
POWERING
POTENTIAL
Welcome!
JavaScript and TypeScript
For Cancer Research UK

Overview

Objectives

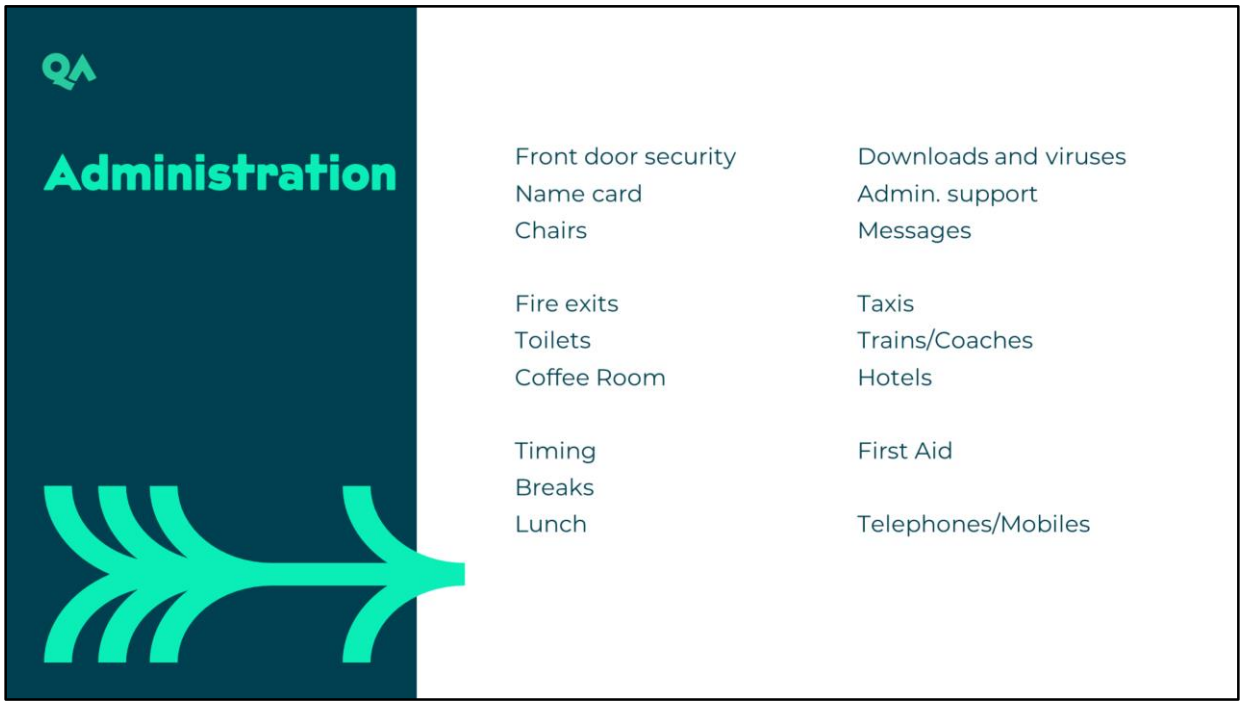
- To explain the aims and objectives of the course

Contents

- Course administration
- Course objectives and assumptions
- Introductions
- Any questions?

Exercises

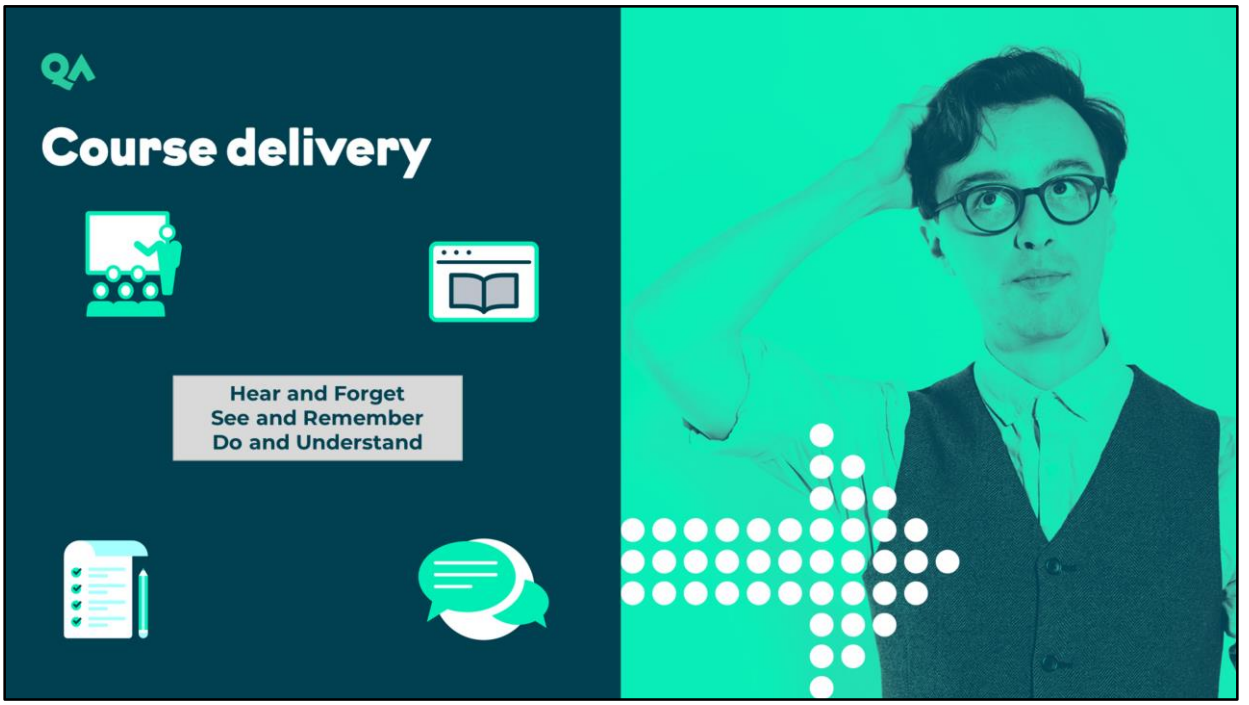
- Locate the exercises
- Locate the help files



We need to deal with practical matters right at the beginning.

Above all, please ask if you have any problems regarding the course or practical arrangements. If we know early on that something is wrong, we have the chance to fix it. If you tell us after the course, it's too late! We ask you to fill in an evaluation form at the end of the course. If you alert us to a problem for the first time on the feedback form at the end of the course, we won't have had the opportunity to put it right.

If this course is being held at your company's site, much of this will not apply or will be outside our control.

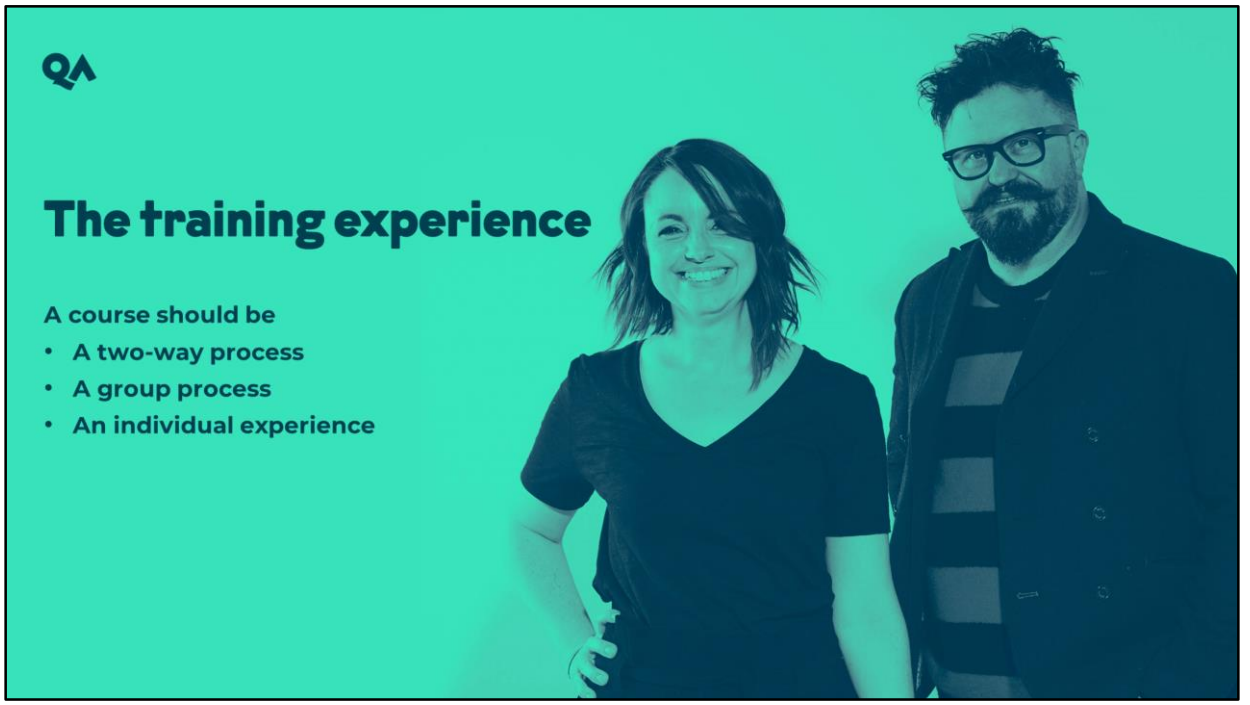


The course will be made up of lecture material coupled with the course workbook, informal questions and exercises, and structured practical sessions. Together, these different teaching techniques will help you absorb and understand the material in the most effective way.

The course notebooks contain all the overhead foils that will be shown, so you do not need to copy them. In addition, there are extra textual comments (like these) below the foils, which are there to amplify the foils and provide further information. Hopefully, these notes mean you will not need to write too much and can listen and observe during the lectures. There is, however, space to make your own annotations too.

The appendices cover material that is beyond the scope of the course, together with some help and guidelines. There are also appendices on bibliography and Internet resources to help you find more information after the course.

In the practical exercise sessions, you will be given the opportunity to experiment and consolidate what has been taught during the lecture sessions. Please tell the instructor if you are having difficulty in these sessions. It is sometimes difficult to see that someone is struggling, so please be direct.



The best courses are not those where the instructor spends all of his/her time pontificating at the front of the class. Things get more interesting if there is dialogue, so please feel free to make comments or ask questions. At the same time, the instructor has to think of the whole group, so if you have many queries, you might be asked to deal with them off-line.

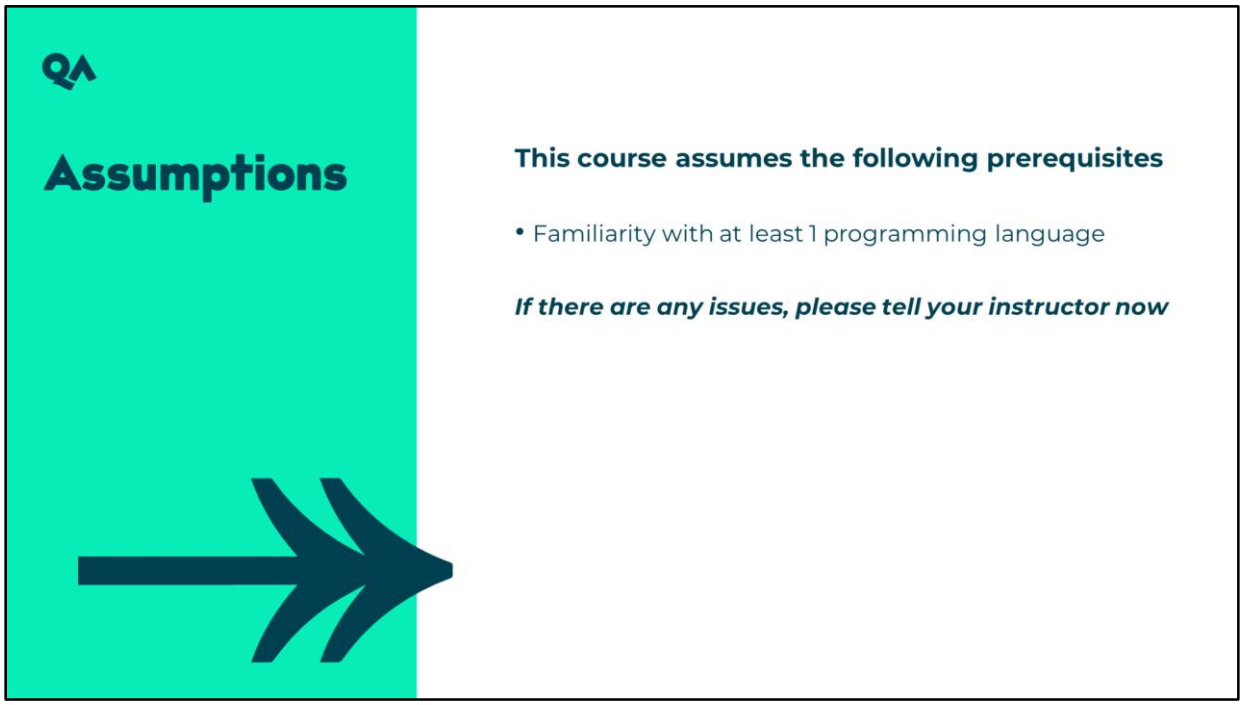
Work with other people during practical exercise sessions. The person next to you may have the answer, or you may know the remedy for them. Obviously do not simply 'copy from' or 'jump-in on' your neighbour, but group collaboration can help with the enjoyment of a course.

We are also individuals. We work at different paces and may have special interests in particular topics. The aim of the course is to provide a broad picture for all. Do not be dismayed if you do not appear to complete exercises as fast as the next person. The practical exercises are there to give plenty of practical opportunities; they do not have to be finished and you may even choose to focus for a long period on the topic that most interests you. Indeed, there will be parts labelled 'if time allows' that you may wish to save until later to give yourself time to read and absorb the course notes. If you have finished early, there is a great deal to investigate. Such “’hacking’ time is valuable. You may not get the opportunity to do it back in the office!

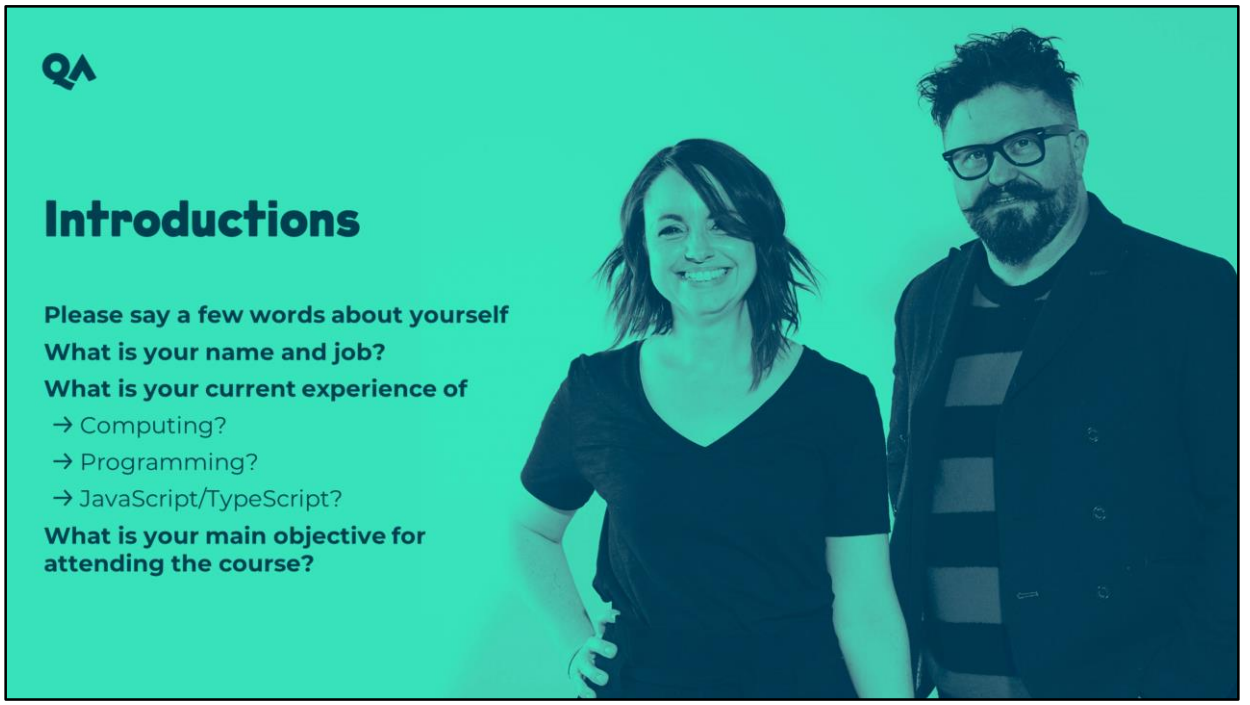
Course Outcomes

By the end of the course, you will be able to:

- Be able to write code using ES2015+ syntax JavaScript including:
  - Variables and constants
  - Primitive and Reference Types
  - Arrays and Collections
  - Loops
  - Functions
  - Object Orientation
  - Asynchronous techniques
- Be able to use TypeScript to make JavaScript more type-safe



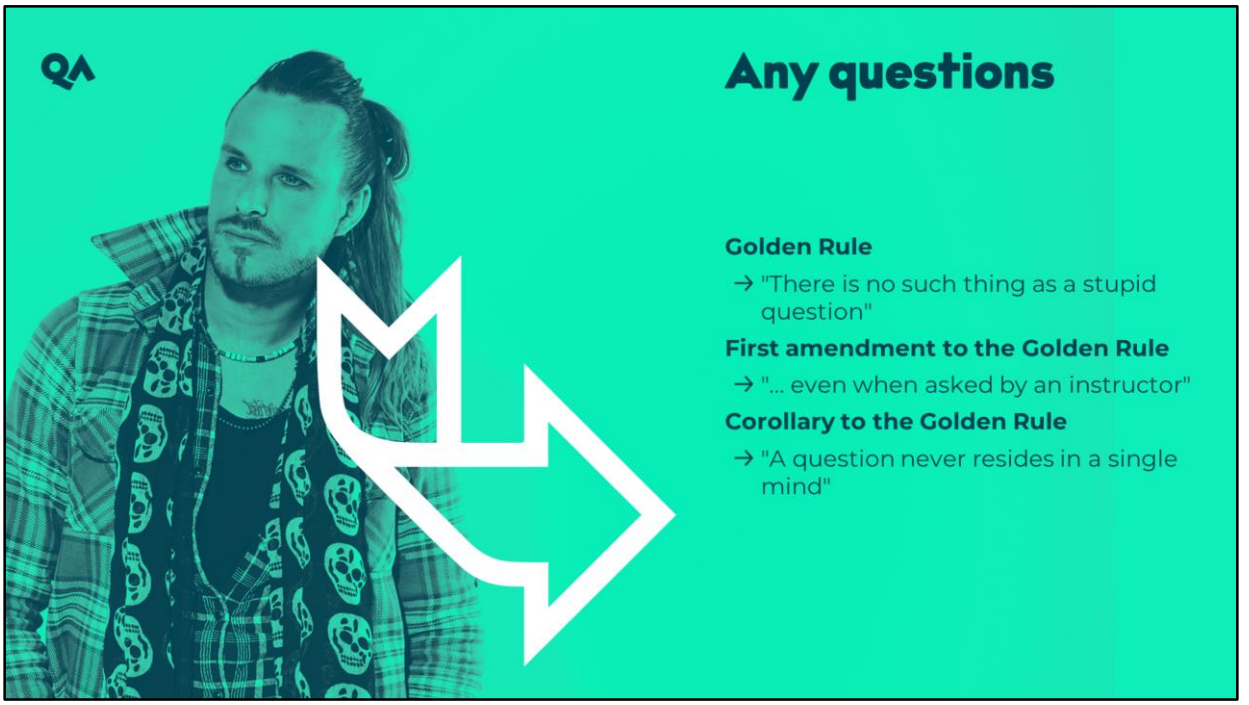
If you are not sure of any of these, please inform the instructor as soon as you can and they will do their best to help you.



One of the great benefits of courses is meeting other people. They may have similar interests, have encountered similar problems and may even have found the solution to yours. The contacts made on the course can be very useful.

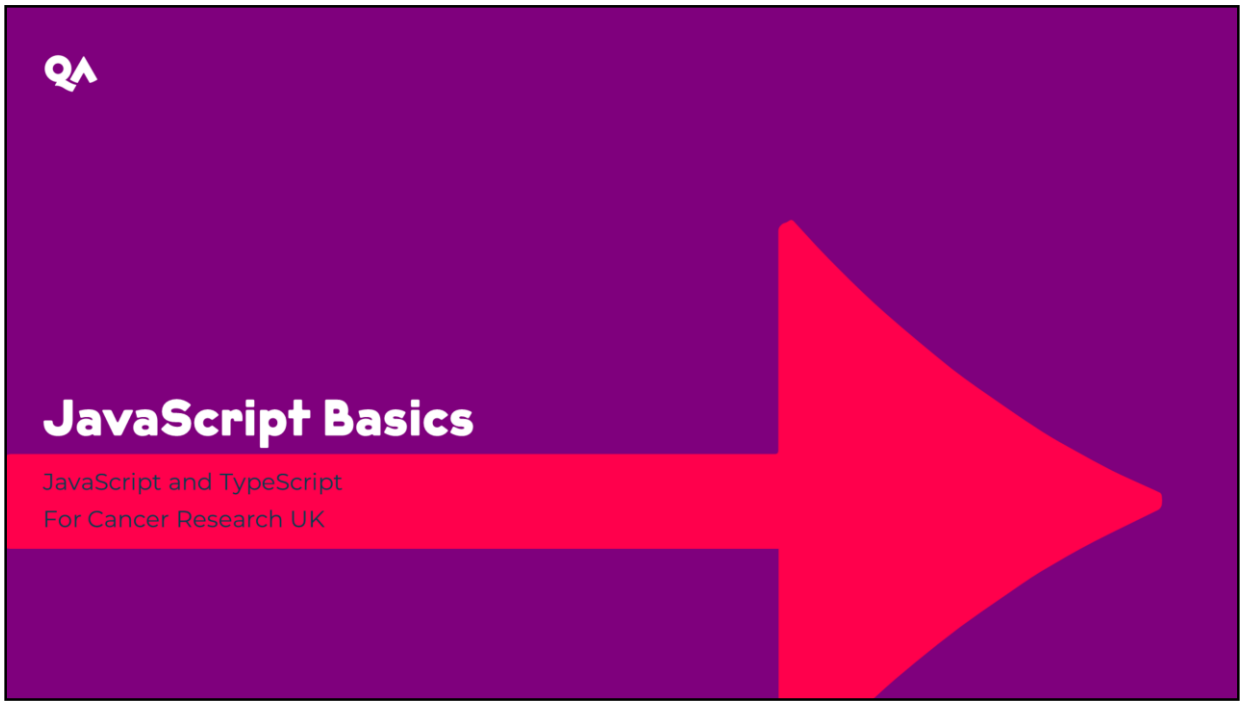
It is useful for us all to be aware of levels of experience. It will help the instructor judge the level of depth to go into and the analogies to make to help you understand a topic. People in the group may have specialised experience that will be helpful to others.

It is worth highlighting particular interests, as we may be able to address them during the course. However, it is a general course that aims to cover a broad range of topics, so the instructor may have to deal with some areas during a coffee break or over lunch.



Please feel free to ask questions.

Teaching is a much more enjoyable and productive process if it is interactive. You will no doubt think of questions during the course. If so, ask them!


JavaScript Basics
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To be aware of the history of JavaScript
- To be able to put JavaScript into a web page

A very brief history

JavaScript has been with us since 1995

- Originally designed for client based form validation
- Three separate versions in IE, Netscape and ScriptEase

Put forward to the ECMA as a proposed standard in 1997 as v1.1

- Ratified in 1998 as ECMAScript
- Implemented in browsers with various degrees of success ever since

Implementation is made up of three parts

- The Core (ECMAScript)
- The DOM (Document Object Model)
- The BOM (Browser Object Model)

Whether you are here as a new web programmer or an old hand server guru forced to consider the front end in a different way because of those dashed, new fangled MVC approaches, JavaScript is a core part of any web developer's arsenal. JavaScript is supported in almost every browser today and increasingly in a universal way; but, as we will see, older browsers cause us real issues with the way we code and what we can do.

