```

Factorial using a `for` loop

```javascript
// calculate n! (e.g. 5x4x3x2x1)
let factorial = 1;
let n = 5;
let number = n;
while (number > 0) {
  factorial = factorial * number;
  number--;
}

console.log(n+"! =", factorial);
```

Factorial using a `while` loop

2. The following program uses a `for` loop to traverse every character in a string entered by the user, counting the vowels as it does so. Encapsulate the code in a function.

```javascript
// prompt the user to enter a string
let inString = prompt("Enter a string:");
let vowels = 0;
let ch;

for(let i = 0; i < inString.length; i ++) {

  // Extract the next character (from position i) ...
  // ... and convert it to upper case
  ch = inString.charAt(i).toUpperCase();

  // if ch is a vowel increment the vowel counter
  if (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U')
    vowels ++;
}
// display the result
console.log("The number of vowels found was", vowels);
```

3. A prime number is a positive integer that has exactly two factors; itself and 1. The short program shown below prompts the user to enter a positive integer and implements an algorithm to determine whether the number is prime or not.

The key to understanding the logic of the program lies behind the Boolean variable called `prime`. The purpose of `prime` is to indicate whether the integer in question is prime or not. A value of `true` at the end of the program indicates that the integer is indeed prime; `false` otherwise.

```
let x = Number(prompt("Please enter an integer: "));
let prime;
if (x <= 0)
  prime = false;
else
  prime = true;

if (x > 2) {
  let denominator = 2;
  while (denominator <= Math.sqrt(x)) {
    if (x % denominator === 0) {
      prime = false;
      break;
    } else {
      denominator++;
    } // end else
  } // end while
} // end if


if (prime)
  console.log(x+" is prime");
else
  console.log(x+" is NOT prime");
```

The user is prompted to enter a number and if it is negative the variable prime is set to false (because negative numbers, by definition, cannot be prime). If the integer is positive the value of prime is initialised to true.

Therefore, the algorithm assumes that the integer is prime and sets about trying to disprove this assumption by looking for factors of the number that was entered.

Encapsulate the above code into a function called `isPrime(x)`. The required behaviour is illustrated in the following example calls to the function.

```
isPrime(13) --> true (because 13 is a prime number)
isPrime(10) --> false (because 10 is not a prime number)
```

Once you have successfully implemented `isPrime` you should attempt to complete these additional tasks:

a) Use the `isPrime` function to display all the prime numbers between 2 and 100 inclusive.

b) Use the `isPrime` function to count and display the number of prime numbers between 2 and 1000.

c) Use `isPrime` to display the first 50 prime numbers.

4. Define a function to calculate the area of a triangle using the formula:

$$area = \frac{1}{2} \times b \times h$$

Use this function to calculate the areas of the triangles displayed below:
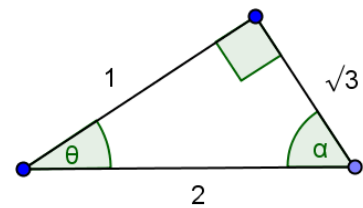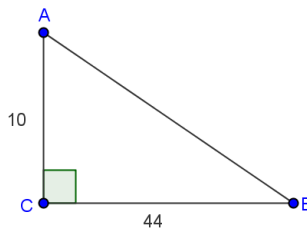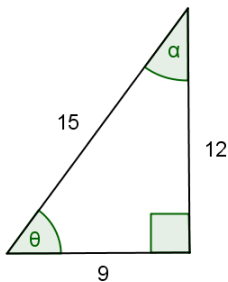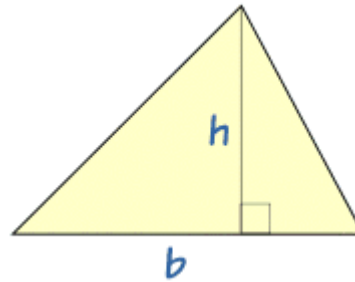
5. The image on the right hand side illustrates the Fahrenheit values for a selection of Celsius values.

Given the formulae below to convert between the two scales write a program that can be used to verify the accuracy of the values shown.

$$f = \frac{9}{5}c + 32 \qquad\qquad c = (f - 32) \times \frac{5}{9}$$

The program will contain two functions named and defined as follows:

`fhar2Cent` - a function that accepts a Fahrenheit value and returns its Celsius equivalent and

`cent2Fhar` - a function that accepts a Celsius value and returns its Fahrenheit equivalent
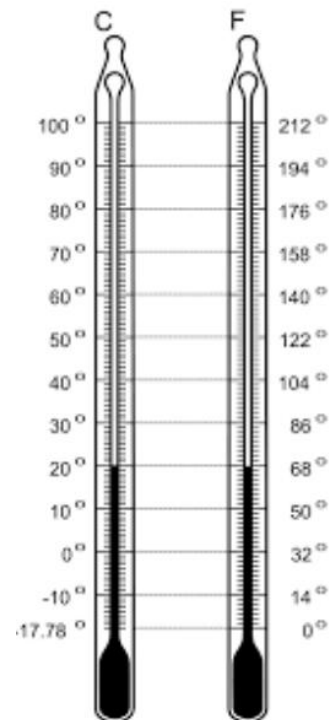
6. Write a program that prompts a user to enter a number of days and then proceeds to compute the number of minutes in that number of days.
   a) extend the program just written to prompt for a number of hours as well as a number of days.

Reflect on all the exercises in this section.

Suggest how you could adapt any of the exercises for use in your own LCCS classroom.
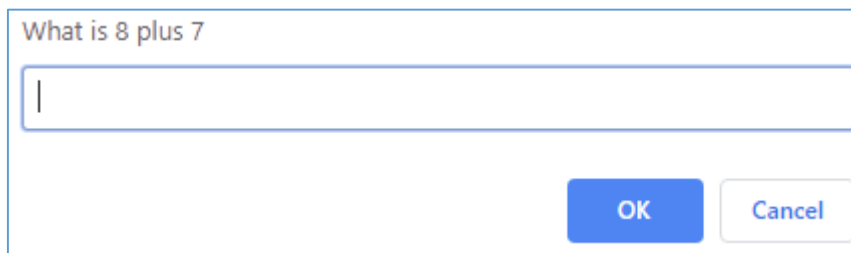
BREAKOUT ACTIVITY 1

Computer Aided Learning

Computer Aided Instruction (CAI) is playing an increasing role in education. For this activity you are required to write a JavaScript application that will help primary school pupils learn arithmetic. We call the application Computer Aided Learning or CAL for short.

You are provided with a program (see listing on the next page) which you will need to make changes to in order to develop an improved solution.

When it is run, the program displays a popup window asking the user to enter the answer to a simple addition problem involving two randomly selected single digit integers between 0 and 9 inclusive. For example,
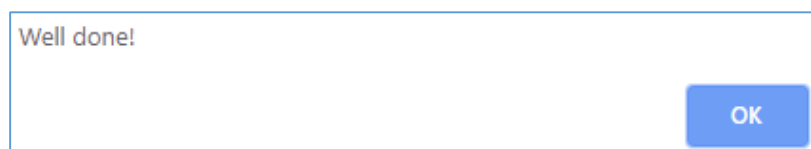
The user types in their response clicks OK. If the response is correct the program displays an encouragement message in a new popup window such as that shown below. The message displayed is picked randomly from the following list (array) of strings:

- Well done!
- Very good!
- Correct!
- Keep it up!
- Nice work!

If the answer is wrong, the program displays the message, *Wrong answer. Try again repeatedly* until the pupil finally enters the right answer.

The program listing you are provided with is shown below.

```javascript
// STEP 1. Generate the question
// Generate 2 random numbers
let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

// STEP 2. Ask the user the question and get a response
// n1 and n2 are converted to strings so that they can be displayed as part ...
// ... of the prompt string
let problemInWords = "What is "+ String(n1) + " + " + String(n2); // Q1
console.log(problemInWords);
let userResponse = Number(prompt(problemInWords));
console.log("User entered %d", userResponse); // this is for debug purposes

// STEP 3. Process the response
// Compute the correct answer. Then ...
// ... as long as the user's answer is different to the computer's answer ...
// ... tell the user they are wrong and ...
// ... ask the user for another response
let correctAnswer = n1 + n2; // Compute the correct answer
while (userResponse != correctAnswer) {
    console.log("Wrong answer! Try again.");
    userResponse = Number(prompt(problemInWords));
    console.log("User entered %d", userResponse); // this is for debug purposes
} // end while

// STEP 4. Display a randomly selected encouragement message
const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice
work!"];
const r = Math.floor(Math.random() * messages.length);
console.log(messages[r]);
alert(messages[r]);
```

CAL v1 Initial Code Listing

Study the above program carefully before trying it out – use the space below to reflect on the code. Demonstrate that you understand the code by answering the questions listed on the next page. Once you are satisfied that you understand how the code works you can then proceed to the attempt the exercises.

What is your initial impression of the above program?

Use the space below to answer each of the following questions before moving on to the exercises in the next section.

1. How does the program generate the two integers to use in the 'sum'? What step(s) are taken to ensure that the integers are between 0 and 9 inclusive?

2. How does the computer know what the correct answer to the 'sum' is?

3. Explain how the + operator behaves differently when its operands are string as opposed to being of a numeric type? (What would happen if you tried to add a string to an integer?)

4. How does the program select a message to display when the user gets the answer right?

5. Identify and state the purpose of each variable (there are seven of them!)

Exercises

Attempt each of the following questions. The questions can be attempted in any order but we recommend carrying them out in the same sequence as they are given. Although it is possible to build the solutions on top of one another you are encouraged to return to the initial code base for each question i.e. implement each solution separately over the initial code listing provided earlier.

1. Change the way the question is displayed so that it uses the operator symbol as opposed to words. So for example display `4 + 2` as opposed to the text *4 plus 2*.

2. Incorporate a statistics feature (i.e. a counter) to keep track of the number of *attempts* it took a user to get the correct answer. This value should be displayed once before the program ends. A sample output is shown here to the right.

```
What is 5 plus 1
User entered 5
Wrong answer! Try again.
User entered 1
Wrong answer! Try again.
User entered 6
You took 3 attempts
Very good!
>
```
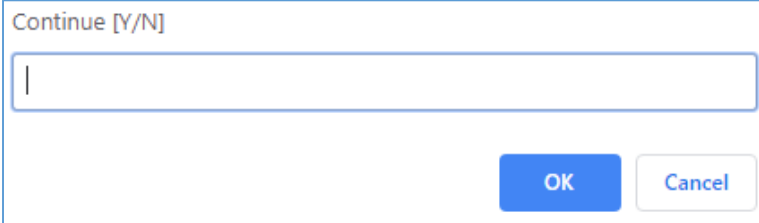
3. Modify the program so that the computer responses for an incorrect answer are randomly selected from the list of messages below. The message should be displayed in an alert box such as that shown below as well as the console (sample output shown).

    – No. Please try again
    – Wrong. Try once more
    – Don't give up
    – No. Keep trying.
    – Incorrect!

```
What is 4 plus 6
User entered 6
Don't give up!
User entered 9
No! Keep trying
User entered 10
Very good!
```
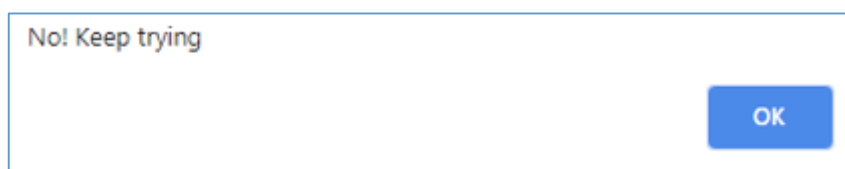
```
No! Keep trying

                              OK
```

4. Modify the program behaviour so that once the user enters the correct answer the user is offered the choice to continue (or quit) i.e. *Continue [Y/N].* (see screenshot) Processing should continue as long as the user enters 'y' or 'Y'
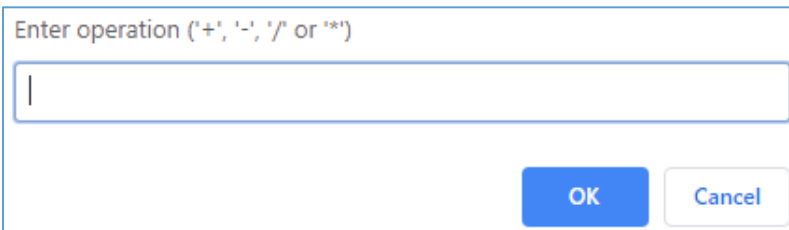
Hint: You will need to introduce a Boolean variable e.g. `quit`, initially set to `false`. The original code will need to wrapped inside a `while` loop. The loop will be executed as long as `quit` is `false`. At the end of the loop you will need to add code find out if the user wants to continue. If the user enters anything other than 'y' or 'Y' then set `quit` to `true` (this will cause the loop to end).

5. Add an option to allow the user to choose a difficulty level (of either 1 or 2) when the program is first started. If the user enters 1 it means that problems should involve single digit integers (i.e. between 0 and 9 inclusive only). If the user enters 2 the application should generate double digit problems (i.e. problems that involve two integers in the range 10 to 99 inclusive). You may write code to validate the user's option if you wish.

Hint: You will need the line of code below which generates a random number between 10 and 99 inclusive.
```
Math.floor(Math.random() * 90) + 10;
```

6. Add an option to allow the user to select the arithmetic operation they wish to use during their session. Use the following encoding: '+' is addition; '-' is subtraction; '*' is multiplication and '/' is division;

7. The current implementation has no exit strategy i.e. the user has no way of telling of telling the system that they don't know the answer. (It repeatedly displays the message *Wrong answer. Try again* until the correct answer is entered.)

   Consequently, the user is left stuck with no way of exiting the system if he/she doesn't know the answer to the question

   Modify the program's behaviour so that any negative value entered as a response is taken to mean that the user does not know the answer to the question presented. When a negative value is entered the program should display the correct answer in the console and then end.

8. Now that you have attempted each question individually try to incorporate all the solutions into a single program.

The solutions to each individual exercise are provided at the end of this section.

---

10 further questions for you to consider

The questions below are designed to provoke ideas for possible enhancements that could be made to CAL at some stage in the future.

1) How can you ensure that the system is fully tested?
2) How could you ensure that all the data entered by the user is validated?
3) Can you come up with an alternative 'exit strategy' to the one proposed in question 7?
4) Would operations involving a mixture of single and double digit operands be possible?
5) How could the system be extended to support other types of operations?
6) How could the system automatically select the type of operation for each question randomly?
7) Could the strategy for setting the difficulty level be made more 'intelligent'? What would this involve?
8) How could the system track the average number of attempts the user takes?
9) What would the system look like as a HTML/CSS web page? Sketch it out.
10) What other extensions could be made to the system?

Extended challenge

Now that you have attempted each question individually try to incorporate all the solutions into a single **modular** program i.e. one that uses functions.

The structure of the main program loop is shown below. Your task is to implement the functions.

Hint: Abstraction and decomposition as the key skills required. You will need to build up the solution function by function.

```javascript
// MAIN PROGRAM STARTS HERE
const correctMessages = ["Well done!", "Very good!", "Correct!", "Keep it up!",
"Nice work!"];
const incorrectMessages = ["No! Please try again", "Wrong! Try once more", "Don't
give up!", "No! Keep trying", "That's incorrect"];

let quit = false;
while (!quit) {
  // Ask the user to enter the difficulty level, the operation and the two operands
  let difficultyLevel = getDifficultyLevel();
  let operation = getOperation();
  let n1 = getOperand(difficultyLevel);
  let n2 = getOperand(difficultyLevel);

  // determine the correct answer from the computer
  let correctAnswer = getCorrectAnswer(n1, n2, operation);

  // keep asking the question until the user gets it right or gives up
  let questionString = "What is "+ String(n1) + operation + String(n2);
  let userResponse = getResponse(questionString);
  while (userResponse != correctAnswer && userResponse > 0) {
    displayRandomMsg(incorrectMessages);
    userResponse = getResponse(questionString);
  }

  // check if the user got it right or 'gave up'
  if (userResponse == correctAnswer) {
    displayRandomMsg(correctMessages);
  } else {
    console.log("Answer not known. The correct answer was", correctAnswer);
    alert("Answer not known. The correct answer was" + String(correctAnswer));

  }

  // Finally, determine whether to keep going or not
  let yesNo = prompt("Continue [Y/N]");
  quit = yesNo.toUpperCase() == "Y" ? false : true;

} // end while
```

The solution to this exercise is shown on page 203.

Reflect on CAL and the exercises just complete in this breakout session. The focus of CAL is very much on early stage arithmetic – can you suggest how CAL could be adapted to assist learning in other subject areas? Discuss is small groups. What are the design implications?

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Suggested Solutions to Breakout #1 (CAL)

The solutions to each of the eight exercise are provided in the following pages. The comments serve to explain the code.

Each solution is implemented over the initial code base – shown again here for convenience.