JavaScript is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.

JavaScript was formalised in the ECMAScript language standard and is primarily used in the form of client-side JavaScript, implemented as part of a Web browser in order to provide enhanced user interfaces and dynamic websites. This enables programmatic access to computational objects within a host environment.

In recent years, it has found usage in non-web programming architectures, such as Acrobat files and Windows 8 Metro applications, demonstrating the amazing versatility of this incredibly flexible and powerful language and server-side programming with tools such as Node.js.

ECMAScript5

All browsers should adhere to the ECMAScript standard

- They do not (the Netscape IE browser wars were messy!)
- ECMAScript standard 3 was mostly implemented
- ECMAScript 4 was not
- ECMAScript 5 was and is widely implemented

ECMAScript 6 was renamed to ECMAScript 2015 to reflect the new annual release schedule of the standard

ECMAScript 2015 (ES2015) was the first revision of the standard in 6 years and so included many new features to the language

ES2016 and beyond have been incremental additions to the language

The 6th edition, officially known as ECMAScript 2015, was finalised in June 2015. This update adds significant new syntax for writing complex applications, including classes and modules, but defines them semantically in the same terms as ECMAScript 5 strict mode. Other new features include iterators and for/of loops, Python-style generators and generator expressions, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises, number and math enhancements, reflection, and proxies (metaprogramming for virtual objects and wrappers).

What can JavaScript do?

Can be used to create cookies

Can asynchronously request data from a server

JavaScript is also widely used as a server-side language, via NodeJS

JavaScript can be used to create desktop applications via tools, such as Electron

JavaScript can be used to create native mobile applications via tools, such as React Native

JavaScript is a scripting language

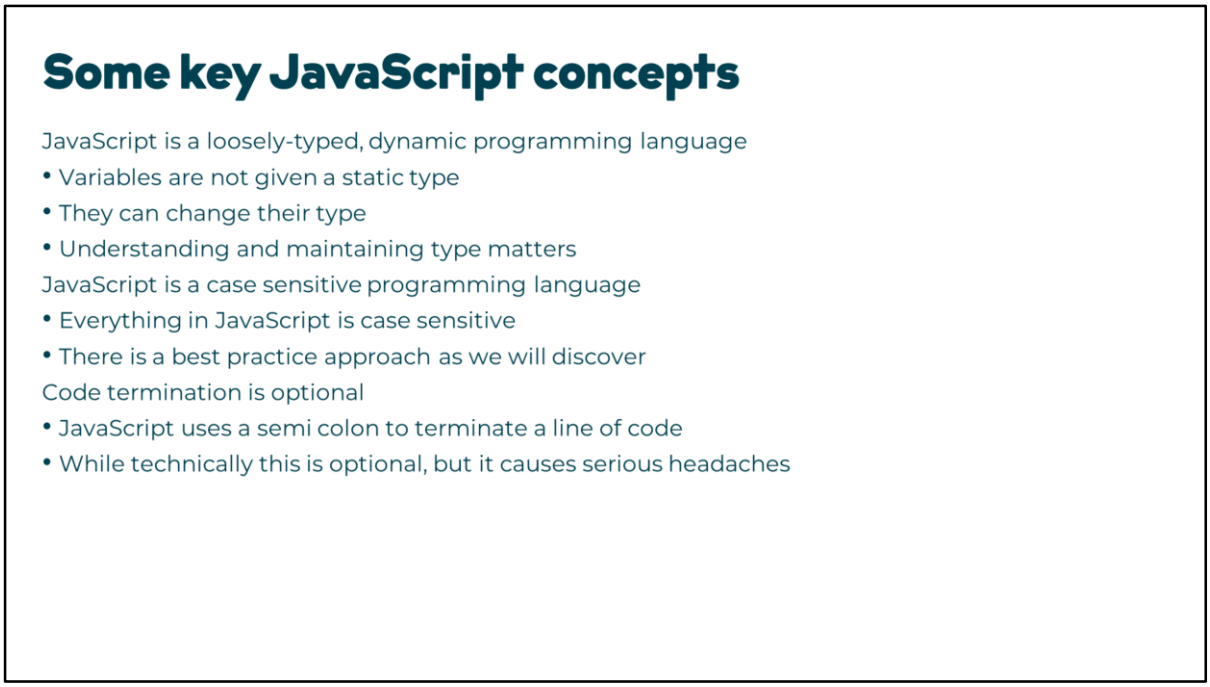
JavaScript gives front-end developers a programming tool

JavaScript can react to events created by the page itself (page loaded) or the user (click)

It can read and change the content of HTML elements, create new content, remove and hide elements

It can be used to validate form input

Can provide access to HTML 5 APIs such as indexeddb, geolocation, canvas and more



As we begin our journey with JavaScript, we will get a few things locked down in our brainboxes that will save us a lot of heartache and pain as the course goes on.

JavaScript is a case-sensitive language. This means…

```
var numberOne = 5;
var NumberOne = 5;
```

… actually creates two separate variables! This is a key fact to remember as you code – be careful as we move on in the development of our code.

JavaScript will work without a semicolon terminating the code in many situations, but this will also cause you real issues as the course draws to a conclusion and we discuss minificiation of script. Please try, therefore, to remember that every instruction needs to be terminated.

Adding script to HTML – embedding

You can either place JavaScript on a page inline

- The closing tag is mandatory

The script can be placed in either the head or body section

- It is executed as soon as the browser renders the script block
- Best practice often places it just before the closing body tag

```
<script>
      // … script goes here …
</script>
```

The browser needs to know the data it is rendering is not just normal text and is actually JavaScript. We achieve this by using the <script> element. You may use as many script elements as you like on a page. Browsers today will assume that your script is client side JavaScript, if no further information is provided.

Script blocks are executed as soon as the browser reaches the code block in the page. The script will then be executed; we will see the importance of this concept as the course progresses.

Adding script to HTML – linking

You can also place JavaScript in a separate file and link to it

- Useful as the script is going to be used on multiple pages
- Requires an additional request to the server
- The requested file is cached by the browser
- Preferred approach to working with script

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

Everything said about inline script processing is also true for linked scripts. The key consideration when using external script references is with performance. Each script file is a separate request to the server for the file. On first request to the page, this can cause some lag in a page rendering. Older browser, IE 6 being a notable example, can only make two asynchronous file requests from a server at a time, so the page load can be uneven. Very often, you compact your files together to create a single script file to assist with this; we will see this strategy later in the course.

The big benefit of this strategy of external linking is that script to be used on multiple pages can be managed centrally, making site maintenance much simpler and, as per usual, with asset caching the initial first cost leads to a quicker load on subsequent visits.

The <noscript> element

Client-side scripting may not be available

The **<noscript>** element will only render its content if

- The browser does not understand **<script>**
- The client has disabled script

```
<noscript>
      If you see this, you do not have JavaScript enabled.
</noscript>
```

Comments

Commenting code is an essential part of programming

Single line comment

```
index = 3; // From here to the end of the line is a comment and ignored
```

Multi-line comment

```
/*
Everything inside these delimiters is treated as
a comment and ignored by the interpreter
*/
```

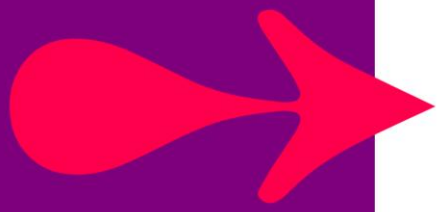
JavaScript has two ways of marking a comment: // starts a comment, which ends at the end of the current line; and multi-line comments, which can be created by inserting the comment text between /* (start-comment) and */ (end-comment).

Note that multi-line comments do not nest. This typical debugging method will cause problems:

```
/*  something wrong in this function
    commented out temporarily to isolate the problem

    function fred()
    {
       /* this function computes mortgage interest
       */
       x = ((y * 3) + 4 )/ 6.291 ^ 3.412
    }

*/
```

The first */ will close all comments. The second */ will be a syntax error.

REVIEW



What is JavaScript?

- A scripting language

What is JavaScript for?

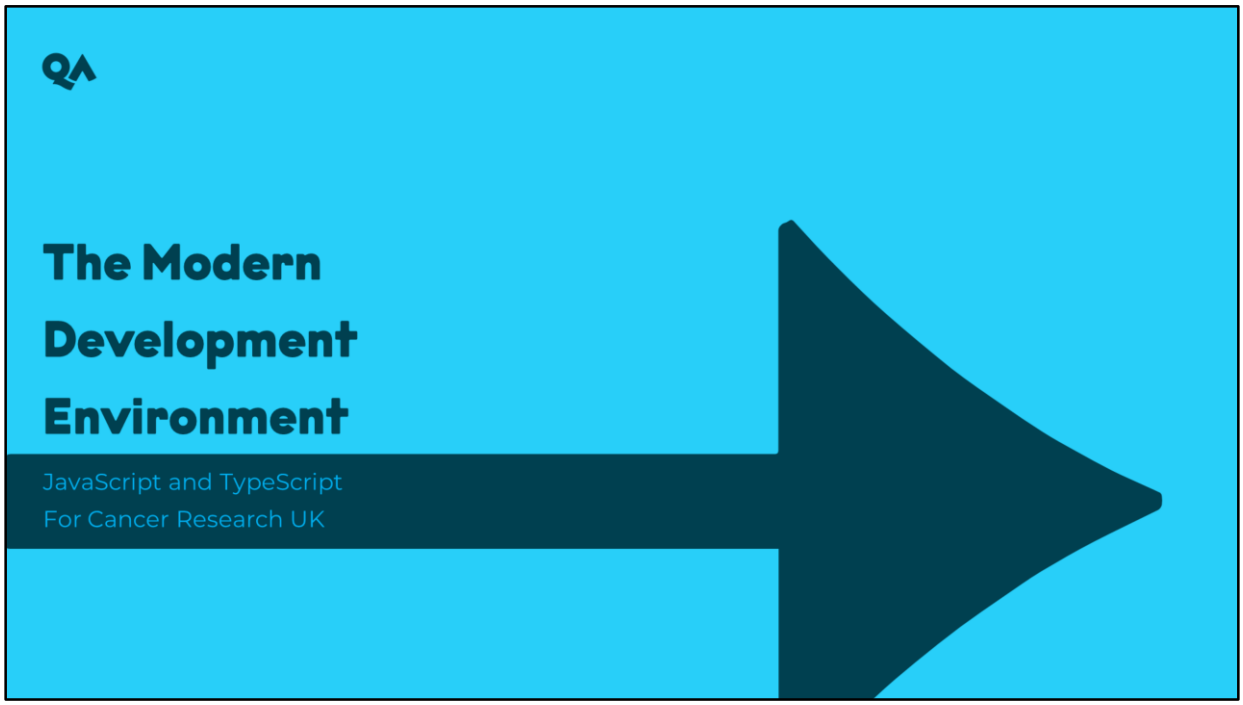
- Building client-side, server-side, desktop and mobile applications

How do we use JavaScript?

- Linked or embedded

Objectives

- To be aware of the history of JavaScript
- To be able to put JavaScript into a web page


The Modern
Development
Environment
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To be able to set up a modern developer's environment

Introduction

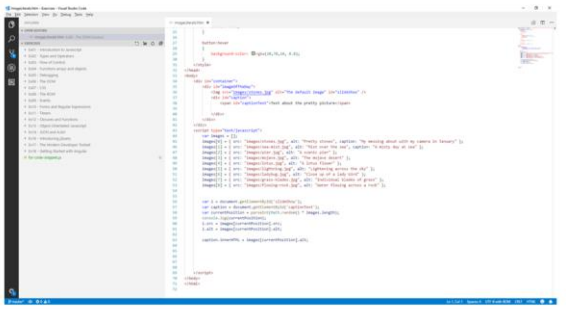
The Open Web Development Stack

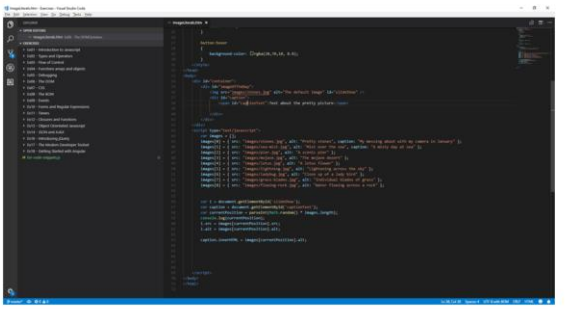
- Integrated Development Environments and Enhanced Text Editors
- Debugging Tools
- Using third-party Packages and dependency management
- Automation support and Continuous Development
- Version control and package management
- Working with the Command Line and Terminal
- Continuous Development and a DevOps Methodology

Walkthrough – using Visual Studio Code

Visual Studio Code is a free, open source enhanced text editor and it was built using JavaScript!

Not to be confused with Visual Studio (a paid-for, full IDE designed specifically for .NET development)

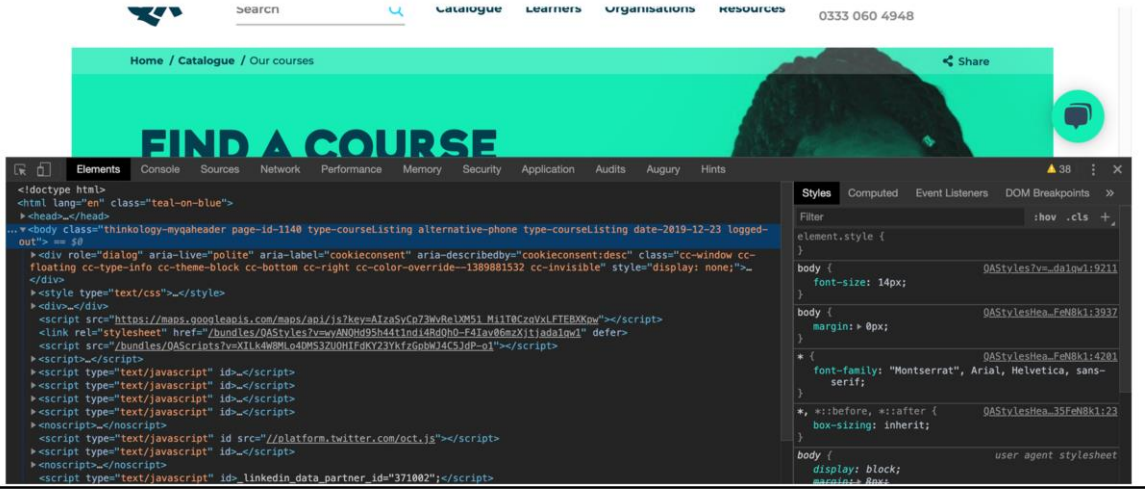




Walkthrough – Chrome development tools

Most browsers have development and debugging tools

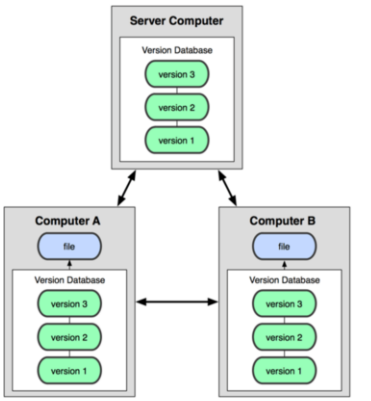
Incredibly useful for testing and management



Distributed Version Control Systems

DVCS do not just check out the most local snapshot of a file

- They mirror the repository
- so each checkout is like a backup of the repository



GIT as a DVCS

GITs origin are in Linux Development and is open source

- Its goals were to create a DVCS system that was:

Fast

Simple

Strong support for non-linear development

Fully distributed

Able to handle large projects like the Linux kernel efficiently

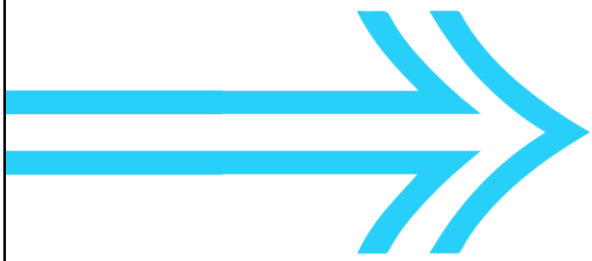
Node & NPM

Node.js is an open source command line tool for server side JS.

- The script is executed by the V8 JavaScript engine.

NPM manages dependencies for an application via the command line.

QuickLab Environment Set

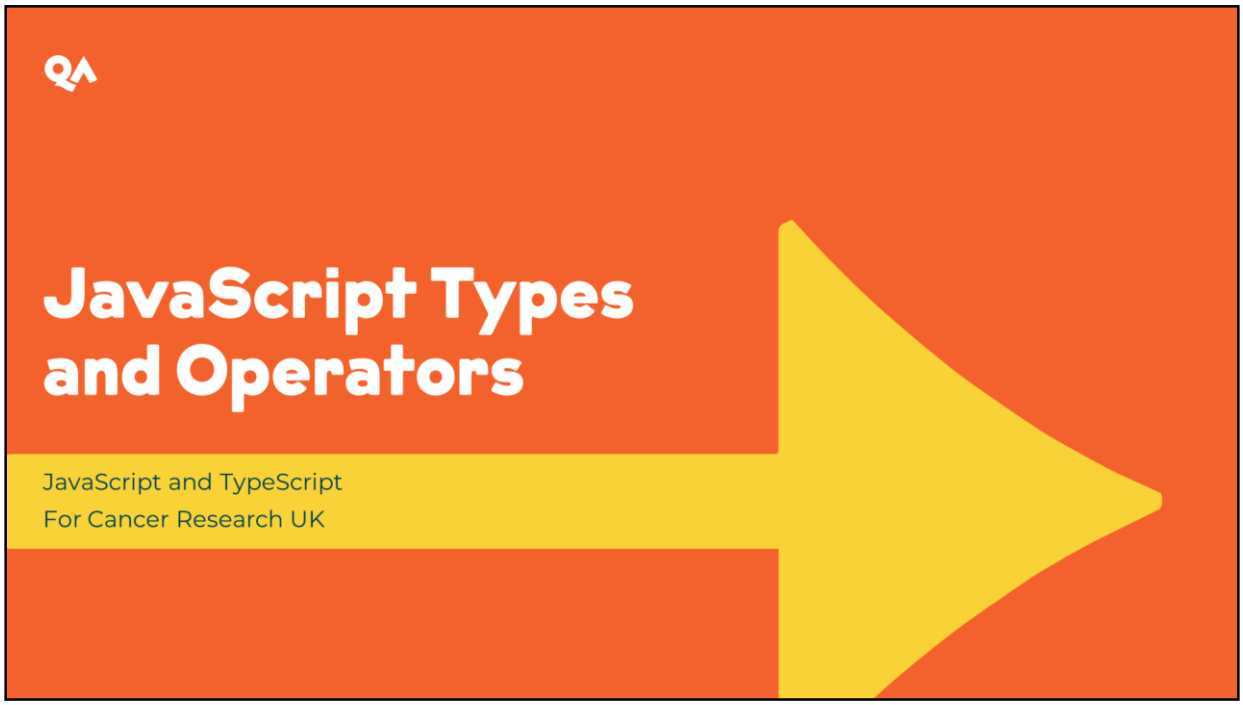


In this QuickLab you will:

- Download, install and/or update VSCode
- Download and Install Node.js and npm

Objectives

- To be able to set up a modern developer's environment


JavaScript Types
and Operators
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To be aware of the primitive and reference types in JavaScript
- To be able to declare constants and variables
- To be able to determine the type of a variable programmatically
- To be able to work with string and number functions
- To understand what operators can be used in JavaScript

Introduction

In this module, you will learn to:

Declare variables

Understand types

- Primitive types
  - Strings
  - Numbers
  - Booleans
  - Undefined
  - Nulls
  - Symbol
- Reference types

Declaring variables

Declaring variables

- const, let and var
- With and without assignment
- Do not use implicit declaration

let – a block-scoped variable (don't worry – we'll discuss what block-scoped means later)

const - the same as **let**, but must be initialised at declaration and cannot be changed

var – a function-scoped variable whose declaration is hoisted and can lead to confusing code! To be avoided now that we have **let** and **const**

What should you use? **const** where possible. **let** when you need it to change

Variable Declarations	
`x = 10;`	`// implicit - DO NOT USE`
`let y;`	`// explicit without assigment`
`let y = 15;`	`// explicit with assignment`
`const z = 10;`	`// constant with assignment`

Variables in JavaScript should be declared with the **var** operator followed by the variable name. If you look above, you will see the variable **x** does not follow this rule. It is valid, just terribly bad practice and has an impact on variable scope as we will examine later.

The second variable, **y**, is valid but currently uninitialised. This is perfectly valid if somewhat dangerous. A JavaScript variable can be assigned at a later point and its type will be set. In fact, that is one of the most interesting things about JavaScript variables – they can change type part way through their life.

We must also be mindful of variable names and try to follow conventions. Variables can start with a $ or an _ but these are reserved for special situations that we will discuss later in the course. For now, our variables will be alphanumeric-based. Most variables will follow a specific naming convention:

- camelCasing – used for variable names
- PascalCasing – used for objects (discussed later)
- sHungarianNotation – where the datatype of the variable prefixes the variable name. This can be useful in JavaScript, when the variable is not set with an explicit type making the code more human readable

It is usually best to have a human-readable variable name, but more experienced developers attempting to obfuscate their code go for terse names.

Declaring variables

Variable names

- Start with a letter , "_" or "$"
- May also include digits
- Are case sensitive
- Cannot use reserved keywords
- E.g. int, else, case

Best practice is to use camelCase for variable names

Variables in JavaScript should be declared with the **var** operator followed by the variable name. If you look above, you will see the variable **x** does not follow this rule. It is valid, just terribly bad practice and has an impact on variable scope as we will examine later.

The second variable, **y,** is valid but currently uninitialised. This is perfectly valid if somewhat dangerous. A JavaScript variable can be assigned at a later point and its type will be set. In fact, that is one of the most interesting things about JavaScript variables – they can change type part way through their life.

We must also be mindful of variable names and try to follow conventions. Variables can start with a $ or an _ but these are reserved for special situations that we will discuss later in the course. For now, our variables will be alphanumeric-based. Most variables will follow a specific naming convention:

- camelCasing – used for variable names
- PascalCasing – used for objects (discussed later)
- sHungarianNotation – where the datatype of the variable prefixes the variable name. This can be useful in JavaScript when the variable is not set with an explicit type making the code more human readable

It is usually best to have a human-readable variable name, but more experienced developers attempting to obfuscate their code go for terse names.

JavaScript types

Primitive data types

- Boolean
- Number
- String
- Undefined
- Null
- Symbol

Object

Dynamically typed

- Data types not declared and not known until runtime
- Variable types can mutate

Interpreted

- Stored as text
- Interpreted into machine instructions and stored in memory as the program runs

As we have discussed, variables in JavaScript can be dynamically created and assigned variables at run time. The JavaScript interpreter in the browser parses the instructions and creates a memory location holding the data.

JavaScript provides a limited number of types that we will explore over the next few pages. If you are used to more full-fat programming languages, like Java or C#, you may be wondering where the doubles and precision numeric types are among others. You simply do not have them in JavaScript and that makes life a little more interesting.

Primitives and Object types

JavaScript can hold two types

Primitives

- Primitive values are immutable pieces of data
- Their value is stored in the location the variable accesses
- They have a fixed length
- Quick to look up

Object

- Objects are collections of properties
- The value stored in the variable is a reference to the object in memory
- Objects are mutable

When we create a variable, you are actually instructing the JavaScript interpreter to grab hold of a location in memory. When a value is assigned to a variable, the JavaScript interpreter must decide if it is a primitive or reference value. To do this, the interpreter tries to decide if the value is one of the primitive types:

- Undefined
- Null
- Boolean
- Number
- String
- Symbol

Each primitive type takes up a fixed amount of space. It can be stored in a small memory area, known as the stack. Primitive values are simple pieces of data that are stored on the stack. This means their value is stored directly in the location that the variable accesses.

If the value is a reference, then the space is allocated on the heap. A reference value can vary in size, so placing it on the stack would reduce the speed of variable lookup. Instead, the value placed in the variable's stack space is the address of a location in the heap where the object is stored.

The typeof operator

The **typeof** operator takes on parameter the value to check

The typeof operator	
`const TYPE_TEST = "string value";`	
`console.log(typeof TYPE_TEST)`	`//outputs "string"`
`console.log(typeof 95)`	`//outputs "number"`

Calling **typeof** on a variable or value returns one of the following

- number
- boolean
- string
- undefined
- symbol
- object (if a null or a reference type)

`typeof` is an incredibly useful operator in the weakly-typed JavaScript language. It allows us to evaluate the type of a variable or value. It is extremely useful with primitives but null and reference types always evaluate to the generic object type.

The undefined type

A variable that has been declared but not initialised is **undefined**

The undefined type	
let age;	
console.log(typeof age);	//returns undefined

A variable that has not been declared will also be **undefined**

- The **typeof** operator does not distinguish between the two

The undefined type	
let boom;	
console.log(typeof boom);	//returns undefined

- It is a good idea to initialise variables when you declare them

A variable that has not been initialised is undefined. As we will see, undefined and null can have a lot of similar properties, if we were to use them in operator statements, so we should preferably give a variable a type even if that type is null.

null is not undefined

`null` and `undefined` are different concepts in JavaScript

- `undefined` variables have never been initialised
- `null` is an explicit keyword that tells the runtime it is 'empty'

```
let userID = null;
console.log(userID); //returns null
```

There is a foobar to be aware of with `null`:

- `undefined` is the value of an uninitialised variable
- `null` is a value we can assign to represent objects that don't exist

```
let userID = null;
console.log(userID == undefined); //returns true
```

NULL is very useful and carries a much more implicit meaning than an undefined and uninitialised variable. If the value of a variable is to be set to a reference object such as an array, be careful in your code because a variable that is undefined will evaluate the same as null.

The Boolean type

Boolean can hold two values – **`true`** and **`false`**

These are reserved words in the language:

```
var loggedOn = false;
console.log(loggedOn); //returns false
```

When evaluated against numbers, you can run into issues

- **`false`** is evaluated as **`0`**
- **`true`** can be evaluated to **`1`**

The Number type

Always stored as 64-bit values
If bitwise operations are performed, the 64-bit value is rounded to a 32-bit value first
There are a number of special values

Constant	Definition
`Number.NaN or NaN`	Not a number
`Number.Infinity or Infinity`	Greatest possible value (but no numeric value)
`Number.POSITIVE_INFINITY`	Positive infinity
`Number.NEGATIVE_INFINITY`	Negative infinity
`Number.MAX_VALUE`	Largest possible number represented in the 64-bits
`Number.MIN_VALUE`	Smallest possible number represented in the 64-bits

In JavaScript, numbers are always stored as 64-bit values. However, if you perform a bitwise operation then the value is truncated to 32-bits. This is important to remember when you require a large bit field. During regular arithmetic, division can always produce a result with fractional parts.

There are a number of special values that are listed above and a series of object functions/methods in the JavaScript documentation at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

Method	Description
`toExponential(n)`	Converts a number into an exponential notation
`toFixed(n)`	Formats a number with n numbers of digits after the decimal
`toPrecision(n)`	Formats a number to x length
`toString(n)`	Converts a Number object to a string
`valueOf()`	Returns the primitive value of a Number object

The String type

Immutable series of zero or more Unicode characters

- Modification produces a new string
- Can use single (') or double quotes (") or backticks (`)
  - BACKTICK are preferred
- Primitive and not a reference type

String concatenation is expensive

Back-slash (\) used for escaping special characters

As a rule, always use backticks (`)

Escape	Output
`\'`	'
`\"`	"
`\\`	\
`\b`	Backspace
`\t`	Tab
`\n`	Newline
`\r`	Carriage return
`\f`	Form feed
`\ddd`	Octal sequence
`\xdd`	2-digit hex sequence
`\udddd`	Unicode sequence (4-hex digits)

As with many other languages, strings in JavaScript are an immutable sequence of zero or more Unicode characters. If we modify a string, for example by concatenation, then a new string is allocated. We can use single or double quotes or backticks in JavaScript.

As is common with many languages, certain special characters must be represented as an escape sequence a back-slash followed either by the character itself, by a significant letter or by a code as shown in the table above.

String Concatenation and Interpolation

Adding 2 (or more strings) is an expensive operation due to the memory manipulation required

To concatenate a string the + operator is used

```
let str1 = "5 + 3 = ";
let value = 5 + 3;
let str2 = str1 + value
console.log(str2); // 5 + 3 = 8
```

Template literals (introduced in ES2015) allow for strings to be declared with JavaScript expressions that are evaluated immediately using ${} notation

```
let str2 = `5 + 3 = ${5 + 3}`;
console.log(str2); // 5 + 3 = 8
```

String functions

The String type has string manipulation methods, including

Method	Description
`indexOf()`	Returns the first occurrence of a character in a string
`charAt()`	Returns the character at the specified index
`toUpperCase()`	Converts a string to uppercase letters

- Where n will be a number with a value of 13

```
let str = "Hello world, welcome to the universe.";
let n = str.indexOf("welcome");
```

Method	Description
`charCodeAt()`	Returns the unicode of the character at the specified index
`concat()`	Joins strings and returns a copy of the joined string
`fromCharCode()`	Converts unicode values to characters
`lastIndexOf()`	Returns the position of last found occurrence from a string
`slice()`	Extracts part of a string and returns a new string
`split()`	Splits a string into an array of substrings
`substr()`	Extracts from a string specifying a start position and number of characters
`substring()`	Extracts from a string between two specified indices
`toLowerCase()`	Converts a string to lower case

Review

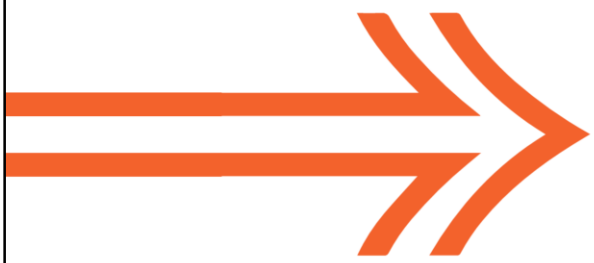
Primitive variables

- Value types

Understand types

- There are six primitive types
- There are 2 other types – Object and Symbol
- Types can mutate

QuickLab 2 – Exploring Types



In this QuickLab you will:

- Create a number variable
- Create and manipulate a string

Operators – assignment and arithmetic

Operators allow us to work with types in tasks such as

- Mathematic operations
- Comparisons

They include

- Assignment:

Assignment	=
Shorthand Assignment	+= -= *= /= %=

- Arithmetic:

Arithmetic	
Addition, subtraction	+ -
Multiplication, division, modulus	* / %
Negation	-
Increment, decrement	++ --
Power	**

JavaScript has all the operators that you would expect for a modern language; in general, they follow the same representation as first became popular in the C language.

In mathematic expressions, there is an order of precedence, e.g. **`5 + 3 * 10`** returns a value of 35 because the multiplication is dealt with before the addition.

The following piece of code uses some of the assignment operators to do the same thing. JavaScript programmers like to do things in as few characters as possible:

```
var x = 0;
x = x+ 1; //x is now 1
x+= 1; //x is now 2
x++; //x is now 3
X**2; //x is now 9
```

Operators – Relational and Boolean

Relational and Boolean operators evaluate to true or false

- Relational:

Relational	
Less than, greater than	`< >`
Less than or equal, greater than or equal	`<= >=`
Equals, not equals	`== === !=`

- Boolean:

Boolean	
AND, OR	`&&  ||`
NOT	`!`

The Boolean logical operators short-circuit

- Operands of `&&` and `||` are evaluated strictly left to right and are only evaluated as far as necessary

The Boolean AND and OR operators have 'short-circuit' evaluation. This means that when an expression involving them is evaluated, it is only evaluated as far as is necessary. For example, consider the expression:

```
if (exprA && exprB)
```

If exprA is false, then exprA && exprB must also be false, so there is sometimes no point evaluating exprB. exprB will only be evaluated if exprA is true; indeed, the && operator will simply return the value of exprB if exprA is true.

Type checking

JavaScript is a loosely-typed language

```
let a = 2;
let b = "two";
let c = "2";
console.log(typeof a);          // logs "number"
console.log(typeof b);          // logs "string"
console.log(typeof c);          // logs "string"
```

JavaScript types can mutate and have unexpected results

```
console.log(a * a);             // logs 4
console.log(a + b);             // logs 2two
console.log(a * c);             // logs 4
console.log(typeof (a * a));    // logs "number"
console.log(typeof (a + b));    // logs "string"
console.log(typeof (a * c));    // logs "number"
```

When we 'add' a string and a number using the + operator, JavaScript assumes we're trying to concatenate the two, so it creates a new string. It would appear to change the number's variable type to string. When we use the multiplication operator (*) though, JavaScript assumes that we want to treat the two variables as numbers.

The variable itself remains the same throughout, it's just treated differently. We can always explicitly tell JavaScript how we intend to treat a variable; but, if we don't, we need to understand just what JavaScript is doing for us. Here's another example:

```
alert(a + c);// alerts 22
alert(a + parseInt(c));// alerts 4
```

Quick exercise – checking for equality and type

Type in a type insensitive language can be 'interesting'

```
let a = 2;
let b = "2";
let c = (a == b);
```

What is the value of **c**? **true** or **false**?

```
var a = 2 ;
var b = "2";
var c = (a === b);          // returns ?
```

There is a strict equality operator, shown as **===**

```
var a = true; var b = 1;
alert(a == b);              // ???
alert(a === b);             // ???
alert(a != b);              // ???
alert(a !== b);             // ???
```

Two = signs together, ==, is known as the equality operator, and establishes a Boolean value. In our example, the variable will have a value of true, as JavaScript compares the values before and after the equality operator, and considers them to be equal. Using the equality operator, JavaScript pays no heed to the variable's type, and attempts to coerce the values to assess them.

Switch out the first equal sign for an exclamation mark, and you have yourself an inequality operator (!=). This operator will return false if the variables are equal, or true if they are not.

In JavaScript 1.3, the situation became even less simple, with the introduction of one further operator: the strict equality operator, shown as ===.

The strict equality operator differs from the equality operator, in that it pays strict attention to type as well as value when it assigns its Boolean. In the above case, d is set to false; while a and b both have a value of 2, they have different types. And, as you might have guessed, where the inequality operator was paired with the equality operator, the strict equality operator has a corresponding strict inequality operator:

```
var f = (a !== b);
```

In this case, the variable will return true, as we know the two compared variables are of different types, though their values are similar.

Type conversion

Implicit conversion is risky – better to safely convert

You can also use explicit conversion

- **`eval( )`** evaluates a string expression and returns a result
- **`parseInt( )`** parses a string and returns an integer number
- **`parseFloat( )`** parses a string, returns a floating-point number

```
let s = "5";
let i = 5;
let total = i + parseInt(s); //returns 10 not 55
```

You can also check if a value is a number using **`isNaN()`**

```
isNaN(s);     // returns true
!isNaN(i);    //returns true
```

As we discovered, type mismatching can cause some serious logical issues while working with JavaScript, and is often better to explicitly take control of type conversion. There are three key functions here:

`eval()` – commonly used to create string arrays. We will examine this function in more depth later in the course.

`parseInt()` – takes a value or variable that is not currently a number and tries to convert its value into a numeric type. Specifically, it is looking for numeric values and any decimal points. It preserves anything to the left of the decimal point, so:

parseInt(55.95) would return 55, note that no rounding has occurred

parseInt("55.95boom!") would also return 55

`parseFloat()` – works as per parseInt(), but preserves numeric values after the decimal point:

parseFloat(55.95) would return 55.95 as a number
parseFloat("55.95boom!") would also return 55.95 also

Both **parseInt** and **parseFloat** return a NaN error object if the conversion can not occur, which you can detect using the **isNan()** function.

Review

Primitive variables

- Value types

Understand types

- There are six primitive types
- There are 2 other types – Object and Symbol
- Types can mutate

Objectives

- To be aware of the primitive and reference types in JavaScript
- To be able to declare constants and variables
- To be able to determine the type of a variable programmatically
- To be able to work with string and number functions


JavaScript Arrays,
Objects and Collections
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To be able to create, access and manipulate arrays
- To be able to declare, access, manipulate and destructure objects
- To be able to create, access and manipulate collections


JavaScript Arrays

Creating arrays

Arrays hold a set of related data, e.g. students in a class

- The default approach is accessed by a numeric index

As with the rest of JavaScript, the elements in arrays are loosely-typed. This means that each separate element of an array can contain a different type. This is useful and dangerous at once and brings us back to the consideration that type matters.

Array indices in JavaScript start at 0. Arrays have a `length` property equal to the length of the array; note that an array of length of 6 will have elements 0 to 5.

JavaScript arrays are *sparse* – this means that there can be holes in an array. The example above shows an array that initially has values assigned to elements number 0 and 5. This results in an array that contains only two elements. Elements numbered 1 to 4 exist but, if referenced, will yield the special value `undefined`.

Note that once a value has been assigned to an element of an array, that array element is now defined, and cannot be removed. Assigning `null` to it changes the value but does not remove the array element. In fact, simply *referring* to the element is enough to create it.

Arrays in JavaScript will grow dynamically from its initial size. It is also possible to shrink an array by assigning a smaller value to the `length` property than it currently holds. Do not assume that elements with indexes greater than or equal to the new value exist.

Creating arrays

Arrays in JavaScript have some idiosyncrasies

- They can be resized at any time
- They index at 0

  So **`Array(3)`** would have elements with indexes 0, 1 and 2
- They can be *sparsely* filled

  Unassigned parts of an array are **`undefined`**
- They can be created in short hand using just square brackets

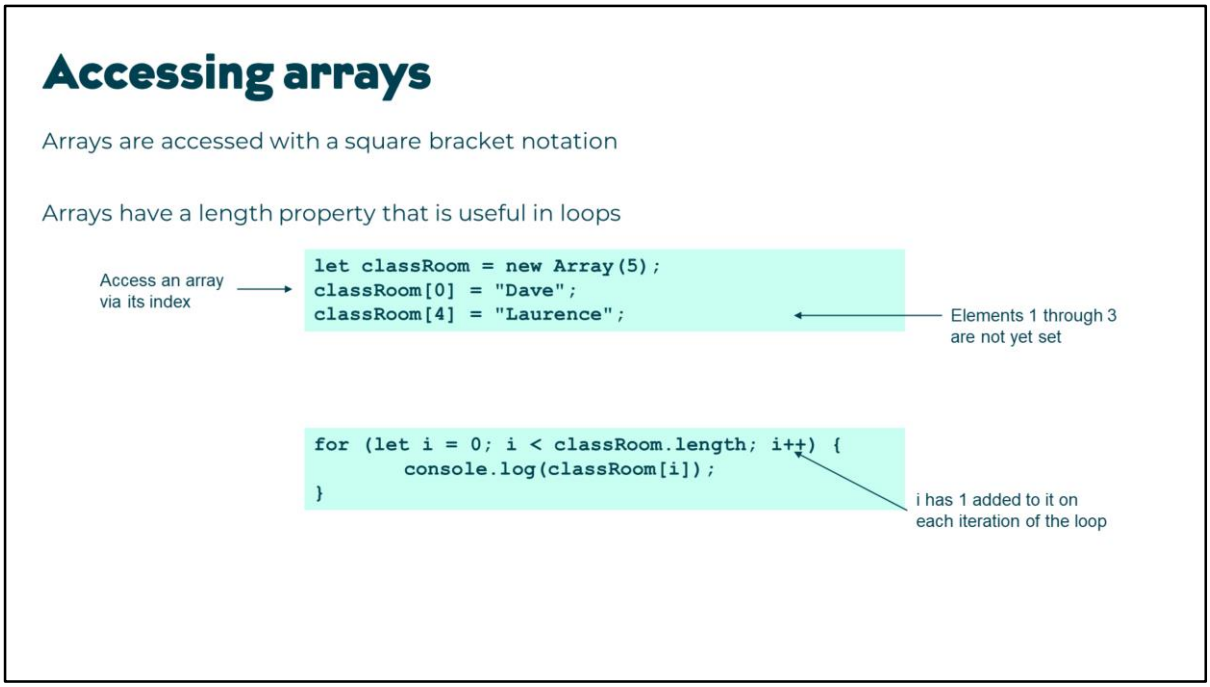
As with the rest of JavaScript, the elements in arrays are loosely-typed. This means that each separate element of an array can contain a different type. This is useful and dangerous at once and brings us back to the consideration that type matters.

Array indices in JavaScript start at 0. Arrays have a `length` property equal to the length of the array; note that an array of length of 6 will have elements 0 to 5.

JavaScript arrays are *sparse* – this means that there can be holes in an array. The example above shows an array that initially has values assigned to elements number 0 and 5. This results in an array that contains only two elements. Elements numbered 1 to 4 exist but, if referenced, will yield the special value `undefined`.

Note that once a value has been assigned to an element of an array, that array element is now defined, and cannot be removed. Assigning `null` to it changes the value but does not remove the array element. In fact, simply *referring* to the element is enough to create it.

Arrays in JavaScript will grow dynamically from its initial size. It is also possible to shrink an array by assigning a smaller value to the `length` property than it currently holds. Do not assume that elements with indexes greater than or equal to the new value exist.



Arrays allow us to store a related set of data. Arrays store data in a list of elements; we access a particular elements by specifying the name of the array and the element index within square brackets, e.g.

```
myArray[0];
```

The first element of the array is always accessed through [0].

In the above example, a 5 element array is created and elements 1 through 3 are not set. In this situation, they will have a value of **`undefined.`** If you remember what we have learnt from the module on types, this makes perfect sense. They have been created but not initialised.

Arrays also have a length property. This will always reflect the exact number of elements an array contains. As you can see, this is immensely useful as it allows us to create loops that will always run as many times as is needed.

Array object methods

Array objects have methods

reverse()

join([separator])

• Joins all the elements of the array into one string, using the supplied separator or a comma

sort([sort function])

• Sorts the array using string comparisons by default

• Optional sort function compares two values and returns sort order

```
let fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
let fruitString = fruit.join("---");

console.log(fruitString); // Apples---Pears---Bananas---Oranges
```

Array objects have various methods:

- **`reverse`** reverses the order of the elements in the array, so that the last element in the array becomes the first, and so on. This method operates directly on the array itself, rather than returning a new array
- **`join`** returns a string formed by joining together all the elements in the array using the supplied separator. If no separator is supplied, a comma is used. The separator may be any string (including an empty one). Undefined array elements are represented by a null string, meaning that two or more separators will appear next to each other
- **`Sort`** sorts the array. If no function argument is supplied, the array elements are temporarily converted to strings and sorted in standard dictionary order. To make the sort generic, it is possible to supply a function as an argument. This function will be called by the sort routine as necessary to compare two values in the array. The function should have the form: `compare(a, b)` and should return a value less than 0 if a should be sorted less than b, 0 if they are equal, and greater than 0 if a should be sorted greater than b

Pop and push array methods

The **push()** method

- Adds a new element to the end of the array
- Array's length property is increased by one
- This method returns the new length of the array

```
let fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
console.log(fruit.push('Lemons'));                          //5

//fruit is now ['Apples', 'Pears', 'Bananas', 'Oranges', 'Lemons']
console.log(fruit);
```

The JavaScript `Array.push()` method adds a new element to the end of the array. When doing so, the array's length property increases by one. After adding the new element to the end of the array, this method returns the new length of the array.

The JavaScript `Array.pop()` method removes the last element from the end of the array. When doing so, the array's length property decreases by one. After removing the last element from the end of the array, this method returns the array element that was removed.

Pop and push array methods

The **pop()** method

- Removes the last element from the end of the array
- The array's length property is decreased by one
- This method returns the array element that was removed

```
let fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
console.log(fruit.pop());                                    //Oranges

//fruit is now ['Apples', 'Pears', 'Bananas']
console.log(fruit);
```

The JavaScript `Array.push()` method adds a new element to the end of the array. When doing so, the array's length property increases by one. After adding the new element to the end of the array, this method returns the new length of the array.

The JavaScript `Array.pop()` method removes the last element from the end of the array. When doing so, the array's length property decreases by one. After removing the last element from the end of the array, this method returns the array element that was removed.

Shift and unshift array methods

The **`unshift()`** method

- Adds a new element to the beginning of the array
- Array's length property is increased by one
- This method returns the new length of the array

```
let fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
console.log(fruit.unshift('Kiwis'));                    //5

//fruit is now ['Kiwis','Apples', 'Pears', 'Bananas', 'Oranges']
console.log(fruit);
```

The JavaScript `Array.shift()` method removes the first element from the beginning of the array. When doing so, the array's length property decreases by one. After removing the first element from the beginning of the array, this method returns the array element that was removed.
The JavaScript `Array.unshift()` method adds a new element to the beginning of the array. When doing so, the array's length property increases by one. After adding the new element to the beginning of the array, this method returns the new length of the array.

Shift and unshift array methods

The **`shift()`** method

- removes the first element from the beginning of the array
- Array's length property is decreased by one
- This method returns the array element that was removed

```
let fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
console.log(fruit.shift());                                    //Apples

//fruit is now ['Pears', 'Bananas', 'Oranges']
console.log(fruit);
```

The JavaScript `Array.shift()` method removes the first element from the beginning of the array. When doing so, the array's length property decreases by one. After removing the first element from the beginning of the array, this method returns the array element that was removed.
The JavaScript `Array.unshift()` method adds a new element to the beginning of the array. When doing so, the array's length property increases by one. After adding the new element to the beginning of the array, this method returns the new length of the array.