```javascript
// STEP 1. Generate the question
// Generate 2 random numbers
let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

// STEP 2. Ask the user the question and get a response
// n1 and n2 are converted to strings so that they can be displayed as part ...
// ... of the prompt string
let problemInWords = "What is "+ String(n1) + " plus " + String(n2);
console.log(problemInWords);
let userResponse = Number(prompt(problemInWords));
console.log("User entered %d", userResponse); // this is for debug purposes

// STEP 3. Process the response
// Compute the correct answer. Then ...
// ... as long as the user's answer is different to the computer's answer ...
// ... tell the user they are wrong and ...
// ... ask the user for another response
let correctAnswer = n1 + n2; // Compute the correct answer
while (userResponse != correctAnswer) {
    console.log("Wrong answer! Try again.");
    userResponse = Number(prompt(problemInWords));
    console.log("User entered %d", userResponse); // this is for debug purposes
} // end while

// STEP 4. Display a randomly selected encouragement message
const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice work!"];
const r = Math.floor(Math.random() * messages.length);
console.log(messages[r]);
alert(messages[r]);
```

CAL v1 Initial Code Listing

As you read through the solutions to each questions in the following pages you will notice some of the code is highlighted in bold. This is used to indicate the parts of the program that were changed, added and/or commented out in order to achieve the solution. You should make an effort to understand each solution before moving on to the next.

Breakout #1 (CAL): Question 1 and Suggested Solution

Change the way the question is displayed so that it uses the operator symbol as opposed to words. So for example display `4 + 2` as opposed to the text *4 plus 2*.

```javascript
// CAL Suggested solution to Question 1

// STEP 1. Generate the question
// Generate 2 random numbers
let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

// STEP 2. Ask the user the question and get a response
// n1 and n2 are converted to strings so that they can be displayed as part ...
// ... of the prompt string
let problemInWords = "What is "+ String(n1) + " + " + String(n2); // Q1 replace
'plus' with '+'
console.log(problemInWords);
let userResponse = Number(prompt(problemInWords));
console.log("User entered %d", userResponse); // this is for debug purposes

// STEP 3. Process the response
// Compute the correct answer. Then ...
// ... as long as the user's answer is different to the computer's answer ...
// ... tell the user they are wrong and ...
// ... ask the user for another response
let correctAnswer = n1 + n2; // Compute the correct answer
while (userResponse != correctAnswer) {
    console.log("Wrong answer! Try again.");
    userResponse = Number(prompt(problemInWords));
    console.log("User entered %d", userResponse); // this is for debug purposes
} // end while

// STEP 4. Display a randomly selected encouragement message
const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice
work!"];
const r = Math.floor(Math.random() * messages.length);
console.log(messages[r]);
alert(messages[r]);
```

Breakout #1 (CAL): Question 2 and Suggested Solution

Incorporate a statistics feature (i.e. a counter) to keep track of the number of *attempts* it took a user to get the correct answer. This value should be displayed once before the program ends. A sample output is shown here to the right.

```
What is 5 plus 1
User entered 5
Wrong answer! Try again.
User entered 1
Wrong answer! Try again.
User entered 6
You took 3 attempts
Very good!
>
```

```javascript
// CAL Suggested solution to Question 2

// STEP 1. Generate the question
// Generate 2 random numbers
let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

// STEP 2. Ask the user the question and get a response
// n1 and n2 are converted to strings so that they can be displayed as part ...
// ... of the prompt string
let problemInWords = "What is "+ String(n1) + " plus " + String(n2);
console.log(problemInWords);
let userResponse = Number(prompt(problemInWords));
console.log("User entered %d", userResponse); // this is for debug purposes

// STEP 3. Process the response
// Compute the correct answer. Then ...
// ... as long as the user's answer is different to the computer's answer ...
// ... tell the user they are wrong and ...
// ... ask the user for another response
let correctAnswer = n1 + n2; // Compute the correct answer
let attempts = 1; // Q2 used to store the number of incorrect attempts
while (userResponse != correctAnswer) {
    console.log("Wrong answer! Try again.");
    userResponse = Number(prompt(problemInWords));
    console.log("User entered %d", userResponse); // this is for debug purposes
    attempts++; // Q2 increment attempts
} // end while

// STEP 4. Display a randomly selected encouragement message
const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice work!"];
const r = Math.floor(Math.random() * messages.length);
console.log(messages[r]);
alert(messages[r]);
console.log("You took", attempts, "attempts"); // Q2 display attempts
```

We use a variable called `attempts` which is initialised to 1 before the `while` loop. The variable is incremented on each loop iteration and finally displayed by the last line of the program.

Breakout #1 (CAL): Question 3 and Suggested Solution

Modify the program so that the computer responses to an incorrect answer are randomly selected from the list of messages shown. The message should be displayed in an alert box as well as the console.

- No. Please try again
- Wrong. Try once more
- Don't give up
- No. Keep trying.
- Incorrect!

```javascript
// CAL Suggested solution to Question 3

// STEP 1. Generate the question
// Generate 2 random numbers
let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

// STEP 2. Ask the user the question and get a response
// n1 and n2 are converted to strings so that they can be displayed as part ...
// ... of the prompt string
let problemInWords = "What is "+ String(n1) + " plus " + String(n2);
console.log(problemInWords);
let userResponse = Number(prompt(problemInWords));
console.log("User entered %d", userResponse); // this is for debug purposes

// STEP 3. Process the response
// Compute the correct answer. Then ...
// ... as long as the user's answer is different to the computer's answer ...
// ... tell the user they are wrong and ...
// ... ask the user for another response

// Q3 - create a list of messages to use if the response in incorrect
const messages1 = ["No! Please try again", "Wrong! Try once more", "Don't give
up!", "No! Keep trying", "That's incorrect"];

let correctAnswer = n1 + n2; // Compute the correct answer
while (userResponse != correctAnswer) {
    // Q3 comment the next line out - it's functionality is replaced by the new
lines below
    // console.log("Wrong answer! Try again.");
    // Q3 Display a randomly selected message telling the user they are wrong
    const index = Math.floor(Math.random() * messages1.length);
    console.log(messages1[index]);
    alert(messages1[index]);

    userResponse = Number(prompt(problemInWords));
    console.log("User entered %d", userResponse); // this is for debug purposes
} // end while

// STEP 4. Display a randomly selected encouragement message
const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice
work!"];
const r = Math.floor(Math.random() * messages.length);
console.log(messages[r]);
alert(messages[r]);
```

Breakout #1 (CAL): Question 4 and Suggested Solution

Modify the program behaviour so that once the user enters the correct answer the user is offered the choice to continue (or quit) i.e. *Continue [Y/N]*. (see screenshot) Processing should continue as long as the user enters 'y' or 'Y'

```javascript
// CAL Suggested solution to Question 4

let quit = false; // Q4 continue as long as the variable quit is false
while (quit == false) {    // Q4 the original code is 'wrapped' inside this while loop

  // STEP 1. Generate the question
  // Generate 2 random numbers
  let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
  let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

  // STEP 2. Ask the user the question and get a response
  // n1 and n2 are converted to strings so that they can be displayed as part ...
  // ... of the prompt string
  let problemInWords = "What is "+ String(n1) + " plus " + String(n2);
  console.log(problemInWords);
  let userResponse = Number(prompt(problemInWords));
  console.log("User entered %d", userResponse); // this is for debug purposes

  // STEP 3. Process the response
  // Compute the correct answer. Then ...
  // ... as long as the user's answer is different to the computer's answer ...
  // ... tell the user they are wrong and ...
  // ... ask the user for another response
  let correctAnswer = n1 + n2; // Compute the correct answer
  while (userResponse != correctAnswer) {
      console.log("Wrong answer! Try again.");
      userResponse = Number(prompt(problemInWords));
      console.log("User entered %d", userResponse); // this is for debug purposes
  } // end while

  // STEP 4. Display a randomly selected encouragement message
  const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice
work!"];
  const r = Math.floor(Math.random() * messages.length);
  console.log(messages[r]);
  alert(messages[r]);

  // Q4 find out if the user wants to continue ...
  // ... if the user enters anything other than Y then ..
  // ... set quit to true (this will cause the loop to end)
  let yesNo = prompt("Continue [Y/N]");
  if (yesNo.toUpperCase() != "Y")
    quit = true;

} // Q4 end of while loop
```

Breakout #1 (CAL): Question 5 and Suggested Solution

Add an option to allow the user to choose a difficulty level (of either 1 or 2) when the program is first started. If the user enters 1 it means that problems should involve single digit integers (i.e. between 0 and 9 inclusive only). If the user enters 2 the application should generate double digit problems (i.e. problems that involve two integers in the range 10 to 99 inclusive). You may write code to validate the user's option if you wish.

```javascript
// CAL Suggested solution to Question 5

let diffLevelQuestion = "Enter difficulty level (1 or 2)";
console.log(diffLevelQuestion);
let difficultyLevel = Number(prompt(diffLevelQuestion));
let n1, n2;
if (difficultyLevel == 1) {
  // Generate 2 random numbers between 0 and 9 incl.
  n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
  n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10
} else {
  // Generate 2 random numbers between 10 and 99 incl.
  n1 = Math.floor(Math.random() * 90) + 10; // 10 <= n1 < 100
  n2 = Math.floor(Math.random() * 90) + 10; // 10 <= n2 < 100
}

/// STEP 1. Generate the question
// Generate 2 random numbers
// Q5 comment out the next two lines (n1 and n2 are set above)
//let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
//let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

// STEP 2. Ask the user the question and get a response
// n1 and n2 are converted to strings so that they can be displayed as part ...
// ... of the prompt string
let problemInWords = "What is "+ String(n1) + " plus " + String(n2);
console.log(problemInWords);
let userResponse = Number(prompt(problemInWords));
console.log("User entered %d", userResponse); // this is for debug purposes

// STEP 3. Process the response
// Compute the correct answer. Then ...
// ... as long as the user's answer is different to the computer's answer ...
// ... tell the user they are wrong and ...
// ... ask the user for another response
let correctAnswer = n1 + n2; // Compute the correct answer
while (userResponse != correctAnswer) {
    console.log("Wrong answer! Try again.");
    userResponse = Number(prompt(problemInWords));
    console.log("User entered %d", userResponse); // this is for debug purposes
} // end while

// STEP 4. Display a randomly selected encouragement message
const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice
work!"];
const r = Math.floor(Math.random() * messages.length);
console.log(messages[r]);
alert(messages[r]);
```

The code to validate the user's option is left as a further challenge!

Breakout #1 (CAL): Question 6 and Suggested Solution

Add an option to allow the user to select the arithmetic operation they wish to use during their session. Use the following encoding: '+' is addition; '-' is subtraction; '*' is multiplication and '/' is division;

```javascript
// CAL Suggested solution to Question 6


// Q6 Get the required operation from the user
let operationQuestion = "Enter operation ('+', '-', '/' or '*')";
console.log(operationQuestion);
let operation = prompt(operationQuestion);

// STEP 1. Generate the question
// Generate 2 random numbers
let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

// STEP 2. Ask the user the question and get a response
// n1 and n2 are converted to strings so that they can be displayed as part ...
// ... of the prompt string
let problemInWords = "What is "+ String(n1) + operation + String(n2); // Q6
console.log(problemInWords);
let userResponse = Number(prompt(problemInWords));
console.log("User entered %d", userResponse); // this is for debug purposes

// STEP 3. Process the response
// Compute the correct answer. Then ...
// ... as long as the user's answer is different to the computer's answer ...
// ... tell the user they are wrong and ...
// ... ask the user for another response
let correctAnswer; // Q6 = n1 + n2; // Compute the correct answer

// Q6 Compute the correct answer for the operation
if (operation === "+") {
    correctAnswer = n1 + n2;
} else if (operation === "-") {
    correctAnswer = n1 - n2;
} else if (operation === "*") {
    correctAnswer = n1 * n2;
} else {
    correctAnswer = n1 / n2;
}

while (userResponse != correctAnswer) {
    console.log("Wrong answer! Try again.");
    userResponse = Number(prompt(problemInWords));
    console.log("User entered %d", userResponse); // this is for debug purposes
} // end while

// STEP 4. Display a randomly selected encouragement message
const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice
work!"];
const r = Math.floor(Math.random() * messages.length);
console.log(messages[r]);
alert(messages[r]);
```

Breakout #1 (CAL): Question 7 and Suggested Solution

The current implementation has no exit strategy i.e. the user has no way of telling of telling the system that they don't know the answer. (It repeatedly displays the message Wrong answer. Try again until the correct answer is entered.)

Consequently, the user is left stuck with no way of exiting the system if he/she doesn't know the answer to the question

Modify the program's behaviour so that any negative value entered as a response is taken to mean that the user does not know the answer to the question presented. When a negative value is entered the program should display the correct answer in the console and then end.

```javascript
// CAL Suggested solution to Question 7

// STEP 1. Generate the question
// Generate 2 random numbers
let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

// STEP 2. Ask the user the question and get a response
// n1 and n2 are converted to strings so that they can be displayed as part ...
// ... of the prompt string
let problemInWords = "What is "+ String(n1) + " plus " + String(n2);
console.log(problemInWords);
let userResponse = Number(prompt(problemInWords));
console.log("User entered %d", userResponse); // this is for debug purposes

// STEP 3. Process the response
// Compute the correct answer. Then ...
// ... as long as the user's answer is different to the computer's answer ...
// ... tell the user they are wrong and ...
// ... ask the user for another response
let correctAnswer = n1 + n2; // Compute the correct answer
while (userResponse != correctAnswer) {
    // Q7 - take a negative response to mean the user doesn't know the answer
    if (userResponse < 0) {
         break; // exit the loop
    } else {
       console.log("Wrong answer! Try again.");
       userResponse = Number(prompt(problemInWords));
       console.log("User entered %d", userResponse); // this is for debug purposes
    }
} // end while

// Q7 - user didn't know the answer so tell them
if (userResponse < 0) {
    console.log("Answer not known. The correct answer was", correctAnswer);
    alert("Answer not known. The correct answer was" + String(correctAnswer));
} else {
    // STEP 4. Display a randomly selected encouragement message
    const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice
work!"];
    const r = Math.floor(Math.random() * messages.length);
    console.log(messages[r]);
    alert(messages[r]);
}
```

Breakout #1 (CAL): Question 8 and Suggested Solution

Now that you have attempted each question individually try to incorporate all the solutions into a single program.

```javascript
// CAL Suggested solution to Question 8

let quit = false; // Q4 continue as long as the variable quit is false
while (quit == false) {   // Q4 the original code is 'wrapped' inside this while
loop

  // Q6 Get the required operation from the user
  let operationQuestion = "Enter operation ('+', '-', '/' or '*')";
  console.log(operationQuestion);
  let operation = prompt(operationQuestion);

  // Q5 solution starts here ...
  let diffLevelQuestion = "Enter difficulty level (1 or 2)";
  console.log(diffLevelQuestion);
  let difficultyLevel = Number(prompt(diffLevelQuestion));
  let n1, n2;
  if (difficultyLevel == 1) {
    // Generate 2 random numbers between 0 and 9 incl.
    n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
    n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10
  } else {
    // Generate 2 random numbers between 10 and 99 incl.
    n1 = Math.floor(Math.random() * 90) + 10; // 10 <= n1 < 100
    n2 = Math.floor(Math.random() * 90) + 10; // 10 <= n2 < 100
  }

  /// STEP 1. Generate the question
  // Generate 2 random numbers
  // Q5 comment out the next two lines (n1 and n2 are set above)
  //let n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
  //let n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10

  // Q5 solution ends here ...

  // STEP 2. Ask the user the question and get a response
  // n1 and n2 are converted to strings so that they can be displayed as part ...
  // ... of the prompt string
  let problemInWords = "What is "+ String(n1) + operation + String(n2); // Q6
  console.log(problemInWords);
  let userResponse = Number(prompt(problemInWords));
  console.log("User entered %d", userResponse); // this is for debug purposes

  // STEP 3. Process the response
  // Compute the correct answer. Then ...
  // ... as long as the user's answer is different to the computer's answer ...
  // ... tell the user they are wrong and ...
  // ... ask the user for another response
  let correctAnswer; // Q6 = n1 + n2; // Compute the correct answer

  // Q6 Compute the correct answer for the operation
  if (operation === "+") {
      correctAnswer = n1 + n2;
  } else if (operation === "-") {
      correctAnswer = n1 - n2;
  } else if (operation === "*") {
      correctAnswer = n1 * n2;
  } else {
      correctAnswer = n1 / n2;
  }
```

```javascript
  let attempts = 1; // Q2 used to store the number of incorrect attempts
  // Q3 - create a list of messages to use if the response in incorrect
  const messages1 = ["No! Please try again", "Wrong! Try once more", "Don't give
up!", "No! Keep trying", "That's incorrect"];

  while (userResponse != correctAnswer) {
      // Q7 - take a negative response to mean the user doesn't know the answer
      if (userResponse < 0) {
          break; // exit the loop
      } else {
        //console.log("Wrong answer! Try again.");
        // Q3 Display a randomly selected message telling the user they are wrong
        const index = Math.floor(Math.random() * messages1.length);
        console.log(messages1[index]);
        alert(messages1[index]);

        userResponse = Number(prompt(problemInWords));
        console.log("User entered %d", userResponse); // this is for debug purposes

        attempts++; // Q2 increment attempts
      }
  } // end while


  // Q7 - user didn't know the answer so tell them
  if (userResponse < 0) {
      console.log("Answer not known. The correct answer was", correctAnswer);
      alert("Answer not known. The correct answer was" + String(correctAnswer));
  } else {
      // STEP 4. Display a randomly selected encouragement message
      const messages = ["Well done!", "Very good!", "Correct!", "Keep it up!",
"Nice work!"];
      const r = Math.floor(Math.random() * messages.length);
      console.log(messages[r]);
      alert(messages[r]);
  }

  console.log("You took", attempts, "attempts"); // Q2 display attempts

  // Q4 find out if the user wants to continue ...
  // ... if the user enters anything other than Y then ..
  // ... set quit to true (this will cause the loop to end)
  let yesNo = prompt("Continue [Y/N]");
  if (yesNo.toUpperCase() != "Y")
    quit = true;

} // Q4 end of while loop
```

Breakout #1 (CAL): Suggested Solution to extended challenge

Now that you have attempted each question individually try to incorporate all the solutions into a single **modular** program i.e. one that uses functions.

The structure of the main program loop is shown (again) below. Your task is to implement the functions.

```javascript
// MAIN PROGRAM STARTS HERE

const correctMessages = ["Well done!", "Very good!", "Correct!", "Keep it up!",
"Nice work!"];
const incorrectMessages = ["No! Please try again", "Wrong! Try once more", "Don't
give up!", "No! Keep trying", "That's incorrect"];

let quit = false;
while (!quit) {
  // Ask the user to enter the difficulty level, the operation and the two operands
  let difficultyLevel = getDifficultyLevel();
  let operation = getOperation();
  let n1 = getOperand(difficultyLevel);
  let n2 = getOperand(difficultyLevel);

  // determine the correct answer from the computer
  let correctAnswer = getCorrectAnswer(n1, n2, operation);

  // keep asking the question until the user gets it right or gives up
  let questionString = "What is "+ String(n1) + operation + String(n2);
  let userResponse = getResponse(questionString);
  while (userResponse != correctAnswer && userResponse > 0) {
    displayRandomMsg(incorrectMessages);
    userResponse = getResponse(questionString);
  }

  // check if the user got it right or 'gave up'
  if (userResponse == correctAnswer) {
    displayRandomMsg(correctMessages);
  } else {
    console.log("Answer not known. The correct answer was", correctAnswer);
    alert("Answer not known. The correct answer was" + String(correctAnswer));

  }

  // Finally, determine whether to keep going or not
  let yesNo = prompt("Continue [Y/N]");
  quit = yesNo.toUpperCase() == "Y" ? false : true;

} // end while
```

The first step towards a solution here is to insert empty functions at the *top* of the program listing shown on the previous page. Empty functions such as these are called **stub functions** and are often used by programmers as placeholders so that they can focus their attention on one aspect of functionality at a time without having to worry about syntax errors caused by missing function definitions.

```javascript
// CAL Suggested solution to Question 9 - function stubs

// A function to return the arithmetic operation
function getOperation() {


} // end getOperation

// A function to return the difficulty level
function getDifficultyLevel() {


} // end getDifficultyLevel

// A function to generate an operand based on the difficulty level
function getOperand(level) {


} // end getOperand


// A function to receive the user's answer
function getResponse(problemInWords) {


} // end getResponse

// A function to compute the correct answer
function getCorrectAnswer(op1, op2, op){


} // getCorrectAnswer


// A function that generates and displays a random message
function displayRandomMsg(messages) {


} // end displayRandomMsg

// MAIN PROGRAM STARTS HERE
```

The code for each function is shown on the following pages.

```javascript
// A function to return the arithmetic operation
function getOperation() {
  let operationQuestion = "Enter operation ('+', '-', '/' or '*')";
  console.log(operationQuestion);

  return prompt(operationQuestion);
} // end getOperation
```

```javascript
// A function to return the difficulty level
function getDifficultyLevel() {
  let diffLevelQuestion = "Enter difficulty level (1 or 2)";
  console.log(diffLevelQuestion);

  return Number(prompt(diffLevelQuestion));
} // end getDifficultyLevel
```

```javascript
// A function to generate an operand based on the difficulty level
function getOperand(level) {
  if (level === 1) {
    // Generate a random number between 0 and 9 incl.
    return Math.floor(Math.random() * 10); // 0 <= n2 < 10
  } else {
    // Generate a random number between 10 and 99 incl.
    return Math.floor(Math.random() * 90) + 10; // 10 <= n2 < 100
  }
} // end getOperand
```

```javascript
// A function to receive the user's answer
function getResponse(problemInWords) {
  console.log(problemInWords);
  let userResponse = Number(prompt(problemInWords));
  console.log("User entered %d", userResponse); // this is for debug purposes
  return userResponse;
} // end getResponse
```

```javascript
// A function to compute the correct answer
function getCorrectAnswer(op1, op2, op){
  let correctAnswer;

  // Compute the correct answer for the operation
  if (op === "+") {
    correctAnswer = op1 + op2;
  } else if (op === "-") {
    correctAnswer = op1 - op2;
  } else if (op === "*") {
    correctAnswer = op1 * op2;
  } else {
    correctAnswer = op1 / op2;
  }

  return correctAnswer;
} // getCorrectAnswer
```

The above functions can be assembled together into a single modular program available here.

BLANK PAGE

13. Client-side JavaScript

So far in this manual we have focused on the main features of JavaScript which are used to support sequence, selection and iteration. These features combine to make up what is commonly referred to as *core JavaScript*. While core JavaScript is important, it only presents a limited view of the language.

The real power and purpose of JavaScript only begin to emerge when it is used to create dynamic and interactive web pages and websites. The term *client-side JavaScript* is used to describe the features of JavaScript which make these types of websites possible.

In this section of the manual we will explore client-side JavaScript but before we start let's just clarify what is meant by the two terms *dynamic web page* and *interactive web page*?

- A dynamic (as opposed to static) web page is a page whose contents and appearance can change as it is being used. Most modern websites contain some pages with dynamic content. This means that content can be different each time you visit the same page; or it may be that the content is changed '*on-the-fly*' while you are looking at a page.

- An interactive website is one which supports user interactivity e.g. playing a game, selecting a product to purchase or entering and submitting data such as a search term, credit card details, or uploading an image to Instagram or Facebook.

Dynamic and interactive websites are made possible by virtue of two technologies – DOM and events. The remainder of this section explores client-side JavaScript support for both.

> **Client-side programming**
> The JavaScript language contains a number of key features that makes it possible to develop dynamic and interactive websites.
>
> These features combine to make up what is commonly referred to as *client-side JavaScript* and the act of writing programs that makes use of the client-side features of JavaScript is called *client-side programming*.

Dynamic web pages – some examples

The simplest way to *inject* dynamic content into a web page from a JavaScript program is to use `document.write`. Simply type the statement and pass in the string you want to see on the web page as an argument.

The examples below illustrate the use of `document.write` to create dynamic web page content. The JavaScript code is shown on the left and a screenshot of the resulting web page is shown on the right.

```
document.write("Hello World");
```
Hello World

The next example illustrates that HTML code can be passed into `document.write.` The `<br>` tag causes a new line to be rendered in the browser window before the second string is displayed.

```
document.write("Hello World<br>");
document.write("Hello World");
```
Hello World
Hello World

In this next example we see the text is marked up as a level 1 heading with the `h1` tag

```
document.write("<h1>Hello World</h1>");
```
Hello World

The next example prompts the user to enter their name and displays a personalised greeting message. The name entered – in this case *Joe* - is marked up in bold.

```
let userName = prompt("Enter your name");
document.write("Hello <b>" + userName + "</b>");
```
Hello **Joe**

PROGRAMMER TIP
Please note that the use of `document.write` is not recommended – we are just using it here to demonstrate the idea of dynamic web page content.

This next example makes use of a user-defined function called `getTimeMsg` to generate a different greeting message depending on the current time. The current time is read from the system using `Date()`. The JavaScript code is included inline between the opening and closing `script` tags in the HTML code as shown.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Display the time</title>
    <meta charset="utf-8">
  </head>
  <body>
    <!- Inline JS for this web page -->
    <script>
      let timeMsg = "The current date and time is " + Date() + "<br><br>";
      document.write(timeMsg); // not recommended!
      let greeting = getTimeMsg();
      document.write(greeting); // not recommended!

      function getTimeMsg() {
        let msg;
        const time = new Date().getHours();
        if (time < 6) {
          msg = "Before 6am";
        } else if (time < 12) {
          msg = "Good morning";
        } else if (time < 18) {
          msg = "Good afternoon";
        } else {
          msg = "Good evening";
        }

        return msg;
      } // end getTimeMsg

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

When the program is run your browser should display a greeting message that looks something like this:

> The current date and time is Mon May 24 2021 23:28:26 GMT+0100 (Irish Standard Time)
>
> Good evening

JS ***Challenge!***
Modify the code so that the message, 'Good night' gets displayed after 10pm

The Document Object Model (DOM)

The Document Object Model (or DOM for short) provides a means for programmers to inject dynamic content into web pages. It is the recommended alternative to using `document.write` which should not be used because when a program calls `document.write` after the web page is loaded the *entire* page gets overwritten which we do not want to happen!

The DOM is a data structure used to represent web pages. To understand the DOM more fully it is useful to consider web pages from the perspectives – firstly from the perspective of the HTML author and secondly from the perspective of the the end-user.

HTML perspective of a web page

This is the view pf a web page as seen by the HTML/CSS author. Every web page is a representation of a HTML file or document. The HTML and/or CSS file(s) must be changed in order to change the content and/or appearance of the web page.

An example HTML file is shown below.

```html
<!DOCTYPE html>
<html>
   <head>
      <title>A simple web page</title>
   </head>
   <body id="body-id">
      <h1>Level 1 heading</h1>

      <ul>
         <li>Hats</li>
         <li>Flags</li>
         <li>Scarves</li>
         <li>Headbands</li>
      </ul>

      <h2>Level 2 heading</h2>
      <p id="para-id">An ordinary paragraph ...</p>
      <br>
      <p>This <b>bold <i>and italic</i></b> text</p>

      <script>

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

KEY POINT: In order to make changes to the content and/or appearance of a web page the HTML/CSS author must change the HTML/CSS code.

End-user view

This is the view of the page as seen by the end-user when they browse to the page. End-users typically have no interest in, and are often not even aware of the HTML/CSS or JavaScript code that is behind the pages they visit.

When a browser loads a page it parses the HTML and renders it according to the rules set out in the HTML specification. Over the years, programmers have interpreted these rules in different ways and this has led to different HTML implementations in different browsers. This is one of the main reasons why the exact same HTML may look different on different browsers.

The HTML code on the previous page is rendered to an end-user as shown here to the right.

Level 1 heading

- Hats
- Flags
- Scarves
- Headbands

Level 2 heading

An ordinary paragraph ...

This **bold** *and italics* text

Programmer's view

This is the view of a web page as seen by the web application developer. When a browser loads a page it constructs an internal memory representation of that page known as the DOM. DOM stands for Document Object Model. The illustration below depicts a simplified DOM representation of the HTML code shown on the previous page.

A DOM tree is a browser's representation of a web page
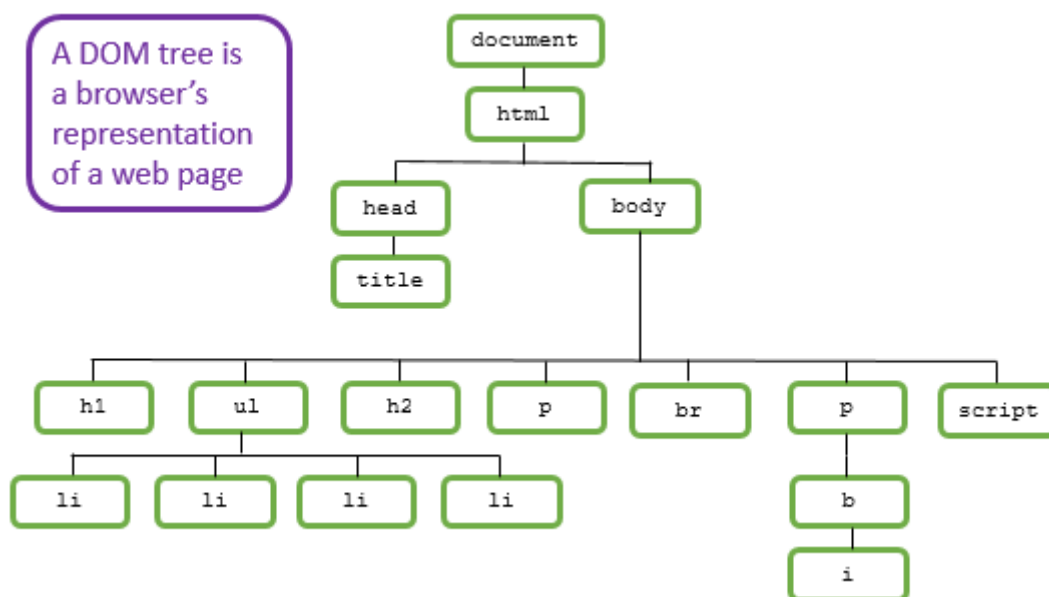
Because this representation resembles an upside down tree, it is referred to as the DOM tree. Every DOM tree is made up of a collection of **nodes** starting with a single **root node** called `document`.

Notice in the illustration at the bottom of the previous page that there is a separate node on the DOM tree for every element in the HTML code on which it is based. These nodes which are highlighted in green are called element nodes. DOM trees can also contain nodes to represent text, attributes and comments but these are all omitted from the diagram for the sake of simplicity and clarity.

KEY POINT: Before a browser can render a page, it builds a DOM tree by parsing the HTML markup.

As can be seen from the illustration, the DOM tree is very hierarchical in nature. It is useful to understand some terminology that describes the relationship between the various nodes. The terms **parent**, **child** and **sibling** are particularly important.

Nodes that appear at the same level on the DOM tree are called sibling nodes. Sibling nodes share the same parent – or to put it another way - each sibling is said to be a child of the same parent.

The tree shown to the right here is a sub-section of the tree shown on the previous page. We can see that the `html` element node contains two children – `head` and `body`. These are siblings of one another.
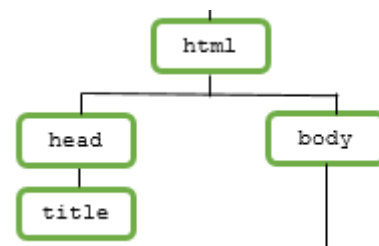

The node for the `head` element contains only one child node – this is an element node for `title`.

DOM nodes can be related to each other as parents, children and siblings

The node for the `body` element contains seven direct children – these are the nodes for the elements that appear inside the body section in the HTML code i.e. `h1, ul, h2, p, br, p` and `script`.

In the HTML code, the unordered list contains four list items. This is reflected in the DOM tree where the element node for `ul` has four children i.e. one child element node for each of the `li` elements. The three views are shown below:

```
<ul>
 <li>Hats</li>
 <li>Flags</li>
 <li>Scarves</li>
 <li>Headbands</li>
</ul>
```
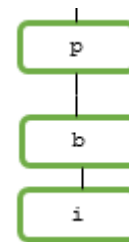
HTML View

DOM view

User View
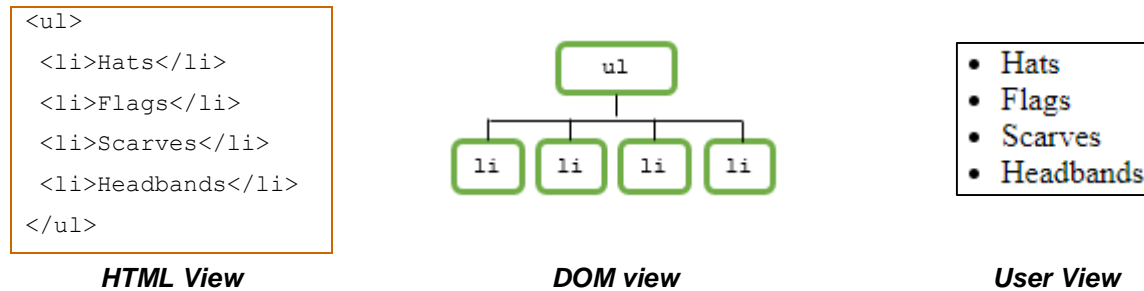
Similarly, the markup relating to the second paragraph tag can be viewed in three ways. First there's the HTML – notice the paragraph tag contains a `bold` tag which itself is marked up in italics using the `i` tag.

```
<p>This <b>bold <i>and italic</i></b> text</p>
```

This HTML is rendered as:

This **bold *and italics*** text

DOM view

The Magic of DOM
To uncover the magic of the DOM we ask the question:
What happens to a web page when the DOM changes?

To answer this question, we need to be aware of one vital piece of information and that is browsers always keep the web page on display in sync with the DOM (automatically).

Therefore, when the DOM changes the browser automatically updates its display to reflect these changes. We will see in the next section how the DOM API can be used by programmers to change DOM trees. The implication of this is that programmers can have full runtime control over the content of web pages.

In short, the DOM makes dynamic web content possible.

The DOM Application Programming Interface (API)

In the previous section we learned that every web page is represented as a DOM tree and that changes to the DOM tree are automatically updated in the browser's display. So for example, if a new paragraph element is somehow added to the DOM tree then the end-user's view displayed in the browser's window will also be updated to reflect this change.

In this section we will look at how the JavaScript DOM APIs can be used to access and change DOM trees, thereby dynamically updating the web pages they represent.

KEY POINT: Browsers keep their display in sync with the DOM. So, when the DOM tree changes the browser automatically updates its display to reflect these changes.

The DOM API is a specification of how programs should interact with the DOM. It lists the names of functions and properties that can be used by programs to navigate and manipulate every aspect of a DOM tree.

The good news is that client-side JavaScript comes with a full implementation of the DOM API. What this essentially means is that JavaScript has a full suite of built-in functions and properties that can be used to manipulate DOM trees. For example,

- the function `createElement` can be used to create a new element node.
- the function `addChild` can be used to add a node at a specific point in the DOM tree.
- the function `removeChild` can be called to delete a node from the DOM tree.
- the function `getElementById` can be used to retrieve an element node from the DOM tree.
- the properties `textContent` and `innerHTML` can be used to change the text content of any node. These properties could potentially be used to change the contents of a heading or a paragraph or a button or any control that can contain text.