Methods Introduced in ES2015

`Array.from()` creates a real Array out of array-like objects

```
let formElements = document.querySelectorAll('input, select, textarea');
formElements = Array.from(formElements);
formElements.push(anotherElement);                                    //works fine!
```

`Array.prototype.find()` returns the first element for which the callback returns true

```
["Chris","Bruford",22].find(function(n) { return n === "Bruford"});        //"Bruford"
```

Similarly **`findIndex()`** returns the index of the first matching element

```
["Chris","Bruford",22].findIndex(function(n) { return n === "Bruford"});   //1
```

`fill()` overrides the specified elements

```
["Chris","Bruford",22,true].fill(null);       //[null,null,null,null]
["Chris","Bruford",22,true].fill(null,1,2);   //["Chris",null,null,true]
```

Methods Introduced in ES2015

`.entries()`, `.keys()` & `.values()` each return a sequence of values via an iterator:

```
let arrayEntries = ["Chris","Bruford",22,true].entries();
console.log(arrayEntries.next().value);                    //[0,"Chris"]
console.log(arrayEntries.next().value);                    //[1,"Bruford"]
console.log(arrayEntries.next().value);                    //[2,22]
```

```
let arrayKeys = ["Chris","Bruford",22,true].keys();
console.log(arrayKeys.next().value);                       //0
console.log(arrayKeys.next().value);                       //1
console.log(arrayKeys.next().value);                       //2
```

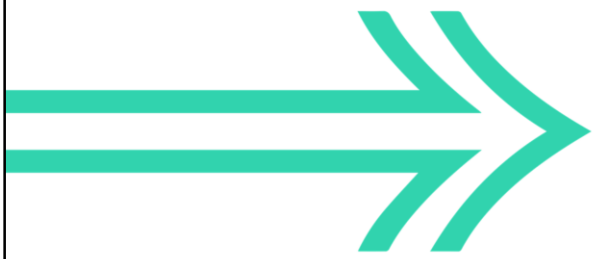
```
let arrayValues = ["Chris","Bruford",22,true].values();
console.log(arrayValues.next().value);                     //"Chris"
console.log(arrayValues.next().value);                     //"Bruford"
console.log(arrayValues.next().value);                     //22
```

for...of loop

The for-of loop is used for iterating over **iterable** objects (more on that later!)
For an array if means we can loop through the array, returning each element in turn

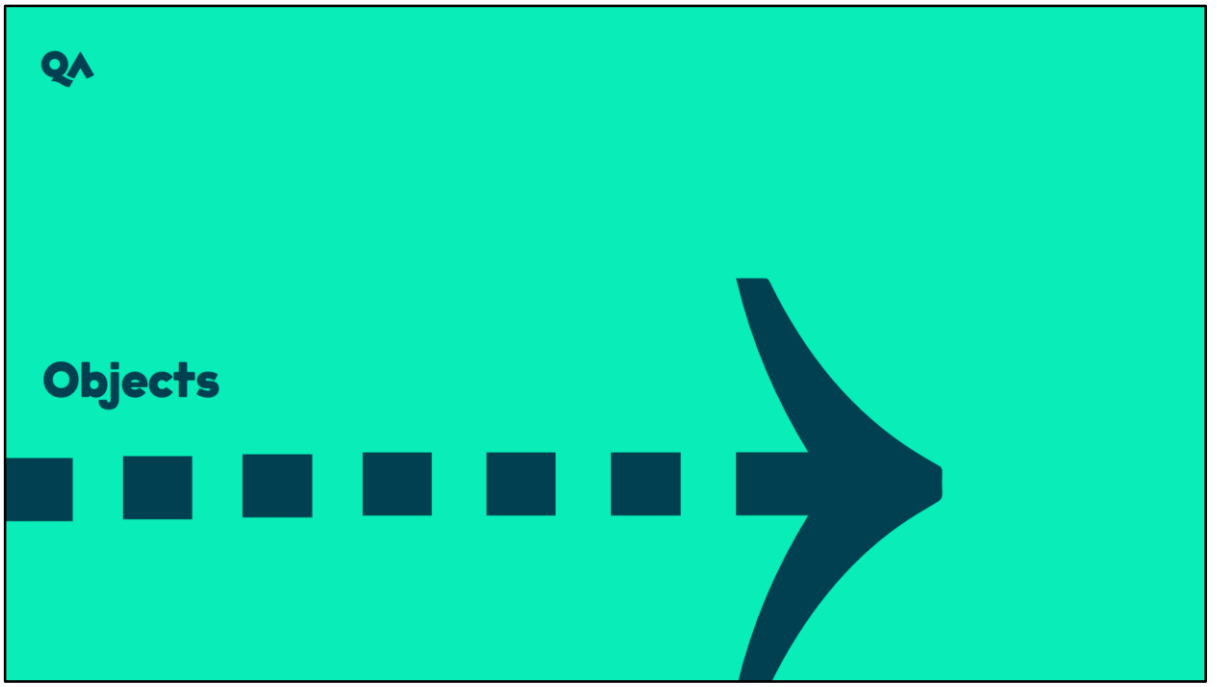
```
//will print 1 then 2 then 3
let myArray = [1,2,3,4];
for (let el of myArray) {
    if (el === 3) break;
    console.log(el);
}
```

QuickLab 2 – Arrays



In this QuickLab you will:

- Access an array
- Manipulate the contents of an array


Objects

Objects – data structures

Objects in JavaScript are key – value pairs

- Where standard arrays are index – value pairs
- Keys are very useful for providing semantic data

```
var student = new Object();
student["name"] = "Caroline";
student["id"] = 1234;
student["courseCode"] = "LGJAVSC3";
```

The object can have new properties added at any time

- Known as an expando property

```
student.email = "caroline@somewhere.com";
```

Many programming paradigms can be used within JavaScript, one of which is object oriented programming. It provides a series of input objects we will shortly be examining that include window and document and their numerous offspring, which are very important – but they are defined by the browser, not by the programmer. We can define our own objects.
Objects are principal objects in JavaScript. If the variable does not refer to a primitive type, it is an object of some kind. In principal, they are very similar to the concepts of arrays but, instead of an indexed identifier, a string-based key is used (in fact – Arrays in JavaScript are nothing more than special objects).

the property…

```
student["name"]
```

can also be read or written by calling:

```
student.name
```

Objects – accessing properties

The key part of an object is often referred to as a property

- It can be directly accessed

```
student.email ;
student["email"];
```

When working with objects, the for in loop is very useful

- key holds the string value of the key
- student is the object
- So it loops for each property in the object

```
for (let key in student) {
    console.log(`${key}:${student[key]}`);
}
```

Objects are collections of properties and every property gets its own standard set of internal properties. (We can think of these as abstract properties – they are used by the JavaScript engine but aren't directly accessible to the user. ECMAScript uses the [[*property*]] format to denote internal properties).

One of these properties is [[Enumerable]]. The for-in statement will iterate over every property for which the value of [[Enumerable]] is true. This includes enumerable properties inherited via the prototype chain. Properties with an [[Enumerable]] value of false, as well as *shadowed* properties – i.e. properties that are overridden by same-name properties of descendant objects – will not be iterated.

Objects – literal notation

There is an alternative syntactic approach to defining objects

```
let student2 = { name: "David", id: 1235, courseCode: "LGJAVSC3" };
```

This can be combined into more complex arrays

- Below is an indexed array containing two object literals
- Note the comma separator

```
let classRoom = [
    { name: "David", id: 1235, courseCode: "LGJAVSC3" },
    { name: "Caroline", id: 1234, courseCode: "LGJAVSC3" }
]
```

The above code is a quick implementation for JavaScript object. It initialises the object and sets three properties.

The second example creates an indexed array of object literals

Quick exercise – objects and arrays

If we define the following data

```
let classRoom = [
    { name: "David", id: 1235, courseCode: "LGJAVSC3" },
    { name: "Caroline", id: 1234, courseCode: "LGJAVSC3" }
]
```

What would we have to add to this code to

- Access the inner object
- Display the key value pair

```
for (var i = 0; i < ???? ; i++) {
    for (var key in classRoom[i]) {
        console.log(key + " : " + classRoom[???][???]);
    }
}
```

Enhanced Object Literals

A shorthand for foo:foo assignments – when the property name is the same as the variable you wish to use for the property's value.

Defining methods

```
let power = 200;
let myCar = {
    power
}
```

Maker super calls

```
let myCar = {
    speed = 0,
    power,
    accelerate() { this.speed = this.power / 2 },
}
```

```
let myCar = {
    …
    toString() { return `Car: ${super.toString()}` }
}
```

Dynamic Property Names

Dynamic property names

```
let power = 200;
n = 0;

let myCar = {
    power,
    ["prop_" + ++n]: n
};
```

Object.assign()

The assign() method has been added to copy enumerable own properties to an object

Can use this to merge objects

```
let obj1 = {a: 1};
let obj2 = {b: 2};
let obj3 = {c: 3};

Object.assign(obj1,obj2,obj3);
console.dir(obj1); //{a: 1, b: 2, c: 3}
```

Or copy objects

```
let obj1 = {a: 1};

let obj2 = Object.assign({},obj1);
console.dir(obj2);
```

Everything is an object

JavaScript is an object based programing language

- All types extend from it
- Including functions
- Function is a reserved word of the language

Theoretically, we could define our functions like this

- Then call it using `doStuff();`

```
let doStuff = new Function('alert("stuff was done")');
```

In the above example, we have added all the functionality as a string

- The runtime will instantiate a new function object
- Then pass a reference to the **doStuff** variable
- Allowing us to call it in the same way as any other function

Objects are the building blocks of the JavaScript language. In fact, it is defined as an *object-based programming language.* Absolutely everything we work with in JavaScript has an object at its core. Consider the following code:

```
let x = 5;
let y = new Number(5);
```

Both code implementations create an object of type number and the variable then receives a memory reference to that object. It is important to remember that JavaScript variables are simply pointers to this object.

This concept extends to functions. In the code block above, we see another way of creating a function. The **new** keyword instantiates a function, with the logic passed in as a string. **doStuff** receives a reference to the function. As long as the variable **doStuff** remains in scope, the function remains available.

(N.B. THIS IS A THEORETICAL APPROACH! Never implement functions like this! Later in the course your instructor will show you a pattern called self-executing functions that create a security vulnerability of incredible risk.


Destructuring

Destructuring: Arrays

Providing a convenient way to extract data from objects and arrays

```
let [first, second, third] = [`I`, `Love`, `JavaScript`];
console.log(first);           // I
console.log(second);          // Love
console.log(third);           // JavaScript
```

We can also use default values

```
let [first, second = 7] = [1];
console.log(first); //1
console.log(second); //7
```

Destructuring: Objects

Basic object destructuring:

```
let myObject = {first: `Salt`, second: `Pepper`};
let {first,second} = myObject;

console.log(first);  //Salt
console.log(second); //Pepper
```

We can rename the variables:

```
let myObject = {first: `Salt`, second: `Pepper`};
let {first: condement1, second: condement2} = myObject;

console.log(condement1); //Salt
console.log(condement2); //Pepper
```

Destructuring: Objects

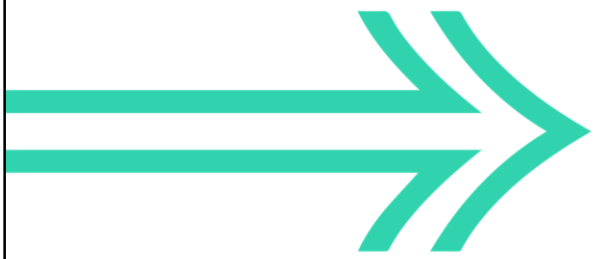
Default values

```
let myObject = {first: `Salt`};
let {first=`Ketchup`, second=`Mustard`} = myObject;

console.log(first);  //Salt
console.log(second); //Mustard
```

Gotcha! Braces on the lhs will be considered a block

```
let a,b;
{a,b} = {a: 5, b: 7};          //syntax error
({a,b} = {a: 5, b: 7});        //okay!
```

QuickLab 3 – Objects



In this QuickLab you will:

- Declare and destructure objects

Review

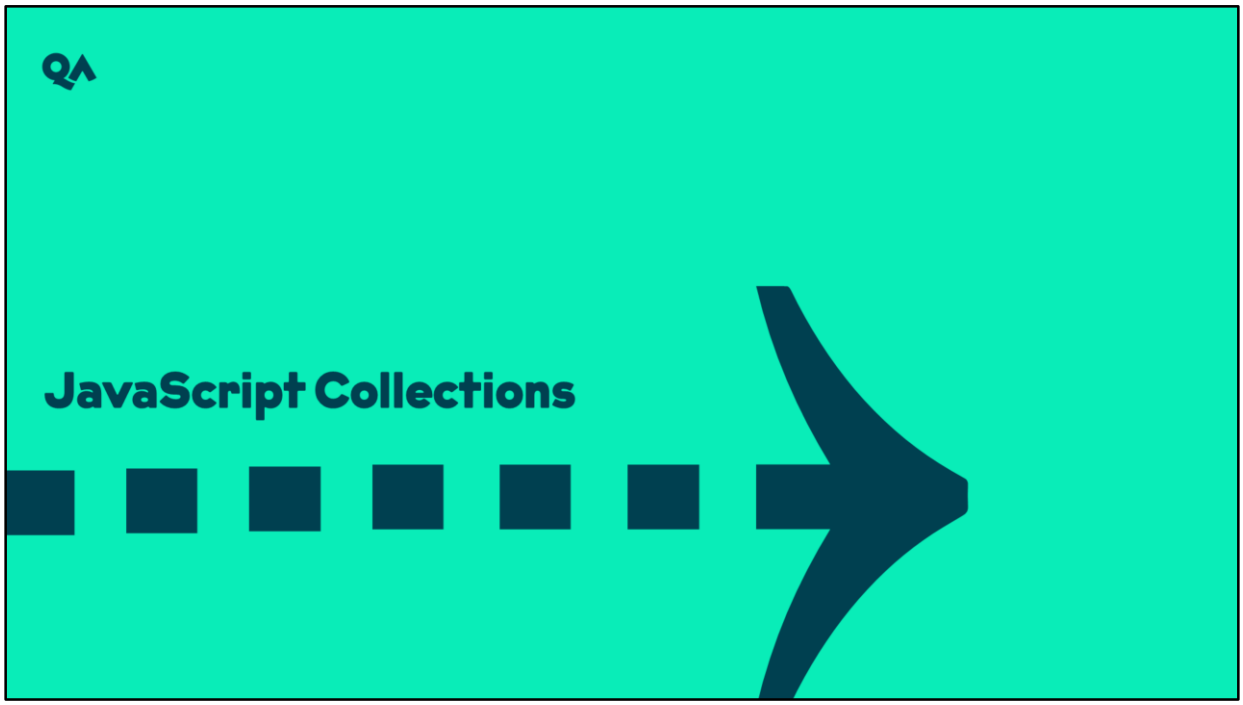
Arrays and Objects are essential collections that allow us to gather data under one roof that can then be acted upon in a coherent and concise manner

JavaScript is an object-based language

- Everything is an object behind the scenes
- Many very useful objects built into JavaScript

We will revisit all three concepts through the course

- Every module in the course builds out of these concepts
- So please speak now if you are unsure on anything!


JavaScript Collections

Maps

Key / Value pairs where both Key and Value can be any type

```
let myMap = new Map([[1,`bananas`],[2,`grapefruit`],[3,`apples`]]);
```

With some helpful methods

```
console.log(myMap.size)                 //3

myMap.set(4, `strawberries`);
console.log(myMap.size);                //4

console.log(myMap.get(4));              //strawberries
console.log(myMap.has(2));              //true

myMap.delete(3);
console.log(myMap.size);                //3

myMap.clear();
console.log(myMap.size);                //0
```

Maps: Iterating

We can iterate over a map using for...of

```
// log all key/value pairs in the map
for (let [key, value] of myMap) {
    console.log(`key: ${key} value: ${value}`);
}
// log all keys in the map
for (let key of myMap.keys()) {
    console.log(`key: ${key}`);
}
// log all values in the map
for (let value of myMap.values()) {
    console.log(`value: ${value}`);
}
// log all entries (key/value pairs) in the map
for (let [key, value] of myMap.entries()) {
    console.log(`key: ${key} value: ${value}`);
}
```

Sets

Sets allow you to store unique values of any type

```
let mySet = new Set();
```

With some helpful methods

```
mySet.add("apples")
mySet.add("bananas")
console.log(mySet.size)                 //2

mySet.add("apples")
console.log(mySet.size)                 //2 (the 2nd apples is not unique)

console.log(mySet.has("apples"));       //true

mySet.delete("apples");
console.log(mySet.size);                //1

mySet.clear();
console.log(myMap.size);                //0
```

Sets: Iterating

We can iterate over a set using for...of

```
// log all key/value pairs in the set
for (let item of mySet) {
    console.dir(item);
}
// log all values in the set
for (let value of mySet.values()) {
    console.log(`value: ${value}`);
}
// same as above for values()
for (let key of mySet.keys()) {
    console.log(`key: ${key}`);
}
// log all entries (key/value pairs) in the set where key and value are the same
for (let [key, value] of mySet.entries()) {
    console.log(`key: ${key} value: ${value}`);
}
```

WeakSets and WeakMaps

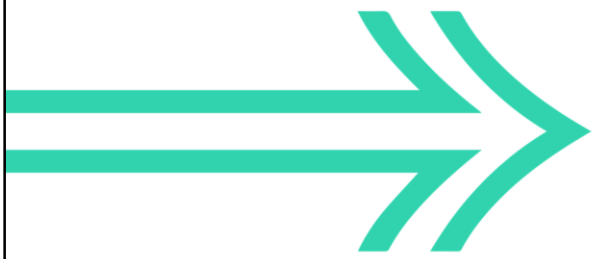
Behave exactly like Map and Set but:

- Do not support iteration methods
- Values in a WeakSet and keys in a WeakMap must be objects

This allows the garbage collector to collect dead objects out of weak collections!

```
// keep track of what DOM elements are moving
let element = document.querySelector(".animateMe");

if (movingSet.has(element)) {
    smoothAnimations(element);
}
movingSet.add(element);
```

QuickLab 3 – Collections



In this QuickLab you will:

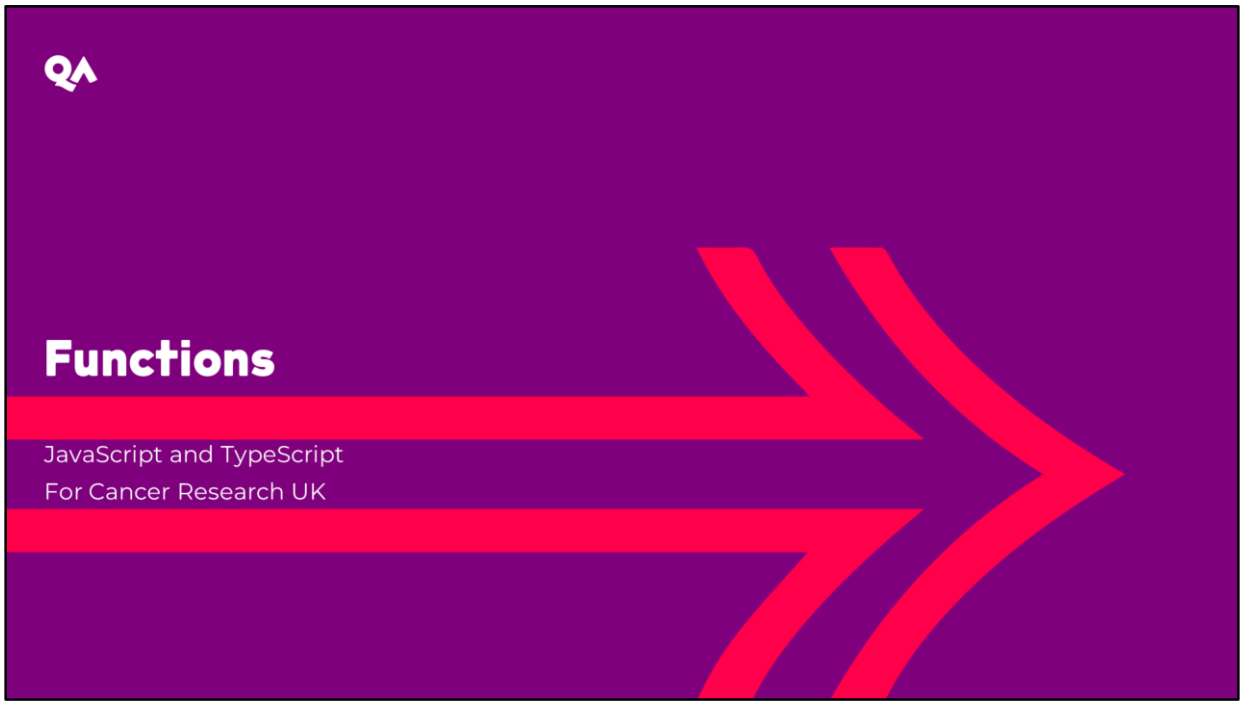
- Create, access and manipulate a collection

Review

Arrays, Objects, Maps and Sets are essential collections that allow us to gather data under one roof that can then be acted upon in a coherent and concise manner

Objectives

- To be able to create, access and manipulate arrays
- To be able to declare, access, manipulate and destructure objects
- To be able to create, access and manipulate collections


Functions
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To understand how to create and call functions in JavaScript
- To understand how scope works in JavaScript
- To be aware of the 'Arrow Function' syntax

Functions

Functions are one of the most important concepts in JavaScript
Functions allow us to block out code for execution when we want

- Instead of it running as soon as the browser processes it
- Also allows us to reuse the same operations repeatedly

  Like console.log();
- Functions are first-class objects and are actually a type of built-in type

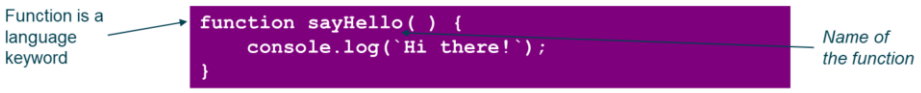
  The keyword function actually creates a new object of type Function

In JavaScript a function is a type. It is a first-class object. Whenever we declare a function, we are actually declaring a variable of type function with the name of the function becoming the name of the variable.

When we invoke a function, we are actually calling a method of that function (a function on the function) called call – the invocation syntax is simply sugar.

Functions – creating

The function keyword is used to create JavaScript functions



Parameters may be passed into a function

```
function sayHelloToSomeone(name) {
    alert(`Hi there ${name}!`);
}
```

It may optionally return a value

```
function returnAGreetingToSomeone(name) {
    return `Hi there ${name}!`
}
```

Functions are created with the reserved word function. The newly created function block has mandatory curled braces and a function name. In the first example above, a simple function has been created that calls an alert and takes no input.

In example two, we are passing in a value this is known as a parameter or argument. It is at this point an optional parameter. If it was passed in blank, it would be undefined as a type. In the exercise, we will consider the idea of passing in the parameter and checking its value.

The first two functions just go off and do 'something'; they would often be known as 'sub routines' and are effectively a diversion from the main program allowing us to reuse code.

The third example uses the return keyword. It calculates or achieves some form of result and then returns this result to the program. It would normally be used in conjunction with a variable as we will see on the next slide.

Functions – calling

Functions once created can be called
Use the function name
Pass in any parameters, ensuring the order
If the function returns, pass back result

```
sayHelloToSomeone(`Dave`);
let r = returnAGreetingToSomeone(`Adrian`);
```

Parameters are passed in as value based
- The parameter copies the value of the variable
- For a primitive, this is the value itself
- For an object, this is a memory address

The function must be declared before it is called. It must be declared in the current block, a previously rendered block or an external script file you have referenced, or else the function call will raise an error.

Parameters need to be passed in the correct order and remember that JavaScript has no type checking externally. We would need to be sure what is going into the function call is not garbage. Any parameters passed in are local copies and do not refer to the original variable.

The `return` statement has two purposes in a function. The first is as a method of flow control: when a return statement is encountered, the function exits immediately, without executing any code which follows the return statement. The second, and more important use of the return statement, returns a value to the caller by giving the function a value.

If a function contains no return statement, or the return statement does not specify a value, then its return value is the special undefined value, and attempting to use it in an arithmetic expression will cause an error.

Default Values & Rest Parameters

Default values were a long standing problem with a fiddly solution

Can provide a value for the argument and if none is passed to the function, it will use the default.

```
function doSomething(arg1, arg2, arg3=5) {
    return(arg1 + arg2 + arg3);
}
console.log(doSomething(5,5));                    //15
```

If the last named argument of a function is prefixed with ... then it's value and all further values passed to the function will be captured as an array:

```
function multiply(arg1, ...args) {
args.forEach((arg,i,array) => array[i] = arg*arg1);
return args;
}
console.log(multiply(5,2,5,10));                  //[10,25,50]
```

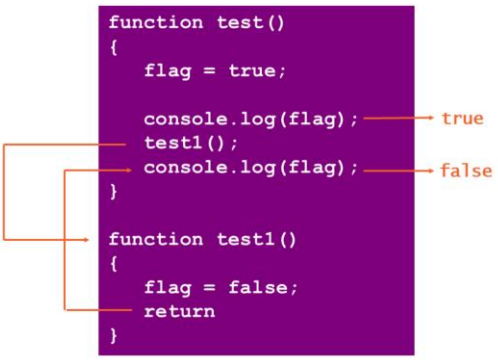
The syntax used as the argument to the args.forEach is known as an Arrow function. This is a shorthand method to declare an anonymous function and is extensively used in JavaScript. The left hand side of the => indicates the arguments that are passed into the function. The right hand side is the code to execute.
This is the same as:

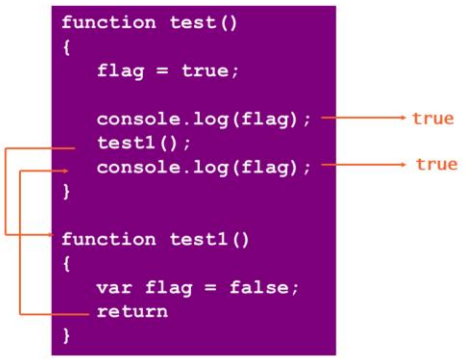
Args.forEach(function(arg, i, array) { array[i] = arg*arg1;});

Functions – scope (1)

Scope defines where variables can be seen

- Use the let keyword to specify scope to the current block
- If you don't use let, then variable has 'global' scope





The **var** keyword has additional semantics: it provides 'scoping' of variables. Variables declared outside the body of a function are always global as expected. Less obvious is that variables created inside a function body are also global. However, variables created explicitly using by the **var** keyword have scope limited to the enclosing function (no matter how deep within the function they are created).

Variables created outside of a function body using the **var** keyword are global. Note that means global to the html page, not just the current SCRIPT block.
In the first example in the slide, setting the value of flag to false in function test1() has the effect of changing the value of the global variable and hence flag in function test().

In the second example, marking the variable flag as **var** in function test1() restricts its scope to that function only. This means that setting the value of the local flag variable does not effect the value of flag set in function test().