In the next section we will look at example code that uses these APIs

TEACHER TIP
A browser's built-in DOM editor is a great way to demonstrate how changes to the DOM tree are immediately updated in the browser's display window.

DOM Example 1

In this example we will dynamically add a single `h1` element to a web page.

```html
<! You should rename this file index.html to run it -->

<!DOCTYPE html>
<html>
  <head>
    <title>DOM Example 1</title>
  </head>

  <body id="body-id">

    <script>

      // create a new h1 element
      let h1Elem = document.createElement("h1");
      // set the h1 element's text content
      h1Elem.textContent = "This is a level 1 heading!";

      // insert the new h1 element as a child of body
      let bodyElem = document.getElementById("body-id");
      bodyElem.appendChild(h1Elem);

    </script>
  </body>
</html>
```
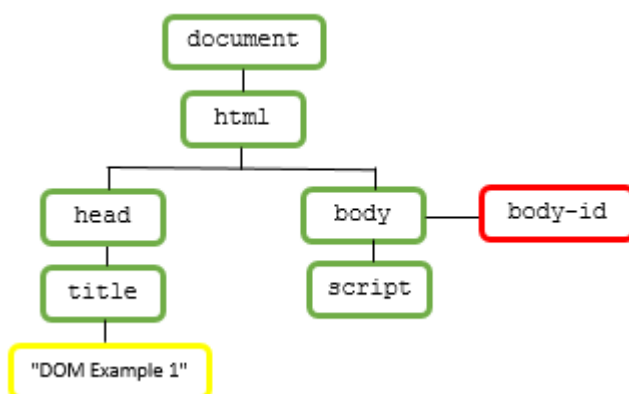
The initial DOM tree representation for the HTML is shown below. This is the representation *before* the JavaScript code is executed. At this point, if the page was rendered in a browser it would be blank.

The green nodes are all used to represent HTML elements. Red and yellow are used to represent attribute and text nodes respectively.

Note also that the `id` attribute on the `body` tag has a value of *body-id*. This will be used in the JavaScript code when it calls the function `getElementById`.

The illustration below depicts the DOM representation *after* the JavaScript code is executed. As you can see the `h1` element has been added to the DOM tree by the JavaScript code.
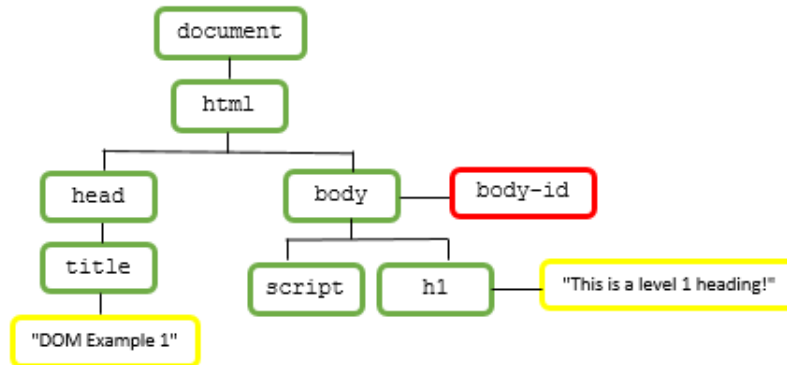
Because the original HTML did not contain any `h1` element we say the heading has been added *dynamically*. This has been done by using JavaScript to change the structure of the DOM tree. Specifically, the JavaScript code creates a new `h1` element using the `createElement` API. It then calls `appendChild` to attach the new element to the DOM tree as the last child of the `body` element. To do this the program must first know where the body element is. This is done using `getElementById` passing in the `id` of the `body` element (`body-id`) as shown in the JavaScript code on the previous page.

> **PROGRAMMER TIP**
> The most straightforward way to retrieve a DOM element is to call `getElementById` passing in the element's HTML `id` attribute as an argument

In this example the function `getElementById` is used in order to retrieve the handle of the `body` element. The function works by finding an element whose `id` attribute matches the argument passed into it. In the HTML code for example the `id` attribute for `body` is *body-id*. Therefore, when *body-id* is passed as an argument into `getElementById` the DOM representation for `body` was returned.

> **PROGRAMMER TIP**
> `appendChild` inserts a node as the last child of the parent node on which it acts

DOM Example 2

This example demonstrates the use `querySelector` as an alternative to `getElementById` in order to retrieve a DOM element.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>DOM Example 2</title>
  </head>
<body id="body-id" class="body-class">

    <script>

    // create a new h1 element
    let h1Elem = document.createElement("h1");
    // set the h1 element's text content
    h1Elem.textContent = "This is a level 1 heading!";

    // insert the new h1 element as a child of body
    let bodyElem = document.querySelector("#body-id"); // or .body-class
    bodyElem.appendChild(h1Elem);

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

As was the case with the previous example, the code here demonstrates how an element can be dynamically added to a page. The `querySelector` API accepts an element's **selector value** as an argument - the value of either an element's `id` or `class` attribute are commonly used as selector values.

Notice in the above code that argument passed into querySelector is the same as the value of the id attribute for body preceded by a hash symbol (#) i.e. #body-id. The hash tag tells JavaScript that id is being used as the selector value.

To use the `class` attribute, the selector must be preceded by a dot as shown here. (This example assumes the `class` attribute for `body` has a value of *body-class*.)

```
let bodyElem = document.querySelector(".body-class");
```

PROGRAMMER TIP
`getElementById` and `querySelector` can be used interchangeably. The former is simple to use because no two elements can have the same `id`. The advantage of `querySelector` is that it can be used to target more elements. We will use `getElementById` as our default technique for accessing elements from within a running JavaScript program as it has been around longer and is therefore supported more in older browsers.

DOM Example 3

In this example we demonstrate how to insert *three* heading elements to the body section of a HTML page.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>DOM Example 3</title>
  </head>
  <body id="body-id">

    <script>

        let bodyElem = document.getElementById("body-id");

        let h1Elem = document.createElement("h1");
        h1Elem.textContent = "This is a level 1 heading!";
        bodyElem.appendChild(h1Elem);

        let h2Elem = document.createElement("h2");
        h2Elem.textContent = "This is a level 2 heading!";
        bodyElem.appendChild(h2Elem);

        let h4Elem = document.createElement("h4");
        h4Elem.textContent = "This is a level 4 heading!";
        bodyElem.appendChild(h4Elem);

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

Each time we want to add a new element we need to first create it, then set its content and finally put it on the DOM tree

Notice how in the examples so far the `textContent` property has been used to set the text of the heading elements that are being created. Strictly speaking this is incorrect as text nodes should be used to represented text in a DOM tree. The following code is considered better.

```
let bodyElem = document.getElementById("body-id");

let h1Elem = document.createElement("h1");
var textNode = document.createTextNode("Level 1
heading!");
h1Elem.appendChild(textNode);
bodyElem.appendChild(h1Elem);
```

Challenge!
Modify the above code so that the text for h2 and h4 is represented on text nodes (rather than as a `textContent` property)
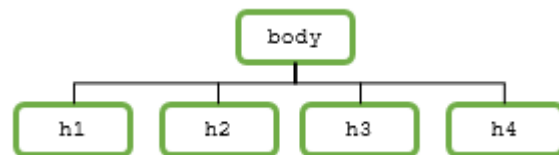
DOM Example 4

In this example we demonstrate how to create and insert a `h3` element between the `h2` and `h4` elements created in the previous example. By following this example you will learn a technique for dynamically inserting a new element between any two elements.

Assume that a DOM tree already exists and it has three elements - `h1`, `h2` and `h4` as depicted on left hand DOM tree below. We want to insert a `h3` element so that the DOM tree is changed as depicted on the right hand side.
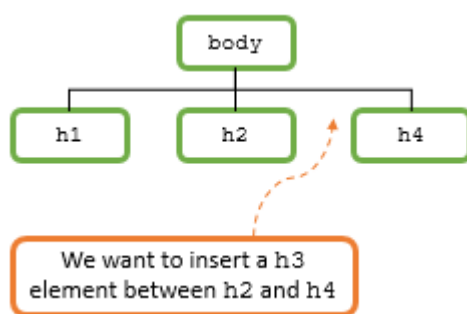
DOM tree before insertion   **DOM tree after insertion**

One way to achieve the requirement is to use `insertAdjacentElement` as shown in the code below. The call to `insertAdjacentElement` inserts the element referenced by `h3Elem` after the end of `h2Elem` in the DOM tree.

```
let h3Elem = document.createElement("h3");
h3Elem.textContent = "This is a level 3 heading!";
h2Elem.insertAdjacentElement('afterend', h3Elem);
```

```
<!-- beforebegin -->
<h2>
  <!-- afterbegin -->
  Level 2 heading!
  <!-- beforeend -->
</h2>
<!-- afterend -->
```

The valid values for the relative position argument are `beforebegin`, `afterbegin`, `beforeend` and `afterend`. These are depicted above.

Challenge!
Before clicking on octocat to see the full code listing for this example, see if you can can figure out by yourself the code that should precede the code that is shown.

Hint: You will need to start by retrieving a handle for the body element, then create the h1, h2 and h4 elements along with their text content.

DOM Example 5

This example demonstrates how to generate an unordered list 'on-the-fly' using JavaScript.

The JavaScript code is provided below (you will need to hook this in with the HTML file). The content of each list is taken from an array of strings called `strs` which is declared at the top.

```
<script>
    let strs = [ "Milk", "Bread", "Butter", "Crisps" ];

    // Retrieve the body element - we will attach the ul element to this later
    let bodyElem = document.getElementById("body-id");

    // Create an element for ul
    let unorderedList = document.createElement("ul");
    for (let i in strs) {

      let listItem = document.createElement("li");
      //anchor.innerText = strs[i];
      listItem.textContent = strs[i];
      unorderedList.appendChild(listItem);
    }

    // Attach the ul element to the body
    bodyElem.appendChild(unorderedList);
</script>
```
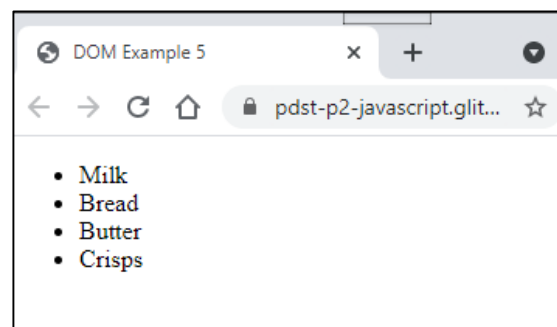
The illustration below depicts a section of the DOM tree along with the actual list as it is rendered by the browser.
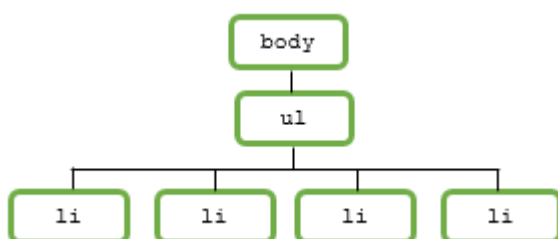
Experiment!
- *See if you can achieve the same result using the same lines of code but arranged in a slightly different order*
- *See if you can create another type of list e.g. an orderd list or a definition list*
- *How would you adapt the code to create a list of paragraphs?*

Events

An event is a programming term used to describe something that happens while a program is running. In the context of web development events occur continually when we are interacting with websites through our browser.

Events are typically (but not always) initiated by an end-user e.g. a user clicks on a mouse button to submit a form, and have very specific JavaScript names e.g. `click`. When an event is initiated it is said to be *raised* (or *fired*).

From a programming perspective there are two aspects to events – *handlers* and *listeners*.

An event handler is the code in your program that will be run when an event is fired. Handlers are usually implemented as JavaScript functions in your program.

In order to ensure your handler gets called when its event gets fired you first need to register it with the JavaScript sub-system. **Once a handler has been registered JavaScript will 'listen' for the event it is registered to handle**. When the event is raised the handler will be called.

HTML/CSS, JavaScript code and the JavaScript/browser sub-system technologies combine together to form an architecture for event handling which is depicted in the illustration below.
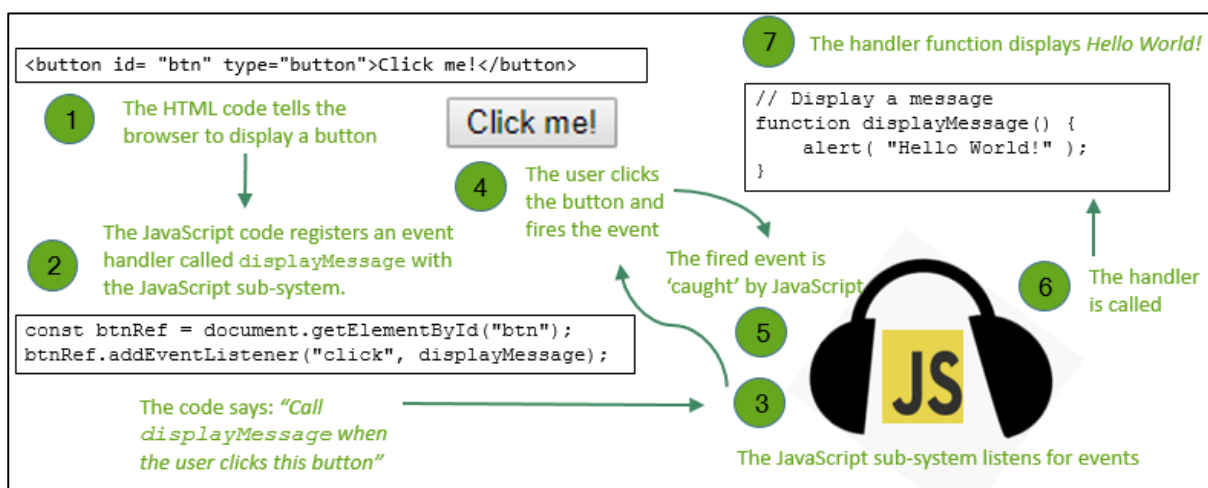
The previous illustration depicts how events are processed by JavaScript for a simple button click. When the button is clicked an event is fired and the handler displays the message *Hello World!* in an alert popup.

This processing is explored in more detail using the code shown below.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My first event handler</title>
  </head>
  <body id="body-id">
    <button id="btn-id" type="button">Click me!</button>

    <script>

      // 1. Retrieve the button handle
      const btnRef = document.getElementById("btn-id");

      // 2. Attach an event click handler to the time button
      btnRef.addEventListener("click", displayMessage);

      // 3. Handler for time button
      function displayMessage() {
          alert("Hello World!")
      }

    </script>

  </body>
</html>
```

The name of the handler is displayMessage and the event it handles is called 'click'

There are *three* key steps:

1. The button tag tells the system to display a button with the text *Click me!*. Note that the `id` attribute for the button in *btn-id*.

2. The JavaScript code registers an event handler called `displayMessage` for the button. The handler is said to be attached to the button. In this example, the type of event handler created is called a `click` event handler. It responds to user mouse clicks.

3. When the button is clicked by the user the handler is called. This in turn displays an alert box with the text *Hello World!*

Because there can be an indefinite amount to time between steps 2 and 3 (the user may *never* click the button!) this type of processing is called **asynchronous event handling**. The system is primed to listen out for click events on the button, and if one happens it then calls the function `displayMessage`.

When the code is run the browser displays a window that looks something like this:
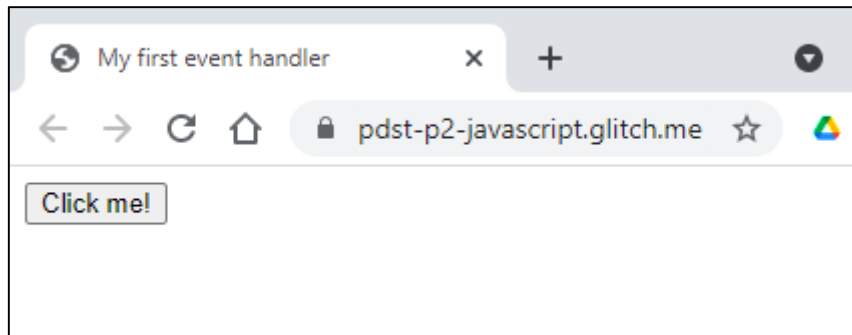
When the user clicks on the button the browser responds by running the handler (i.e. `displayMessage`). The popup is displayed with the message *Hello World!* as shown here.

KEY POINTS:
- ✓ Event handlers are user-defined functions that are invoked when an event is fired
- ✓ Handlers need to be registered for specific events e.g. `click`
- ✓ You can register a handler with the JavaScript API `addEventListener`

PROGRAMMER TIP
Use `addEventListener` as the preferred method to register event listeners.

We will now explore some more examples of how to register event handlers.

Events Example 1

In the example we write a handler to display the current time in response to a user clicking a button on a web page.

The page initially contains a button with the text *Click to see the current time*. There is also a paragraph with the text *PLACEHOLDER* directly beneath the button.

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Event Example 1</title>
  </head>
  <body id="body-id">

    <button id="time" type="button">Click to see the current time</button>
    <p id="demo">PLACEHOLDER</p>

    <script>

    // Attach a handler to the time button
    const timeBtn = document.getElementById("time");
    timeBtn.addEventListener("click", dispTime);

    // Handler for time button
    function dispTime() {
        document.getElementById("demo").textContent = Date();
    }

    </script>
  </body>
</html>
```
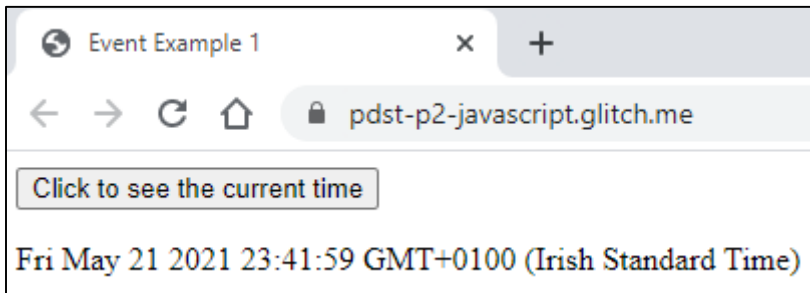
PROGRAMMER TIP
Unless submitting form data to a server it is recommended to set the `type` attribute of `button` controls to *button*

When the user clicks on the button the current time will be rendered as a HTML paragraph situated directly underneath the button on the page as shown here.

Event Example 1

pdst-p2-javascript.glitch.me

Click to see the current time

Fri May 21 2021 23:41:59 GMT+0100 (Irish Standard Time)

Notice from the code that:

a) The `id` attribute of the button is *time* – this is marked in red.

b) The `id` attribute of the paragraph element is *demo* – this is marked in blue. The idea here is to use this a as a placeholder by the JavaScript code to use in order to display the time when the button is clicked.

c) The name of the event handler is `dispTime` – this is highlighted in yellow.

Experiment!
Instead of using `timeBtn.addEventListener("click", dispTime);` to add a listener try any of the following ....
```
timeBtn.addEventListener("dblclick", dispTime);
timeBtn.addEventListener("mousedown", dispTime);
timeBtn.addEventListener("mouseover", dispTime);
```

Events Example 2

In the previous example the time is always displayed in the same position on the page. This means that when the button is clicked, the previously displayed time is overwritten with the new value for time.

In this example the times appear underneath each other as new paragraphs on the page. This is depicted below.

Click to see the current time	Click to see the current time
Sat May 22 2021 00:07:04 GMT+0100 (Irish Standard Time)	Sat May 22 2021 00:07:04 GMT+0100 (Irish Standard Time)
Sat May 22 2021 00:06:43 GMT+0100 (Irish Standard Time)	Sat May 22 2021 00:06:43 GMT+0100 (Irish Standard Time)
Sat May 22 2021 00:06:46 GMT+0100 (Irish Standard Time)	Sat May 22 2021 00:06:46 GMT+0100 (Irish Standard Time)
Sat May 22 2021 00:06:47 GMT+0100 (Irish Standard Time)	Sat May 22 2021 00:06:47 GMT+0100 (Irish Standard Time)
	Sat May 22 2021 00:07:04 GMT+0100 (Irish Standard Time)

| **Before click** | **After click** |

The effect is achieved by modifying the event handler, `dispTime`, from the previous example to the one shown in the listing below.

```
// Display the time
function dispTime() {
    let dateTimeStr = Date();
    document.getElementById("demo").textContent = dateTimeStr;

    // Add the time as the last child of body
    let bodyElem = document.getElementById("body-id");
    let newPara = document.createElement("p");
    newPara.textContent  = dateTimeStr;
    bodyElem.appendChild(newPara);
}
```

The handler calls `createElement` to construct a new DOM paragraph element. The paragraph text is set using its `textContent` property. Finally, `appendChild` is called to insert the paragraph element as child element of the body. The paragraph is placed on the DOM tree at the end of all elements that appear inside the element `body`.

Events Example 3

The screenshot shown below illustrates the idea of the next example.

Text Entry Demo

Enter some text [|] [OK]

I heard the news today...

... oh boy!

When the user types text into the entry field and clicks `OK` the text is displayed on the web page. Each piece of new text is appended as a new paragraph under a `div` section which is declared inside the body of the HTML code.

The HTML and JavaScript code for this page is shown below.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Event Example 3</title>

  </head>
  <body>
    <h1>Text Entry Demo</h1>

    <label for="txt-field">Enter some text</label>
    <input type="text" id="txt-field">
    <button id="ok-btn" type="button">OK</button>

    <!-- an empty div section as a structural placeholder for text -->
    <div id="result-div"></div>

    <script>

      const textField = document.getElementById('txt-field');
      const okBtn = document.getElementById('ok-btn'); // retrieve the OK button
      okBtn.addEventListener('click', buttonClicked); // attach the handler

      // Handler code to execute when the OK button gets clicked
      function buttonClicked() {

        // Add the entered data as the last child of the result division
        let divElem = document.getElementById("result-div");
        let newPara = document.createElement("p");
        newPara.textContent = textField.value;
        divElem.appendChild(newPara);

        textField.focus(); // set the focus back to the text field
        textField.value = ""; // clear the contents of the text field
      }

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

> **PROGRAMMER TIP**
> The pairing between the label's `for` attribute and the input's `id` attribute enables users to select the input area by clicking anywhere on the label.

> **PROGRAMMER TIP**
> Placing structural elements such as `div` at strategic places in a HTML file can provide your JavaScript program with a quick and convenient way to access certain parts of the DOM tree.

Events Example 4

This example is pretty much the same as the previous one – this time instead of appearing as paragraphs, the text is rendered as list items in an unordered (i.e. a bulleted) list. Here's the JavaScript code – remember it needs to be hooked in with the HTML.

```javascript
        const textField = document.getElementById('txt-field');
        textField.focus(); // set the focus to the text field

        // Attach a listener to the ok button - in one step!
        document.getElementById('ok-btn').addEventListener('click', buttonClicked);

        function buttonClicked() {
           let dataEntered = textField.value;

           let listItem = document.createElement("li");
           listItem.textContent = dataEntered;

           // Add the entered data as the last child of an unordered list
           let unorderedList = document.getElementById("ul");

           // If the list doesn't exist create it and ....
           // ... insert it as a child of the result division
           if (unorderedList == null) {
              let divElem = document.getElementById("result-div");
              unorderedList = document.createElement("ul");
              divElem.appendChild(unorderedList);
           }

           unorderedList.appendChild(listItem);

           textField.focus();
           textField.value = "";
        }
```

> The first time the code is run it will create a **ul** element and attach it to **div**

> **PROGRAMMER TIP**
> Use the `value` property to retrieve or set the value of a field on a form

Explanation: Each time the user clicks the OK button a new list item element (`li`) is appended as a child of the `ul` element. Each `li` is contained by the same `ul` parent. Of course, in order to add each `li` to a `ul`, the `ul` itself must exist. The code ensures that the `ul` exists by testing whether its handle is `null` or not. If the `ul` handle is `null` then the `ul` is created and added to the `div` section of the HTML document.

The behaviour of the code when the user types *HTML* into the entry field is illustrated below.

List Demo

Enter some text [   ] OK

- Python
- JavaScript

List Demo

Enter some text [   ] OK

- Python
- JavaScript
- HTML
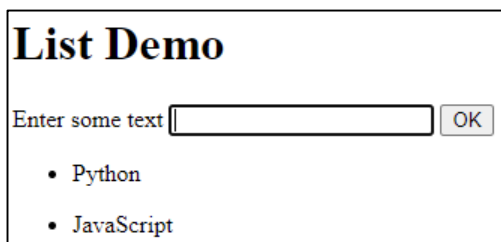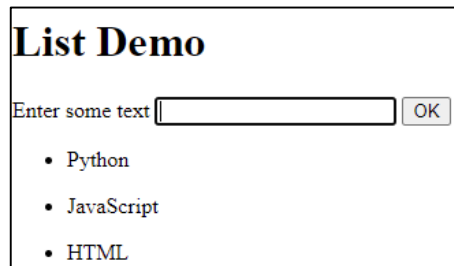
Before user clicks OK   *After user enters "HTML" and clicks OK*

Events Example 5

This example is a 'number guessing game' – the initial HTML page is based on the previous two examples. When the game is started the browser looks something like this.

The JavaScript code generates a random integer between 1 and 10 inclusive and every time the user submits a guess, the program responds with one of two feedback messages – either *Congratulations!* or *Wrong!* – depending on whether the number entered was correct or not.

The HTML code below declares a placeholder called `rightOrWrong` which can be used in the JavaScript code (shown on the next page) to add the appropriate feedback message.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Event Example 5 </title>
  </head>
  <body>
    <h1>Number guessing game</h1>
    <p>Guess a number between 1 and 10 (incl.)</p>
    <label for="guessField">Enter a guess: </label>
    <input type="text" id="guessField">
    <button id="submit-btn" type="button">Submit guess</button>
    <p id="rightOrWrong"></p>

    <script>

      <!-- See next page for JavaScipt -->

    </script>

  </body>
</html>
```

> The `rightOrWrong` identifier in the HTML file gives the JavaScript code a location in the DOM tree to display the feedback message

The JavaScript code is fairly straightforward. A handler called `checkGuess` is attached to the submit button. The handler is invoked when a 'click' event is fired. This happens only when the user clicks the submit button.

Most of the processing is done inside the handler – the code reads the value entered by the user and checks to see if it is the same as the computer's secret number. The appropriate feedback message (i.e. *Congratulations* or *Wrong!*) is then set as the text content on the page's paragraph element which is used to display the feedback. (Recall from the HTML on the previous page that this 'feedback paragraph' is identified by the value *rightOrWrong*)

```javascript
// pick a random number between 1 and 10
let secretNumber = Math.floor(Math.random() * 10) + 1;

// Setup an event handler for the submit button
const submitBtn = document.getElementById("submit-btn");
submitBtn.addEventListener('click', checkGuess);

// Handler - check the guess
function checkGuess() {
  const feedback = document.getElementById('rightOrWrong');
  const guessField = document.getElementById('guessField');

  // Read the user's guess
  let userGuess = Number(guessField.value);

  if (userGuess === secretNumber) {
    feedback.textContent = 'Congratulations!';
  } else {
    feedback.textContent = 'Wrong!';
  }

  guessField.value = ''; // blank the field
  guessField.focus(); // give it the focus
}
```

> **Suggest _three_ enhancements that could be made to the 'number guessing game' system before proceeding to the next breakout activity.**

1.

2.

3.

BREAKOUT ACTIVITY 2

In the tasks that follow, you are required to make a number of enhancements to the 'number guessing game' program presented as example 5 in the previous section. Each task builds on the code from the previous - so for Task 1 you should start by making a copy of the code from the last example (i.e. events example 5 – number guessing game)

Most of the code you need to complete each task is provided – you just need to figure out what it does, how to use it and where to put it in order to complete the task.

Task 1

a) Add a checkbox to the page to display/hide the number that the user is trying to guess i.e. the computer generated number which is stored in the variable `secretNumber`.

A check box has
been added here
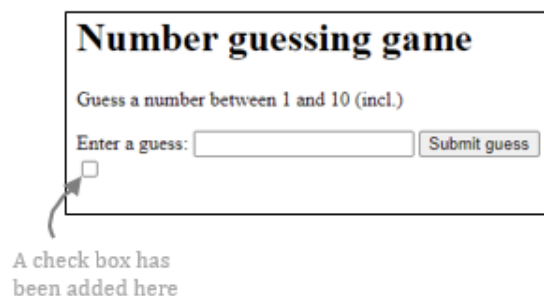
An example of how the checkbox should behave is shown below. The left hand side shows the situation when the user clicks on the checkbox on - the secret number is displayed as the text content of the checkbox's label. The checkbox acts as a toggle so when the user switches the checkbox off the secret number is hidden. This is depicted on the right.
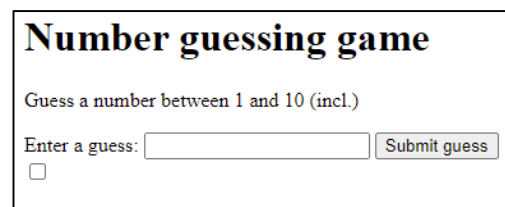
When the user checks the box
the secret number is displaued

PDST
Professional Development | An tSeirbhís um Fhorbairt
Service for Teachers | Ghairmiúil do Mhúinteoirí

HTML CSS JS
5 3 J

b) Change the style property of the feedback message so that it is displayed in green if the guess entered by the user is correct and red otherwise. These two scenarios are illustrated below.

Number guessing game

Guess a number between 1 and 10 (incl.)

Enter a guess: [   ] [Submit guess]
☑ 5

Congratulations! You got it right!

Number guessing game

Guess a number between 1 and 10 (incl.)

Enter a guess: [   ] [Submit guess]
☑ 5

Wrong!

The code you need to complete this task is provided in a number of separate code blocks shown below. All you have to do is figure out where to incorporate each block into your own code base. (Remember your code base is the code for events example 5 from the previous section.)

CODE BLOCK 1

```
    feedback.style.backgroundColor = 'red'; // in need of a home!
```

CODE BLOCK 2

```
function toggleNumber() {
    const computerNumber = document.getElementById('computerNumber');
    if (checkbox.checked) {
      computerNumber.style.visibility = 'visible';
      computerNumber.textContent = secretNumber;
    } else {
      computerNumber.style.visibility = 'hidden';
    }
}
```

CODE BLOCK 3

```
    <br>
    <input id="showNumberCkBx" type="checkbox">
    <label id="computerNumber"></label>
```

CODE BLOCK 4

```
    feedback.style.backgroundColor = 'green'; // help – I'm lost!
```

CODE BLOCK 5

```
let checkbox = document.getElementById("showNumberCkBx");
checkbox.addEventListener('click', toggleNumber);
```

Task 2

For this task you are required to create a results section in your page. The purpose of the results section is to display the following information:

a) A list of all the user's previous guesses

b) A helpful message to tell the user whether their guess was too low, too high or just right

c) A statistic at the end with the total number of guesses (you will need to declare and maintain a variable to keep track of the user's guesses – call it `guessCount`)

The screenshots below show the required behaviour.

When the system starts it should look something like this.

Notice there has been a paragraph added with the text *Previous Guesses:*.

The user clicks on the checkbox to reveal the secret number (in this case 3). Now let's say the user enters two values - the first 2 is too low and the second 4 which is too high.

The user guesses 4 which is too low   The user guesses 4 which is too high

Notice that the incorrect guesses are displayed in an unordered list underneath the previous guesses paragraph.

Finally, when the user enters the correct number a screen similar to the one shown below is displayed. Notice the last line shows the total number of guesses the user made.

Number guessing game

Guess a number between 1 and 10 (incl.)

Enter a guess: [   ] [Submit guess]

Congratulations! You got it right!

☑ 3

Just right!

Previous Guesses:

- 2
- 4
- 3

You took 3 guesses
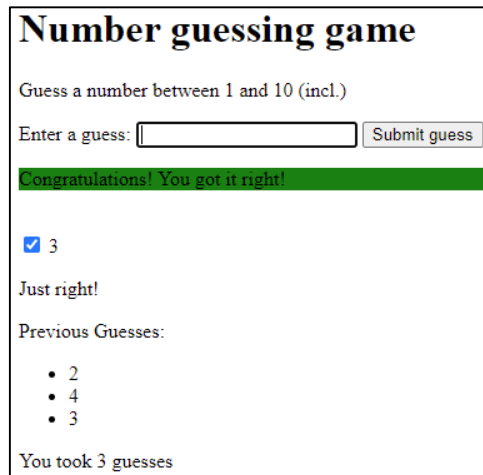
The blocks of code you will need are provided below in no particular order – you need to place them at the correct locations in the code resulting from Task 1. The solution for task 1 is available in the appendix.

CODE BLOCK 1

```
<div id="results">
  <p id="lowOrHi"></p>
  <p>Previous Guesses:</p>
  <ul id="prevGuesses"></ul>
  <p id="stats"></p>
</div>
```

Note: the position on the screen where the feedback messages are displayed may be slightly enhanced by *moving* the paragraph placeholder for `rightORWrong` directly before the `lowOrHi` paragraph (inside the `div` block) in this code block. That is ..