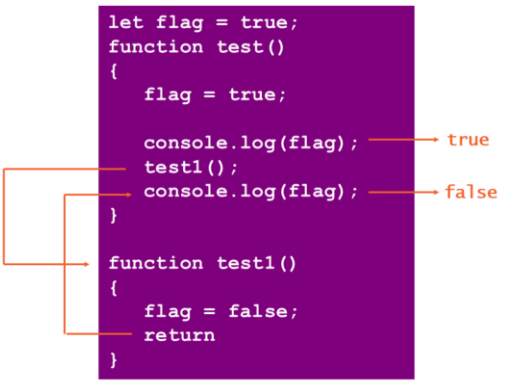
Improper use of scoping can be a source of major debugging headaches.

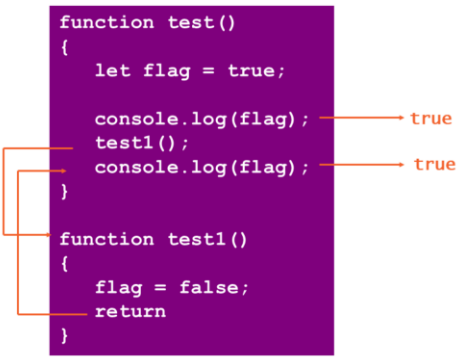
Functions – scope (2)

In the code sample to the left the flag variable is explicitly defined at a higher level

In the code sample to the right it is declared in the scope of test

- Can test1 see it?

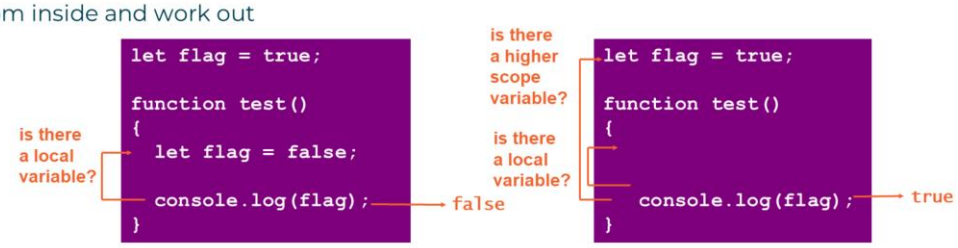




Functions – local vs. global scope

Scope Chains define how an identifier is looked up

- Start from inside and work out



What happens if there is not a local or global variable?

- One is added to global scope!

The scoping rules and the way identifiers are resolved follow an inside to outside flow in JavaScript.

If this code is executing in a function, then the first object to be checked is the function itself: local variables and parameters. After that, the 'global' store of variables is checked.

We discuss functions in greater detail later, but while we are talking about scope: variables declared in a formal parameter list are implicitly local.

```
function test(flag)      // This flag has local scope
{
  flag = false
}
```

The global object

Global object for client-side JavaScript is called window

- Global variables created using the var keyword are added as properties on the global object (window)
- Global functions are methods of the current window
- Current window reference is implicit

```
var a = 7;
alert(b);
```

```
window.a = 7;
window.alert(b);
```

These are equivalent

- Global variables created using the let keyword are NOT added as properties on the window

The global object

Unless you create a variable within a function or block it is of global scope

- The scope chain in JavaScript is interesting
- JavaScript looks up the object hierarchy not the call stack
  This is not the case in many other languages
- If a variable is not seen in scope, it can be accidently added to global
  Like the example in the previous slide

Closures

According to the documentation:

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

- Essentially, a closure is created when a function is declared inside another function
- This pattern is used widely across noteworthy Frameworks and Libraries such as ReactJS

Arrow Functions

Can be declared as **`const`** (or **`let`**) setting a variable name to be a function

This is becoming a common pattern in JavaScript

- Will see it used in Angular, React, etc

```
const someFunction = () => { // Some implementation code };
someFunction();

const someOtherFunction = someArgument => { console.log(someArgument); }
someOtherFunction(`A value to pass in`);                    // outputs value of
                                                                  someArgument

const automaticallyReturningArrowFunction = (num1, num2) => ( num1 * num2 );
console.log(automaticallyReturningArrowFunction(10, 10))    // outputs 100
```

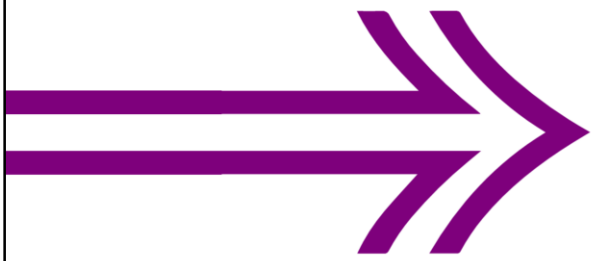
Review

Functions allow us to create re-usable blocks of code

Scope is a critical concept to understand and utilise in your JavaScript programming career

Functions are first-class objects, meaning we can pass them round as we would other objects and primitives

QuickLab 5 – Functions

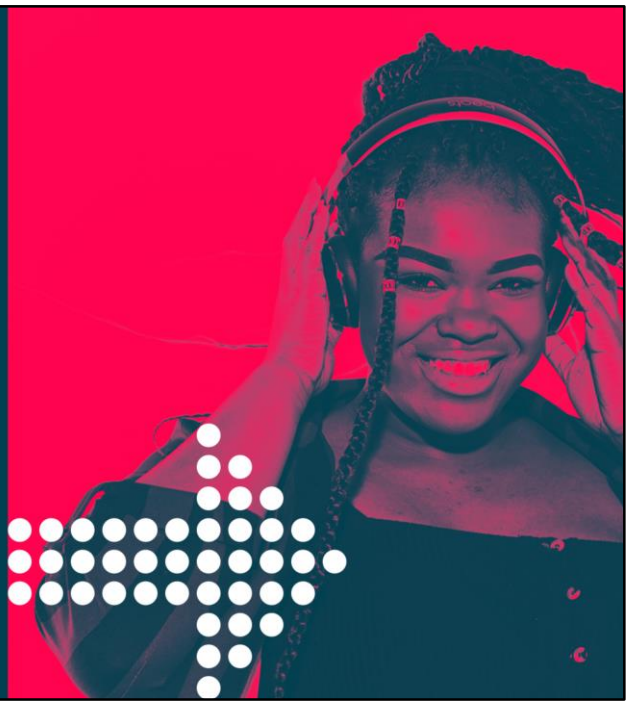


In this QuickLab you will:

- Create functions and investigate scope

Objectives

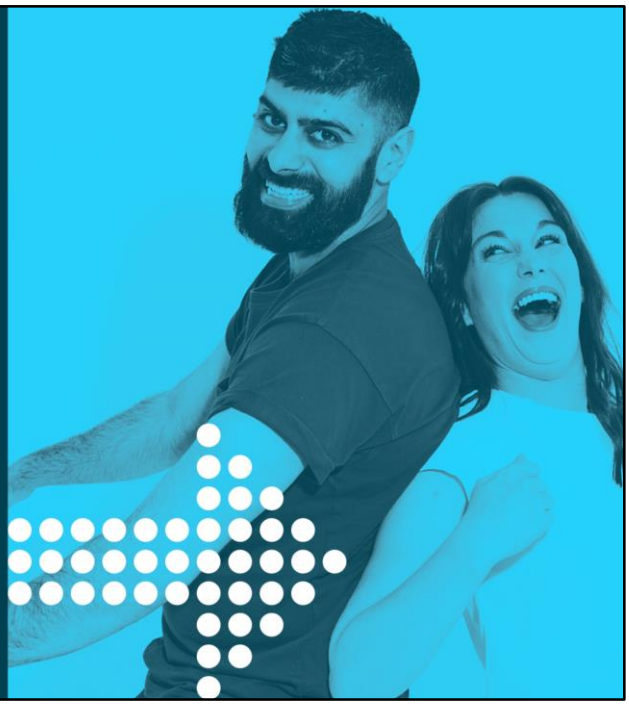
- To understand how to create and call functions in JavaScript
- To understand how scope works in JavaScript
- To be aware of the 'Arrow Function' syntax




Object Oriented
JavaScript
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To refresh how objects are defined in JavaScript
- To understand CLASS definition using ES2015+ syntax
- To be able to implement inheritance in JavaScript ES2015+ syntax



JavaScript objects (1)

Everything in JavaScript is an object

- Functions, dates, DOM elements
- How we extend the language with our own types

Creating...

- Use **new** keyword or **{ }**

Properties

- Use *dot notation* or *object literal* notation

```
let myBike = new Object(); // using new
myBike.make = "Honda";
myBike.model = "Fireblade";

let myBike2 = {             // using {}
    make: "Honda",
    model: "Fireblade"
};

myBike.make = "Yamaha"; // Dot
myBike["model"] = "R1"; // Obj Lit
```

Everything in JavaScript is an Object. This includes functions, dates, strings and DOM Elements. This is how we are able to extend the language with our own types.

To create an object, we use the ***new*** keyword or we can simply use empty curly braces if we are creating an instance of the object type.

Objects' members are stored as an associative array. We can access members of this array using dot notation or via indexer syntax as shown above.

This gives us great flexibility and makes creating custom objects a simple process; however, there are more useful and better-performing ways to create reusable objects as we'll discover.

JavaScript objects (2)

Use **`for...in`** loop to iterate over the properties

```
var myBike = {
    make: "Honda",
    model: "Fireblade",
    year: 2008,
    mileage: 12500,
}

for (let propName in myBike) {
    print `${propName}: ${myBike[propName]}`;
}
```

To iterate over the members of the associative array, we can use a `for…in` loop as shown in the first example above. This is the basis for a reflection-like mechanism in JavaScript.

As well as simple properties, as Functions are first-class objects, we can add methods by having a property of our object with the type of function as in the second and third examples above.

The third example is the preferred method as it enables us to declare both properties and methods in the same block, making our code more readable.

Classes

Syntactic sugar over prototypal inheritance

Gotcha: NOT hoisted like functions

Executed in strict mode

Private properties are prefixed with an underscore

- Purely convention as there is no notion of private scope for properties in JavaScript

```
class Car {
  constructor (wheels, power) {
    this._wheels = wheels;
    this._power = power;
    this._speed = 0;
  }

  accelerate(time) {
    this._speed = this._speed + 0.5*this._power*time;
  }
}
const myCar = new Car(4, 20); //constructor called
```

Constructor method is called when the class is instantiated through the “new” keyword.

Methods can be created inside the class definition without using the function keyword or assigning to “this”.

See the Appendix for more information about Prototypal Inheritance.

Accessing Properties

Object Oriented Programming has a concept called Encapsulation

- This means 'private' properties should not be accessible directly
- Should use 'accessor' or 'getter' methods to retrieve the value
- Should use 'mutator' or 'setter' methods to change the value

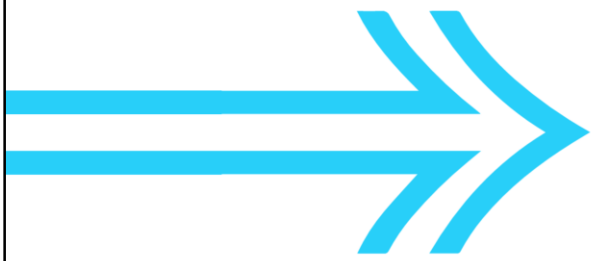
Allows the class to decide if the value is permissible

Two ways to achieve in JavaScript:

- Write a method called **`getPropertyName()`** that **returns** the value of the property
- If value can be changed, write a method called **`setPropertyName()`** with logic to change the value
- Use **`get`** and **`set`** keywords instead of these functions

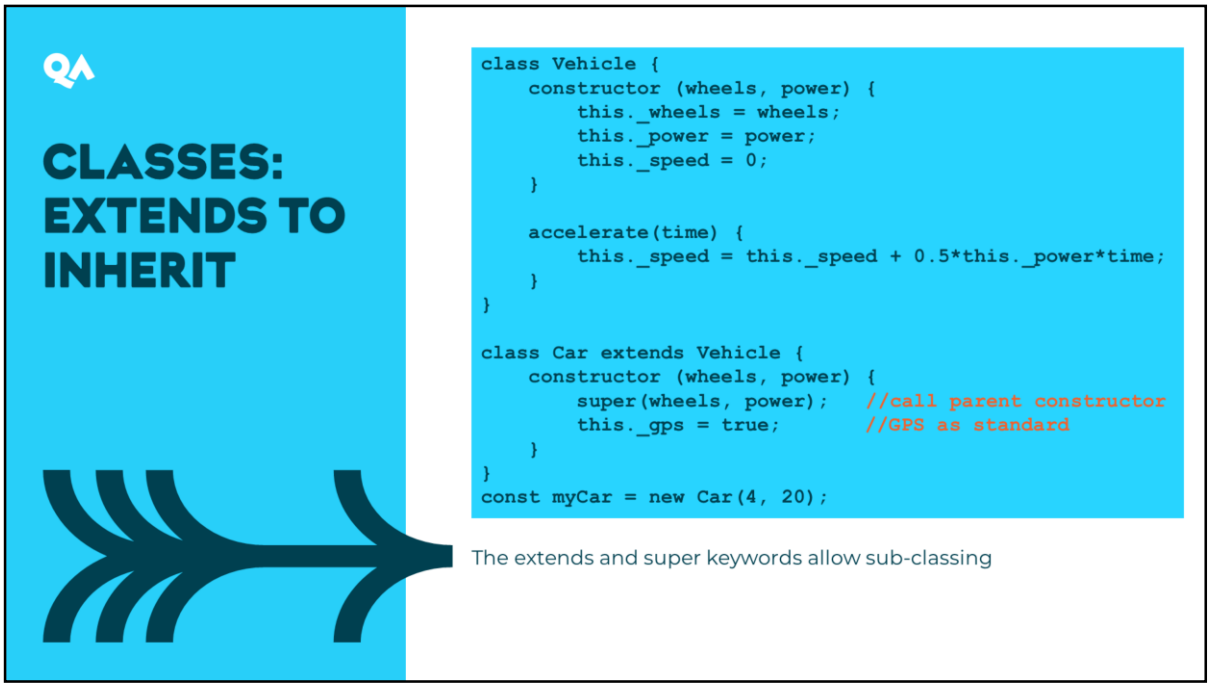
GOTCHA: cannot use the property name as the 'name' of the function

QuickLab 6 – Creating a Class



In this QuickLab you will:

- Create a class
- Add getters and setters
- Create an instance of the class



Notice that the car has all the properties of a Vehicle plus its own property of _gps.

GOTCHA! : “this” will be undefined before you call “super()” in a subclass

Inheritance in action – Custom Error handler

JavaScript has in inbuilt Error object (along with many other inbuilt objects)

- Through inheritance, we can create our own error types

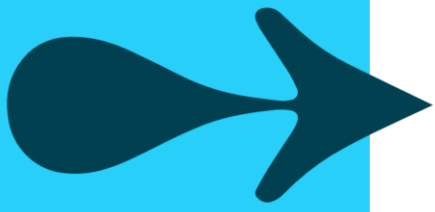
```
function DivisionByZeroError(message) {
    this.name = "DivisionByZeroError";
    this.message = (message || "");
}

DivisionByZeroError.prototype = new Error();
```

Customised error messages can b e generated using the built-in exception types. However, another approach is to create new exception types by extending the existing “Error” type. Because the new type inherits from “Error”, it can be used like the other built-in exception types.

The following example looks at the problem of dealing with division by zero. Instead of using an “Error” or “RangeError” object, we are going to create our own type of exception.

In this example, we are creating the “DivisionByZeroError” exception type. The function in the example acts as the constructor for our new type. The constructor takes care of assigning the “name” and “message” properties.

CLASSES: STATIC



- The **`static`** keyword allows for method calls to a **`class`** that hasn't been instantiated
- Calls to a **`static`** function of an instantiated class will throw an error

```
class Circle {
    constructor (radius, centre) {
        this.radius = radius;
        this.centre = centre;
    }

    static area(circle) {
        return Math.PI * Math.pow(circle.radius,2);
    }
}

const MY_CIRCLE = new Circle(5,[0,0]);
console.log(Circle.area(myCircle));
//78.53981633974483
```

Sealing objects to prevent expando errors

- Extensibility of objects can be toggled
- Turning off extensibility prevents new properties changing the object

→ `Object.preventExtensions( obj )`

→ `Object.isExtensible( obj )`

```
let obj = {
    name: "Dave";
};
print(obj.name); //Dave

console.log(Object.isExtensible(obj));
// true

Object.preventExtensions(obj);

obj.url = "http://ejohn.org/";
//Exception in strict mode
//(silent fail otherwise)

console.log(Object.isExtensible(obj));
//false
```

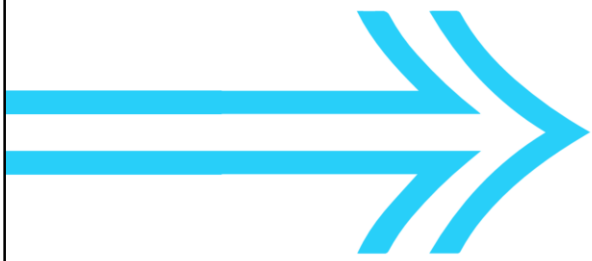
A feature of ECMAScript 5 is that the extensibility of objects can now be toggled. Turning off extensibility can prevent new properties from getting added to an object.

ES5 provides two methods for manipulating and verifying the extensibility of objects.

```
Object.preventExtensions( obj )
Object.isExtensible( obj )
```

preventExtensions locks down an object and prevents any future property additions from occurring.

QuickLab 7 – Extending a Class



In this QuickLab you will:

- Extend a base class
- Use a super call
- Override methods from the base class

Objectives

- To refresh how objects are defined in JavaScript
- To understand CLASS definition using ES2015+ syntax
- To be able to implement inheritance in JavaScript ES2015+ syntax


QA
Appendix:
Prototypal Inheritance
(A JavaScript History
Lesson)

Prototypal Inheritance

Inheritance in JS is achieved through prototypes

- Every object has a prototype from which it can inherit members

Prior to ES2015, inheritance was a 'messy' business with lots of confusing code to achieve what has been shown using classes and the extends keyword!

The following slides show how this was achieved...

Functions as Constructors

Constructors

- A function used in conjunction with the **new** operator
- In the body of the constructor, **this** refers to the newly-created instance
- If the new instance defines methods, within those methods **this** refers to the instance

```
function Dog() {
    this._name = '';
    this._age = 0;
}
var dog = new Dog();
```

Private properties are prefixed with an underscore

- Purely convention as there is no notion of private scope for properties in JavaScript

The idea of a constructor (code that is run when an object is first instantiated/initialised) is nothing new in the object oriented world. In JavaScript, this is simply a function that is used in conjunction with the **new** operator.

This is the preferred approach to creating custom objects. Within the body of the constructor, **this** refers to the newly-created instance; so, by using the **this** keyword, we can set instance properties.

(e.g. **this.propertyName= "some Value";**)

If the new instance defines any methods – other functions – then those methods will have the same context (i.e. the new instance).

The prototype object

Holds all properties that will be inherited by the instance

- Defines the structure of an object
- All new instances inherit the members of the prototype
- References to objects/arrays added to prototype shared
- Slightly slower as instance is searched before prototype
- General practice

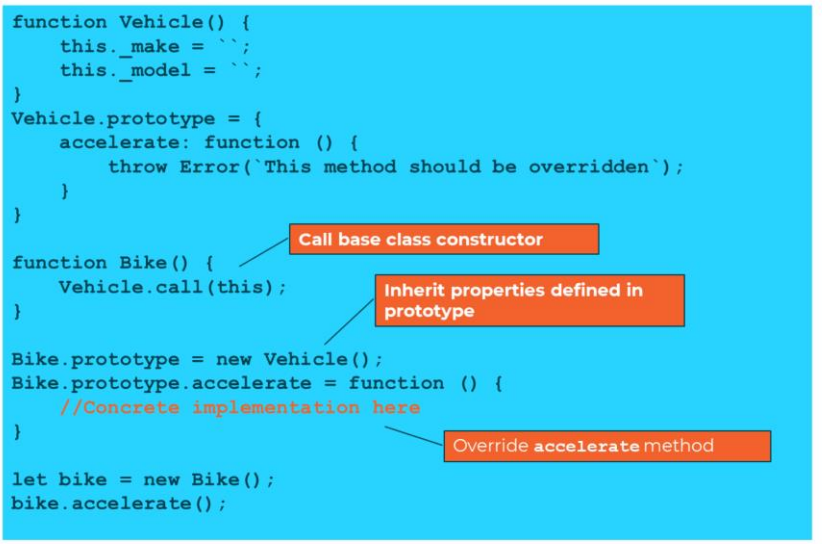
→ Declare members in the constructor

→ Declare methods in the prototype

```
function Dog() {
    this._name = "";
    this._age = 0;
}

Dog.prototype.speak = function () {
    alert("Woof!");
}

let dog = new Dog();
dog.speak(); // alerts "Woof!"
```

The built-in prototype object holds all properties (including methods) that will be inherited by the instance. In essence, this defines the structure of the object and is comparable to a class. All new instances of the object will inherit the members of the prototype. References to data structures – such as objects/arrays – that are added to the prototype are shared between instances.
Accessing these shared properties is slightly slower as the instance is searched before the prototype, but it does mean that we save on memory allocation.
The general advice is to declare members in the constructor (private fields) and to declare methods (functions) within the prototype. This way gives the separation between data and behaviour that we expect.

Inheritance in JavaScript



The above example shows a way of simulating inheritance using pure JavaScript. First, we declare the constructor of the base class Vehicle, then the prototype is defined with an abstract method (accelerate) where we simply throw an Error as this should be overridden by derived classes. After that, we create a constructor for the Bike class that will inherit from Vehicle.

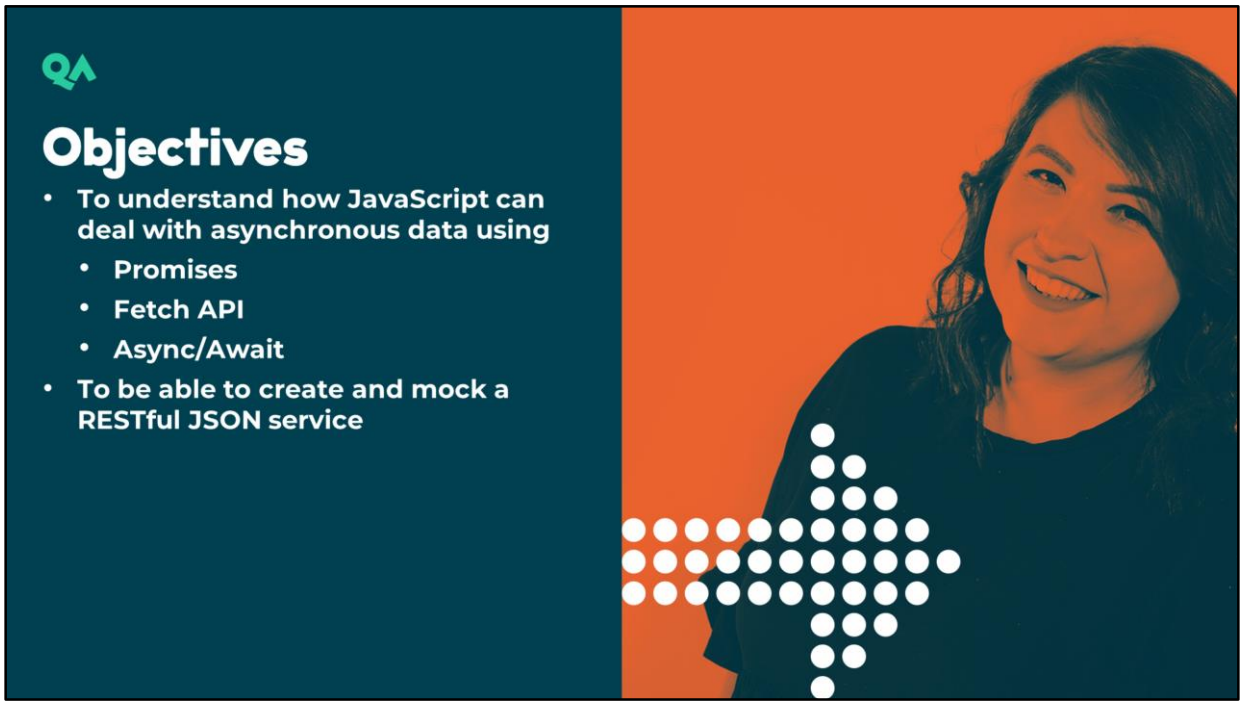
We use the call method, which allows us to change the context of the call so that **this** will refer to a different object (we pass in our new Bike – **this**). So, when the Vehicle constructor is called, the _make and _model will be added to the new Bike – we have inherited the field members.