```
<div>
  <p id="rightOrWrong"></p>
  <p id="lowOrHi"></p>
```

CODE BLOCK 2

```
// Display each user's guess in the division for prevGuesses
let unorderedList = document.getElementById("prevGuesses");
let newListItem = document.createElement("li");
newListItem.textContent = guessField.value;
unorderedList.appendChild(newListItem);
```

CODE BLOCK 3

```
// Display a helpful message to the user
let helpfulMsgField = document.getElementById("lowOrHi");
if (userGuess < secretNumber) {
    helpfulMsgField.textContent = 'Too low!' ;
} else if (userGuess > secretNumber) {
    helpfulMsgField.textContent = 'Too high!';
} else if (userGuess === secretNumber) {
    helpfulMsgField.textContent = 'Just right!';
}
```

CODE BLOCK 4

```
function displayStats() {
    // Display the number of guesses
    let statsPara = document.getElementById("stats");
    statsPara.textContent = "You took "+guessCount+" guesses";
}
```

CODE BLOCK 5

```
      let guessCount = 0;
```

CODE BLOCK 6

```
      guessCount++;
```

CODE BLOCK 7

```
    displayStats(); // call the function to display the number of guesses
```

Task 3

Add 'Game Over' processing. For this task you are required to add a 'New game' button to your page. The two screenshots below illustrate how the system should behave before and after the button is clicked.

The first screen depicts a normal run of the game – the user has just guessed the correct number on the third attempt. Notice that the New game button is enabled because the game has ended.

Before the New game button is clicked

The next screen on the right displays what happens when the New game button is clicked – the screen is cleared and the system returns to its initial state ready to play a new game.

After New game button is clicked

The code blocks you will need are provided below. All you need to do is arrange them into the correct places. Remember to use the solution to the previous task as your code base.

CODE BLOCK 1

```
const newGameBtn = document.getElementById('newgame-btn');
newGameBtn.addEventListener('click', newGame);
```

CODE BLOCK 2

```
<button id="newgame-btn" type="button">New game</button>
```

CODE BLOCK 3

```
function newGame() {
  secretNumber = Math.floor(Math.random() * 10) + 1;
  guessCount = 0;

  submitBtn.disabled = false;
  newGameBtn.disabled = true;

  document.getElementById('computerNumber').textContent = "";
  document.getElementById("showNumberCkBx").checked = false;
  document.getElementById('rightOrWrong').textContent = "";
  document.getElementById("lowOrHi").textContent = "";
  document.getElementById("prevGuesses").innerHTML = "";
  document.getElementById("stats").textContent = "";
}
```

CODE BLOCK 4

```
function gameOver() {
  submitBtn.disabled = true;
  newGameBtn.disabled = false;
  displayStats();
}
```

CODE BLOCK 5

```
    // Start a new game
    newGame(); // Task 3
```

CODE BLOCK 6

```
        gameOver(); // Task 3
```

Hint: You will need to include this final code block to call the `gameOver` function somewhere inside the function `checkGuess`.

Task 4

In this task you will add two spin buttons to provide users with the ability to set a minimum and maximum value for the secret number. The secret number generated by the program will be between the minimum and maximum values as decided by the user.
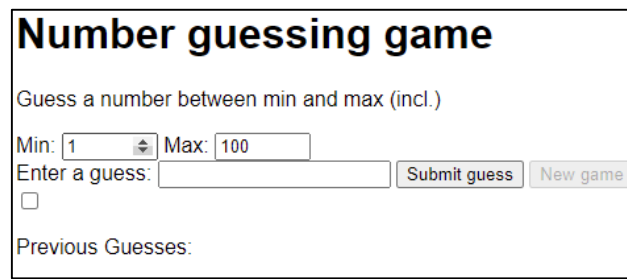
You will need to change the prompt on the screen from this ..

Guess a number between 1 and 10 (incl.)

.... to this ...

Guess a number between min and and max (incl.)
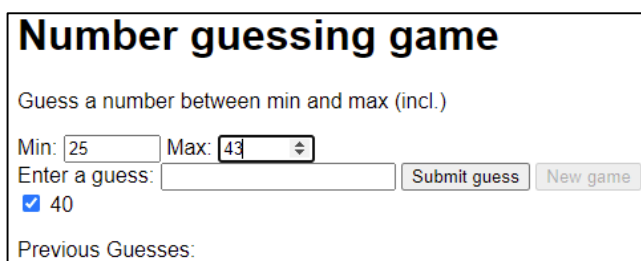
The initial screen in the final system should look something like this.

The initial values of the Min and Max spin button have been set to 1 and 100 respectively. When the mouse is floated over the spin buttons you will notice an up/down arrow appear to its right hand side. This arrow can be used to scroll up and down through different numbers in steps of 1. In this way the user can control the values for the minimum and maximum values without having to use the keyboard (i.e. by using the mouse only).

The screenshot below shows what the screen would look like if the user had configured the game to select a secret number in the range 25 to 43. The program generates a new secret number every time the user changes the range. This ensures that the secret number will always be within the required range.

The `secretNumber` generated should always be between the values selected in the spinners for min and max

The following code blocks contain all the code you will need in order to complete this task. The HTML code is used to change the *appearance* of the page and the JavaScript code is used to change the *behaviour* of the page.

Don't forget that this code should be layered over the solution to the previous task.

CODE BLOCK 1

```
// Create a listener for the two spinners (the same one will do them both)
document.getElementById('low').addEventListener('change', setSecretNo);
document.getElementById('high').addEventListener('change', setSecretNo);
```

CODE BLOCK 2

```
    <div>
      <label for="low">Min: </label>
      <input id="low" type="number" min="1" max="100" step="1" value="1">
      <label for="high">Max: </label>
      <input id="high" type="number" min="1" max="100" step="1" value="100">
    </div>
```

CODE BLOCK 3

```
// Handler for the spinners
function setSecretNo() {

  minValue = Number(document.getElementById('low').value);
  maxValue = Number(document.getElementById('high').value);
  secretNumber = Math.floor(Math.random() * (maxValue - minValue +1)) + minValue;

  // update the secret number on the display
  document.getElementById('computerNumber').textContent = secretNumber;

}
```

CODE BLOCK 4

```
  // This code needs to be grafted in the function 'newGame'
  //secretNumber = Math.floor(Math.random() * 10) + 1; // Commented out for task 4
  minValue = Number(document.getElementById('low').value);
  maxValue = Number(document.getElementById('high').value);
  secretNumber = Math.floor(Math.random() * (maxValue - minValue +1)) + minValue;
```

CODE BLOCK 5

```
    //let secretNumber = Math.floor(Math.random() * 10) + 1;
    let secretNumber; // Comment the preceding line and replaces with this
    let minValue, maxValue; // Task 4
```

Reflect on all the interactive tasks you just completed. Describe any situation where you would have used a different approach to complete any of the tasks? Are there any further enhancements you can think of making? Suggest how you could adapt any of the exercises for use in your own LCCS classroom.

Task 5 (Online Computer Aided Learning System, OCALS)

For this final task of the breakout session you are required to 'webify' the Computer Aided Learning (CAL) system that you developed in breakout 1. We will brand the new system, Online Computer Aided Learning System, or **OCALS** for short!

The requirement can be accomplished by grafting the code you have from 'number guessing game' with the completed code from the first breakout. Before starting you should have a think about what OCALS might look like when it is finished. You will need to plan and design your solution carefully before starting to implement it using HTML/CSS and JavaScript code. Use the design process diagram from the LCCS specification document and the questions below to scaffold your thinking.
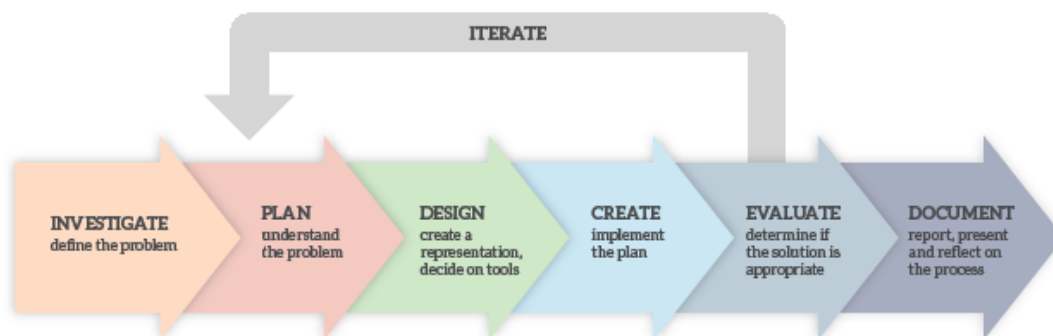
Start by scoping your system – what will it do? Put yourself in the position of the end-user. What functionality would the typical end-user like to have? Use this to determine what's in and what's out (and what's in between). You will also need to consider your data flows – what information is needed? What do I need to capture from the end-user? What data can the system generate?

By this stage it may be a good idea to sketch up a few ideas for the user interface (the screenshots shown below and on the next page might inspire some ideas – try to be original!). What will the system look like to the end-user? Design some use cases - these can be used as a basis for your test design. By now you should be ready to start low-level design. What UI controls/widgets will you use? What events will need to handled by your system? Which parts of your site will be static and which parts will be dynamic?

Once you have completed your requirements and design specifications you will be ready to move on to implementation and testing phase of the project. Good luck!

Online Computer Aided Learning System (OALS)

Example user interfaces / Inspiration ideas

Example User Interface #1

Example User Interface #2

Example User Interface #3

BLANK PAGE

Suggested Solution to Breakout #2

Breakout #2: Task 1 (abridged) and Suggested Solution

a) Add a checkbox to the page to display/hide the number that the user is trying to guess i.e. the computer generated number which is stored in the variable `secretNumber`.

Number guessing game

Guess a number between 1 and 10 (incl.)

Enter a guess: [   ] [Submit guess]

☐

A check box has
been added here

b) Change the style property of the feedback message so that it is displayed in green if the guess entered by the user is correct and red otherwise.

Recall that each solution is layered over the previous starting from the initial code base provided as the solution to example 5 – the number guestting game. The code highlighted in bold shows the additional code required for each task. We start with the HTML for task 1.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Guess Game - Task 1 Solution</title>
  </head>
  <body>
    <h1>Number guessing game</h1>
    <p>Guess a number between 1 and 10 (incl.)</p>
    <label for="guessField">Enter a guess: </label>

    <input type="text" id="guessField">
    <button id="submit-btn" type="button">Submit guess</button>
    <p id="rightOrWrong"></p>

    <!-- Task 1 -->
    <br>
    <input id="showNumberCkBx" type="checkbox">
    <label id="computerNumber"></label>

    <script>
      <!-- Copy the JavaScript code on the next page here -->
    </script>
  </body>
</html>
```

The JavaScript code for task 1 is now shown. Again, remember that the code highlighted in bold is the new code that was needed for this solution.

```html
<script>
  // pick a random number between 1 and 10
  let secretNumber = Math.floor(Math.random() * 10) + 1;

  // Setup an event handler for the submit button
  const submitBtn = document.getElementById("submit-btn");
  submitBtn.addEventListener('click', checkGuess);

  let checkbox = document.getElementById("showNumberCkBx"); // Task 1
  checkbox.addEventListener('click', toggleNumber); // Task 1

  // Handler - check the guess
  function checkGuess() {
    const feedback = document.getElementById('rightOrWrong');
    const guessField = document.getElementById('guessField');

    // Read the user's guess
    let userGuess = Number(guessField.value);

    if (userGuess === secretNumber) {
      feedback.textContent = 'Congratulations! You got it right!';
      feedback.style.backgroundColor = 'green'; // Task 1
    } else {
      feedback.textContent = 'Wrong!';
      feedback.style.backgroundColor = 'red'; // Task 1
    }

    guessField.value = ''; // blank the field
    guessField.focus(); // give it the focus
  }

  // Task #1
  function toggleNumber() {
    const computerNumber = document.getElementById('computerNumber');
    if (checkbox.checked) {
      computerNumber.style.visibility = 'visible';
      computerNumber.textContent = secretNumber;
    } else {
      computerNumber.style.visibility = 'hidden';
    }
  }

</script>
```

Breakout #2: Task 2 (abridged) and Suggested Solution

For this task you are required to create a results section in your page. The purpose of the results section is to display the following information:

a) A list of all the user's previous guesses

b) A helpful message to tell the user whether their guess was too low, too high or just right

c) A statistic at the end with the total number of guesses (you will need to declare and maintain a variable to keep track of the user's guesses – call it `guessCount`)

The HTML code is shown below.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Guess Game - Task 2 Solution</title>
  </head>
  <body>
    <h1>Number guessing game</h1>
    <p>Guess a number between 1 and 10 (incl.)</p>
    <label for="guessField">Enter a guess: </label>
    <input type="text" id="guessField">
    <button id="submit-btn" type="button">Submit guess</button>


    <!-- Task 1 -->
    <br>
    <input id="showNumberCkBx" type="checkbox">
    <label id="computerNumber"></label>

    <!-- Task 2 -->
    <div id="results">
      <p id="rightOrWrong"></p>
      <p id="lowOrHi"></p>
      <p>Previous Guesses:</p>
      <ul id="prevGuesses"></ul>
      <p id="stats"></p>
    </div>

    <!-- JavaScript -->
    <script>

      <!-- Copy the JavaScript code on the next page here -->

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

The JavaScript solution for task 2 is shown here.

```
<!-- JavaScript -->
<script>

  // pick a random number between 1 and 10
  let secretNumber = Math.floor(Math.random() * 10) + 1;

  // Setup an event handler for the submit button
  const submitBtn = document.getElementById("submit-btn");
  submitBtn.addEventListener('click', checkGuess);

  let checkbox = document.getElementById("showNumberCkBx"); // Task 1
  checkbox.addEventListener('click', toggleNumber); // Task 1

  let guessCount = 0; // Task #2

  // Handler - check the guess
  function checkGuess() {
    const feedback = document.getElementById('rightOrWrong');
    const guessField = document.getElementById('guessField');

    // Read the user's guess
    let userGuess = Number(guessField.value);

    guessCount++; // Task 2

    if (userGuess === secretNumber) {
      feedback.textContent = 'Congratulations! You got it right!';
      feedback.style.backgroundColor = 'green'; // Task 1
      displayStats(); // Task 2
    } else {
      feedback.textContent = 'Wrong!';
      feedback.style.backgroundColor = 'red'; // Task 1
    }

    // Task #2
    // Display a helpful message to the user
    let helpfulMsgField = document.getElementById("lowOrHi");
    if (userGuess < secretNumber) {
        helpfulMsgField.textContent = 'Too low!' ;
      console.log("LOW");
    } else if (userGuess > secretNumber) {
        helpfulMsgField.textContent = 'Too high!';
              console.log("LOW");
    } else if (userGuess === secretNumber) {
        helpfulMsgField.textContent = 'Just right!';
    }

    // Task #2
    // Display each user's guess in the division for prevGuesses
    let unorderedList = document.getElementById("prevGuesses");
    let newListItem = document.createElement("li");
    newListItem.textContent = guessField.value;
    unorderedList.appendChild(newListItem);
```

```javascript
        guessField.value = ''; // blank the field
        guessField.focus(); // give it the focus
    }


    // Task #2
    function displayStats() {
        // Display the number of guesses
        let statsPara = document.getElementById("stats");
        statsPara.textContent = "You took "+guessCount+" guesses";
    }



    // Task #1
    function toggleNumber() {
        const computerNumber = document.getElementById('computerNumber');
        if (checkbox.checked) {
          computerNumber.style.visibility = 'visible';
          computerNumber.textContent = secretNumber;
        } else {
          computerNumber.style.visibility = 'hidden';
        }
    }

</script>
```

Breakout #2: Task 3 (abridged) and Suggested Solution

Add 'Game Over' processing. For this task you are required to add a 'New game' button to your page. The two screenshots below illustrate how the system should behave before and after the button is clicked.

The HTML code is shown below.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Guess Game - Task 3 Solution</title>
  </head>
  <body>
    <h1>Number guessing game</h1>
    <p>Guess a number between 1 and 10 (incl.)</p>
    <label for="guessField">Enter a guess: </label>
    <input type="text" id="guessField">
    <button id="submit-btn" type="button">Submit guess</button>

    <!-- Task 3 -->
    <button id="newgame-btn" type="button">New game</button>

    <!-- Task 1 -->
    <br>
    <input id="showNumberCkBx" type="checkbox">
    <label id="computerNumber"></label>

    <!-- Task 2 -->
    <div id="results">
      <p id="rightOrWrong"></p>
      <p id="lowOrHi"></p>
      <p>Previous Guesses:</p>
      <ul id="prevGuesses"></ul>
      <p id="stats"></p>
    </div>

    <!-- JavaScript -->
    <script>

<!-- Copy the JavaScript code on the next page here -->

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

The JavaScript solution for task 3 is shown here.

```html
<!-- JavaScript -->
<script>

  // pick a random number between 1 and 10
  let secretNumber = Math.floor(Math.random() * 10) + 1;

  // Setup an event handler for the submit button
  const submitBtn = document.getElementById("submit-btn");
  submitBtn.addEventListener('click', checkGuess);

  const newGameBtn = document.getElementById('newgame-btn'); // Task 3
  newGameBtn.addEventListener('click', newGame); // Task 3

  let checkbox = document.getElementById("showNumberCkBx"); // Task 1
  checkbox.addEventListener('click', toggleNumber); // Task 1

  let guessCount = 0; // Task #2

  // Start a new game
  newGame(); // Task 3

  // Handler - check the guess
  function checkGuess() {
    const feedback = document.getElementById('rightOrWrong');
    const guessField = document.getElementById('guessField');

    // Read the user's guess
    let userGuess = Number(guessField.value);

    guessCount++; // Task 2

    if (userGuess === secretNumber) {
      feedback.textContent = 'Congratulations! You got it right!';
      feedback.style.backgroundColor = 'green'; // Task 1
      gameOver(); // Task 3
      //displayStats(); // Task 2 - commented out on task 3
    } else {
      feedback.textContent = 'Wrong!';
      feedback.style.backgroundColor = 'red'; // Task 1
    }


    // Task #2
    // Display a helpful message to the user
    let helpfulMsgField = document.getElementById("lowOrHi");
    if (userGuess < secretNumber) {
        helpfulMsgField.textContent = 'Too low!' ;
    } else if (userGuess > secretNumber) {
        helpfulMsgField.textContent = 'Too high!';
    } else if (userGuess === secretNumber) {
        helpfulMsgField.textContent = 'Just right!';
    }
```

```javascript
    // Task #2
    // Display each user's guess in the division for prevGuesses
    let unorderedList = document.getElementById("prevGuesses");
    let newListItem = document.createElement("li");
    newListItem.textContent = guessField.value;
    unorderedList.appendChild(newListItem);


    guessField.value = ''; // blank the field
    guessField.focus(); // give it the focus
  }

  // Task #3
  function newGame() {
    secretNumber = Math.floor(Math.random() * 10) + 1;
    guessCount = 0;

    submitBtn.disabled = false;
    newGameBtn.disabled = true;

    document.getElementById('computerNumber').textContent = "";
    document.getElementById("showNumberCkBx").checked = false;
    document.getElementById('rightOrWrong').textContent = "";
    document.getElementById("lowOrHi").textContent = "";
    document.getElementById("prevGuesses").innerHTML = "";
    document.getElementById("stats").textContent = "";
  }

  // Task #3
  function gameOver() {
    submitBtn.disabled = true;
    newGameBtn.disabled = false;
    displayStats();
  }

  // Task #2
  function displayStats() {
    // Display the number of guesses
    let statsPara = document.getElementById("stats");
    statsPara.textContent = "You took "+guessCount+" guesses";
  }


  // Task #1
  function toggleNumber() {
    const computerNumber = document.getElementById('computerNumber');
    if (checkbox.checked) {
      computerNumber.style.visibility = 'visible';
      computerNumber.textContent = secretNumber;
    } else {
      computerNumber.style.visibility = 'hidden';
    }
  }

</script>
```

Breakout #2: Task 4 (abridged) and Suggested Solution

In this task you will add two spin buttons to provide users with the ability to set a minimum and maximum value for the secret number. The secret number generated by the program will be between the minimum and maximum values as decided by the user.