Then we inherit the properties of Vehicle by setting the prototype of Bike to a new Vehicle instance.

Finally, we override the accelerate method by redefining it on the Bike prototype. We can then use the Bike class and it will have all the capabilities of a Bike that inherits from Vehicle.

However, there are some drawbacks to this method – we have simulated the inheritance but have no idea (from a user perspective) of the base class or the inheritance chain. Nor can we call base implementations without detailed knowledge of the class structure.


Asynchronous
JavaScript
JavaScript and TypeScript
For Cancer Research UK



Angular Logo from: https://angular.io/presskit

What is Asynchronous JavaScript?

A methodology for creating rich Internet applications
- Used to create highly-responsive applications
- Rich content and interactions

A client-focused model
- Uses client-side technologies – JavaScript, CSS, HTML

A user-focused model
- Asynchronous behaviour based on user interactions
- 'User-first' development model

An asynchronous model
- Communications with the server are made asynchronously
- User activity is not interrupted

Users of web applications increasingly expect a user experience that provides a high-level of interactivity on as close a level to a desktop application as possible.

The typical server-bound web application cannot easily provide this because it has to defer to the server for any significant updates to the user interface.

Ajax enables us to create a highly-responsive, rich user interface. It does this by enabling several design options through standard-based technologies.

Key to this are the client-side technologies: HTML with JavaScript and CSS for the styling of our pages. Due to developments and standardisation of these technologies, we can change the focus of our application from server-centric to client-centric.

We focus our development efforts on user interactions with our application and have a user-driven, event-based model, so that we can concentrate as much as possible on the user experience.

Through asynchronous communication with the server, we are able to retrieve data or HTML fragments that we can then insert into the DOM programmatically. The main benefits of asynchronous communication in this scenario are that we are

only transmitting a small amount of information and are semi-coupled with the server; more importantly, user activity and interactions with our application are not interrupted during this request process.

Four principles of Asynchronous JavaScript

The browser hosts an application

- A richer document is sent to the browser
- JavaScript manages the client-side interaction with the user

The server delivers data

- Requests for data, not content, are sent to the server
- Less network traffic and greater responsiveness

User interaction can be continuous and fluid

- The client is able to process simple user requests
- Near instantaneous response to the user

When we work in an Asynchronous environment, we have to think differently to how we may usually think with a server-bound application. We need to change our idea of where the application is running from the server to the client. When we add Asynchronous functionality to a web application, we will be adding a certain amount of code that we would not add normally to a typical server-bound web application. This means that the browser will essentially be hosting an application and not simply content. We will be sending a richer document to the browser, including JavaScript files and then we will manage interaction from the client by making requests for data (not content) and using this to update the DOM within the browser. This mechanism creates less network traffic and, therefore, allows us to show greater responsiveness within our application.

User interaction is simplified in a similar way to event-driven desktop applications in that the client browser using our JavaScript code is able to process the simple (or even relatively complex) requests and make an asynchronous request for data (if necessary), providing a near-instantaneous response to the user due to the decreased traffic and potentially lower processing overhead on the server.

However, we will be writing a lot more code and constructing an application rather than just a series of effects on the client, so we are in the realm of real coding and we need to take a disciplined approach to this.

Client-centric development model

Primarily implemented on the client
- Presentation layer driven from client script
- Uses HTML, CSS and JavaScript

This means
- First request
  A smarter, more interactive application is delivered from the server
- Subsequently
  Less interaction between the browser and the server

Which
- Encourages greater interaction with the user
- Provides a richer, more intuitive experience

As mentioned, there are two main development models that we can consider within Asynchronous applications.

The first is the client-centric development model.

This model has the application mostly implemented on the client with the user interface drive from client script with JavaScript driving the DOM.

At first request, we will have a "smarter" application delivered from the server. By this, we mean that more code will be downloaded so that the client browser can do more of the processing of the application itself. In turn, this reduces the amount of interaction that must happen during the course of the applications life – this encourages a much greater interaction with the user and provides a richer, more intuitive experience.

This is the model we would use for a pure Asynchronous application, where we are communicating with data and manipulating the DOM programmatically.

Server-centric development model

Primarily implemented on the server

- Application logic and most UI decisions remain on the server

This means

- First request

  A regular page is retrieved from the server
- Subsequently

  Incremental page updates are sent to the client

Which

- Reduces latency and increases interactivity
- Gives the opportunity to keep core UI and application logic on the server

The second model is the server-centric development model.

This is also known as, the partial page update model.

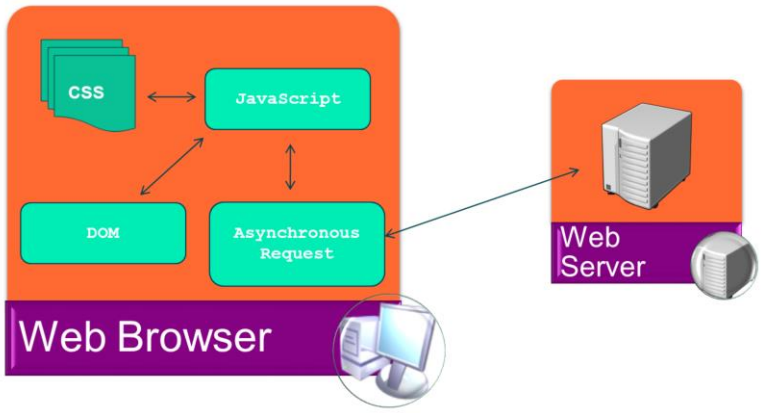
Most of the application logic and UI decisions are still made by the server. On application startup, a normal page is retrieved from the server. After that page has been retrieved, the client interacts with the page and requests are sent to the server for fragments of HTML with which to update the page.

This happens incrementally, so reducing latency and increasing the level of interactivity possible without a full-page refresh. It also gives the opportunity to keep the core login of the Application and UI on the server which may be a required decision for some sensitive applications.

Within any one application, it is possible to have the two models working side-by-side on different pages with one page more suited to the server-centric model and the other more suited to the client-centric model. It would be very unusual and indeed very confusing to mix the two approaches on the same page.

Asynchronous JavaScript-enabling technologies

CSS, DOM, JavaScript and an Asynchronous Request API



The diagram above shows the main enabling technologies of Asynchronous JavaScript.

At the client end, we have the web browser, which will host the document that is requested. This document is made available via a Document Object Model (DOM), so that we can programmatically manipulate the document using JavaScript and also hook into the events of the browser, document and constituent elements.

In addition, we can use an Asynchronous Request API to initiate requests to a server in order to retrieve data, then use that data through JavaScript to update the document through the DOM.


JSON

JavaScript Object Notation (JSON)

Lightweight data-interchange format

- Compared to XML

Simple format

- Easy for humans to read and write
- Easy for machines to parse and generate

JSON is a text format

- Programming language independent
- Conventions familiar to programmers of the C-family of languages, including C# and JavaScript

Transferring data can be a cumbersome task. XML is all well and good; however, it requires a DOM parser in order to read/write and is not easily realised into object format.

JavaScript Object Notation (or JSON) is a lightweight data interchange format that is easy to read and write and, more importantly, easy for machines to parse and to generate.

JSON is a text format that is programming-language-independent and uses conventions familiar to C family programmers. To JavaScript, JSON looks and behaves as an associative array and so can be parsed (using eval) and turned into a fully functioning object, which is very easily consumed.

JSON structures

```
{
  "results": [
    {
      "home": "React Rangers",
      "homeScore": 3,
      "away": "Angular Athletic",
      "awayScore": 0
    },
    {
      "home": "Ember Town",
      "homeScore": 2,
      "away": "React Rangers",
      "awayScore": 2
    }
  ]
}
```

Universal data structures supported by most modern programming languages

A collection of name/value pairs

- Realised as an object (associative array)

An ordered list of values

- Realised as an array

JSON object

- Unordered set of name/value pairs
- Begins with { (left brace) and ends with } (right brace)
- Each name followed by a : (colon)
- Name/Value pairs separated by a , (comma)

JSON consists of structures that are supported by most modern programming languages and so is immediately accessible to most.

It is an associative array (name/value pairs) and can contain an ordered list of values as an array.

The overall JSON object consists of an unordered list of name/value pairs contained within curly braces with each name and value pair separated by a colon and the name/value pairs separated by a comma.

JSON and JavaScript

JSON is a subset of the object literal notation of JavaScript

- Can be used in the JavaScript language with no problems

```
let myJSONObject = {
    "searchResults": [
        {
            "productName": "Aniseed Syrup",
            "unitPrice": 10
        },
        {
            "productName": "Alice Mutton",
            "unitPrice":
            39
        }
    ]
};
```

JSON is a subset of the object-literal notation of JavaScript and so can be used (as shown above) in JavaScript with no problems.

The object realised in the above example can be accessed using either dot or subscript operators as shown in the second example.

The JSON object

The JSON object is globally available

- The **parse** method takes a string and parses it into JavaScript objects
- The **stringify** method takes JavaScript objects and returns a string

Makes working with JSON data a trivial affair

```
let obj = JSON.parse('{"name":"Adrian"}');
console.log(obj.name);                          //logs Adrian
```

```
let str = JSON.stringify({ name: "John" });
```

There are a series of overloaded methods for the type:

JSON.parse(text) – Converts a serialised JSON string into a JavaScript object.

JSON.parse(text, translate) – Uses a translation function to convert values or remove them entirely.

JSON.stringify(obj) – Converts an object into a serialised JSON string.

JSON.stringify(obj, ["white", "list"]) – Serialises only a specific white list of properties.

JSON.stringify(obj, translate) – Serialises the object using a translation function.

JSON.stringify(obj, null, 2) – Adds the specified number of spaces to the output, printing it evenly.

RESTful services

RESTful services are commonly used to supply data to web applications

- **RE**presentational **S**tate **T**ransfer

Essentially they are a server, possibly attached to a Database that returns the requested data:

- Make a request to a URL – can CRUD
  - Create
  - Read
  - Update
  - Delete
- Response will be in the form of JSON

Mocking a RESTful service

json-server is an npm package that allows you to:

"Get a full fake REST API with zero coding in less than 30 seconds"

Need to install the package (globally if it will be used frequently)

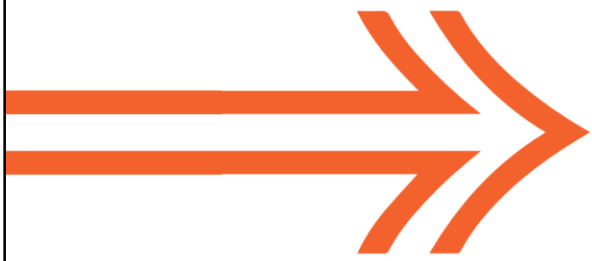
Need to supply it with a properly-formed .json file

Runs on http://localhost:3000 by default (can be changed when spinning up)

Allows full CRUD requests and saves changes to .json file

https://www.npmjs.com/package/json-server

QuickLab 8 – Generate some JSON



In these parts of the QuickLab, you will:

- Create a JSON file
- Install and run JSON server


Promises

What is a Promise?

A placeholder for some data that will be available: immediately, some time in the future or possibly not at all

JavaScript is executed from the top down

- Each line of code evaluated and executed in turn

What happens if data needed is potentially not available immediately?

- Most commonly we may be waiting for some data to come from a remote endpoint
- Need some way to be able to execute code when the data is available or deal with the fact that it will never be available

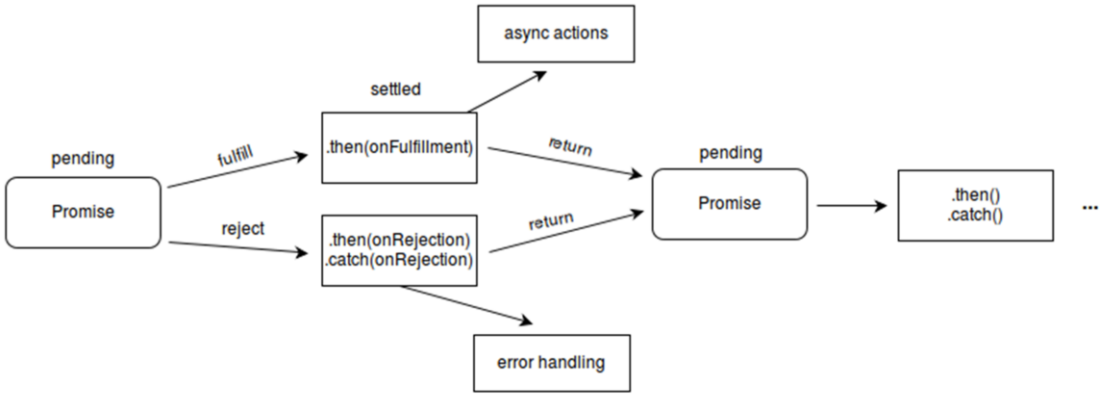
This is the job of a Promise

Promises

A promise is the representation of an operation that will complete at some unknown point in the future

We can associate handlers to the operation's eventual success (or failure)

Exposes .then and .catch methods to handle resolution or rejection



Promises

Construct a new promise passing in an 'executor' function which will be immediately evaluated and is passed both resolve and reject functions as arguments

```
let newPromise = new Promise((resolve, reject) => { });
```

The Promise is in one of three states:

- Pending
- Fulfilled - Operation completed successfully
- Rejected – Operation failed

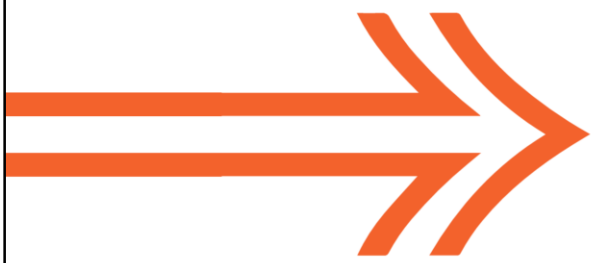
Which we can attach associated handlers too:

- **`.then(onFulfilled, onRejected)`** appends handlers to the original promise, returning a promise resolving to the return of the called handler or the original settled value if the called handler is undefined
- **`.catch(onRejected)`** same as then but only handles the rejected condition

Promises: Example

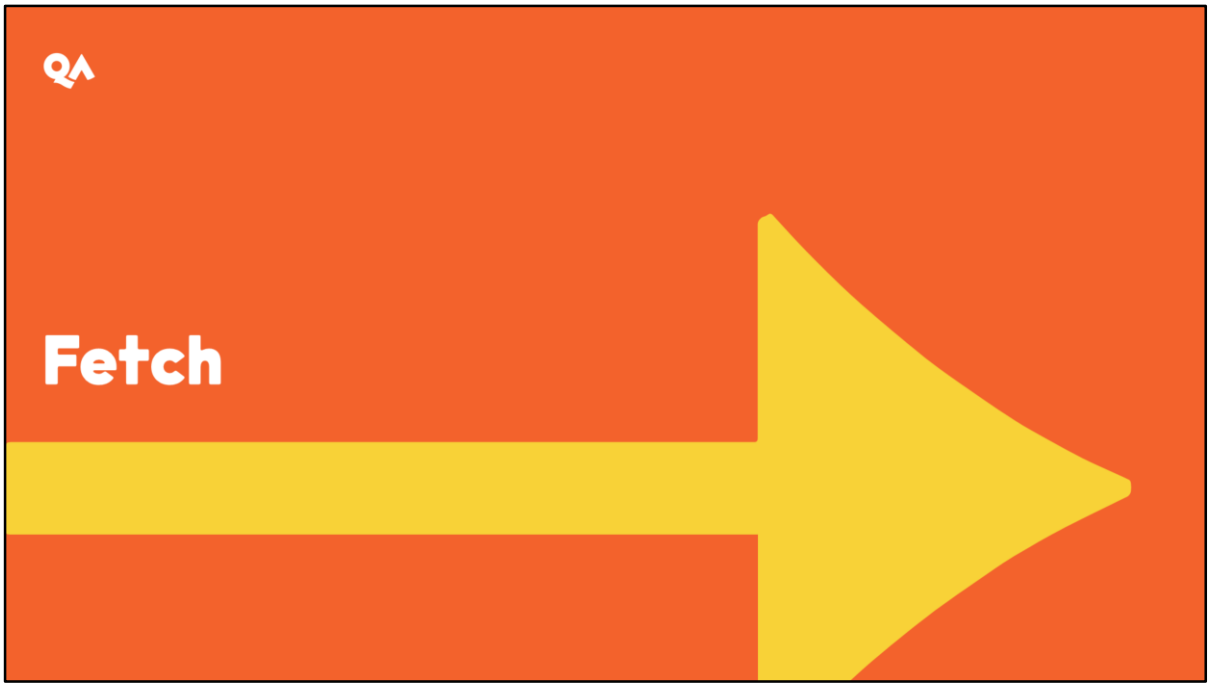
```
let aPromise = new Promise((resolve, reject) => {
    let delayedFunc = setTimeout(() => {
         //whether it resolves or rejects is unknown
        (Math.random() < 0.5) ? resolve("resolved") : reject("rejected");
    }, Math.random() * 5000); //function will return sometime: 0-5s
});

aPromise
    .then(
        //resolved
        data => {
            console.log(v);
        },
        //rejected
        error => {
            console.log(v);
        }
    );
```

QuickLab 9– Promises



In these parts of the QuickLab, you will:

- Experiment with Promises


Fetch

Fetch

"The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global `fetch()` method that provides an easy, logical way to fetch resources asynchronously across the network"

In short, **Fetch** provides the functionality hitherto provided by `XMLHttpRequest`

It greatly simplifies making requests and dealing with responses

Fetch requests return **Promises**

Fetch is supported from Chrome 42, Edge 14, Firefox 39, Safari 10.1, Opera 29

XMLHttpRequest is an older technology, generally used to implement AJAX (Asynchronous JavaScript And XML). See the appendix for information on how Asynchronous requests using the XMLhttpRequest object are made.

Fetch

Making a **`fetch`** request can be as simple as passing a URL and chaining appropriate .then and .catch methods onto the return

```
fetch('https://www.qa.com/courses.json')
    .then(response => response.json())
    .then(myJson => console.log(myJson))
    .cach(err=> console.error(err))
```

Note how we don't have to use **`JSON.parse`** as response objects have a **`.json()`** method which returns a **`Promise`** that resolves to with the result of parsing the body text of the response as JSON

By default, a **`fetch`** request is of type **`GET`**

Fetch – Full Example

We can make more complex requests using the second argument, an init object that allows us to control a number of aspects of the request – including any data we wish to include with it

```
fetch(url, {
    body: JSON.stringify(data),
    // must match 'Content-Type' header
    cache: 'no-cache',
    // *default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, same-origin, *omit
    headers: {
        'content-type': 'application/json'
    },
    method: 'POST',           // *GET, POST, PUT, DELETE, etc
    mode: 'cors',             // no-cors, cors, *same-origin
    redirect: 'follow',       // manual, *follow, error
    referrer: 'no-referrer', // *client, no-referrer
})
.then(response => response.json())
.then(myJSON => console.log(myJSON))
.catch(err => console.log(err));
```

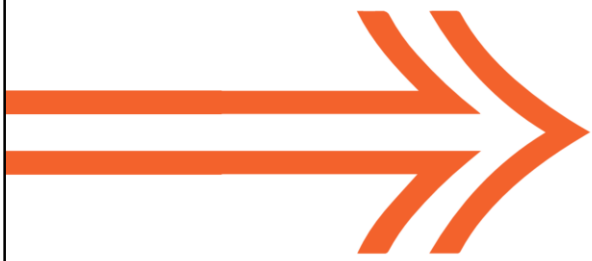
Fetch

A **fetch** promise does not **reject** on receiving an error code from the server (such as 404) instead it **resolves** and will have a property **response.ok = false**.

To correctly handle **fetch** requests we would need to also check whether the server responded with a **response.ok === true**

```
fetch(url)
    .then(response => {
        if (response.ok) {
            //do things
        }
        else {
            //handle error
        }
    });
```

QuickLab 10– Fetch



In these parts of the QuickLab, you will:

- Use the Fetch API to retrieve some data

Async Functions

An **`async`** function will return a **`Promise`** which **resolves** with the value returned by the function, or **rejected** with any uncaught exceptions

An **`async`** function can contain an **`await`** expression which **pauses** the execution of the **`async`** function until completion of the **`Promise`** and then resumes

Async Example

```
async function asyncFunc1() {
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log(`Async function 1`);
            resolve();
        },3000);
    });
}
```

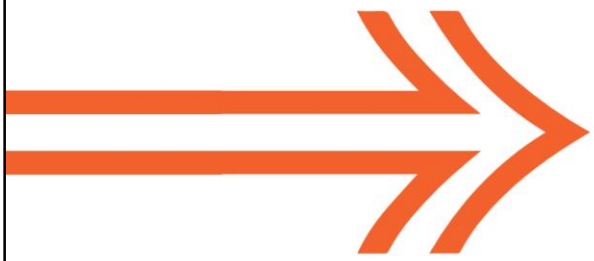
```
async function asyncFunc2() {
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log(`Async function 2`);
            resolve();
        },2000);
    });
}
```

```
async function asyncFunc3() {
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log(`Async function 3`);
            resolve();
        },1000);
    });
}
```

```
async function doThings() {
    await asyncFunc1();
    await asyncFunc2();
    await asyncFunc3();
    return "All done";
}

doThings().then(console.log);
```

```
Async function 1
Async function 2
Async function 3
All done
```

QuickLab 11 – Async/Await



In these parts of the QuickLab, you will:

- Replace Promise chains with async/await calls

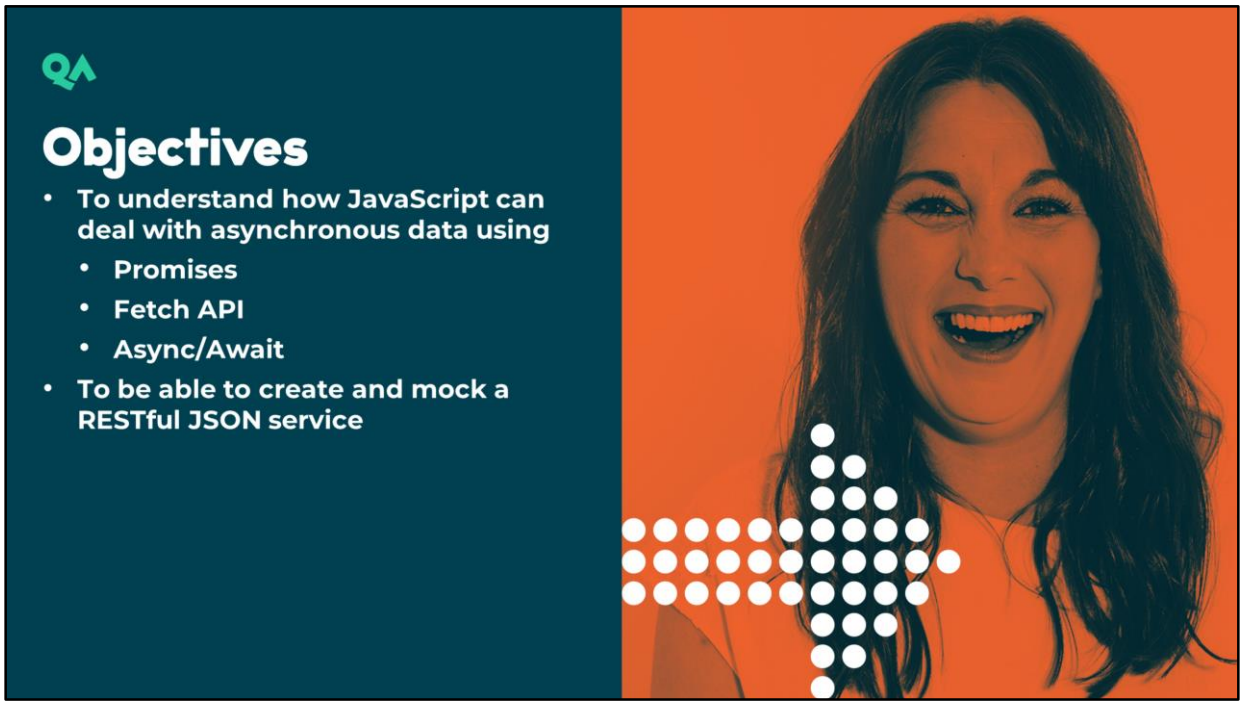
Review

Asynchronous JavaScript is...

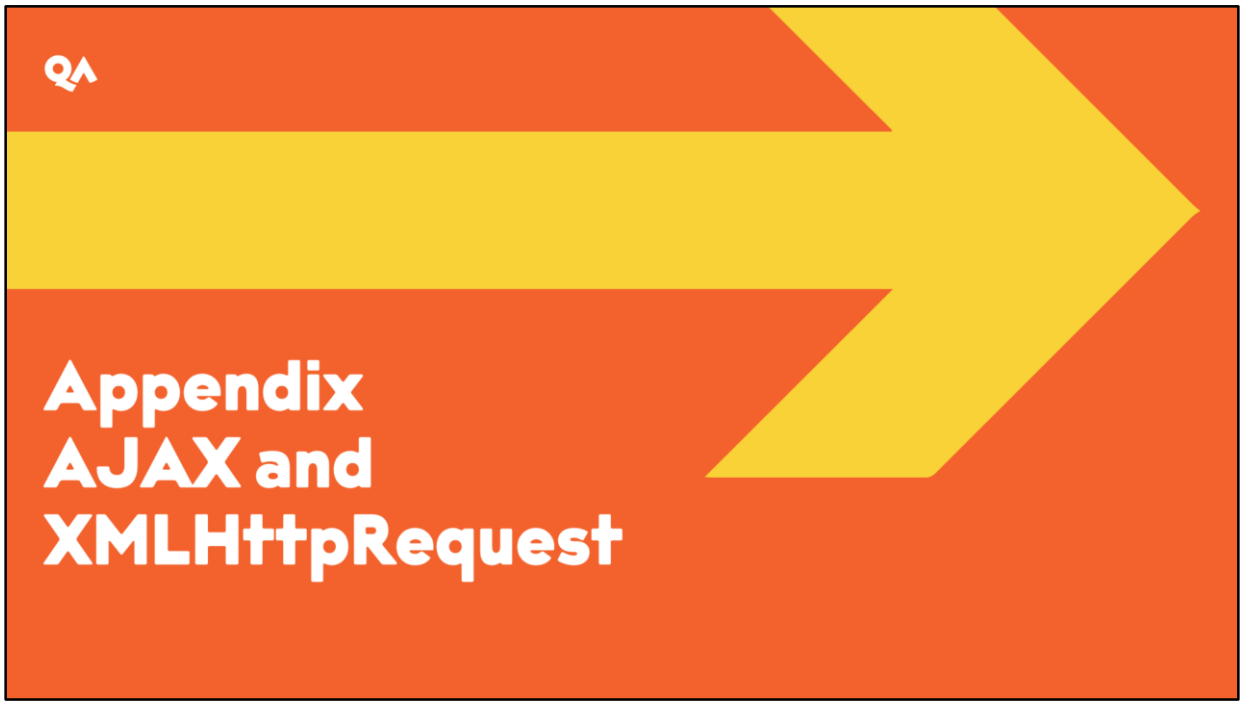
- A methodology for creating rich Internet applications
- A client and user-focused model
- A methodology that enables asynchronous requests

  Fetch API

  `async` functions and the **`await`** declaration



Angular Logo from: https://angular.io/presskit


Appendix
AJAX and
XMLHttpRequest

AJAX and XMLHttpRequest

Asynchronous JavaScript And XML

- Still used commonly to describe asynchronous calls

XMLHttpRequest

- Object required to make asynchronous calls in ES5 and below

XMLHttpRequest – overview

- Request headers can be added
- Response headers can be read
- Support in all modern browsers

Handles the request process

- w3c specification
- See http://www.w3.org/TR/XMLHttpRequest/
- Defines an API that provides scripted client functionality for transferring data between a client and a server

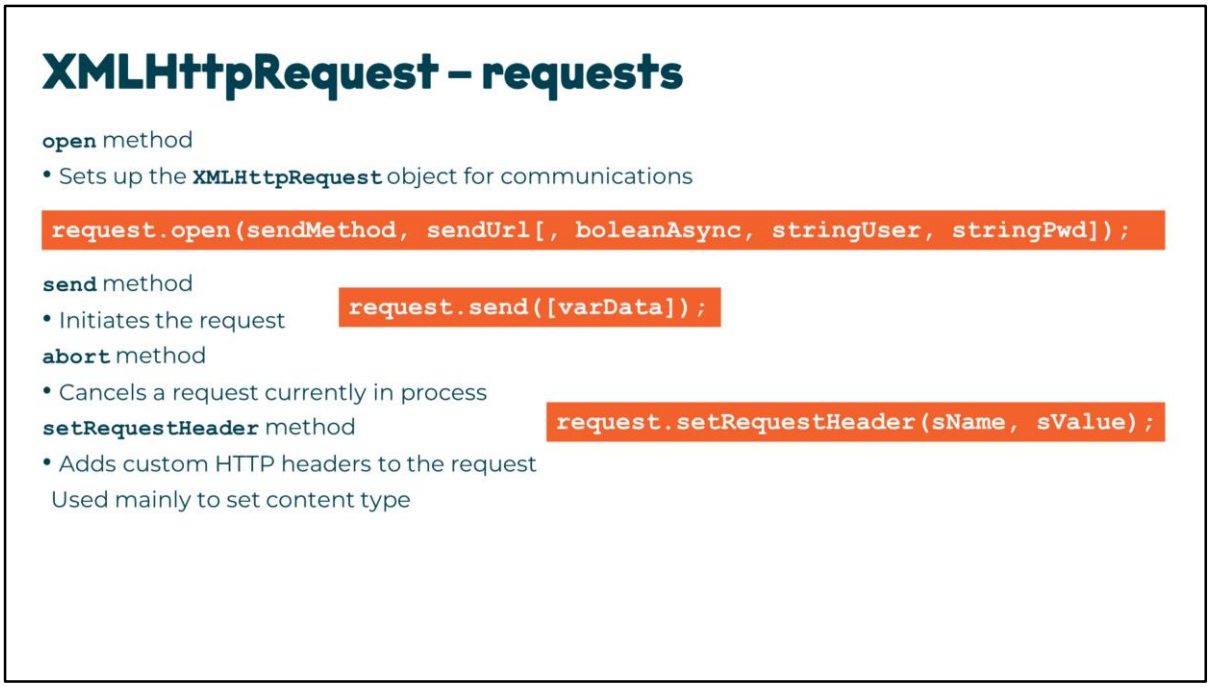
Benefits

- Simple to use
- Can be used for any request type, e.g. GET, POST
- Can be used synchronously or asynchronously

The w3c document referenced above defines an interface that is implemented by the XMLHttpRequest object within conformant browsers. This includes all modern browsers.

The object contains a simple API for creating most types of request (GET, POST, HEAD, PUT, DELETE, OPTIONS) and can be used over HTTP or HTTPS. It can be used for making synchronous or asynchronous requests.

Like any typical HTTP request, we can manipulate the headers by adding extra entries to the request and we can read the response headers.



There are four named request methods; open, send, abort and setRequestHeader.

The open method takes a string indicating the method (GET, POST etc...). It also requires the Url as a string. The other parameters are optional; however to make an asynchronous call you will need to pass true as the bAsync parameter. You can also, optionally, provide a username and password to use for authentication.

The send method initiates the request and has three ways of invocation. You can send the request without any data (e.g. when invoking for a GET request). In addition, the varData parameter can contain either a string containing name/value pairs as would be the body of the request, or it can contain an XML Document.

The abort method allows us to cancel a request that is currently processing.

The setRequestHeader method is used to add headers to the request and is used mainly to set the content type when we want to POST data. Both parameters are strings.

XMLHttpRequest – responses

`status` property, **`statusText`** property

- Return the HTTP response code or friendly text respectively

`load` event

- You can listen to this event in IE9 and above rather than check **`readystate`** on every **`readystatechange`** event

`readystatechange` event

- Fires for each stage in the request cycle

`readyState` property – Progress indicator (0 to 4)

- Most important is 4 (Loaded); you can access the data

`responseXXX` property - retrieves the response

→ **`responseText`** – as a string

→ **`responseBody`** – as an array of unsigned bytes

The key to retrieving or handling the response is to hook into the readystatechange event of

This eve ve the con request

You can ndly text des respons s by using getResponseHeader or getAllResponseHeaders.

readyState value	Description
0	Unsent
1	Opened
2	Headers Received
3	Loading
4	Loaded (Done) – data is fully loaded

XMLHttpRequest – example

Using **XMLHttpRequest**

- Create a new **XMLHttpRequest** object
- Set the request details using the open method
- Hook-up the load event to a callback function

→ Easiest way is to use an anonymous function

- send the request

```
let request = new XMLHttpRequest();
request.open(
    "GET",
    "SomeHandler.ashx", true);

request.onload = () => {
    if (request.status == 200) {
        // Do something with
        // request.responseText
    }
}

request.send();
```

In the example above, we are making a request to a simple handler called SomeHandler.ashx.

First, we instantiate an XMLHttpRequest object by calling its constructor.

We then invoke open with our GET method and the Url and state that the request will be asynchronous.

Next, we attach a handler to the load event. You can provide a function name here but, in the example, we have used an arrow function.

We then check to see that the request completed successfully (HTTP response code 200 – OK).

We can then do something with the responseText/responseXML.

Finally, we initiate the request by invoking the send method.


TypeScript Introduction
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To understand what TypeScript is and why use it
- To identify the tools needed to have a scalable development environment
- To be able to set up a scalable development environment

What is TypeScript

A TYPED superset of the JavaScript language

Compiles to plain JavaScript

- ECMAScript 3 by default or newer environments

Static type checking

- Types are optional and inferred

Maintained by an open source community (keeps up to date with JavaScript developments)

- TypeScript compiler implemented in TypeScript – can be used in any JavaScript host
- Held on GitHub – specification

TypeScript is a trademark of Microsoft Corporation

Why TypeScript?

Enable IDEs to provide a richer environment for spotting common errors

- Compiler can catch errors during development rather than have things fail at run time

Use modern JavaScript in projects immediately while still providing the broadest browser support

Types are optional! Rename your **.js** files to **.ts** now and you'll still get back valid **.js**

- Migrating to TypeScript can be done gradually

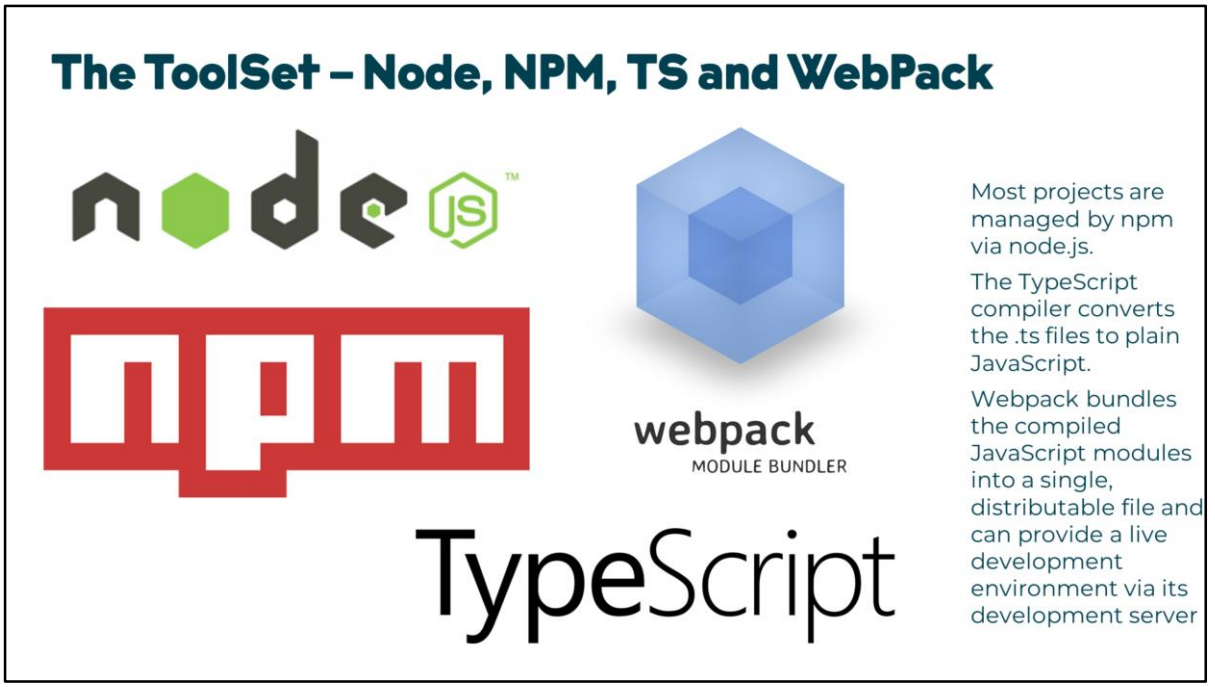
Types help document your code for the next developer (maybe you!)

Is used as an integral part of Angular development

- Can also be used in React and other libraries and frameworks

TypeScript – THE ONLY LESSON!!!

TypeScript is a developer's tool to help make more type-safe JavaScript applications
Valid JavaScript in TypeScript produces valid JavaScript when compiled
Compilation and IDE errors are only visible to the developer
Ignoring the compilation and IDE errors when valid JavaScript is produced is counter-productive!



Node.js is an open source command line tool for server side JS
The script is executed by the V8 Javascript engine

NPM manages decencies for an application via the command line

Installing TypeScript

Installing TypeScript is as simple as running an npm command

```
npm install -g typescript
```

Compiling our TypeScript files can then be as simple as running the TypeScript compiler from the command line

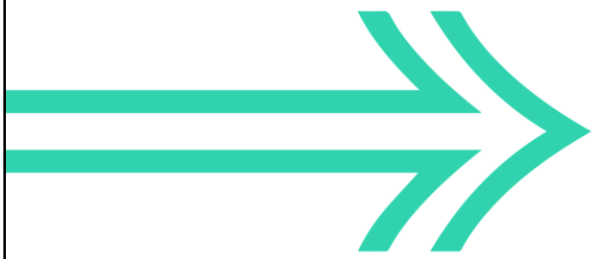
```
tsc intro.ts
```

This command takes our **intro.ts** file and compiles it to **intro.js**

```
//intro.ts
function hello(name) {
  console.log(`Hello ${name}`);
}

hello('World');
```

```
//intro.js
function hello(name) {
    console.log("Hello " + name);
}

hello('World');
```

QuickLab 12 – TypeScript Hello World



In this part of the QuickLab, you will:

- Install the typescript CLI
- Write a Hello World application
- Compile your .ts file to .js using the TypeScript compiler
- Run the JavaScript using NodeJS

Developer Tools

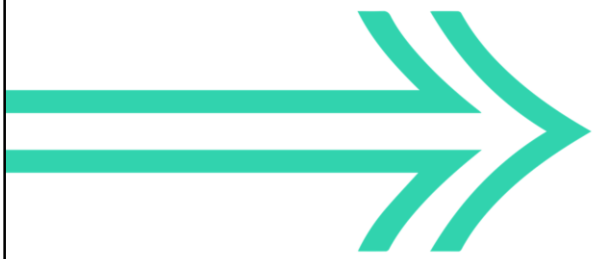
The typescript CLI is a quick way to get up and running but is not particularly scalable.

For the duration of this course we're going to be as lightweight on our tooling as possible, so we can focus on learning TypeScript

The only other tool we will use will be Webpack

- Almost ZERO configuration
- Just need to tell Webpack how to find and deal with TypeScript files!

QuickLab 13 – TypeScript Developer Environment

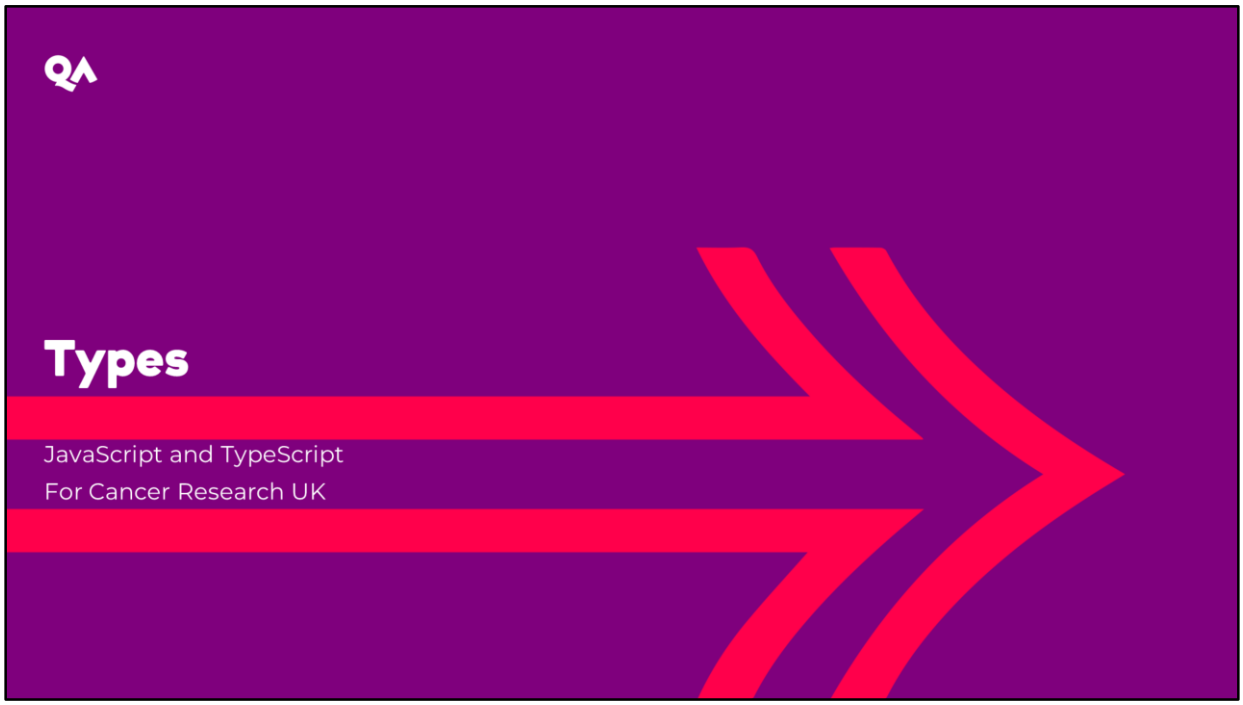


In this part of the QuickLab, you will:

- Set up a scalable development environment using Webpack

Objectives

- To understand what TypeScript is and why use it
- To identify the tools needed to have a scalable development environment
- To be able to set up a scalable development environment


Types
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To understand how TypeScript applies types to variables
- To be aware of additional types used in TypeScript
  - Array
  - Tuple
  - Enum
  - Any
  - Void
  - Never
  - Object
  - Unknown (introduced in TypeScript v3.0)

Primitive Types – just like JavaScript

PRIMITIVES

`boolean`

- true and false values only

`number`

- Floating point values

`string`

- 'single' "double" or `backticks`

`null`

- To define a value that does not exist

`undefined`

- The value of a variable that has never been assigned a value

`symbol`

- Values of this type can be used to make anonymous object properties

Adding Types to variable declarations

A colon and the type to assign to the variable name are added after it

```
// Boolean
let isDone:boolean = false;

// String
let fullName: string = `John Smith`;
let greeting: string = `Hello, my name
is ${fullName}`;
```

```
// Number
let decimal: number = 6;
let float: number = 100.232;
let exponential: number = 3.2E1234;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

Types – Arrays

Arrays

- Statically typed arrays declared in one of two ways

Familiar to JavaScript developers, the square brackets

```
let list: number[] = [1, 2, 3];
let list2: string[] =[`Pie`, `Gravy`];
```

Familiar to C# and Java developers, the angle bracket notation

```
let list: Array<number> = [1, 2, 3];
let list2: Array<string> = [`Pie`, `Gravy`];
```

Types – Tuple

Express an array of fixed (or variable) length but differing types

→ ? on a type means that it is optional

→ If optional is in middle, all following must have ? too

```
//Declaration
let person:[string, number, number?];

//Initialisation
let person = [`John`,21]
// OK

let person = [21, `John`]
// Error
```

Accessing Tuples

→ When accessing element with known index, correct type is retrieved too:

```
console.log(person[0].substr(1));
// OK
console.log(person[1].substr(1));
// Error
```

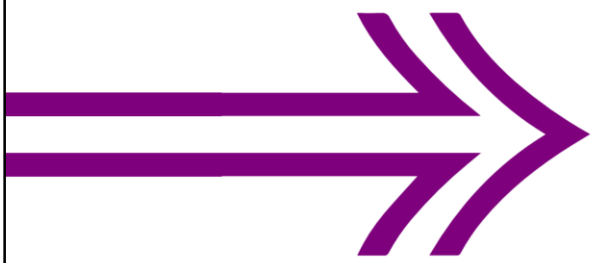
Adding to Tuples

→ Defined length can still be achieved with correct type – but not more

```
person[2] = `Smith`; // Error
person[2] = 1234;    // OK
person[3] = 1234;    // Error
person[3] = `Smith`; // Error
```

In --strictNullChecks mode, ? means undefined is included in the element

QuickLab 14 – Tuples



In this QuickLab, you will:

- Create and manipulate tuples

Types – Enum

Friendly names for numeric values

Automatically numbered from 0

```
enum Color {Red, Green, Blue};
let c: Color = Color.Green; //1
```

Can start from any number

```
enum Color {Red=4, Green, Blue};
let c: Color = Color.Green; //5
```

Or number them all manually

```
enum Color {Red=4, Green=8, Blue=16};
let c: Color = Color.Green; //8
```

And go from numeric value to the name

```
enum Color {Red=4, Green=8, Blue=16};
let c: string = Color[8]; //Green
```

Types – Any

All types are subtypes of Any

→ Gives us a route to describe variables that we do not know the type of e.g. from the user or 3rd party libraries

```
let thing: any = `Thing T. Thing`;
thing = false; // OK
```

This is also handy to start opting in to type checking during compilation

→ And can be helpful if you know parts of a type, but not all

```
let list: any[] = [1, true, `thing`];
```

Types – Void

In some ways, the opposite of **any** this type is the absence of any type at all.

→ Often the type of functions that don't return a value

```
function absenceOfThing(): void {
       alert(`Thing has gone AWOL`);
}
```

→ Declaring variables of type void is not useful

→ Can only assign null and undefined to them

```
Let unusable: void = undefined;
unusable = null;   // OK
Unusable = true;   // Error
```

Types – Null and Undefined

Both primitive types in JavaScript
→ Not extremely useful on their own

```
let u: undefined = undefined;
let n: null = null;
```

Subtypes of all other types
→ Can assign null and undefined to anything

```
let aNumber: number = undefined;
let aString: string = null;
```

Can use strictNullChecks compiler option to ensure that this can only happen if required
→ Use union types

```
//tsconfig.json
…
  "strictNullChecks": true
…
```

```
let aNumber: number = null; // Error
let aString: string|null = `Hi`;
aString = null; // OK
```

Types – Never

never is a subtype of *every type* but *nothing* is a subtype of **never**

never represents the type of values that never occur

```
//functions that never return
function notEver(): never {
       while (true) {}
}

//functions that always throw
function alwaysThrow(): never {
       throw new Error(`throwing`);
}
```

Object

Used to represent anything that is not a primitive

→ Allows better representation of APIs, e.g Object.create

```
declare function create(o: object|null): void;
create({prop: 0});   // OK
create(null);        // OK
create(42);          // Error
create(`string`);    // Error
```

Types – Type Assertion

Sometimes as developers we need to override the compiler

→ Usually when an entity is more specific than its current type

TypeScript provides two syntaxes

```
//angle-bracket syntax
let thing: any = `Thing T. Thing`;
let nameLength: number =
(<string>thing).length;