The HTML code is shown below.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Guess Game - Task 4 Solution</title>
  </head>
  <body>
    <h1>Number guessing game</h1>
    <!-- Task 4 -->
    <!--p>Guess a number between min and max (incl.)</p-->
    <p>Guess a number between min and max (incl.)</p>

    <div>
      <label for="low">Min: </label>
      <input id="low" type="number" min="1" max="100" step="1" value="1">
      <label for="high">Max: </label>
      <input id="high" type="number" min="1" max="100" step="1" value="100">
    </div>

    <label for="guessField">Enter a guess: </label>
    <input type="text" id="guessField">
    <button id="submit-btn" type="button">Submit guess</button>

    <!-- Task 3 -->
    <button id="newgame-btn" type="button">New game</button>

    <!-- Task 1 -->
    <br>
    <input id="showNumberCkBx" type="checkbox">
    <label id="computerNumber"></label>

    <!-- Task 2 -->
    <div id="results">
      <p id="rightOrWrong"></p>
      <p id="lowOrHi"></p>
      <p>Previous Guesses:</p>
      <ul id="prevGuesses"></ul>
      <p id="stats"></p>
    </div>

    <!-- JavaScript -->
    <script>

    <!-- Copy the JavaScript code on the next pages here -->

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

The JavaScript solution for task 4 is shown here.

```
<!-- JavaScript -->
<script>

  // pick a random number between 1 and 10
  //let secretNumber = Math.floor(Math.random() * 10) + 1;
  let secretNumber; // Comment the preceding line and replaces with this
  let minValue, maxValue; // Task 4

  // Setup an event handler for the submit button
  const submitBtn = document.getElementById("submit-btn");
  submitBtn.addEventListener('click', checkGuess);

  const newGameBtn = document.getElementById('newgame-btn'); // Task 3
  newGameBtn.addEventListener('click', newGame); // Task 3

  let checkbox = document.getElementById("showNumberCkBx"); // Task 1
  checkbox.addEventListener('click', toggleNumber); // Task 1

  // Create a listener for the two spinners (the same one will do them both)
  document.getElementById('low').addEventListener('change', setSecretNo); // Task 4
  document.getElementById('high').addEventListener('change', setSecretNo); // Task 4


  let guessCount = 0; // Task #2

  // Start a new game
  newGame(); // Task 3

  // Handler - check the guess
  function checkGuess() {
    const feedback = document.getElementById('rightOrWrong');
    const guessField = document.getElementById('guessField');

    // Read the user's guess
    let userGuess = Number(guessField.value);

    guessCount++; // Task 2

    if (userGuess === secretNumber) {
      feedback.textContent = 'Congratulations! You got it right!';
      feedback.style.backgroundColor = 'green'; // Task 1
      gameOver(); // Task 3
      //displayStats(); // Task 2 - commented out on task 3
    } else {
      feedback.textContent = 'Wrong!';
      feedback.style.backgroundColor = 'red'; // Task 1
    }

    // Task #2
    // Display a helpful message to the user
    let helpfulMsgField = document.getElementById("lowOrHi");
    if (userGuess < secretNumber) {
        helpfulMsgField.textContent = 'Too low!' ;
    } else if (userGuess > secretNumber) {
```

```javascript
            helpfulMsgField.textContent = 'Too high!';
        } else if (userGuess === secretNumber) {
            helpfulMsgField.textContent = 'Just right!';
        }


        // Task #2
        // Display each user's guess in the division for prevGuesses
        let unorderedList = document.getElementById("prevGuesses");
        let newListItem = document.createElement("li");
        newListItem.textContent = guessField.value;
        unorderedList.appendChild(newListItem);



        guessField.value = ''; // blank the field
        guessField.focus(); // give it the focus
    }

    // Task #3
    function newGame() {

        // Task 4
        //secretNumber = Math.floor(Math.random() * 10) + 1; // Commented out for
task 4
        minValue = Number(document.getElementById('low').value);
        maxValue = Number(document.getElementById('high').value);
        secretNumber = Math.floor(Math.random() * (maxValue - minValue +1)) +
minValue;

        guessCount = 0;

        submitBtn.disabled = false;
        newGameBtn.disabled = true;

        document.getElementById('computerNumber').textContent = "";
        document.getElementById("showNumberCkBx").checked = false;
        document.getElementById('rightOrWrong').textContent = "";
        document.getElementById("lowOrHi").textContent = "";
        document.getElementById("prevGuesses").innerHTML = "";
        document.getElementById("stats").textContent = "";
    }

    // Task #3
    function gameOver() {
      submitBtn.disabled = true;
      newGameBtn.disabled = false;
      displayStats();
    }

    // Task #2
    function displayStats() {
        // Display the number of guesses
        let statsPara = document.getElementById("stats");
        statsPara.textContent = "You took "+guessCount+" guesses";
    }


    // Task #1
```

```
    function toggleNumber() {
        const computerNumber = document.getElementById('computerNumber');
        if (checkbox.checked) {
          computerNumber.style.visibility = 'visible';
          computerNumber.textContent = secretNumber;
        } else {
          computerNumber.style.visibility = 'hidden';
        }
    }


    // Task #4
    // Handler for the spinners
    function setSecretNo() {

      minValue = Number(document.getElementById('low').value);
      maxValue = Number(document.getElementById('high').value);
      secretNumber = Math.floor(Math.random() * (maxValue - minValue +1)) +
minValue;

      // update the secret number on the display
      document.getElementById('computerNumber').textContent = secretNumber;

    }

  </script>
```

Breakout #2: Task 5 (abridged) and Suggested Solution

For this final task of the breakout session you are required to 'webify' the Computer Aided Learning (CAL) system that you developed in breakout 1. We will brand the new system, Online Computer Aided Learning System, or **OCALS** for short!

We present two sample solutions – version 1 and version 2.

Version 1

The user interface for version 1 is illustrated below. As you can see the difficulty level and the operation are selected from two separate list boxes. (These are created using the HTML `select` element.)

The problem itself is generated by the program in response to the user clicking on the *Generate a problem* button.

The user continues to enter an answer until they get a correct response. Every user response (attempts) are displayed as list items in an unordered list and the total number of attempts is displayed when the user enters the correct response.

Online Computer Aided Learning V1

Select a difficulty level

Easy (single digit)
Hard (double digit)

Choose an operation

Addition
Subtraction
Multiplication
Division

Generate a problem

What is 4 * 9?

Enter your answer: | Submit answer

Correct!

Previous Answers:

- 32
- 13
- 49
- 36

You took 4 attempts.

Recall from the initial requirements that the system displays a random message each time the user enters their answer. Depending on whether the answer is correct or not, the computer message will come from one of the two following lists.

Correct Messages: "Well done!", "Very good!", "Correct!", "Keep it up!", and "Nice work!"
Incorrect Messages "No! Please try again", "Wrong! Try once more", "Don't give up!", "No! Keep trying", and "That's incorrect"

The HTML code for OCALS v1 is presented below.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Online CAL V1</title>
    <!-- import the webpage's stylesheet -->
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1>Online Computer Aided Learning V1</h1>

    <p>Select a difficulty level</p>
    <select id="difficultyLevel" size="2">
      <option selected value="easy">Easy (single digit)</option>
      <option value="hard">Hard (double digit)</option>
    </select>


    <p>Choose an operation</p>
    <select id="operation" size="4">
      <option selected>Addition</option>
      <option>Subtraction</option>
      <option>Multiplication</option>
      <option>Division</option>
    </select>

    <br><br>
    <button id="generate-btn" type="button">Generate a problem</button>
    <p id="problem"></p>

    <label for="answerField">Enter your answer: </label>
    <input type="text" id="answerField">
    <button id="submit-btn" type="button">Submit answer</button>

    <div id="results">
      <p id="rightOrWrong"></p>
      <p>Previous Answers:</p>
      <ul id="prevAnswers"></ul>
      <p id="stats"></p>
    </div>


    <script>

    <!-- import the webpage's javascript file -->
    <script src="task5SolutionOCALSv1.js"></script>
    <!-- The source for task5SolutionOCALSv1.js is on the next pages -->

    </script>

  </body>
</html>
```

The JavaScript solution for OCALS v1 is shown here.

```javascript
// Online Computer Aided Learning System (OCALS) v1
console.log("Welcome to OCALS v1");

// Setup an event handler for the difficultyLevel selection
const diffLevelSelection = document.getElementById("difficultyLevel");
diffLevelSelection.addEventListener('change', setDiffLevel);

// Setup an event handler for the operation selection
const operationSelection = document.getElementById("operation");
operationSelection.addEventListener('change', setOperation);

// Setup an event handler for the operation selection
const generateBtn = document.getElementById("generate-btn");
generateBtn.addEventListener('click', generateProblem);
generateBtn.disabled = false;

// Setup an event handler for the submit answer button
const submitBtn = document.getElementById("submit-btn");
submitBtn.addEventListener('click', checkAnswer);
submitBtn.disabled = true;

// Global variables
let difficultyLevel = 0, operation = 0, correctAnswer = 0;
let nrAttempts = 0; // used to store the number of attempts

// Declare two global arrays of messages
const correctMsgs = ["Well done!", "Very good!", "Correct!", "Keep it up!", "Nice
work!"];
const incorrectMsgs = ["No! Please try again", "Wrong! Try once more", "Don't give
up!", "No! Keep trying", "That's incorrect"];




// Handler for the difficulty level
function setDiffLevel() {
  console.log("setDiffLevel");
  difficultyLevel = document.getElementById("difficultyLevel").selectedIndex;
  console.log(difficultyLevel);

}


// Handler for the opertion
function setOperation() {
  console.log("setOperation");
  operation = document.getElementById("operation").selectedIndex;
  console.log(operation);

}
```

```javascript
// Handler for the generate problem button
function generateProblem() {
  console.log("generateProblem");

  generateBtn.disabled = true;
  submitBtn.disabled = false;

  let n1, n2;

  if (difficultyLevel === 0) {
    // Generate 2 random numbers between 0 and 9 incl.
    n1 = Math.floor(Math.random() * 10); // 0 <= n1 < 10
    n2 = Math.floor(Math.random() * 10); // 0 <= n2 < 10
  } else { // assume diff level is 1
    // Generate 2 random numbers between 10 and 99 incl.
    n1 = Math.floor(Math.random() * 90) + 10; // 10 <= n1 < 100
    n2 = Math.floor(Math.random() * 90) + 10; // 10 <= n2 < 100
  }

    // Assign the operator
  let operator;
  if (operation === 0) {
      operator = "+";
      correctAnswer = n1 + n2;
  } else if (operation === 1) {
      operator = "-";
      correctAnswer = n1 - n2;
  } else if (operation === 2) {
      operator = "*";
      correctAnswer = n1 * n2;
  } else { // assume division is 3
      operator = "/";
      correctAnswer = n1 / n2;
  }


  let problemPara = document.getElementById("problem");
  let problemText = "What is "+String(n1)+" "+operator+" "+String(n2)+"?";
  problemPara.textContent = problemText;
  console.log(problemText);

  // Clear the results from the previous problem
  document.getElementById('rightOrWrong').textContent = "";
  document.getElementById("prevAnswers").innerHTML = "";
  document.getElementById("stats").textContent = "";

  nrAttempts = 0; // reset the number of attempts taken
  document.getElementById('answerField').focus();

}


// Handler - check the answer
function checkAnswer() {
  const feedback = document.getElementById('rightOrWrong');
  const answerField = document.getElementById('answerField');
```

```javascript
  // Read the user's guess
  let userAnswer = Number(answerField.value);

  nrAttempts++;

  if (userAnswer === correctAnswer) {
    feedback.textContent = getRandomMsg(correctMsgs);
    feedback.style.backgroundColor = 'green';
    generateBtn.disabled = false;
    submitBtn.disabled = true;
    displayStats();
  } else {
    feedback.textContent = getRandomMsg(incorrectMsgs);
    feedback.style.backgroundColor = 'red';
  }


  // Display each user's guess in the division for prevGuesses
  let unorderedList = document.getElementById("prevAnswers");
  let newListItem = document.createElement("li");
  newListItem.textContent = answerField.value;
  unorderedList.appendChild(newListItem);


  answerField.value = ''; // blank the field
  answerField.focus(); // give it the focus
}

// A function that generates and displays a random message
function getRandomMsg(messages) {
  const r = Math.floor(Math.random() * messages.length);
  console.log(messages[r]);
  return(messages[r]);
} // end displayRandomMsg


// A function to display the final statistics
function displayStats(){
  let statsPara = document.getElementById("stats");
  let statsText = "You took "+nrAttempts+" attempts.";
  statsPara.textContent = statsText;

} // end displayStats
```

Version 2

This is what the user interface for version 2 looks like.

The user selects an operation from one of the options – Addition, Subtraction, Multiplicatio and Division – shown in the radio group towards the top of the screen.

The difficulty level (easy/hard) is not explicitly set. Instead the user can control the range of values the computer should use for each operand.

The screenshow above depicts a scenario where the user has selected Multiplication. The first operand is in the range 1-100 and the range of the second operand has been set to 1-12. The user has clicked on the Generate a problem button (which is then disabled) and the computer generates and displays the problem – in this case, *What is 96 * 5*?

Four possible answers are presented using 4 radio buttons at the bottom of the screen. When the user selects the correct answer - only one of the answers shown are correct! – the *Generate a problem* button is (re)enabled and the user can continue.

The HTML and JavaScript code for this system are shown on the next pages.

Here is the HTML code for OCALS v2.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Online CAL V2</title>
    <!-- import the webpage's stylesheet -->
    <link rel="stylesheet" href="style.css">

  </head>
  <body>
    <h1>Online Computer Aided Learning V2</h1>
    <p>Select an operation</p>

    <form id="opForm">
      <label><input type="radio" name="opForm" value="0" checked> Addition</label>
      <label><input type="radio" name="opForm" value="1"> Subtraction</label>
      <label><input type="radio" name="opForm" value="2"> Multiplication</label>
      <label><input type="radio" name="opForm" value="3"> Division</label>
    </form>

    <div>
      <p>Set the range of the 1st operand</p>
      <label for="low1">Min: </label>
      <input id="low1" type="number" min="1" max="100" step="1" value="1">
      <label for="high1">Max: </label>
      <input id="high1" type="number" min="1" max="100" step="1" value="100">
    </div>

    <div>
      <p>Set the range of the 2nd operand</p>
      <label for="low2">Min: </label>
      <input id="low2" type="number" min="1" max="100" step="1" value="1">
      <label for="high2">Max: </label>
      <input id="high2" type="number" min="1" max="100" step="1" value="100">
    </div>

    <div>
     <button id="generate-btn" type="button">Generate a problem</button>
     <p id="problem"></p>
    </div>

    <form id="resultForm">
     <label><input type="radio" name="resultForm" id="radio0" value="0"> </label><br>
     <label><input type="radio" name="resultForm" id="radio1" value="1"> </label><br>
     <label><input type="radio" name="resultForm" id="radio2" value="2"> </label><br>
     <label><input type="radio" name="resultForm" id="radio3" value="3"> </label><br>
    </form>

    <p id="feedback"></p>

    <script>

    <!-- import the webpage's javascript file -->
    <script src="task5SolutionOCALSv2.js"></script>
    <!-- The source for task5SolutionOCALSv2.js is on the next pages -->

    </script>

  </body>
</html>
```

The JavaScript solution for OCALS v2 is presented here.

```javascript
// Online Computer Aided Learning System (OCALS) v2
console.log("Welcome to OCALS v3");
displayResultForm("none"); // Hide the results section

// Setup an event handler for the operation selection
const formCtrl = document.getElementById("opForm");
formCtrl.addEventListener('click', setOperation);

// Create a listener for the two spinners for the 1st number
document.getElementById('low1').addEventListener('change', setN1Range);
document.getElementById('high1').addEventListener('change', setN1Range);

// Create a listener for the two spinners for the 2nd number
document.getElementById('low2').addEventListener('change', setN2Range);
document.getElementById('high2').addEventListener('change', setN2Range);

// Setup an event handler for the operation selection
const generateBtn = document.getElementById("generate-btn");
generateBtn.addEventListener('click', generateProblem);
generateBtn.disabled = false;

// Setup an event handler for the operation selection
const resultForm = document.getElementById("resultForm");
resultForm.addEventListener('click', processResult);


// Global variables
const incorrectMsgs = ["No! Please try again", "Wrong! Try once more", "Don't give
up!", "No! Keep trying", "That's incorrect"];
let operation = 0; // Addition is the default
let correctBtnID = 0;
let minValue1 = 0, minValue2 = 0, maxValue1 = 100, maxValue2 = 100;

// Handler for the operation
function setOperation() {
  console.log("setOperation - radio clicked");
  let form = document.getElementById("opForm");
  operation = form.elements["opForm"].value;
  console.log(operation);

}


// Handler for min/max of operand 1
function setN1Range() {
  minValue1 = Number(document.getElementById('low1').value);
  maxValue1 = Number(document.getElementById('high1').value);
}


// Handler for min/max of operand 2
function setN2Range() {
  minValue2 = Number(document.getElementById('low2').value);
  maxValue2 = Number(document.getElementById('high2').value);
```

```
}

// Handler for the generate problem button
function generateProblem() {
  console.log("generateProblem");
  let correctAnswer = 0;
  generateBtn.disabled = true;

  let n1 = Math.floor(Math.random() * (maxValue1 - minValue1 +1)) + minValue1;
  let n2 = Math.floor(Math.random() * (maxValue2 - minValue2 +1)) + minValue2;
    // Assign the operator
  let operator;
  if (operation == 0) {
      operator = "+";
      correctAnswer = n1 + n2;
  } else if (operation == 1) {
      operator = "-";
      correctAnswer = n1 - n2;
  } else if (operation == 2) {
      operator = "*";
      correctAnswer = n1 * n2;
  } else { // assume division is 3
      operator = "/";
      correctAnswer = n1 / n2;
  }


  let problemPara = document.getElementById("problem");
  let problemText = "What is "+String(n1)+" "+operator+" "+String(n2)+"?";
  problemPara.textContent = problemText;
  console.log(problemText);

  displayResultForm("block"); // Show the results section

  // https://stackoverflow.com/questions/45259854/change-the-text-next-to-a-radio-
button-using-javascript
  // Dynamically set the content of the button's text node
  //document.getElementById('radio0').nextSibling.textContent = "ans1";
  //document.getElementById('radio1').nextSibling.textContent = "ans2";
  //document.getElementById('radio2').nextSibling.textContent = "ans3";
  //document.getElementById('radio3').nextSibling.textContent = "ans4";

  correctBtnID = Math.floor(Math.random() * 3); // a number between 0 and 3 incl
  for (let i=0; i<4; i++) {
    let radioID="radio"+i;
    if (i == correctBtnID) {
      document.getElementById(radioID).nextSibling.textContent =
String(correctAnswer);
    } else {
      let randomAnswer = Math.floor(Math.random() * 500) + 1;
      document.getElementById(radioID).nextSibling.textContent =
String(randomAnswer);
    }
  }

} // end generateProblem
```

```javascript
function processResult() {

  const feedback = document.getElementById('feedback');

  const resultForm = document.getElementById("resultForm");
  let selectedBtnValue = resultForm.elements["resultForm"].value;
  console.log(selectedBtnValue); // undefined

  if (selectedBtnValue == correctBtnID) {
    console.log("Correct");
    displayResultForm("none"); // Hide the results section
    generateBtn.disabled = false; // Enable the Generate a problem button
    document.getElementById("problem").textContent = ""; // Clear the problem
    feedback.textContent = ""; // Clear the feedback
    document.getElementById("radio"+correctBtnID).checked = false;
  } else {
    console.log("Incorrect");
    feedback.textContent = getRandomMsg(incorrectMsgs);
    feedback.style.backgroundColor = 'red';
  }

} // processResult

// Show or hide the results section
function displayResultForm(displayStyle) {
  let resultForm = document.getElementById("resultForm");
  resultForm.style.display = displayStyle;
} // displayResultForm


// A function that generates and displays a random message
function getRandomMsg(messages) {
  const r = Math.floor(Math.random() * messages.length);
  console.log(messages[r]);
  return(messages[r]);
} // end displayRandomMsg
```

Style sheet for OCALS

The following style sheet was used in the two versions of OCALS just presented. Both versions imported this from an external stylesheet file called `style.css`.

```css
/* CSS files add styling rules to your content */

body {
  font-family: "Benton Sans", "Helvetica Neue", helvetica, arial, sans-serif;
  margin: 2em;
}



canvas {
display: block;
}

label {
  font-weight: bold;
}

div {
  padding-bottom: 20px;
}

input[type="text"] {
  padding: 5px;
  width: 150px;
}
```

BLANK PAGE

Appendix

JavaScript Keywords

The full list of JavaScript reserved words is shown in the table below:

```
await       debugger    false       instanceof  this        void

break       default     finally     let         throw       while

case        delete      for         new         true        with

catch       do          function    null        try         yield

class       else        if          return      typeof

const       export      import      super       undefined

continue    extends     in          switch      var
```

ECMAScript 2018 keywords

The following words should also be treated as reserved words (even though they are not)

```
true, false, let, null, undefined
boolean, byte, char, double, float, long,
arguments, eval, parseInt, parseFloat
Infinity, NaN, isNaN, isFinite,
Array, Boolean, Date, Error, Function, JSON, Math, Number,
Object, String
```

Notes:
1) You should avoid using any of these words as identifiers for variables and functions in your JavaScript programs
2) The above list is by no means complete but should serve as a good guide.
3) If unsure, you should consult the Mozilla JavaScript Language reference

Arithmetic Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Remainder	x % y
**	Power	x ** y
++	Unary Increment	x++
--	Unary Decrement	y--

Common arithmetic operators

Compound Assignment Operators

Operator	Name	Example	Same as
+=	Addition and Assignment	x += y	x = x + y
-=	Subtraction and Assignment	x -= y	x = x - y
*=	Multiplication and Assignment	x *= y	x = x * y
/=	Division and Assignment	x /= y	x = x / y
%=	Remainder and Assignment	x %= y	x = x % y
**=	Exponentiation and Assignment (not recommended)	x **= y	x = x ** y

Common Compound Assignment Operators

Operator Precedence

Operator	Name
++, --	Postfix Increment/Decremenet
++, --, !, +, -	**Prefix Increment/Decremenet, Logical NOT, Unary Plus, Unary Negation**
**	**Exponentiation**
*, /, %	Multiplication, Division, Remainder
+, -	Addition, Subtraction
<, <=, >, >=	Less Than, Less Than Or Equal To, Greater Than, Greater Than Or Equal To
==, !=	Equality, Inequality,
===, !==	Strict Equality, Strict Inequality
&&	Logical AND
||	Logical OR
=, *=, /=, %=, +=, -=	Aassignments (simple and compound)
,	Comma

Precedence of Common JavaScript Operators

Comparison Operators

Operator	Name	Description
==	Equality	Returns true if both operands have the same value (after conversion if necessary)
===	Strict equality	Returns true if both operands have the same value and type
!=	Inequality	Returns true if both operands have different values (after conversion if necessary)
!==	Strict inequality	Returns true if both operands have different values and/or types (with no conversion)
>	Greater than	Returns true if the left operand is > the right operand
>=	Greater than or equal to	Returns true if the left operand is >= the right operand
<	Less than	Returns true if the left operand is < the right operand
<=	Less than or equal to	Returns true if the left operand is <= the right operand

JavaScript comparison operators

Logical Operators and Truth Tables

Operator	Name	Syntax
!	Logical NOT	!expr
&&	Logical AND	expr1 && expr2
||	Logical OR	expr1 || expr2

JavaScript Logical Operators

A	!A
false	true
true	false

Truth table for logical NOT

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

Truth table for logical AND

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

Truth table for logical OR

Common `Array` Methods[12]

Method name	Description
`arrA.concat(arrB)`	Returns a new array made up of the elements of `arrA` followed by the elements of `arrB`.
`arrA.indexOf(item)`	Returns the index of the first occurrence of the value specified by `item` in `arrA`. If the item is not found the method returns `-1`
`arrA.lastIndexOf(item)`	Starting from the end, returns the index of the first occurrence of the value specified by `item` in `arrA`. If `item` is not found the method returns `-1`
`arrA.join([separator])`	Returns all the elements of the array joined together as a string. The default value of the optional separator is a comma.
`arrA.push(items)`	Appends one or more elements (as specified by `items`) to the end of `arrA` and returns the new length of the array.
`arrA.pop()`	Removes the last element of `arrA`. Returns the element removed or `undefined` if the array was empty
`arrA.shift()`	Removes the first element of `arrA`. Returns the element removed or `undefined` if the array was empty
`arrA.unshift(items)`	Inserts one or more elements (as specified by items) to the start of `arrA` and returns the new length of the array.
`arrA.sort()`	Sorts the elements of array in place and returns the sorted array (in alphabetical order)
`arrA.reverse()`	Sorts the elements of array in place and returns the sorted array (in alphabetical order)
`arrA.slice([i1, [i2])`	Returns a new array made up of the elements of `arrA` from `i1` up to but not including `i2`. If `i1` is not specified it is taken to be zero; if `i2` is not specified it is taken to be `arrA.length`. The contents of the original array are unchanged.
`arrA.splice(i, [n, [items]])`	Adds/replaces/remove elements from an array *in place*. `i` is the starting index, `n` is the number of elements to remove and `items` are the new elements. Returns a new array with any removed elements. (If no elements are removed an empty array is returned.)

---

[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Common `Date` Methods[13]

The following methods can be used for getting information from a date object `d`.

Method name	Description
`d.getDate()`	Returns the day of the month (1-31) for the specified date according to local time
`d.getDay()`	Returns the day of the week (0-6) for the specified date according to local time.
`d.getFullYear()`	Returns the year (4 digits for 4-digit years) of the specified date according to local time.
`d.getHours()`	Returns the hour (0-23) in the specified date according to local time.
`d.getMinutes()`	Returns the minutes (0-59) in the specified date according to local time.
`d.getMonth()`	Returns the month (0-11) in the specified date according to local time.
`d.getSeconds()`	Returns the seconds (0-59) in the specified date according to local time.
`d.getTime()`	Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC (negative for prior times).

---

[13] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

Common `Math` Methods[14]

Method	Description	Examples	Result
`Math.round(x);`	Returns x rounded up or down to the nearest integer	`Math.round(9.7);`	10
		`Math.round(9.3);`	9
`Math.ceil(x);`	Returns the nearest integer greater than or equal to x	`Math.ceil(9.7);`	10
		`Math.ceil(9.3);`	10
`Math.floor(x);`	Returns the nearest integer less than or equal to x	`Math.floor(9.7);`	9
		`Math.floor(9.3);`	9
`Math.pow(x, y);`	Returns x raised to the power of y.	`Math.pow(2,5);`	32
		`Math.pow(5,2);`	25
`Math.sqrt(x);`	Returns the square root of x	`Math.sqrt(25);`	32
		`Math.sqrt(-25);`	NaN
`Math.cbrt(x);`	Returns the cube root of x	`Math.cbrt(64);`	4
		`Math.cbrt(-64);`	-4
`Math.abs(x);`	Returns the absolute value of x	`Math.abs(25);`	25
		`Math.abs(-25);`	25
`Math.max(x, …)`	Returns the maximum of a list of 1 or more numbers	`Math.max(1,-2,-1);`	1
`Math.min(x, …)`	Returns the minimum of a list of 1 or more numbers	`Math.min(1,-2,-1);`	-2

Some example uses of `Math.random` are given in the table below:

Example	Description
`Math.floor(Math.random() * 10);`	Returns an integer r such that: $0 \leq r < 10$
`Math.floor(Math.random() * 11);`	Returns an integer r such that: $0 \leq r \leq 10$
`Math.floor(Math.random() * 10) + 1;`	Returns an integer r such that: $1 <= r \leq 10$

---

[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Common `Number` Methods[15]

Method name	Description

`Number.isNaN(x)`	Returns `true` if the given value is `NaN` and its type is `Number`; otherwise, `false`.
	
	`Number.isNaN("Joe")` → `false`
	`Number.isNaN("999")` → `false`
	`Number.isNaN(999)` → `false`
	`Number.isNaN(9.99)` → `false`
	`Number.isNaN(999/0)` → `false`
	`Number.isNaN(Infinity)` → `false`
	`Number.isNaN(NaN)` → `true`

`Number.isFinite(x)`	Returns `true` if the number passed in is a finite number; `false` otherwise
	
	`Number.isFinite("Joe")` → `false`
	`Number.isFinite("999")` → `false`
	`Number.isFinite(999)` → `true`
	`Number.isFinite(9.99)` → `true`
	`Number.isFinite(999/0)` → `false`
	`Number.isFinite(Infinity)` → `false`
	`Number.isFinite(NaN)` → `false`

`Number.isInteger(x)`	Returns `true` if the number passed in is an integer (or a decimal number that can be represented as an integer (e.g. 2.0); `false` otherwise
`Number.parseFloat(value)`	Returns a floating point number parsed from `value`. If the `value` cannot be converted to a number, `NaN` is returned This method has the same functionality as the global `parseFloat()` function:
`Number.parseInt(value, [base])`	Returns an integer parsed from `value` in the `base` provided. If `value` cannot be converted to a number, `NaN` is returned. This method has the same functionality as the global `parseInt()` function:

---

[15] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

The following methods work on any numeric expression `x`.

Method name	Description
`x.toExponential()`	Returns a string representation of `x` in exponential notation `19.64738.toExponential()` → `1.964738e+1`

`x.toFixed(len)`	Returns a string representation of `x` with `len` digits after the decimal point
	
	`19.64738.toFixed()` → `20`
	`19.64738.toFixed(1)` → `19.6`
	`19.64738.toFixed(2)` → `19.65`
	`19.64738.toFixed(3)` → `19.647`

`x.toPrecision(len)`	Returns a string representation of `x` rounded to `len` significant digits
	
	`19.64738.toPrecision(0)` → `19.64738`
	`19.64738.toPrecision(2)` → `20`
	`19.64738.toPrecision(4)` → `19.65`
	`19.64738.toPrecision(6)` → `19.6474`

Common `String` Methods[16]

Method name	Description
`strA.concat(strB)`	Returns a new string made up of the characters of `strA` followed by the characters of `strB`.
`str.charAt(index)`	Returns a new string made up of the character at the specified `index` in `str` (or an empty string if `index` is out of bounds)
`str.charCodeAt(index)`	Returns the Unicode code of the character at the specified index in `str` (or `NaN` if `index` is out of bounds)
`str.toUpperCase()`	Returns a new string with all the characters of `str` converted to upper case
`str.toLowerCase()`	Returns a new string with all the characters of `str` converted to lower case
`str.indexOf(item [,fromIndex])`	Returns the index of the first occurrence of the value specified by `item` in `str`. Unless `fromIndex` is specified the search starts at index zero. If `item` is not found the method returns `-1`
`str.lastIndexOf(item [,fromIndex])`	Starting from the end (or at `fromIndex`) and working backwards, this method returns the index of the first occurrence of the value specified by `item` in `str`. If `item` is not found the method returns `-1`
`str.slice([i1, [i2])`	Returns a new string made up of the characters of `str` from `i1` up to but not including `i2`. If `i1` is not specified it is taken to be zero; if `i2` is not specified it is taken to be `str.length`. The contents of the original string are unchanged.
`str.replace(old, new)`	Replaces all occurrences of `old` in `str` with `new`.
`str.split([separator])`	Returns an array of strings split at the point denoted by the `separator`
`str.trim()`	Creates a new string based on `str` with leading and trailing whitespaces removed. Note `trimStart()` removes only leading whitespaces and `trimEnd()` removes only trailing whitespaces

---

[16] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

References

Websites

Mozilla	https://developer.mozilla.org/en-US/docs/Web/JavaScript https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference
Web Demystified	https://www.youtube.com/playlist?list=PLo3w8EB99pqLEopnunz-dOOBJ8t-Wgt2g
W3Schools	https://www.w3schools.com/js/default.asp
TutorialsPoint	https://www.tutorialspoint.com/javascript/
JavaScript.info	https://javascript.info/js
Douglas Crockford's JavaScript	http://crockford.com/javascript/
Oracle	https://developer.oracle.com/javascript
geeksforgeeks	https://www.geeksforgeeks.org/javascript-tutorial/
edX JavaScript MOOC	https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2018/course/
Glitch	https://glitch.com/
ECMAScript 2018 Language Specification	https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

Books/Notes

1) JavaScript The Definitive Guide, David Flanagan, O'Reilly, 2011

2) Heads First JavaScript Programming, Eric Freeman and Elisabeth Robson, O'Reilly, 2014 (Companion website: https://www.wickedlysmart.com/hfjs/)

3) JavaScript and jQuery, Jon Duckett, Wiley and Sons, Inc. 2014

4) Eloquent JavaScript, Marijn Haverbeke, no starch press, 2019

5) Many of the common student misconceptions are taken from *"Misconceptions and the Beginner Programmer"* by Juha Sorva which appears as Chapter 13 in *Computer Science Education, Perspectives on Teaching and Learning in School*, edited by Sentence, Barendsen, and Schulte, Bloomsbury, 2018.

6) Many of the teacher tips are taken from *"Teaching Programming"* by Michael E. Caspersen which appears as Chapter 9 in *Computer Science Education Perspectives on Teaching and Learning in School*, edited by Sentence, Barendsen, and Schulte, Bloomsbury, 2018.

7) Professional Notes on Programming, Teaching and Learning, Joe English.

BLANK PAGE

BLANK PAGE

BLANK PAGE