//as-syntax
let thing: any = `Thing T. Thing`;
let nameLength: number = (thing as
string).length;
```

Unknown top-type

Type-safe counterpart of **any**

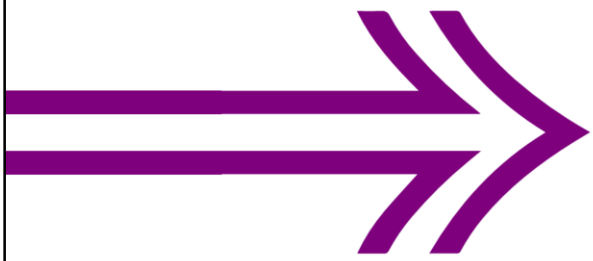
Anything is assignable to **unknown**

unknown isn't assignable to anything but itself and **any** without a type-assertion or control flow based narrowing

No operations are permitted on an **unknown** without assertion or narrowing

```
let x: unknown = `Hi`;  // OK
x = 42;
x !== 10;     // OK
x <= 22;      // Error - equality
              // operators only
(x as number) >= 10 // OK
```

QuickLab 15 – Type Assertion

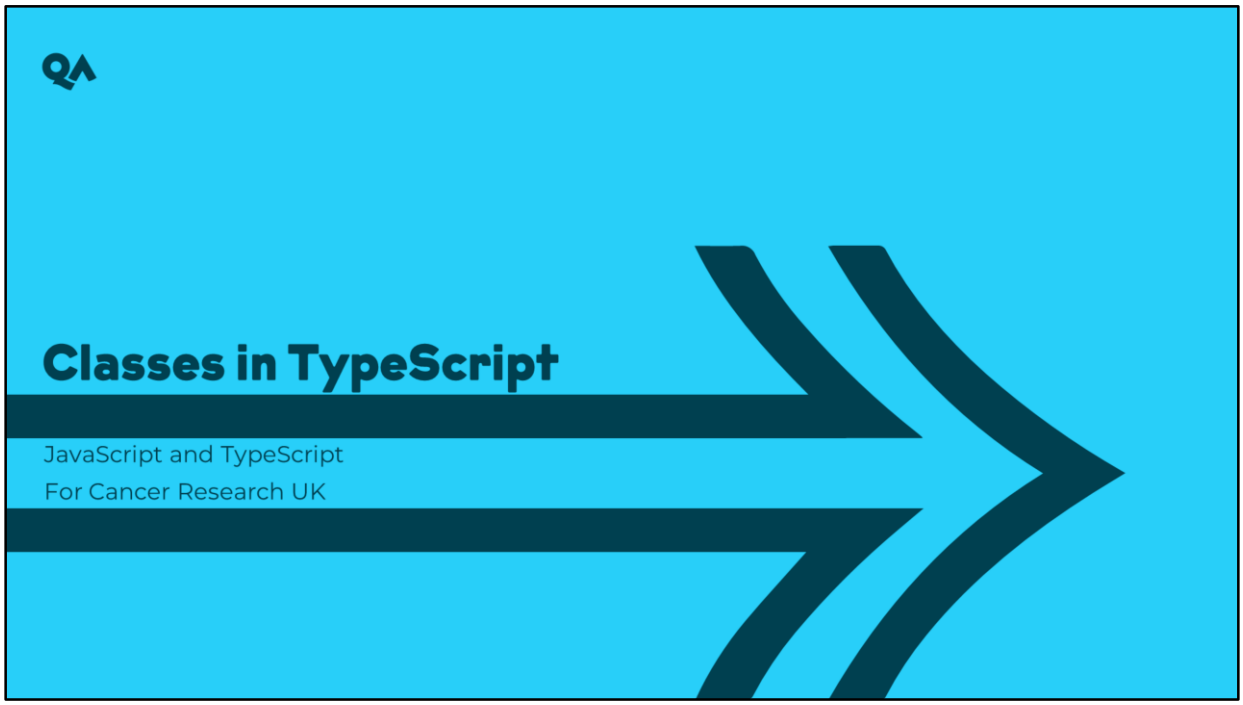


In this QuickLab, you will:

- Use the unknown and type assertion when working with variables

Objectives

- To understand how TypeScript applies types to variables
- To be aware of additional types used in TypeScript
  - Array
  - Tuple
  - Enum
  - Any
  - Void
  - Never
  - Object
  - Unknown (introduced in TypeScript v3.0)


Classes in TypeScript
JavaScript and TypeScript
For Cancer Research UK

Objectives

- To understand how TypeScript works with the JavaScript implementation of classes and inheritance
- To understand how the allowed access modifiers work
- To be able to use get and set with classes
- To understand how abstract classes are implemented
- To be able to use the static keyword

Classes

Syntactic sugar over prototypal inheritance

Gotcha: NOT hoisted like functions

Executed in strict mode

Part of the JavaScript specification

→ Without typing on class members!

```
class Car {
  wheels: number;
  power: number;
  speed: number = 0;

  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }

  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
let myCar = new Car(4, 20); //constructor called
```

Constructor method is called when the class is instantiated through the “new” keyword

Methods can be created without using the function keyword or assigning to “this”

Classes: Extends

```
class Vehicle {
  wheels: number;
  power: number;
  speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
}
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) {
    super(wheels, power);  // Call the parent constructor
    this.gps = true;       // GPS as standard
  }
}
let myCar = new Car(4, 20);
```

The extends and super keywords allow sub-classing

Part of the JavaScript specification

GOTCHA! : “this” will be undefined before you call “super()” in a subclass

Classes: Modifiers (Public, Private, Protected)

```
class Car {
  public wheels: number;
  public power: number;
  public speed: number = 0;
  public constructor (wheels, power) {
    this.wheels = wheels;
    this.power = power;
}

  public accelerate(time) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

let myCar = new Car(4, 20); // constructor called
```

JavaScript has no implementation of Public, Private and Protected – TypeScript does.

Public is the default behavior but can be specified.

Classes: Modifiers (Public, Private, Protected)

```
class Car {
  private wheels: number;
  private power: number;
  private speed: number = 0;
  constructor (wheels, power) {
    this.wheels = wheels;
    this.power = power;
  }
  public accelerate(time) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
let myCar = new Car(4, 20);
console.log(myCar.speed); // Error `speed` is private
```

Private modifier prevents access from outside the class

Classes: Modifiers (Public, Private, Protected)

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  protected speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) { super(wheels, power); }
  public showSpeed() {
    return `Current speed: ${this.speed}`
  }
}
let myCar = new Car(4, 20);
console.log(myCar.showSpeed());
console.log(myCar.speed);             // Error
```

Protected modifier acts much like private except protected members can be accessed by deriving classes.

Classes: Modifiers (Public, Private, Protected)

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  protected speed: number = 0;
  protected constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) { super(wheels, power); }
  public showSpeed() {
    return `Current speed: ${this.speed}`
  }
}

let myCar = new Car(4, 20);
let myVehicle = new Vehicle(4,20); // Error constructor is protected
```

We can protect constructors to enable extension but not instantiation

Constructor Declaration and ASSIGNMENT

```
class Car {
  constructor(
    public wheels: number,
    public power: number,
    public make: string,
    public speed: number = 0      // default
  ) {}
let myCar = new Car(4, 200, `Ford`);
console.log(myCar.speed);  // 0
```

TypeScript allows class variables to be declared and assigned in shorthand.

Simply include the access modifier in the constructor argument and leave the function body empty.

Classes: Structural Types

TypeScript is a structural type system – if the types of all members ae compatible, then the types are compatible.

Except for **`private`** and **`protected`** members.

Classes: Structural Types

```
class Vehicle {
  public wheels: number;
  public power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}
class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
}
class RCCar {
  public wheels: number;
  public power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

let myCar = new Car(4, 20);
let myRCCar = new RCCar(4,5);
let vehicle = new Vehicle(4,15);

vehicle = myCar;                    //ok
vehicle = myRCCar;                  //ok
```

Classes: Structural Types

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}
class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
}
class RCCar {
  protected wheels: number;
  protected power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

let myCar = new Car(4, 20);
let myRCCar = new RCCar(4,5);
let vehicle = new Vehicle(4,15);
vehicle = myCar;       //ok
vehicle = myRCCar;     //Error: RCCar is not a subclass of Vehicle
```

Classes: Readonly

Readonly properties must be initialised at their decleration or in the constructor

```
class Vehicle {
  readonly wheels: number;
  readonly power: number;
  protected speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
}
  accelerate(time: number) {
    this.speed = this.speed + 0.5 * this.power * time;
  }
}

class Car extends Vehicle {
  readonly gps: Boolean = true;
  constructor (wheels, power) {
    super(wheels, power);
  }
}

let myCar = new Car(4, 20);
myCar.wheels = 3; //error - readonly property
```

Classes: Parameter Properties

Parameter properties stop us repeating ourselves quite so much by creating and initialising a property in one place.

By using a modifier in the parameter we create a property.

```
class Vehicle {
  protected speed: number = 0;
  constructor (
    readonly wheels: number,
    readonly power: number
  ) {}
  accelerate(time: number) {
    this.speed = this.speed + 0.5 * this.power * time;
  }
}

class Car extends Vehicle {
  readonly gps: Boolean = true;
  constructor (wheels, power) {
    super(wheels, power);
  }
}

let myCar = new Car(4, 20);
console.log(myCar.wheels); //4
```

Classes: Getters and Setters

Changing properties directly can often be a bad idea, leading to tightly coupled code

Getters and Setters allow us to:

- Encapsulate our implementation
- Add logic to properties

Classes: Getters and Setters

```
class Car {
  private _speed: number = 0;
  constructor (readonly wheels: number, readonly power:
number)
  {}
  get speed(): number {
    return this._speed;
  }
  set speed(newSpeed: number) {
    if (newSpeed && newSpeed > -30 && newSpeed <= 150) {
      this._speed = newSpeed;
    }
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5 * this.power * time;
  }
}
let myCar = new Car(4, 20);
console.log(myCar.speed);  //0
myCar.speed = 100;
console.log(myCar.speed);  //100
myCar.speed = 151;
console.log(myCar.speed);  //100
myCar._speed = 151         // Error
```

Static Properties

We can create static members that are visible on the class itself rather than its instances

Useful for data and behaviour that does not change depending on instance

```
class Car {
  private speed: number = 0;
  static count: number = 0;
  constructor (
    readonly wheels: number, readonly power: number)
  {
    Car.count += 1;
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5 * this.power*time;
  }
}

for (let i = 0; i < 10; i++) {
  new Car(4,20);
}

console.log(Car.count); //10
```

Abstract Classes

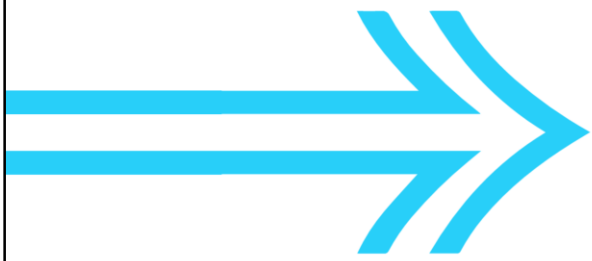
Abstract classes allow us to create base classes from which other classes may be derived.

Abstract classes cannot be instantiated themselves.

Abstract classes provide implementation details

```
abstract class Vehicle {
  wheels: number;
  power: number;
  speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  abstract accelerate(time: number): void;
}

class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
  public accelerate(time: number): void {
     this.speed = this.speed + 0.5  * this.power * time;
  }
}

let myCar = new Car(4, 20);
myCar.accelerate(5);
let myVehicle = new Vehicle(4,20); //Error
```

QuickLab 16 – Classes



In this QuickLab, you will:

- Create classes and their instances
- Experiment with access modifiers and the abstract keyword

Objectives

- To understand how TypeScript works with the JavaScript implementation of classes and inheritance
- To understand how the allowed access modifiers work
- To be able to use get and set with classes
- To understand how abstract classes are implemented
- To be able to use the static keyword

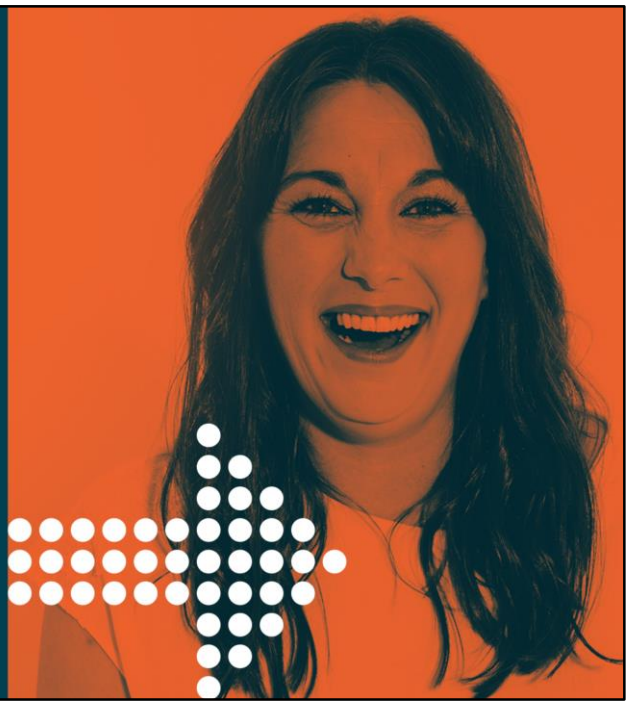

Interfaces in
TypeScript
JavaScript and TypeScript
For Cancer Research UK

Objectives

To understand how to create and use Interfaces with TypeScript

To be able to use inheritance with Interfaces

To understand how excess property checks are done



Interfaces

Interfaces behave like contracts, when we sign (implement) it we must follow its rules.

Interfaces are like abstract classes with (only) abstract methods and properties. There is no actual data or code within.

```
interface Car {
  speed: integer;
  power: integer;
}
```

Interfaces

Interfaces are duck typed (or structural subtyped)

The compiler simply checks we have at least the required members

```
interface Car {
  speed: number;
}

function parkCar(car: Car) {
  car.speed = 0;
  console.log(`Car is parked`);
}

parkCar({speed: 50, power: 200});
```

Optional Properties

Sometimes we may want to hint at a property that is not required.

These are also valid in classes and functions

```
interface Car {
  speed: number;
  fluxCapacitor?: boolean;
  powerOutput?: number;
}

function timeTravel(car: Car) {
  if (car.fluxCapacitor && car.powerOutput >= 1.21) {
    car.speed = 88;
    console.log('Travelling to 1955')
  }
}

timeTravel({
  speed: 50,
  fluxCapacitor: true,
  powerOuput: 1.21
});
```

Optional Properties

This helps prevent runtime errors created by typos

```
interface Car {
  speed: number;
  fluxCapacitor?: boolean;
  powerOutput?: number;
}

function timeTravel(car: Car) {
  //pwerOutput does not exist on Car
  if (car.fluxCapacitor && car.pwerOutput >= 1.21) {
    car.speed = 88;
    console.log('Travelling to 1955')
  }
}

timeTravel({
  speed: 50,
  fluxCapacitor: true,
  powerOuput: 1.21
});
```

Excess Property Checks

Passing in object literals can lead to silent errors, so TypeScript treats object literals with caution and does an excess property check

```
interface Car {
  speed: number;
  fluxCapacitor?: boolean;
  powerOutput?: number;
}

function timeTravel(car: Car) {
if (car.fluxCapacitor && car.powerOutput >= 1.21) {
    car.speed = 88;
    console.log(`Travelling to 1955`)
  }
}

timeTravel({
  speed: 50,
  fluxCapacitor: true,
  pwerOutput: 1.21 //pwerOutput not expected in type `Car`
});
```

Excess Property Checks

To override this we can either:

- Use type assertion
- Add a type index signature
- Assign to a variable first

Should we be trying to get around this check?

Function Types

Interfaces can describe the wide range of shapes that JS objects can take, including functions

Note the name of the parameter in the implementation need not match the name in the interface

```
interface Log {
  (error: string): void;
}

let logError: Log = function(err: string)
{
  console.log(err);
}

logError(`test`);
```

Indexable Types

Similar to Function Types, we can describe types we can index into.

You can have both string and number indexers, but the type returned from number indexers must be a subtype of the type returned from the string indexer (because myGarage[1] === myGaray["1"])

```
interface GarageArray {
  [index: number]: string;
}

let myGarage: GarageArray = [
  `Ford Fiesta`,
  `Audi A3`,
  `Toyota Prius`
]
```

```
interface GarageArray {
  [index: string]: number;
}

let myGarage: GarageArray = {
  "Ford Fiesta": 1,
  "Audi A3": 2,
  "Toyota Prius": 4
}
```

Class Types

Ensuring our class meets a particular contract is one of the most common uses of interfaces in C# and Java. This is possible in TypeScript.

We can define both properties and methods

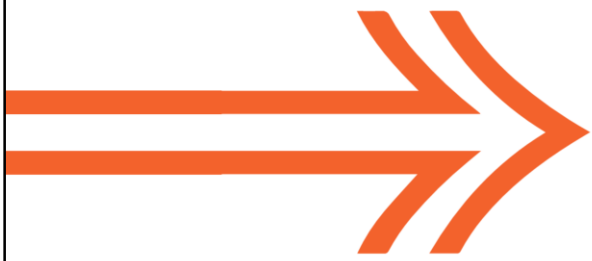
```
interface Car {
  power: number;
  speed: number;
  accelerate(t: number) :void;
}

class FastCar implements Car {
  speed: number = 0;
  constructor(public power: number) {  }

  accelerate(time: number): void {
    this.speed = this.speed + 0.5 * this.power * time
  }
}
```

Extending Interfaces

Classes can extend multiple interfaces giving us fine control over our reusable components

```
interface Vehicle {
  wheels: number;
  colour: string;
}

interface Car extends Vehicle {
  power: number;
  speed: number;
  accelerate(t: number) :void;
}

class FastCar implements Car {
  speed: number = 0;
  wheels: number = 4;
  constructor(public power: number, public colour: string)
{  }

  accelerate(time: number): void {
    this.speed = this.speed + 0.5 * this.power * time
  }
}
```

QuickLab 17 – Interfaces



In this QuickLab, you will:

- Create an interface and make classes implement them

Objectives

To understand how to create and use Interfaces with TypeScript

To be able to use inheritance with Interfaces

To understand how excess property checks are done


Modules in JavaScript and TypeScript
JavaScript and TypeScript
For Cancer Research UK

Objectives

To be aware of the JavaScript ES2015+ module syntax for importing and exporting

Modules - Nomenclature

Pre typescript 1.5 there was a concept of “internal modules” and “external modules”

ECAMScript2015 introduced “modules” to JavaScript and so TypeScript has changed its terminology to match.

Internal modules are now “namespaces”

External modules are now simply “modules”

Modules

Modules run in their own scope, avoiding pollution of the global scope

Only what is exported is exposed externally

Only what is imported is usable internally

Any declaration can be exported through using the export keyword

```
//vehicles.ts
export interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}

export class Car implements Vehicle {
  wheels = 4;
  constructor(
    public make:string,
    public model:string)
  {}

  accelerate(time: number) {…}
}
```

Modules: Export Statements

Export statements can be used to export under a different name

```
//vehicles.ts
interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}

class Car implements Vehicle {
  wheels = 4;
  constructor(
    public make:string,
    public model:string)
  {}
  accelerate(time: number) {…}
}

export { Vehicle };
export { Car as BaseCar };
```

Modules: Default Export

Optionally a default export can be specified

Two syntaxes available

A file can only have 1 default export statement

```
//vehicles.ts
export default interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}
```

```
// cars.ts
class Car implements Vehicle {
  wheels = 4;
  constructor(public make:string, public
model:string){}
  accelerate(time: number) {…}
}

export { Car as default };
```

Modules: Importing

Importing is just as simple as exporting

We use the import keyword with one of the following forms:

```
import { Vehicle, Car } from
'./vehicles';

let myCar = new Car(`Ford`,`Fiesta`);
```

→ We can rename on import

```
import { Car as BasicCar} from
'./vehicles';

let myCar = new BasicCar(`Ford`,
`Fiesta`);
```

→ Or import the whole file!

```
import * as vehicles from './vehicles';

let myCar = new vehicles. Car(`Ford`,
`Fiesta`);
```

→ Importing the default is the simplest form

```
import Car from './vehicles';
```

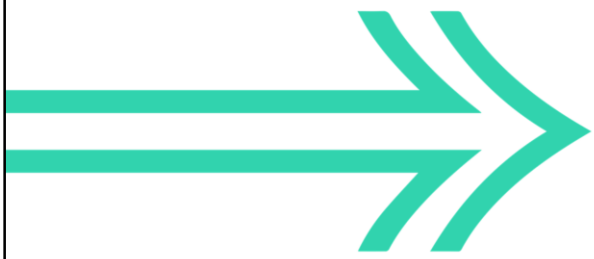
Modules: Code Creation

The TypeScript compiler is not a module loader. It will compile your code to whatever module format you tell it to.

A module loader will be required to then make your code ready for deployment

- This is what we have been using Webpack for although we have only been using a single file so far

QuickLab 18 – Modules



In this QuickLab, you will:

- Separate out code into modules and import where needed

Objectives

To be aware of the JavaScript ES2015+ module syntax for importing and exporting

POWERING POTENTIAL

Course Outcomes

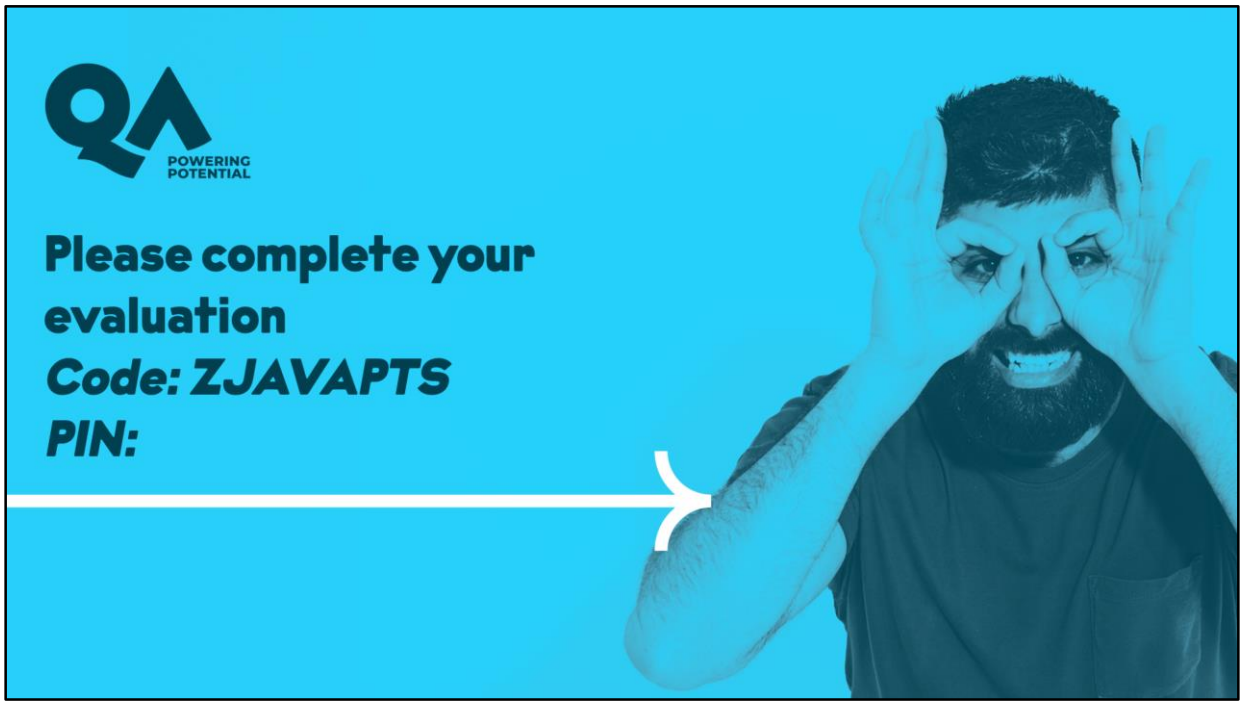
By the end of the course, you will be able to:

- Be able to write code using ES2015+ syntax JavaScript including:
  - Variables and constants
  - Primitive and Reference Types
  - Arrays and Collections
  - Functions
  - Object Orientation
  - Asynchronous techniques
- Be able to use TypeScript to make JavaScript more type-safe

POWERING POTENTIAL

Hope you enjoyed this training course.

ed.wright@qa.com


POWERING
POTENTIAL
Please complete your
evaluation
Code: ZJAVAPTS
PIN: