Introduction

WEB DEVELOPMENT FUNDAMENTALS – JAVASCRIPT

Overview

- Objectives
  - To explain the aims and objectives of the course
- Contents
  - Course administration
  - Course objectives and assumptions
  - Introductions
  - Any questions?
- Exercise
  - Locate the exercises
  - Locate the help files

Administration

- Front door security
- Name card
- Chairs

- Fire exits
- Toilets
- Coffee Room

- Timing
- Breaks
- Lunch

- Downloads and viruses
- Admin. support
- Messages

- Taxis
- Trains/Coaches
- Hotels

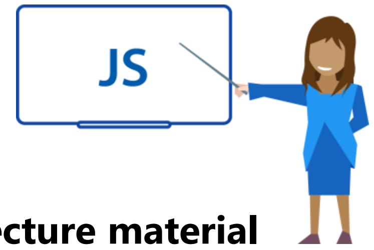
- First Aid

- Telephones/Mobiles

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.

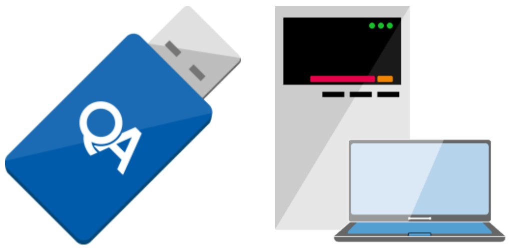
Course delivery



Lecture material

Hear and Forget
See and Remember
Do and Understand

Course workbooks



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.

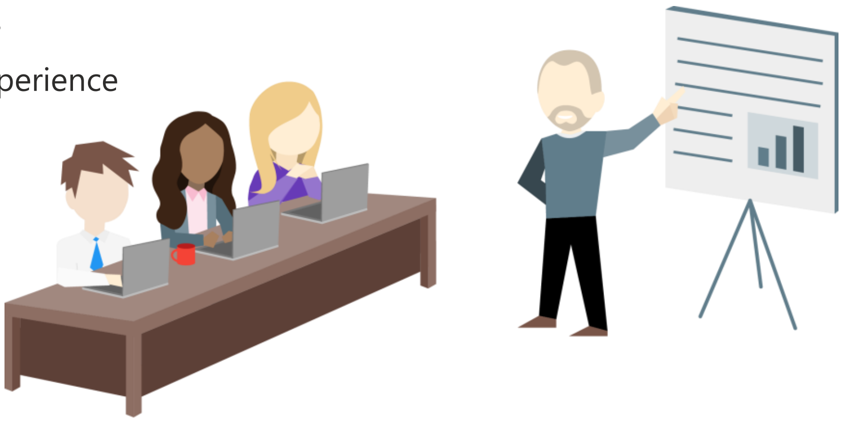
The appendices cover material that is beyond the scope of the course, together with some help and guidelines. There are also appendices on bibliography and Internet resources to help you find more information after the course.

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.

A training experience

A course should be

- A two-way process
- A group process
- An individual experience



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 aims and objectives

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

- Set up a development environment for programming in modern JavaScript
- Manage and use JavaScript types and data structures effectively
- Control the flow of programs using loops and conditional code
- Use JavaScript alongside HTML, manipulating and changing the DOM
- React to events to make web pages respond to user interaction, including form handling
- Produce and use basic Object Oriented JavaScript
- Work with asynchronous data using JavaScript

Assumptions

- This course assumes the following prerequisites
  - Delegates ***MUST*** have HTML and CSS skills equivalent to those provided by the Web Development Fundamentals – HTML and CSS course
  - Familiarity with programming would be beneficial, we recommend Programming Foundations to new developers
- If there are any issues, please tell your instructor now

Introductions

Please say a few words about yourself

- What is your name and job?
- What is your current experience of
  - Computing?
  - Programming?
  - Web development?
- What is your main objective for attending the course?

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.

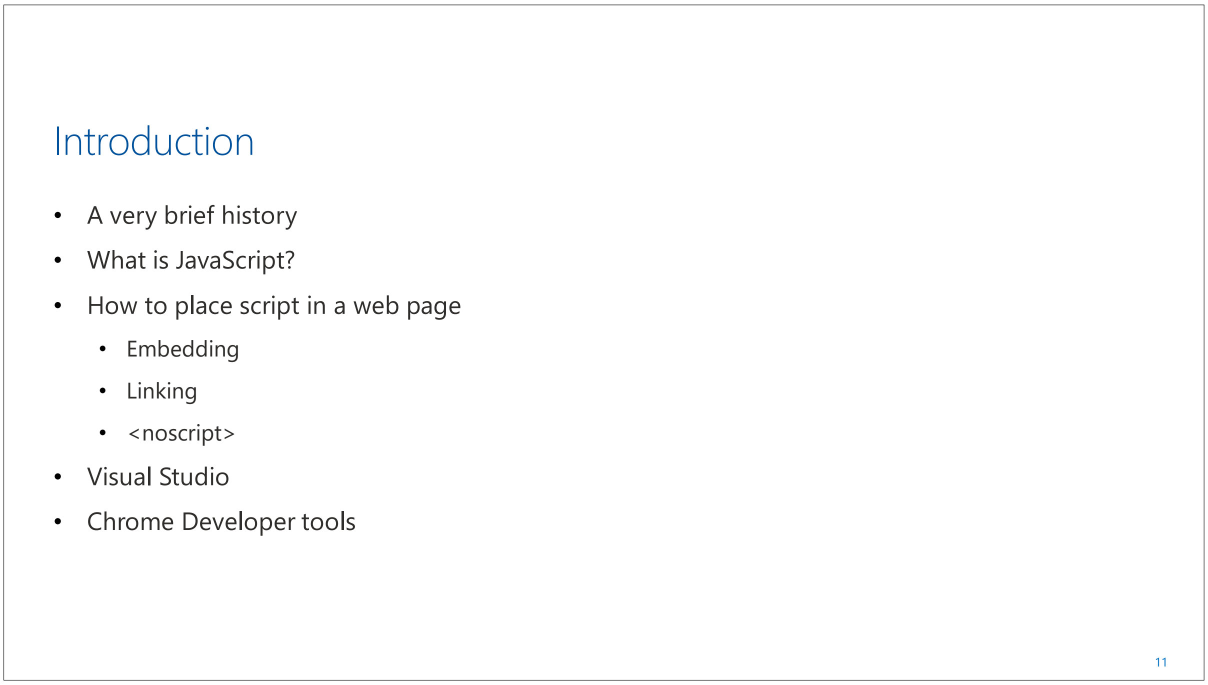
Any questions

- Golden Rule
  - "There is no such thing as a stupid question"
- First amendment to the Golden Rule
  - "... even when asked by an instructor"
- Corollary to the Golden Rule
  - "A question never resides in a single mind"

questions during the course. If so, ask them!

Introduction to JavaScript

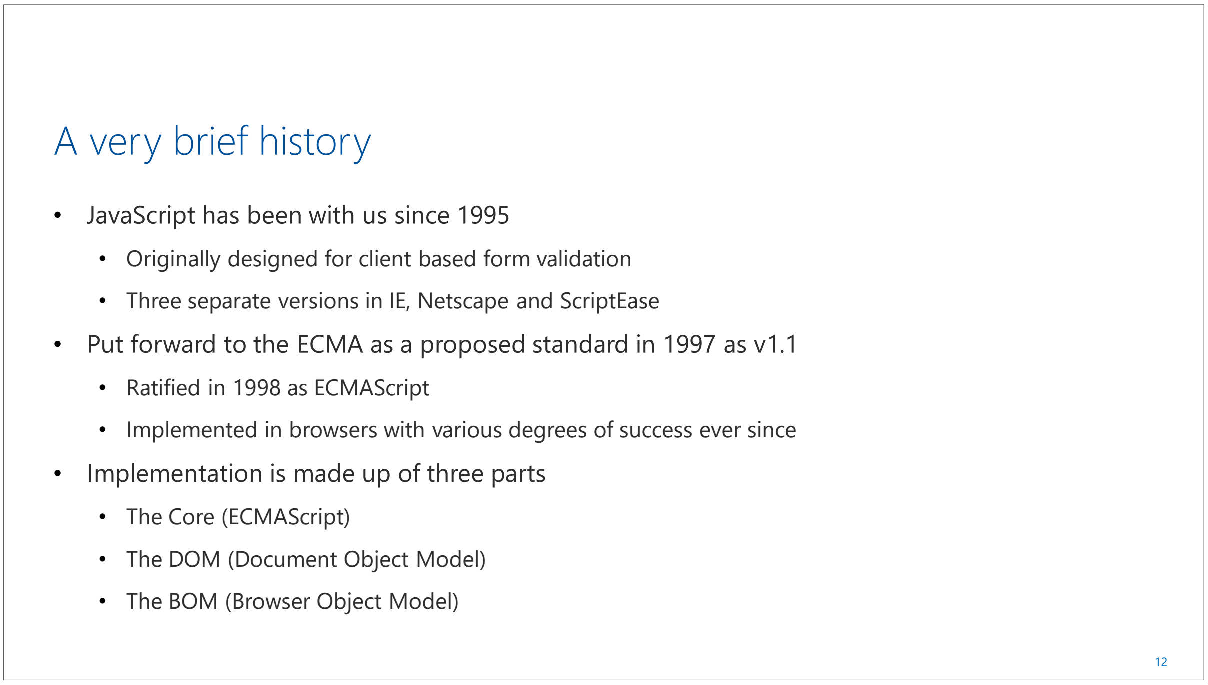
WEB DEVELOPMENT FUNDAMENTALS - JAVASCRIPT



Introduction

- A very brief history
- What is JavaScript?
- How to place script in a web page
  - Embedding
  - Linking
  - <noscript>
- Visual Studio
- Chrome Developer tools

11



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)

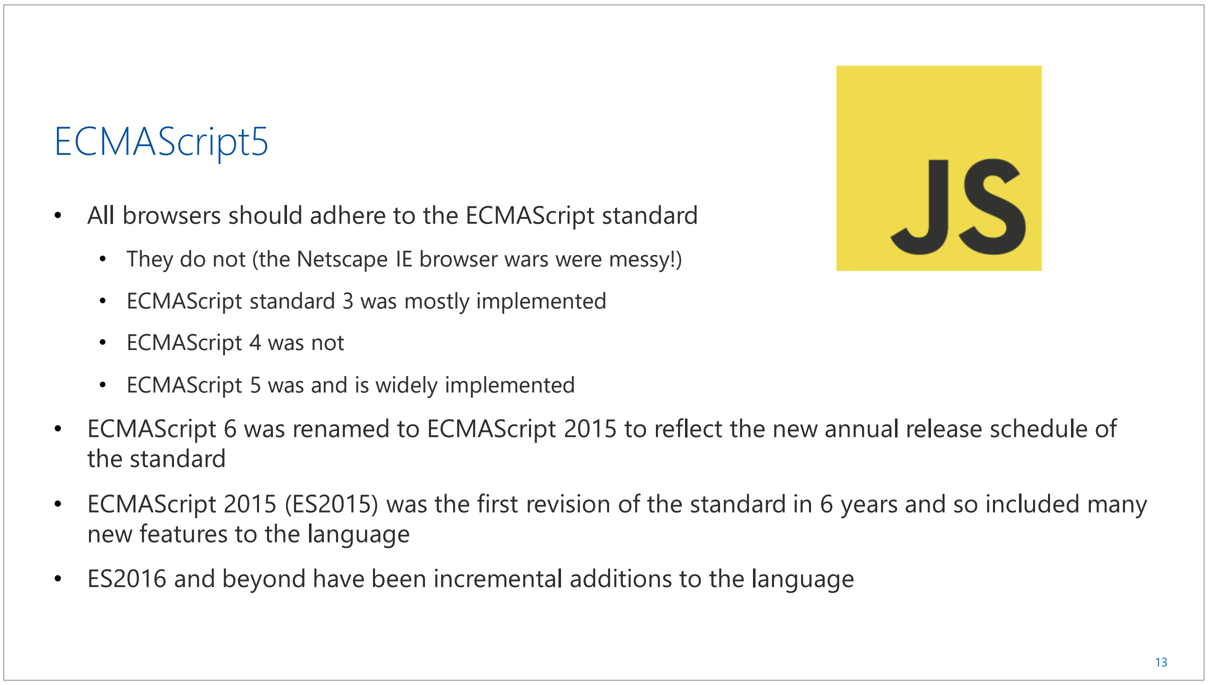
12

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.

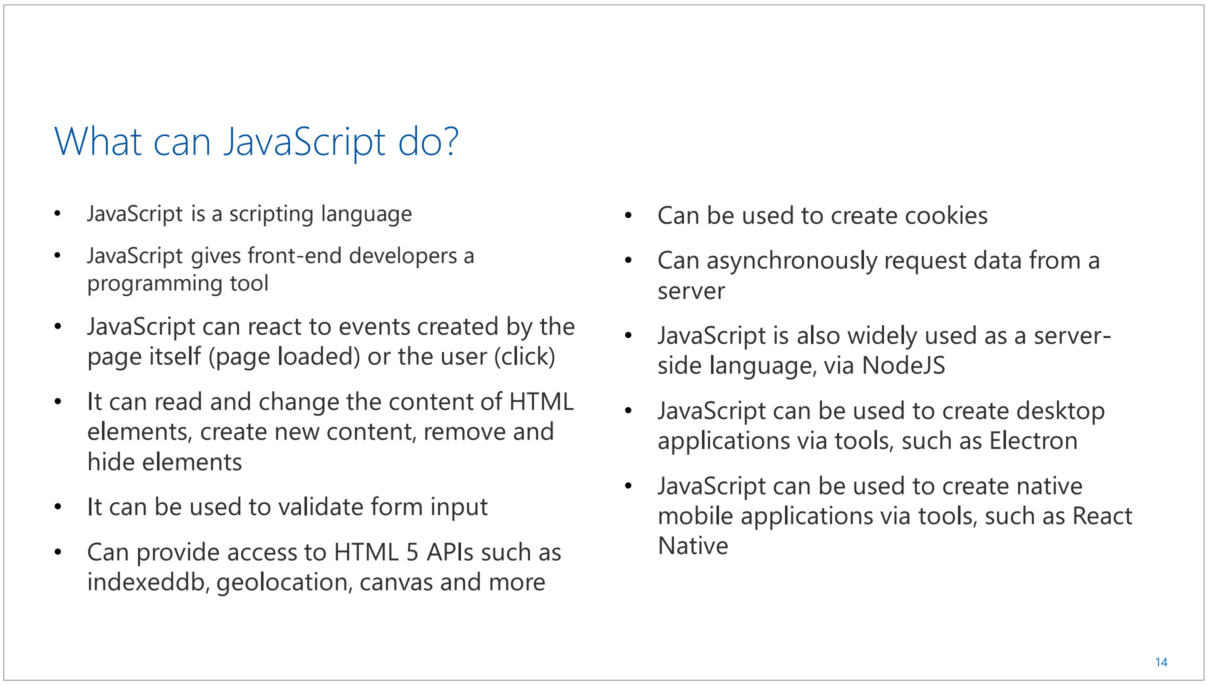


ECMAScript5

- 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

13

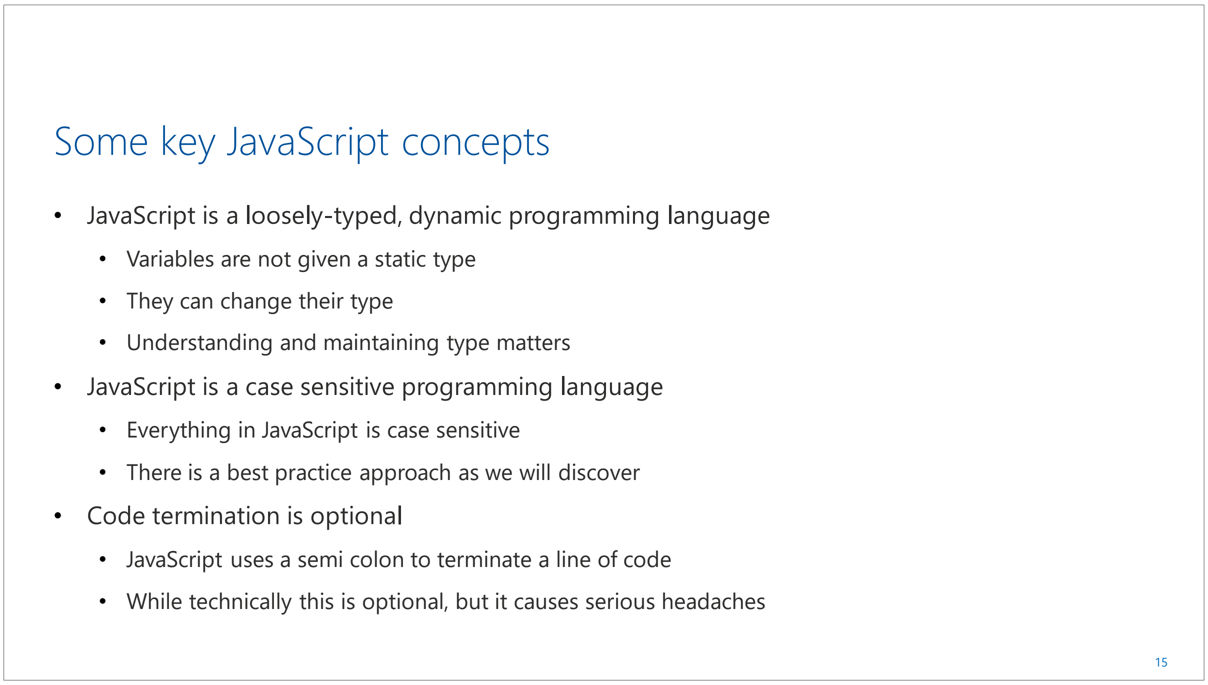
The 6th edition, officially known as ECMAScript 2015, was finalised in June 2015. This update adds significant new syntax for writing complex applications, including classes and modules, but defines them semantically in the same terms as ECMAScript 5 strict mode. Other new features include iterators 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?

- 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
- 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

14



Some key JavaScript concepts

- JavaScript is a loosely-typed, dynamic programming language
  - Variables are not given a static type
  - They can change their type
  - Understanding and maintaining type matters
- JavaScript is a case sensitive programming language
  - Everything in JavaScript is case sensitive
  - There is a best practice approach as we will discover
- Code termination is optional
  - JavaScript uses a semi colon to terminate a line of code
  - While technically this is optional, but it causes serious headaches

15

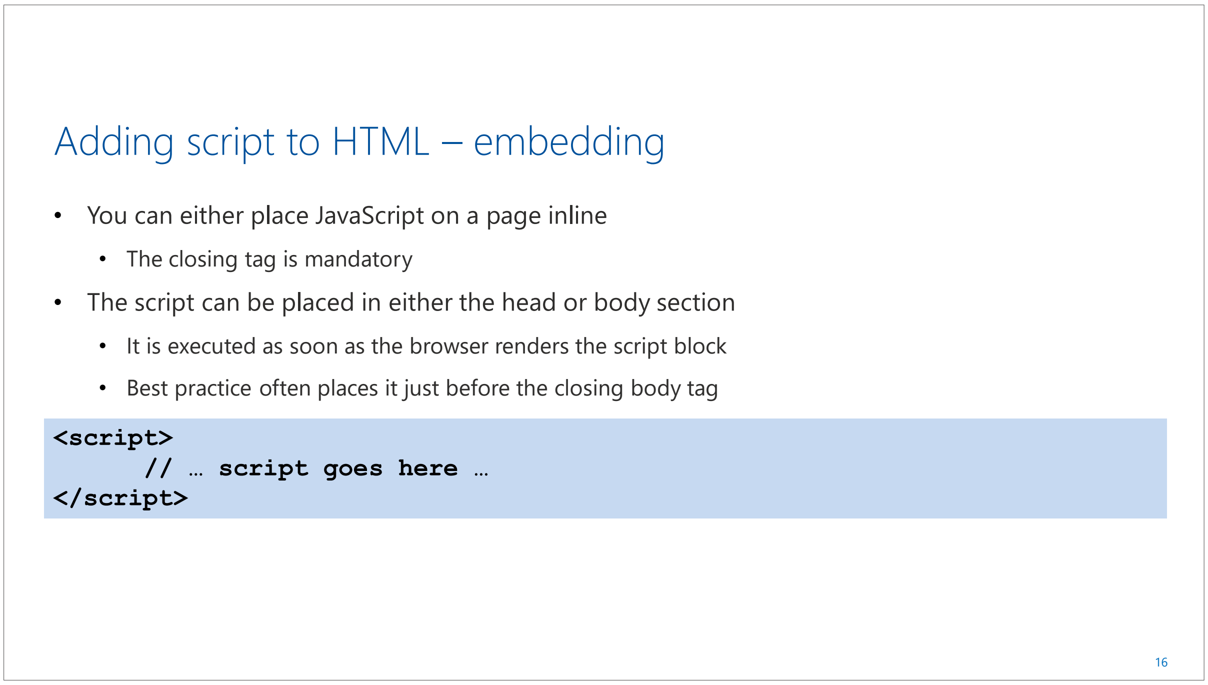
As we begin our journey with JavaScript, we will get a few things locked down in our brainboxes that will save us a lot of heartache and pain as the course goes on.

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

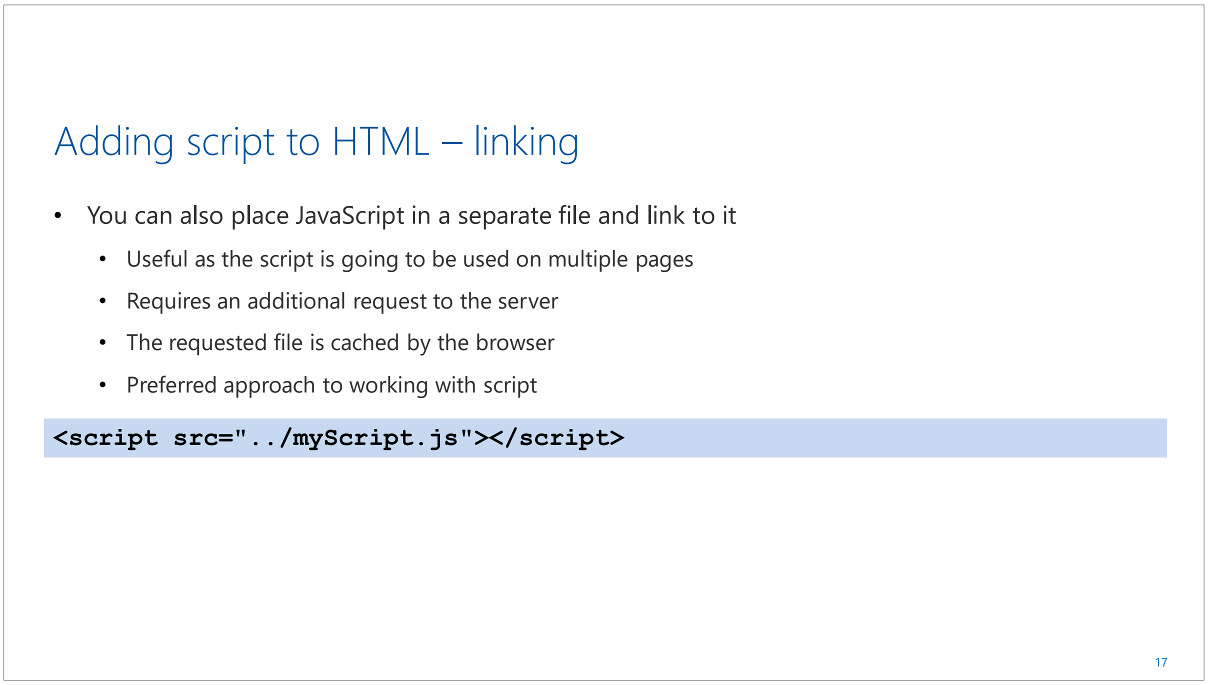
- You can either place JavaScript on a page inline
  - The closing tag is mandatory
- The script can be placed in either the head or body section
  - It is executed as soon as the browser renders the script block
  - Best practice often places it just before the closing body tag

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

16

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

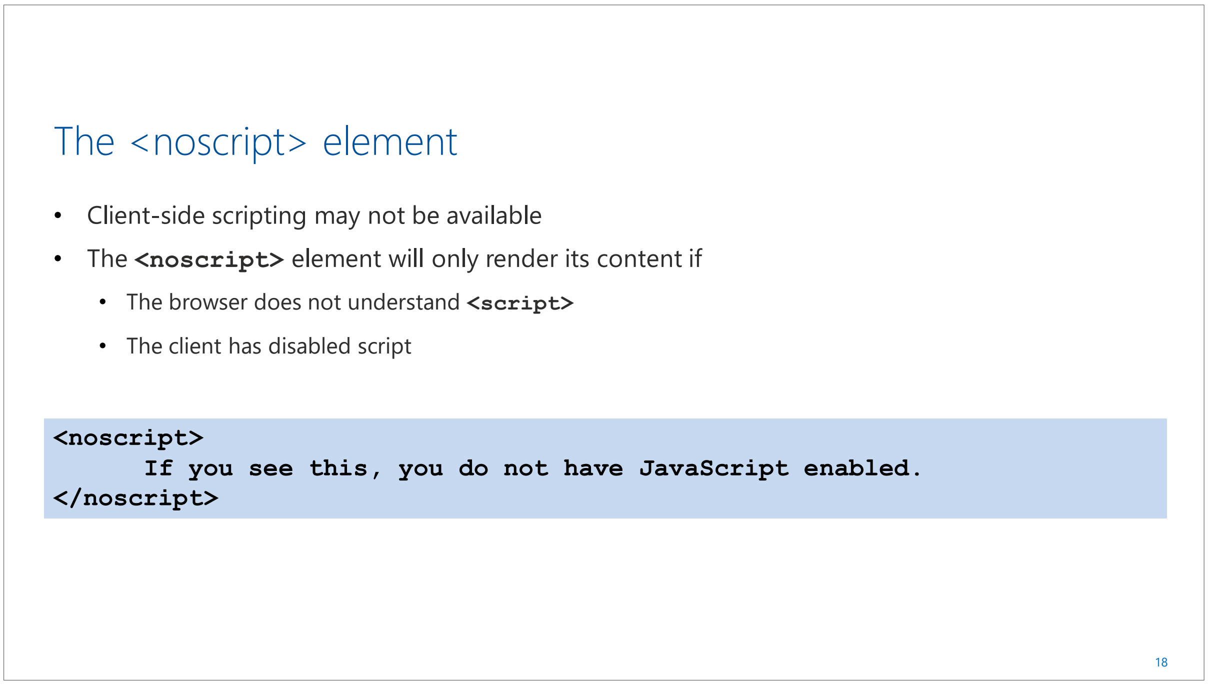
- You can also place JavaScript in a separate file and link to it
  - Useful as the script is going to be used on multiple pages
  - Requires an additional request to the server
  - The requested file is cached by the browser
  - Preferred approach to working with script

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

17

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.

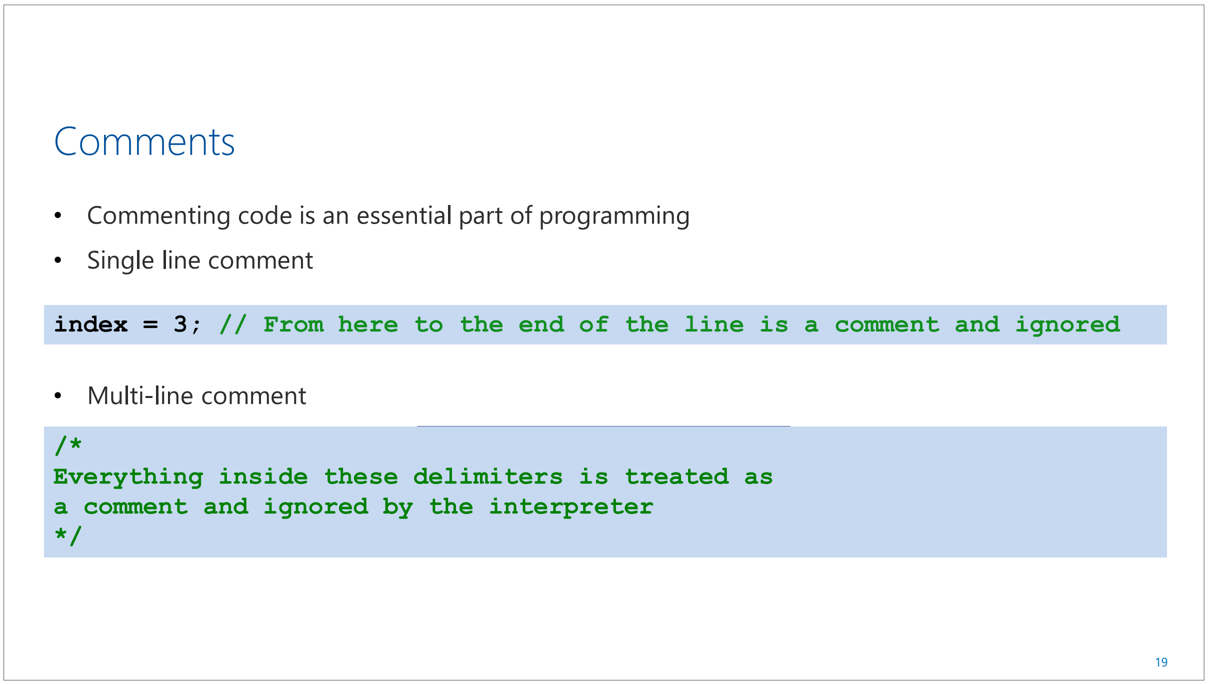


The <noscript> element

- 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>
```

18



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
*/
```

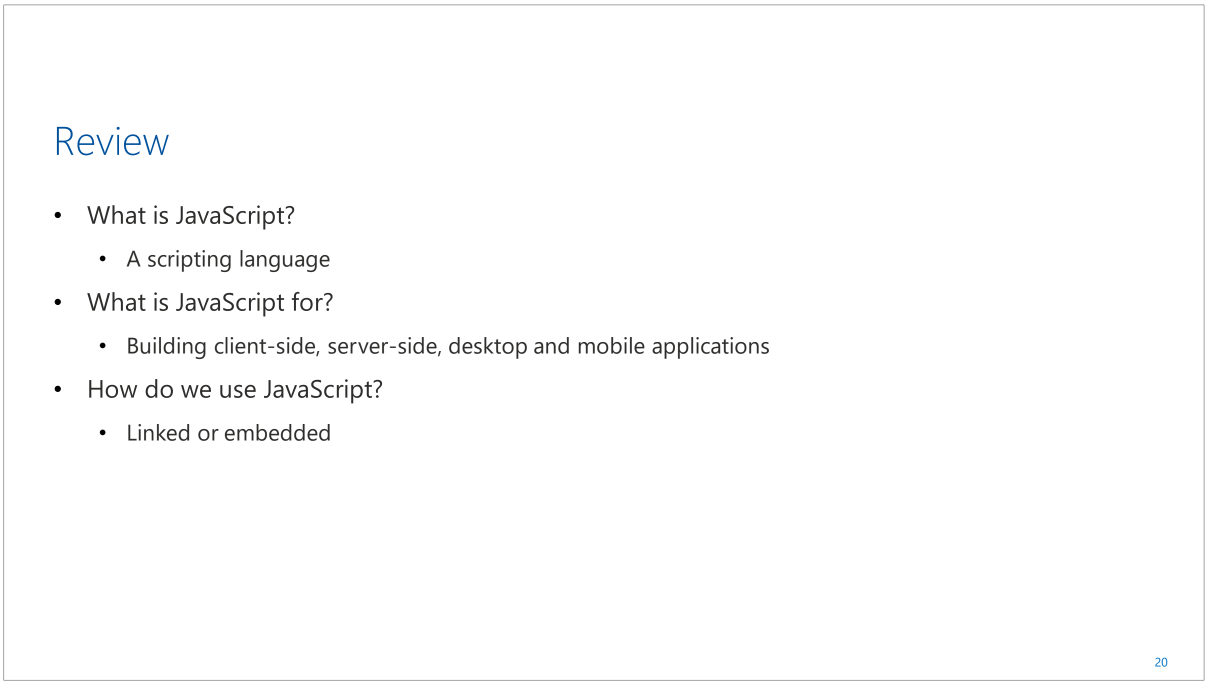
19

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.



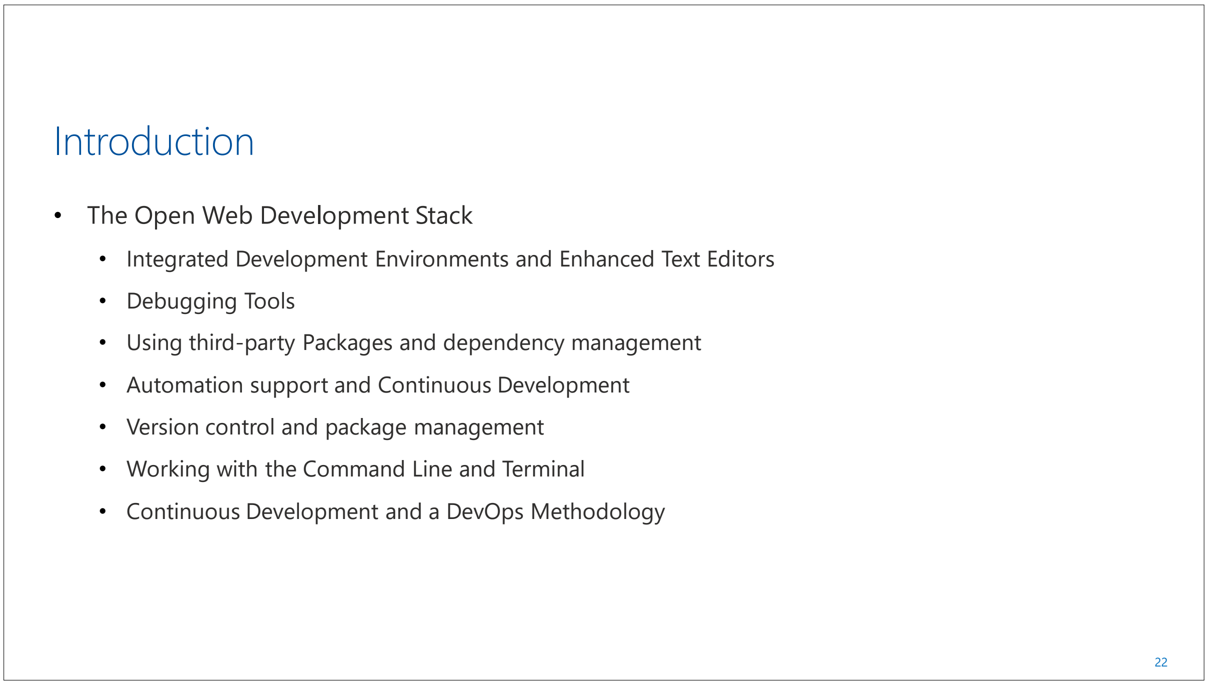
Review

- 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

20

Web Development Tools

JAVASCRIPT FUNDAMENTALS



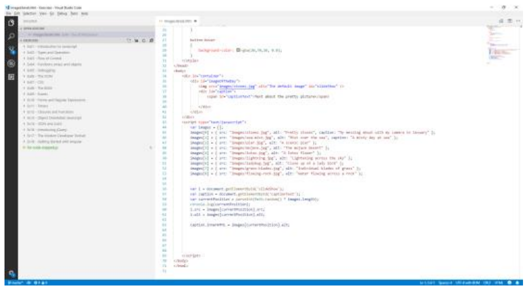
Introduction

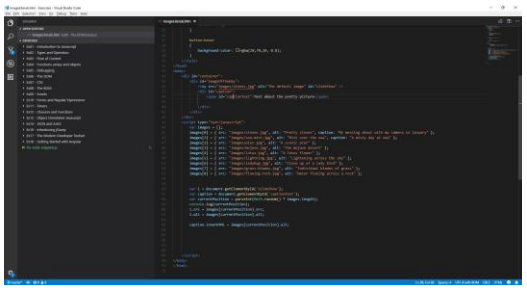
- The Open Web Development Stack
  - Integrated Development Environments and Enhanced Text Editors
  - Debugging Tools
  - Using third-party Packages and dependency management
  - Automation support and Continuous Development
  - Version control and package management
  - Working with the Command Line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

22

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)

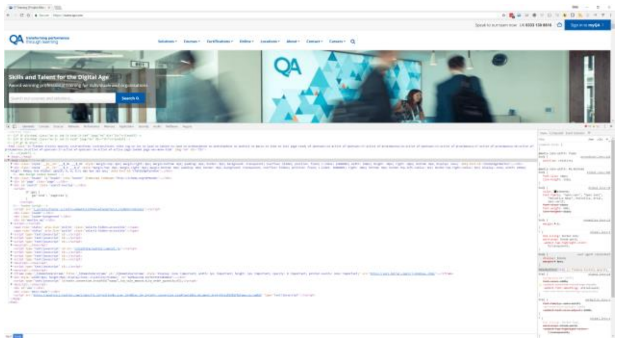




23

Walkthrough – Chrome development tools

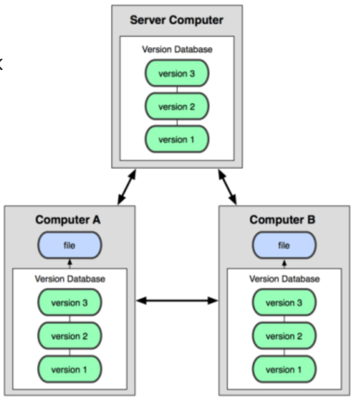
- Most browsers have development and debugging tools
- Incredibly useful for testing and management



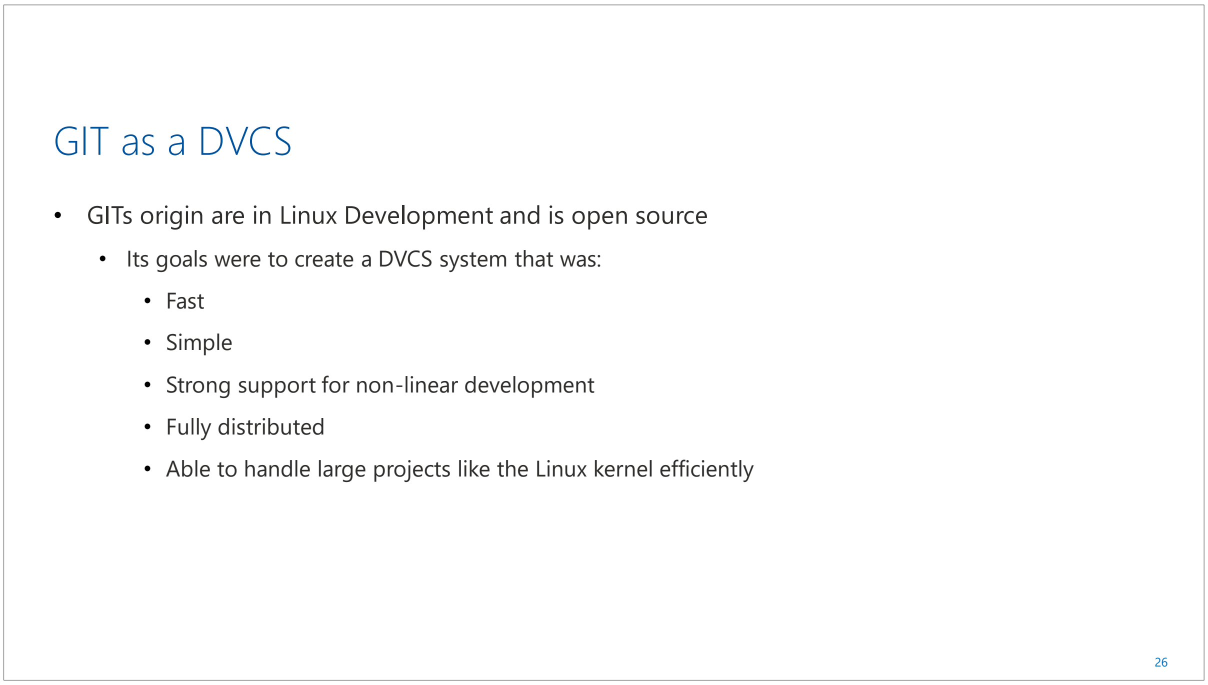
24

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 back



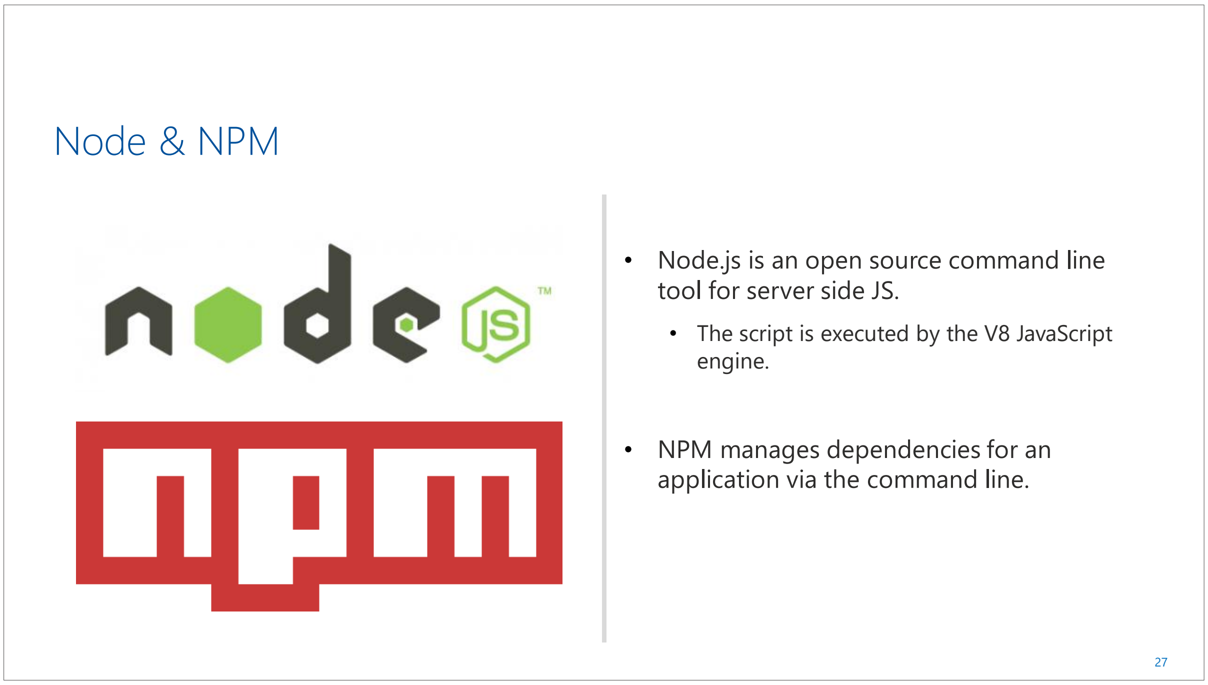
25



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

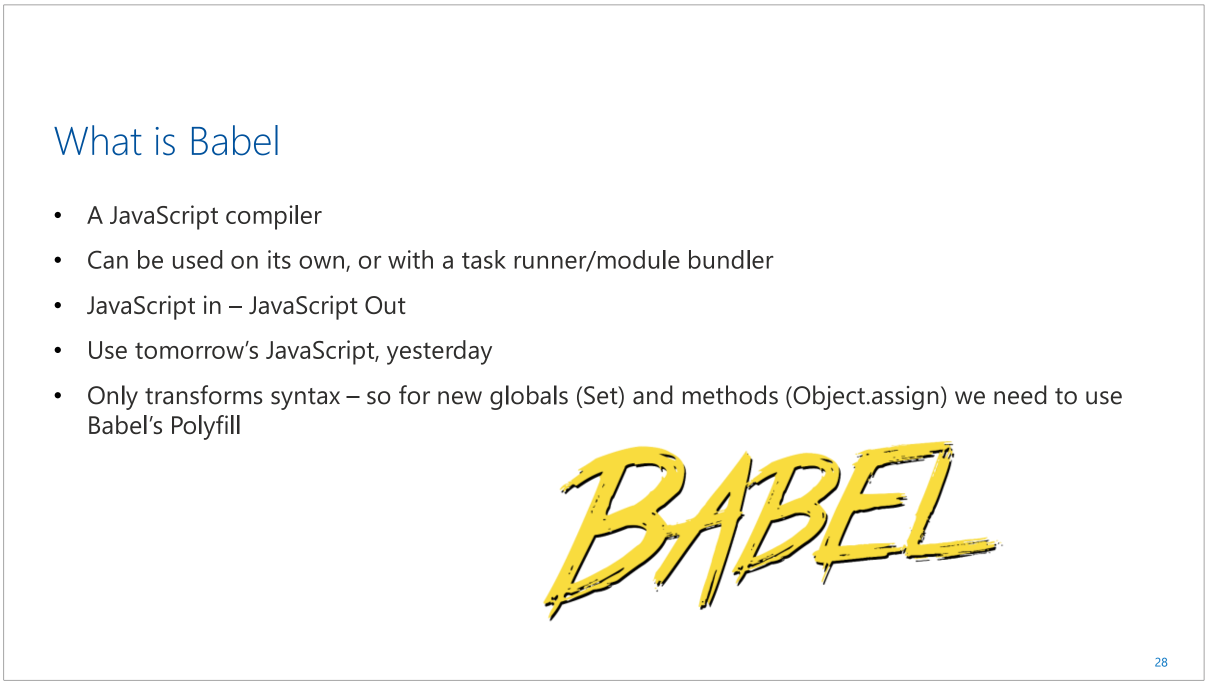
26



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.

27



What is Babel

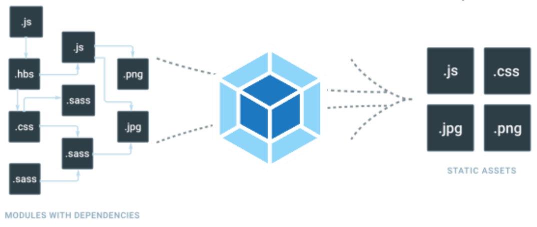
- A JavaScript compiler
- Can be used on its own, or with a task runner/module bundler
- JavaScript in – JavaScript Out
- Use tomorrow's JavaScript, yesterday
- Only transforms syntax – so for new globals (Set) and methods (Object.assign) we need to use Babel's Polyfill

28

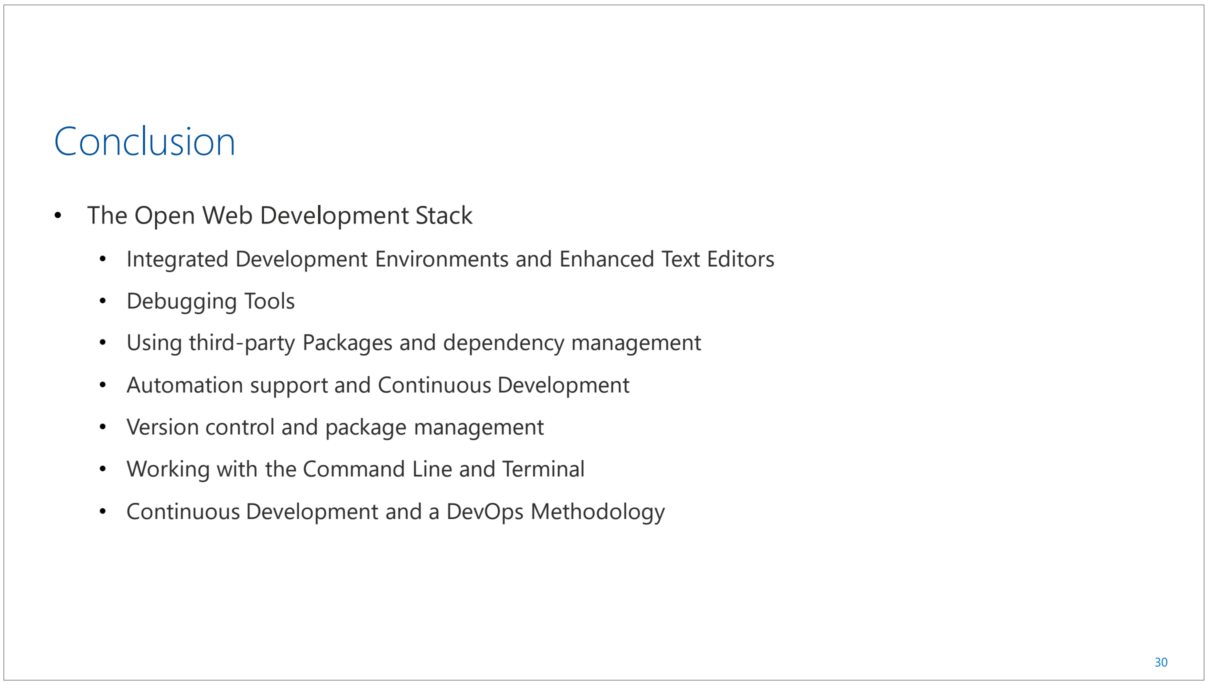
Babel transpiles ES2015+ Syntax back to ES3+ syntax and they have created a polyfill to create the new globals and methods found in ES2015+. This means that tomorrow's JavaScript works in yesterday's browsers!

webpack

- webpack is a module bundler for modern JavaScript applications
- Entry points define where webpack starts. From here it builds a graph of your applications dependencies
- Output tells webpack where to emit the bundled code
- Loaders teach webpack how to handle assets that aren't JavaScript
- Plugins are used to perform actions and add custom functionality for our bundled code



29



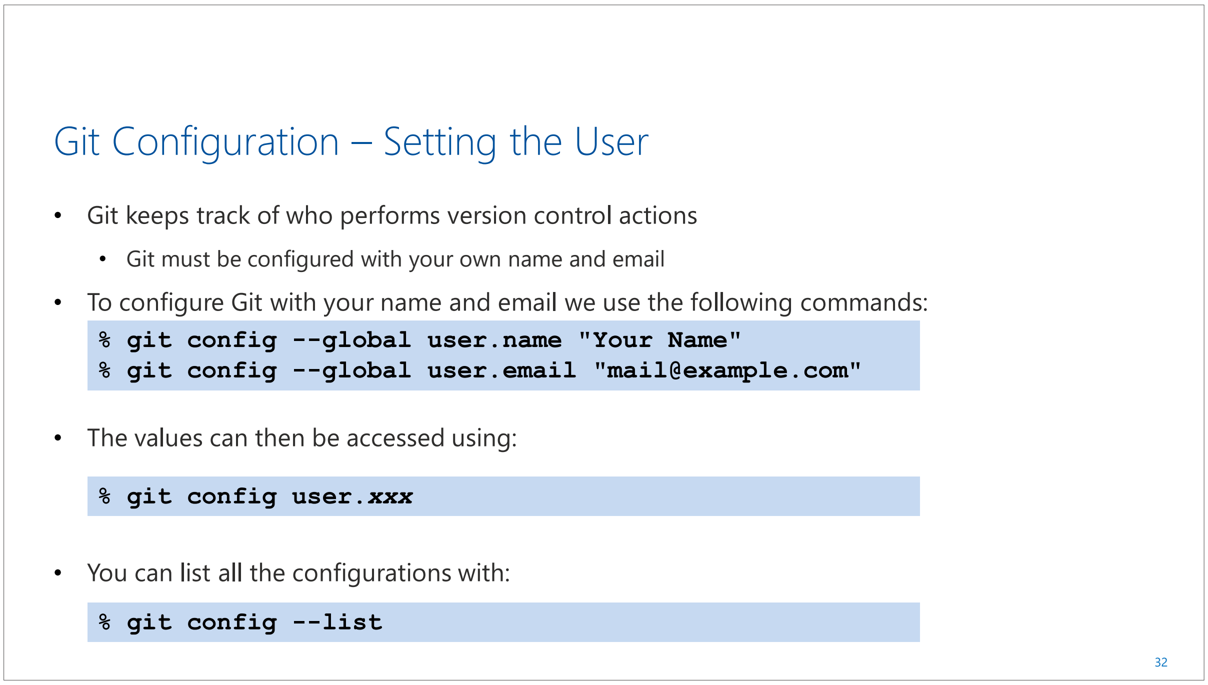
Conclusion

- 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

30

APPENDIX

- GIT Essentials



Git Configuration – Setting the User

- Git keeps track of who performs version control actions
  - Git must be configured with your own name and email
- To configure Git with your name and email we use the following commands:

```
% git config --global user.name "Your Name"
% git config --global user.email "mail@example.com"
```

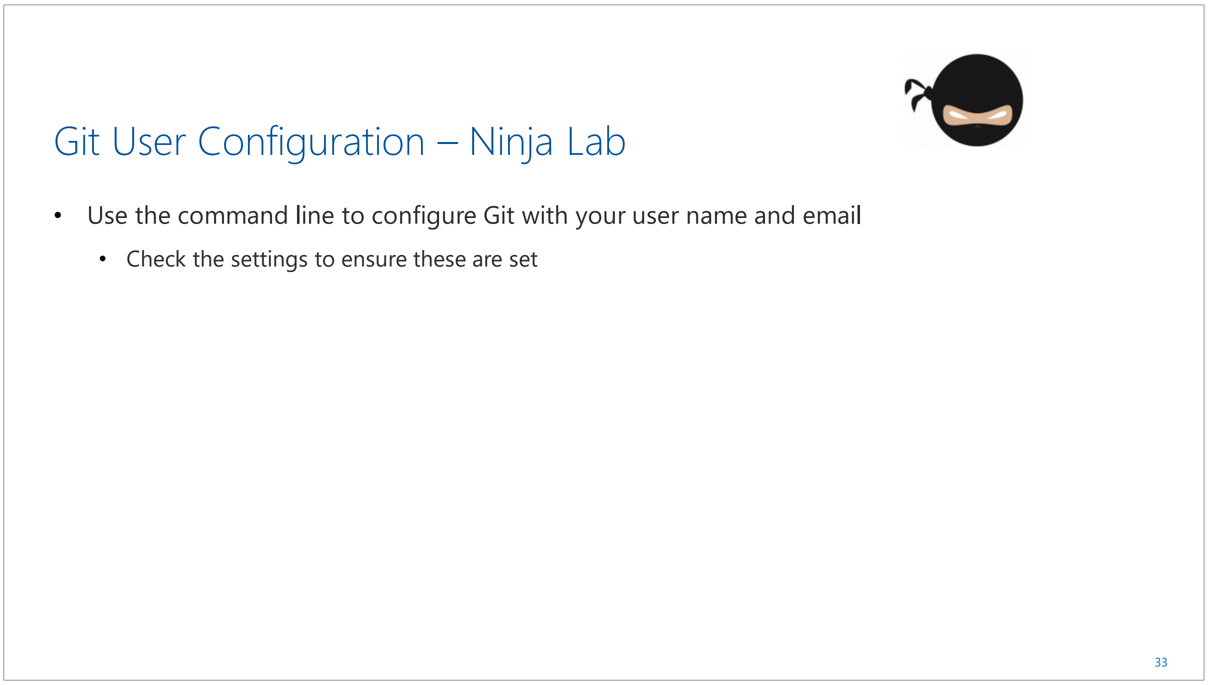
- The values can then be accessed using:

```
% git config user.xxx
```

- You can list all the configurations with:

```
% git config --list
```

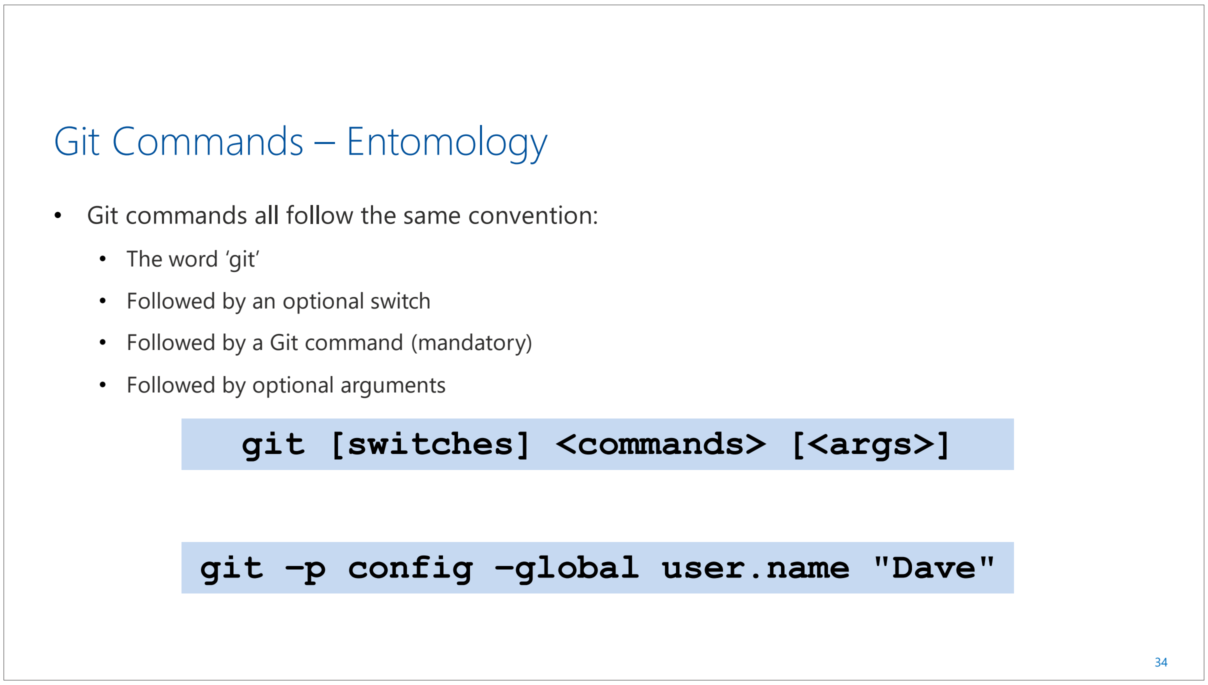
32



Git User Configuration – Ninja Lab

- Use the command line to configure Git with your user name and email
  - Check the settings to ensure these are set

33



Git Commands – Entomology

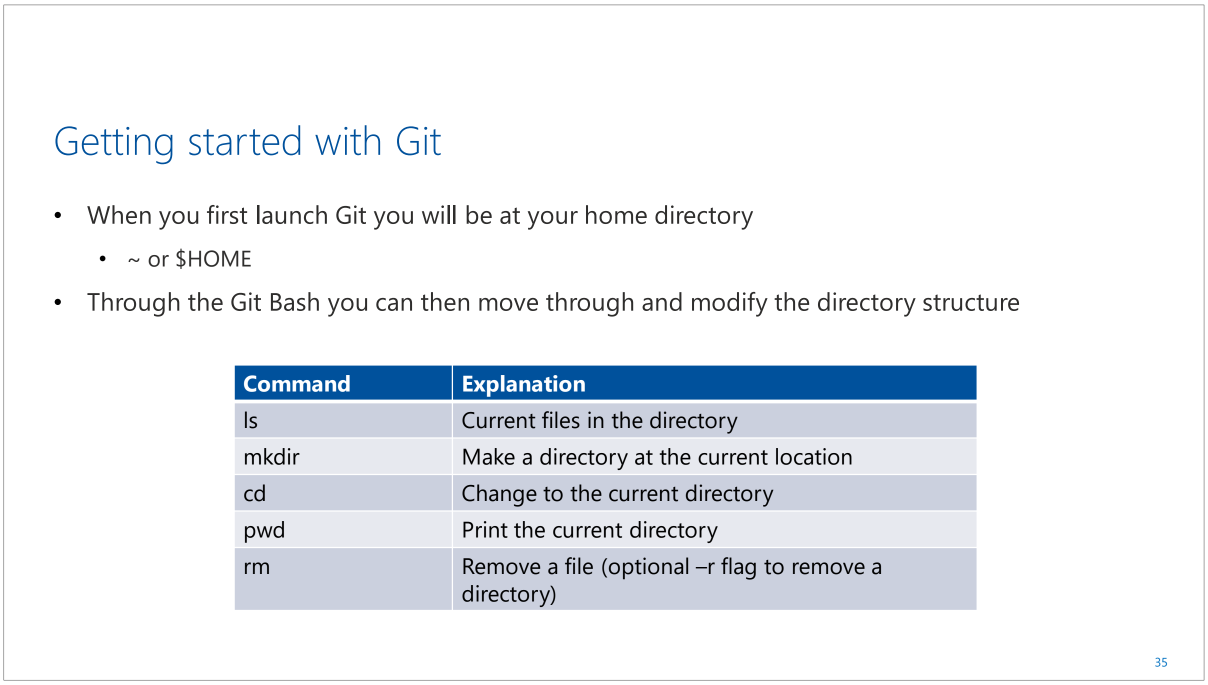
- Git commands all follow the same convention:
  - The word ‘git’
  - Followed by an optional switch
  - Followed by a Git command (mandatory)
  - Followed by optional arguments

```
git [switches] <commands> [<args>]
```

```
git -p config -global user.name "Dave"
```

34

The -p switch paginates the output if needed.

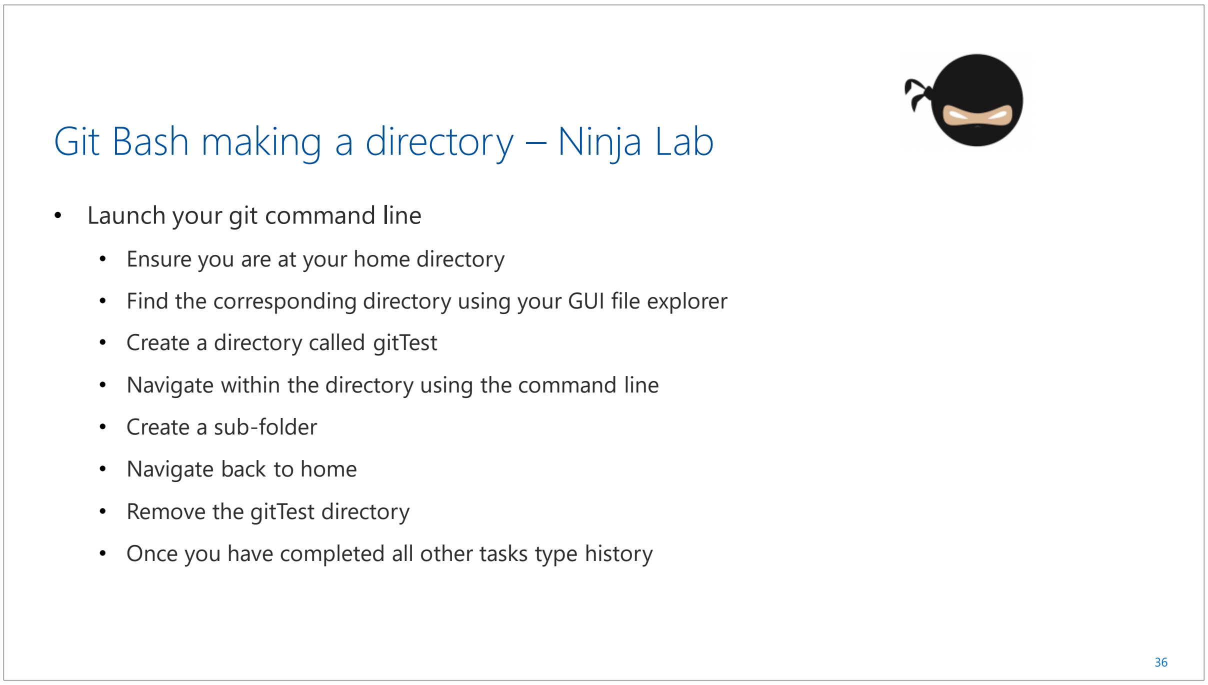


Getting started with Git

- When you first launch Git you will be at your home directory
  - ~ or $HOME
- Through the Git Bash you can then move through and modify the directory structure

Command	Explanation
ls	Current files in the directory
mkdir	Make a directory at the current location
cd	Change to the current directory
pwd	Print the current directory
rm	Remove a file (optional –r flag to remove a directory)

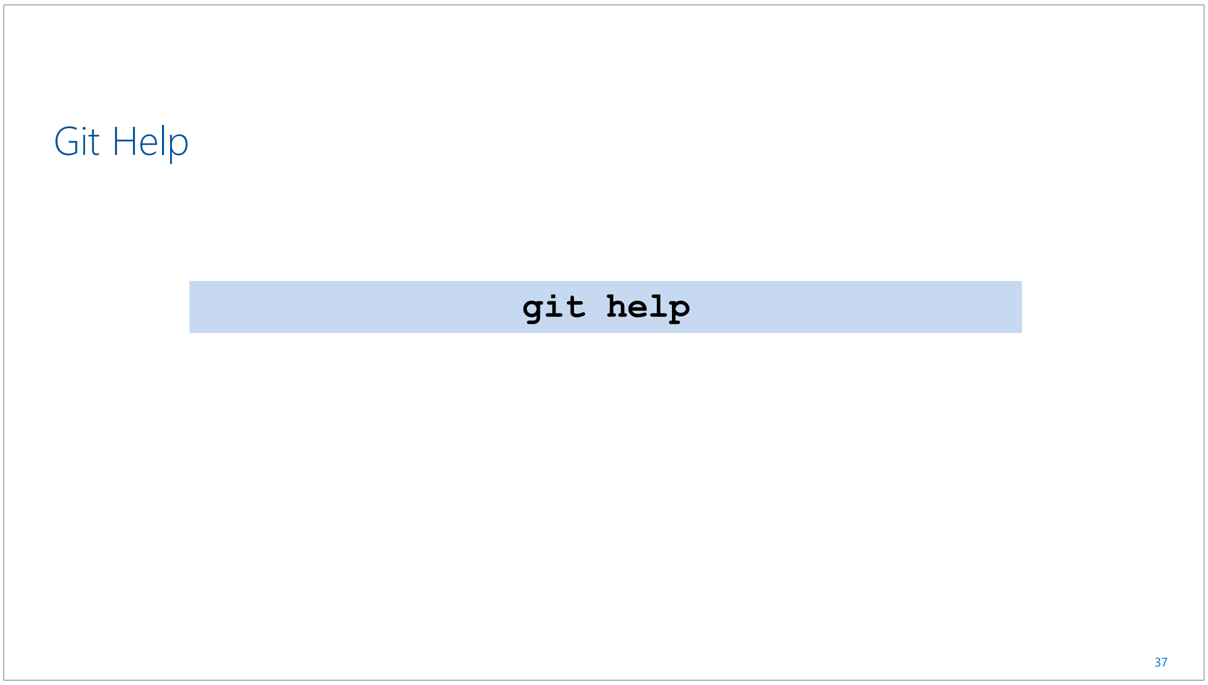
35



Git Bash making a directory – Ninja Lab

- Launch your git command line
  - Ensure you are at your home directory
  - Find the corresponding directory using your GUI file explorer
  - Create a directory called gitTest
  - Navigate within the directory using the command line
  - Create a sub-folder
  - Navigate back to home
  - Remove the gitTest directory
  - Once you have completed all other tasks type history

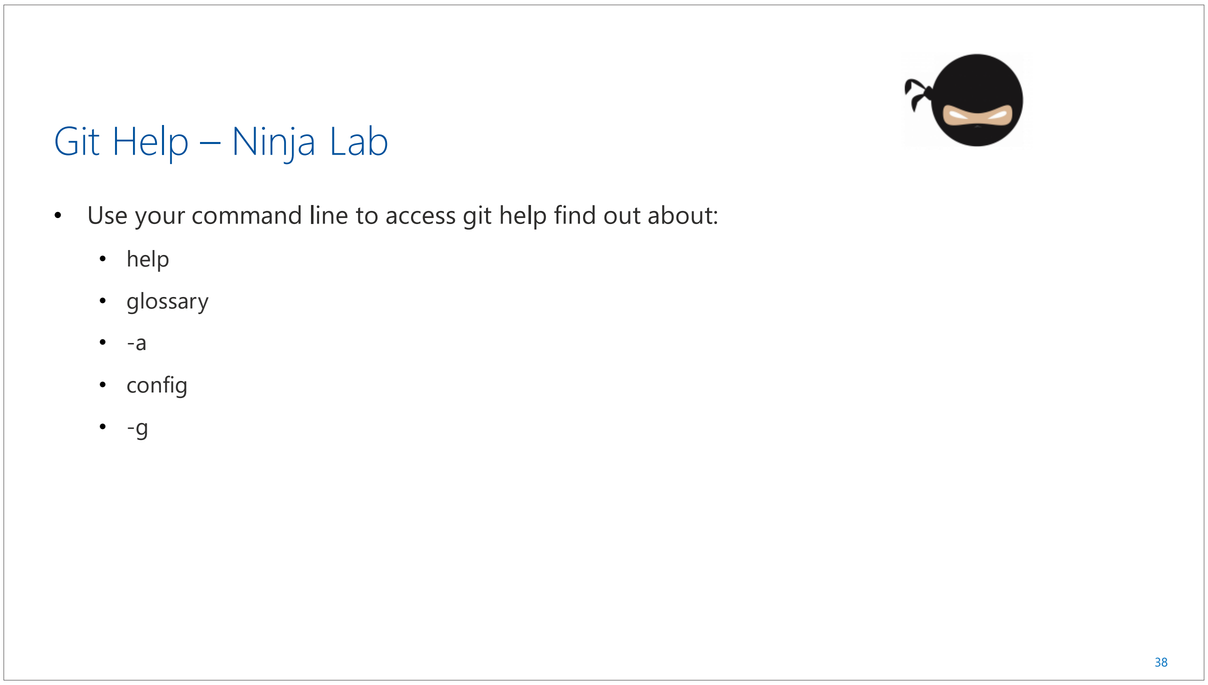
36



Git Help

```
git help
```

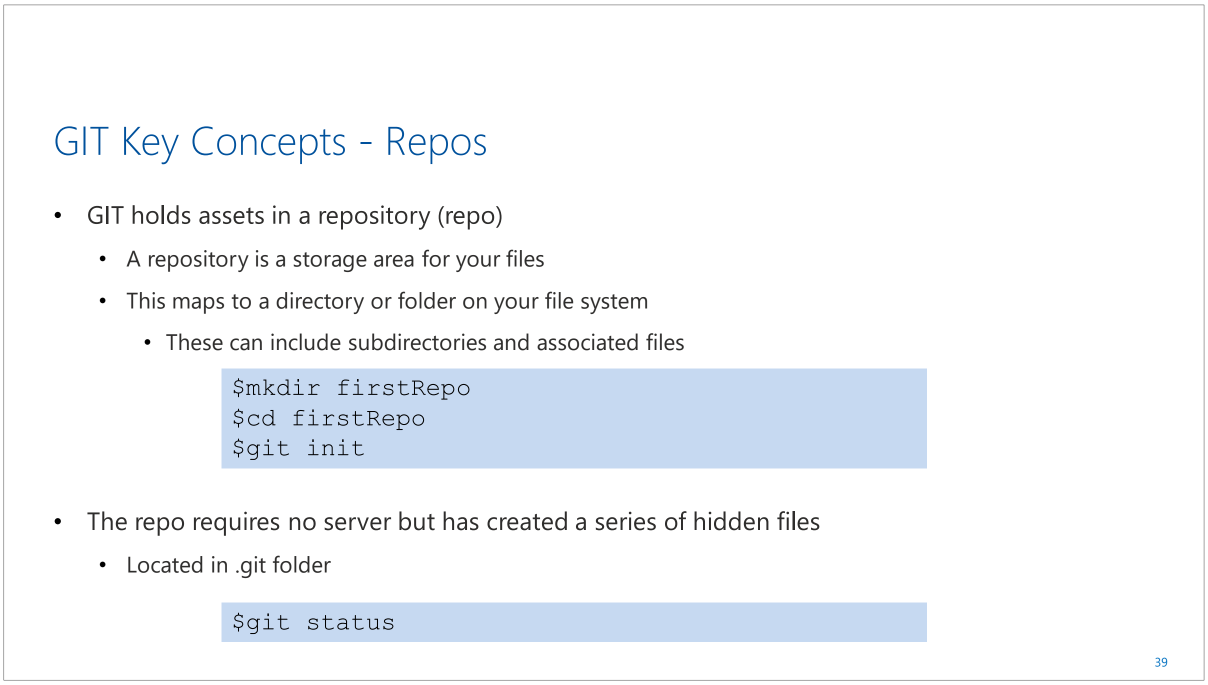
37



Git Help – Ninja Lab

- Use your command line to access git help find out about:
  - help
  - glossary
  - -a
  - config
  - -g

38



GIT Key Concepts - Repos

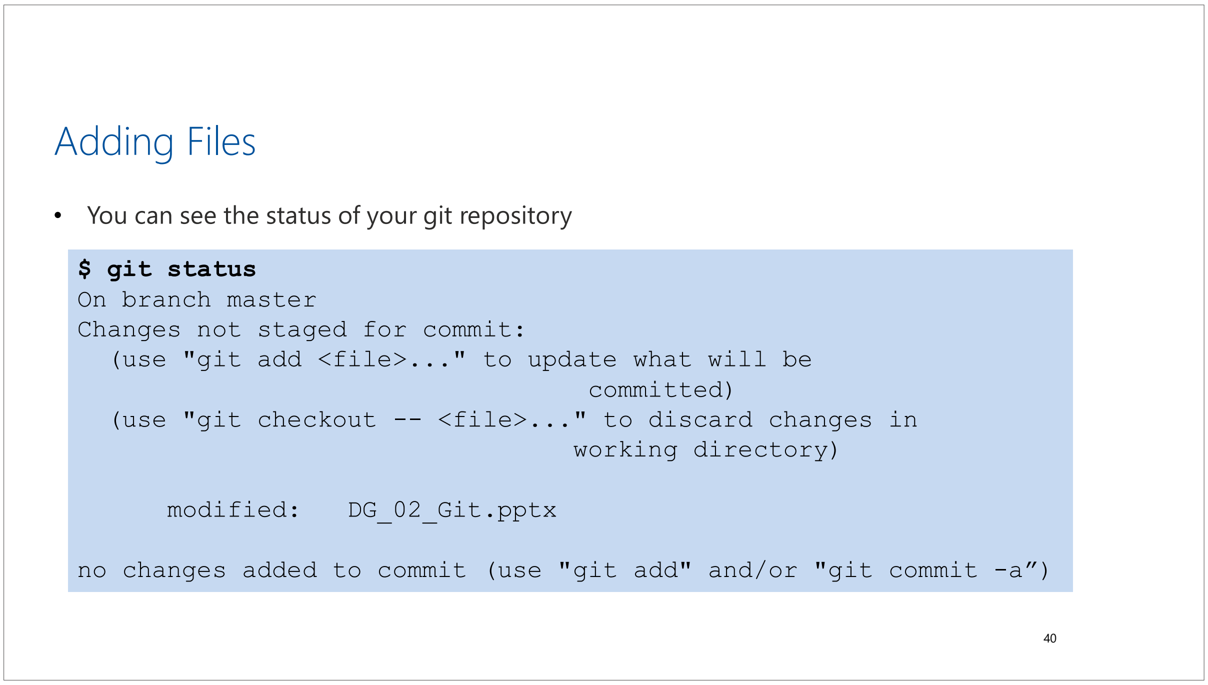
- GIT holds assets in a repository (repo)
  - A repository is a storage area for your files
  - This maps to a directory or folder on your file system
    - These can include subdirectories and associated files

```
$mkdir firstRepo
$cd firstRepo
$git init
```

- The repo requires no server but has created a series of hidden files
  - Located in .git folder

```
$git status
```

39

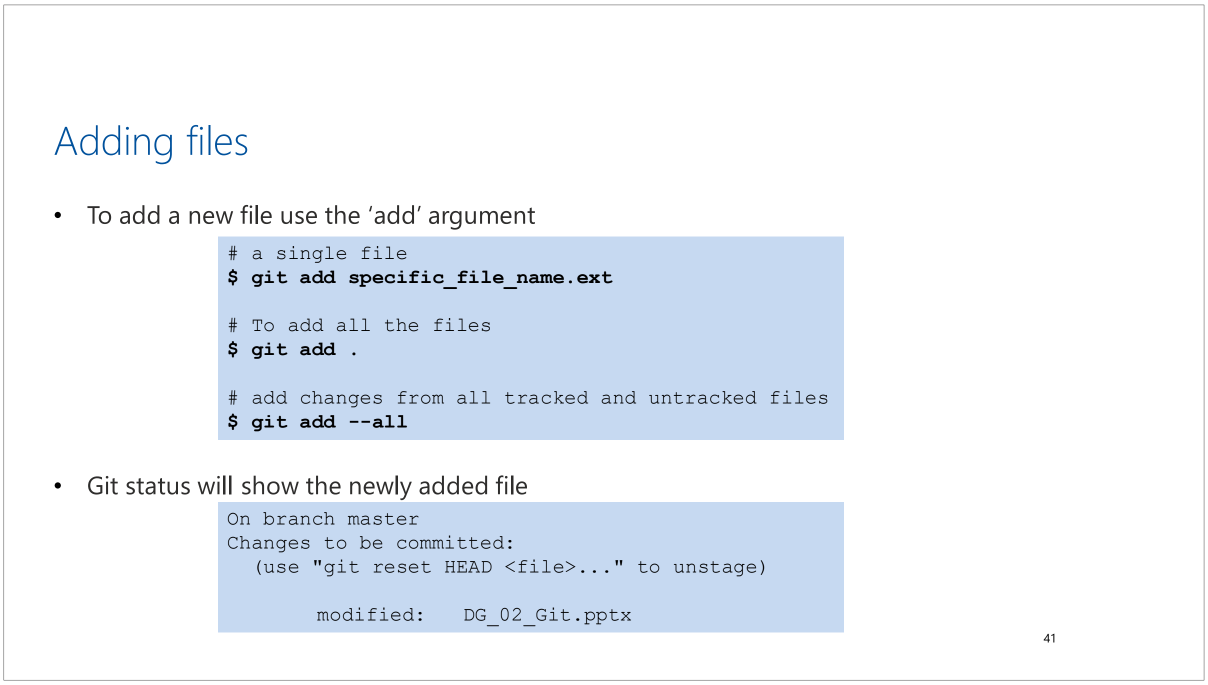


Adding Files

- You can see the status of your git repository

```
$ git status
On branch master
Changes not staged for commit:
   (use "git add <file>..." to update what will be
                                            committed)
   (use "git checkout -- <file>..." to discard changes in
                                           working directory)

        modified:   DG_02_Git.pptx

no changes added to commit (use "git add" and/or "git commit -a”)
```

40



Adding files

- To add a new file use the ‘add’ argument

```
# a single file
$ git add specific_file_name.ext

# To add all the files
$ git add .

# add changes from all tracked and untracked files
$ git add --all
```

- Git status will show the newly added file

```
On branch master
Changes to be committed:
   (use "git reset HEAD <file>..." to unstage)

        modified:   DG_02_Git.pptx
```

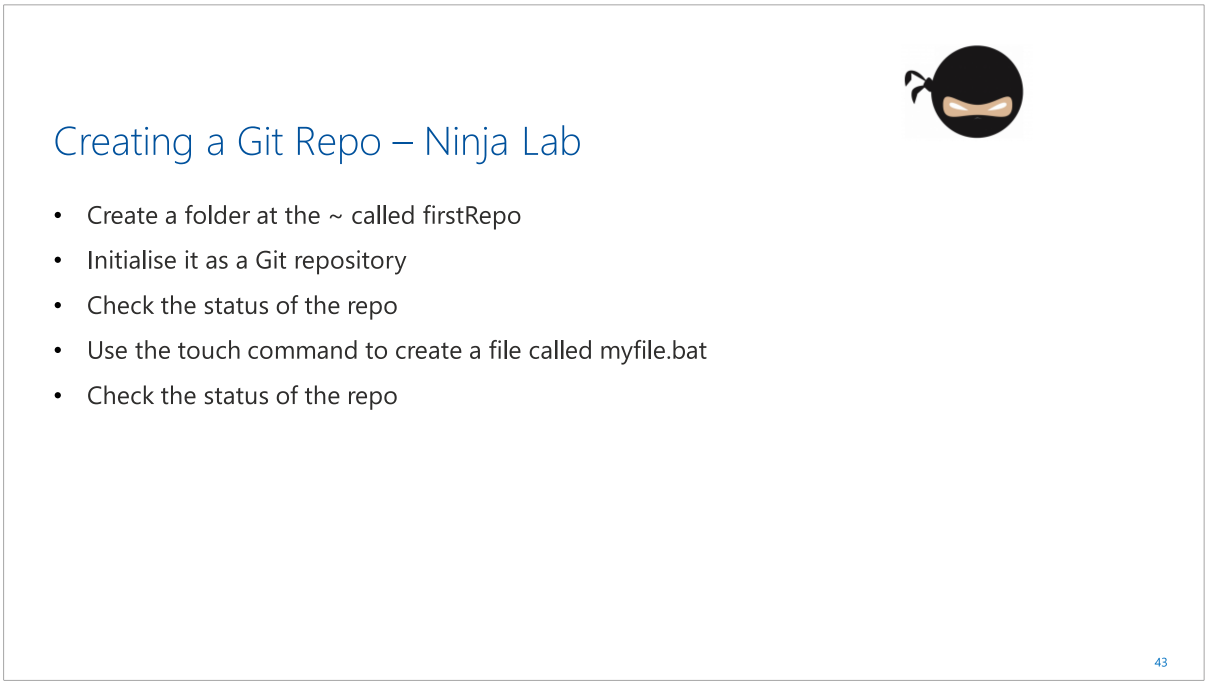
41

Committing files

- To commit the changes, use the “commit” argument
  - You can specify the author with the –a flag
  - The –m flag is used to set a message

```
$ git commit -m "More content for DG02 - git"
[master 93e8300] More content for DG02 - git
 1 file changed, 0 insertions(+), 0 deletions(-)
```

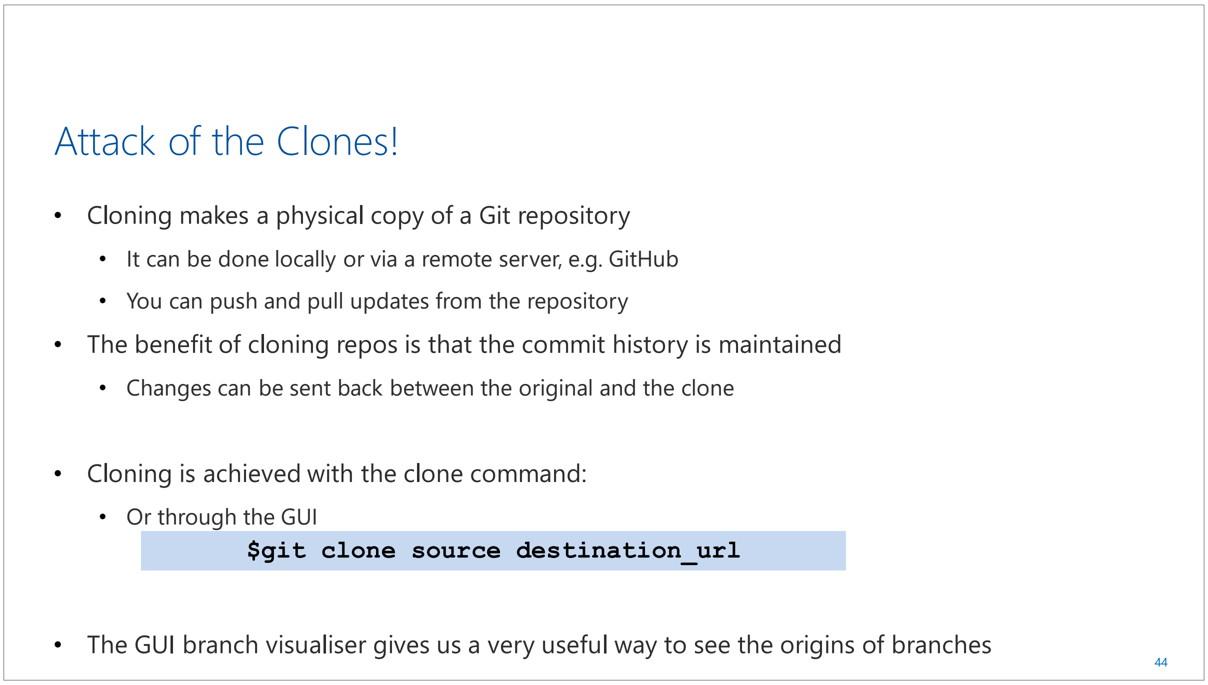
- This is now saved to the local version of your repository



Creating a Git Repo – Ninja Lab

- Create a folder at the ~ called firstRepo
- Initialise it as a Git repository
- Check the status of the repo
- Use the touch command to create a file called myfile.bat
- Check the status of the repo

43



Attack of the Clones!

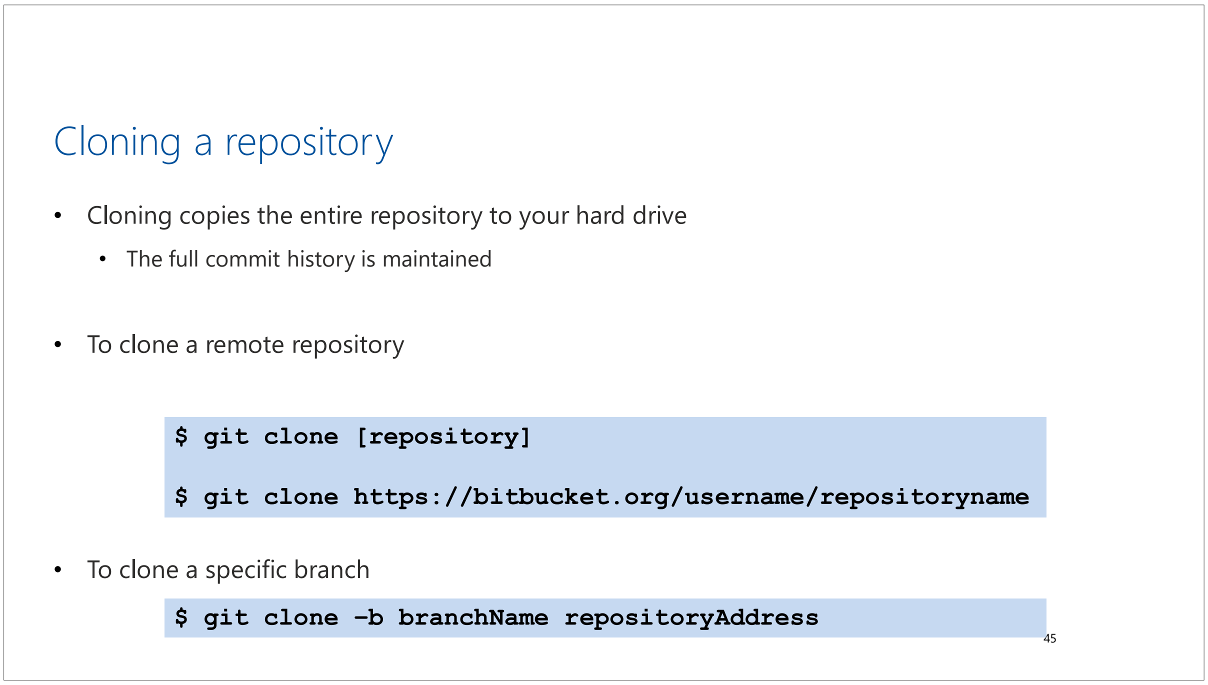
- Cloning makes a physical copy of a Git repository
  - It can be done locally or via a remote server, e.g. GitHub
  - You can push and pull updates from the repository
- The benefit of cloning repos is that the commit history is maintained
  - Changes can be sent back between the original and the clone

- Cloning is achieved with the clone command:
  - Or through the GUI

```
$git clone source destination_url
```

- The GUI branch visualiser gives us a very useful way to see the origins of branches

44



Cloning a repository

- Cloning copies the entire repository to your hard drive
  - The full commit history is maintained

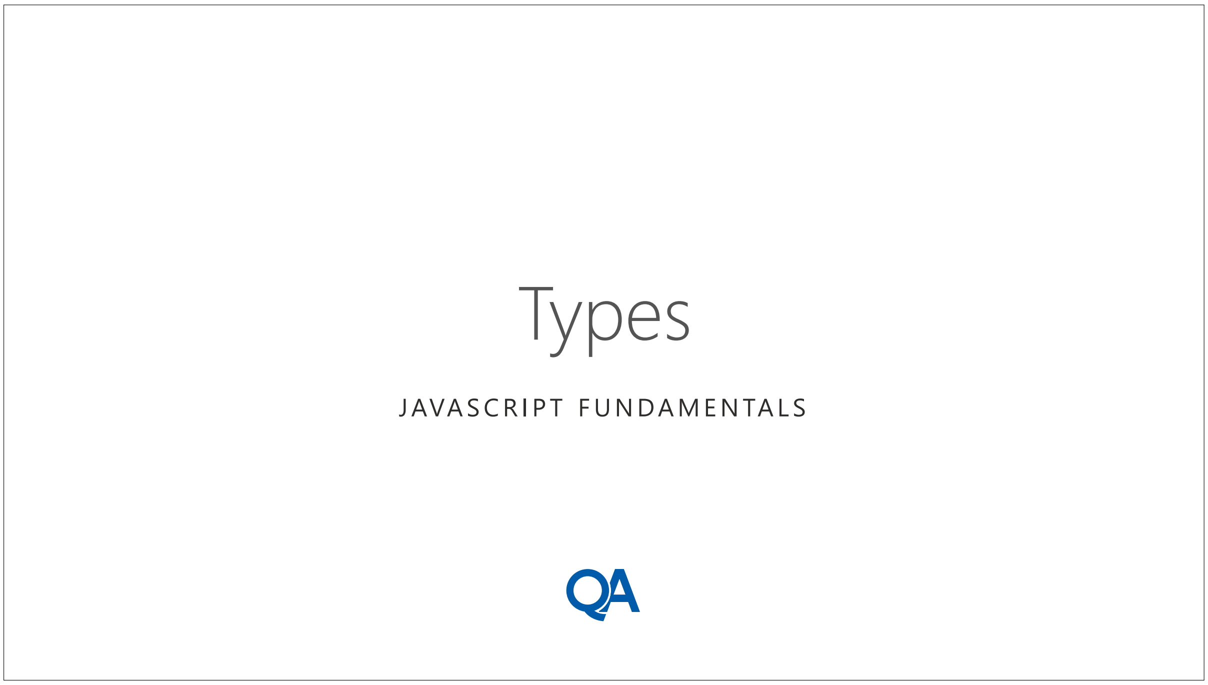
- To clone a remote repository

```
$ git clone [repository]

$ git clone https://bitbucket.org/username/repositoryname
```

- To clone a specific branch

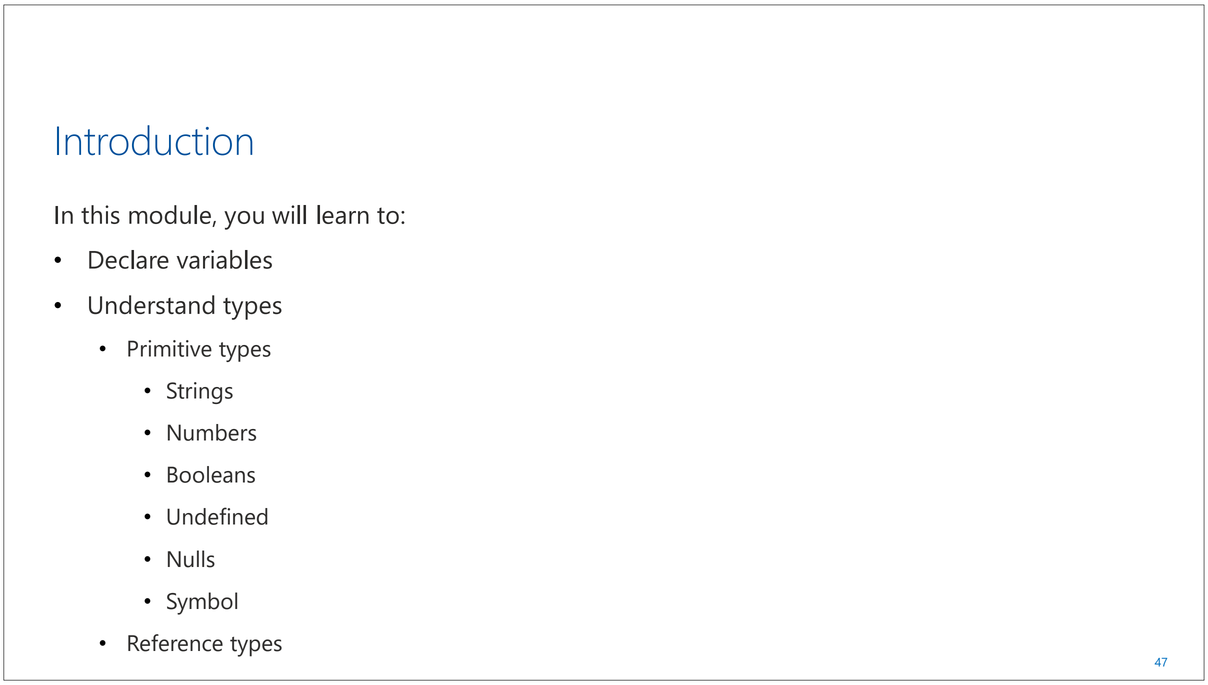
```
$ git clone –b branchName repositoryAddress
```

45



Types

JAVASCRIPT FUNDAMENTALS

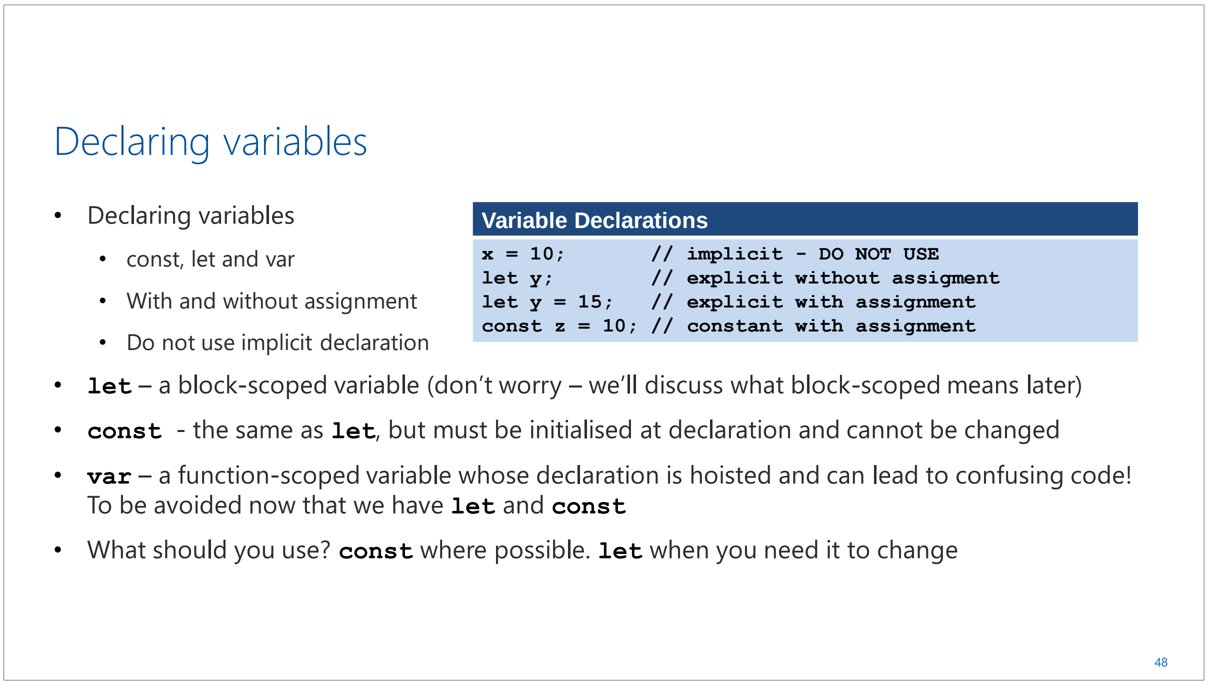


Introduction

In this module, you will learn to:

- Declare variables
- Understand types
    - Primitive types
        - Strings
        - Numbers
        - Booleans
        - Undefined
        - Nulls
        - Symbol
    - Reference types

47



Declaring variables

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

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
```

- **`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

48

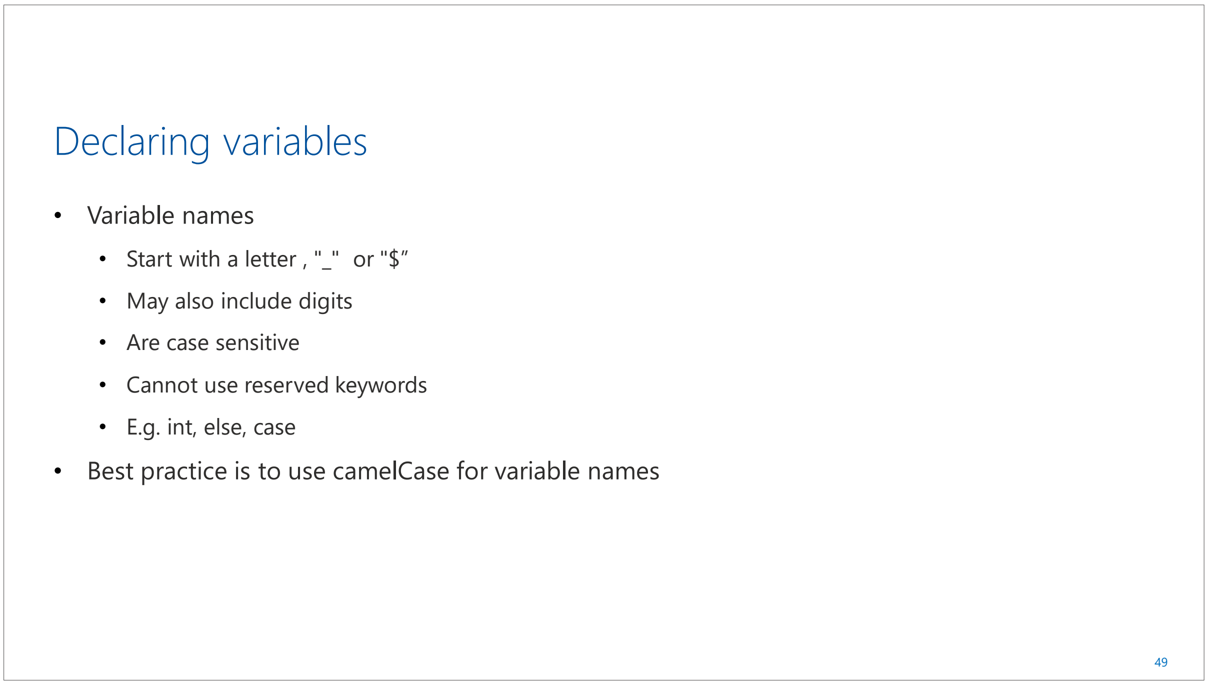
Variables in JavaScript should be declared with the **var** operator followed by the variable name. If you look above, you will see the variable **x** does not follow this rule. It is valid, just terribly bad practice and has an impact on variable scope as we will examine later.

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

49

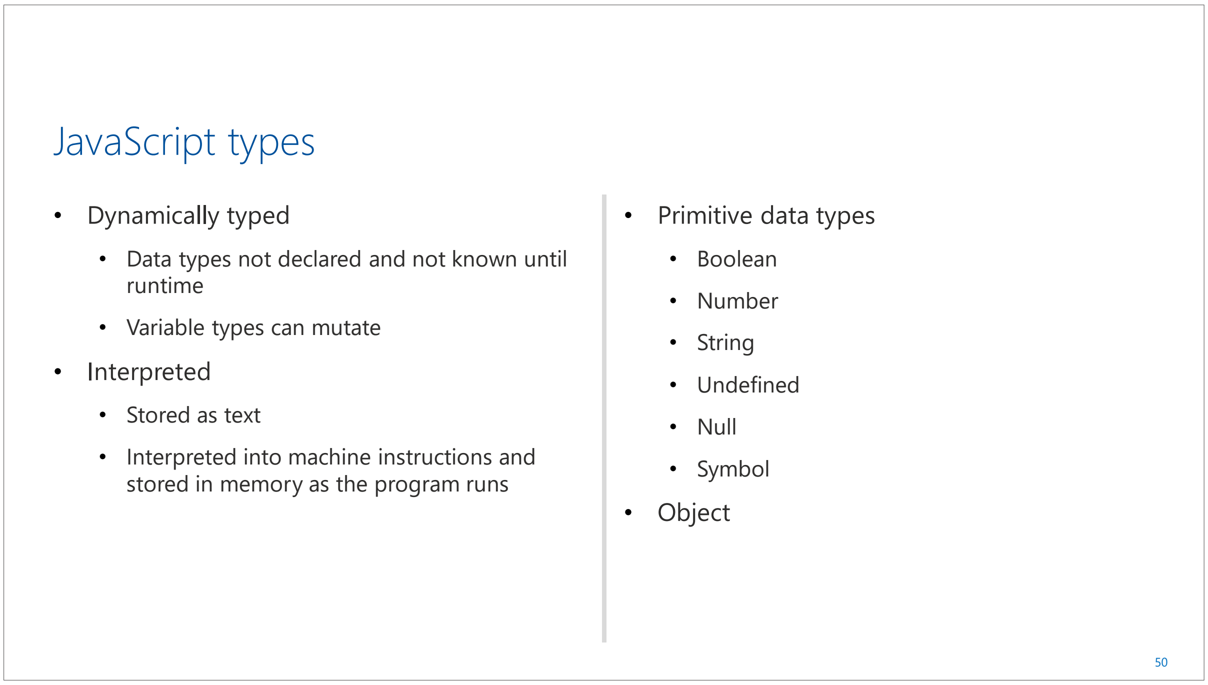
Variables in JavaScript should be declared with the **var** operator followed by the variable name. If you look above, you will see the variable **x** does not follow this rule. It is valid, just terribly bad practice and has an impact on variable scope as we will examine later.

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

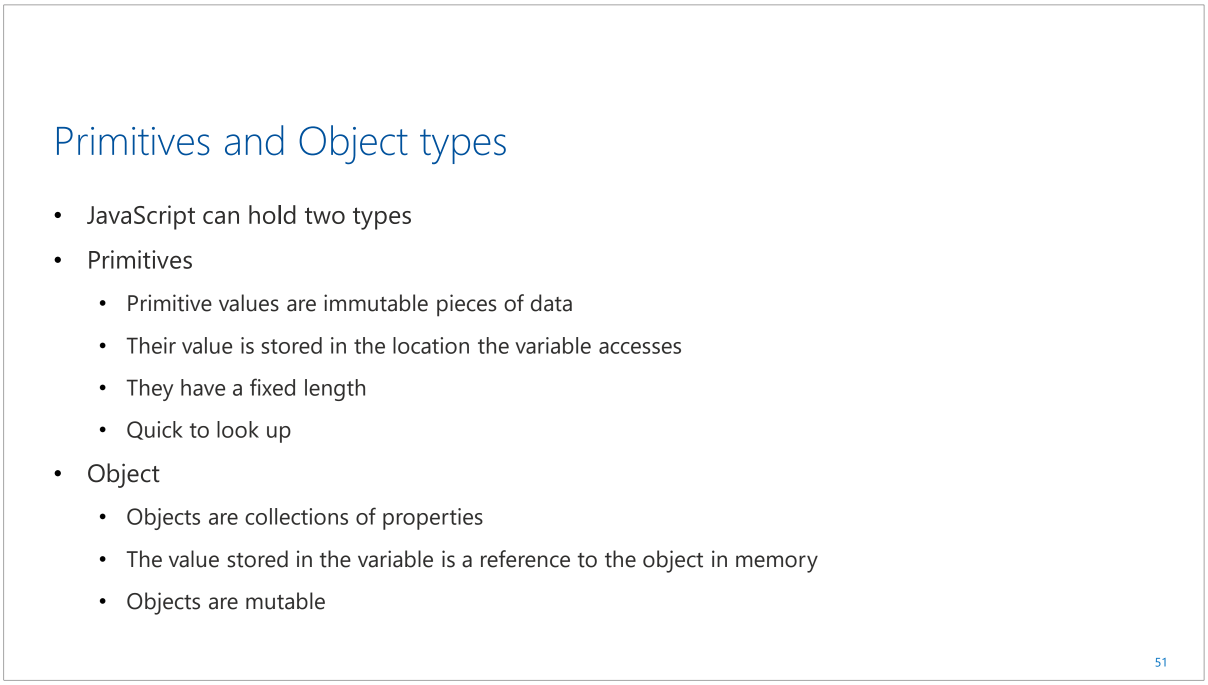
- Dynamically typed
  - Data types not declared and not known until runtime
  - Variable types can mutate
- Interpreted
  - Stored as text
  - Interpreted into machine instructions and stored in memory as the program runs

- Primitive data types
  - Boolean
  - Number
  - String
  - Undefined
  - Null
  - Symbol
- Object

50

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

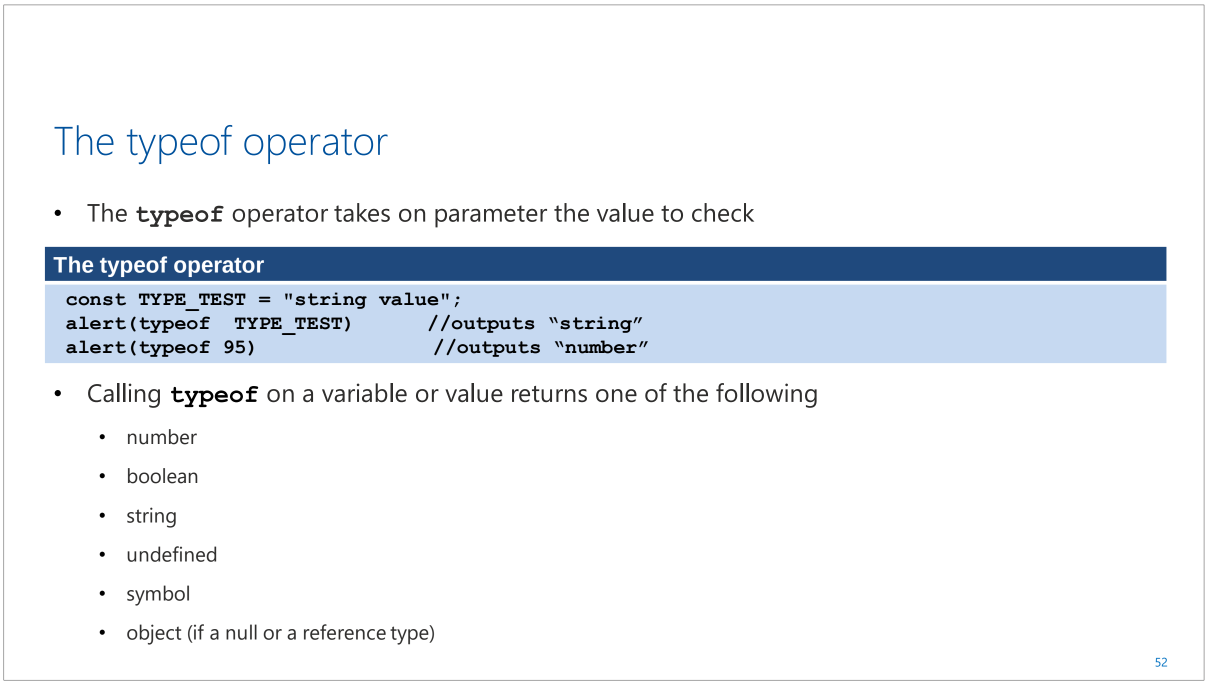
51

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

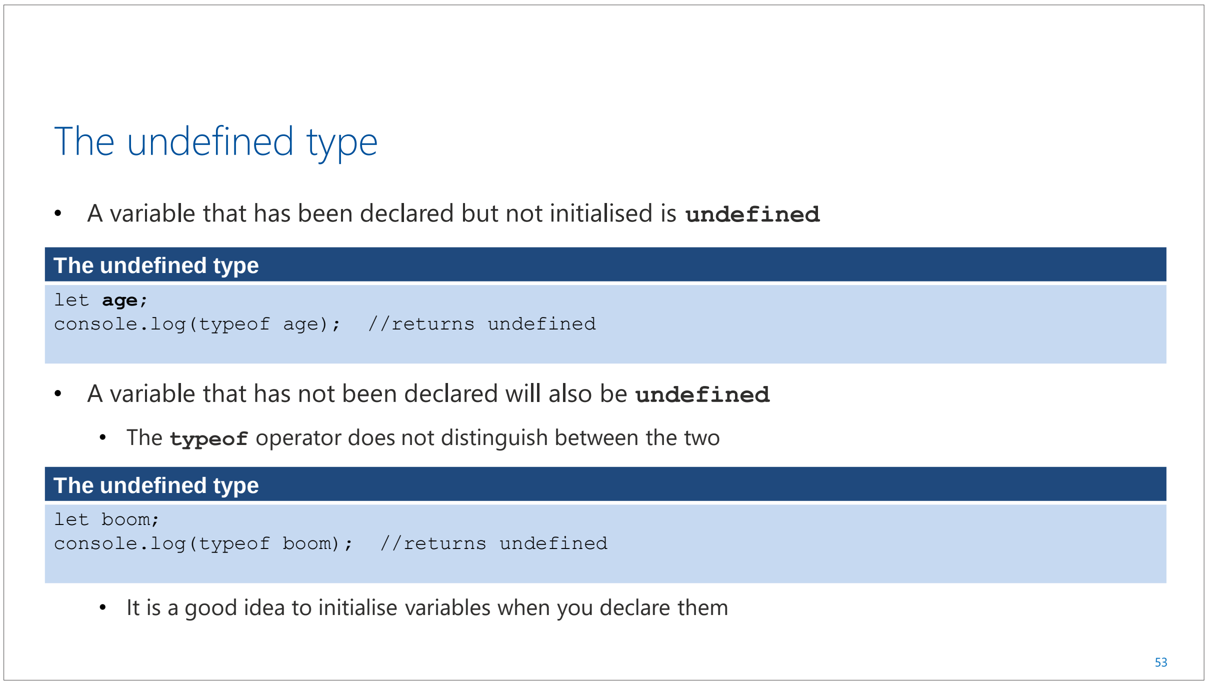
The typeof operator

```
const TYPE_TEST = "string value";
alert(typeof  TYPE_TEST)      //outputs "string"
alert(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)

52

`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

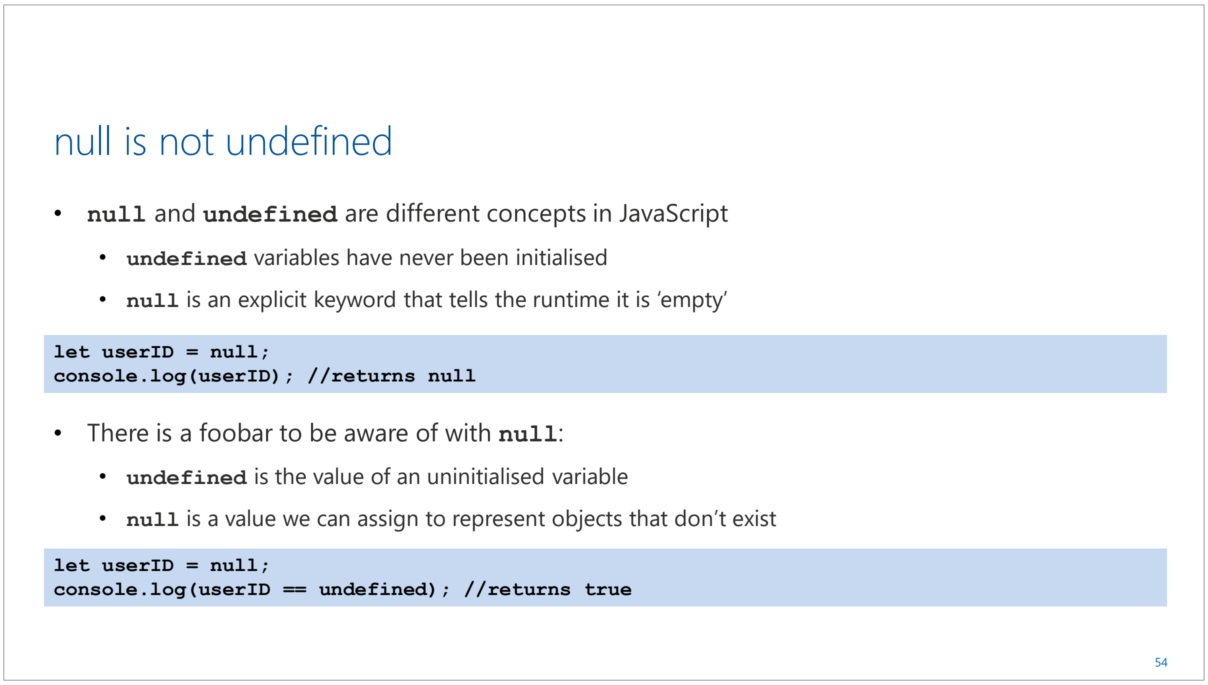
The undefined type

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

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

53

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'

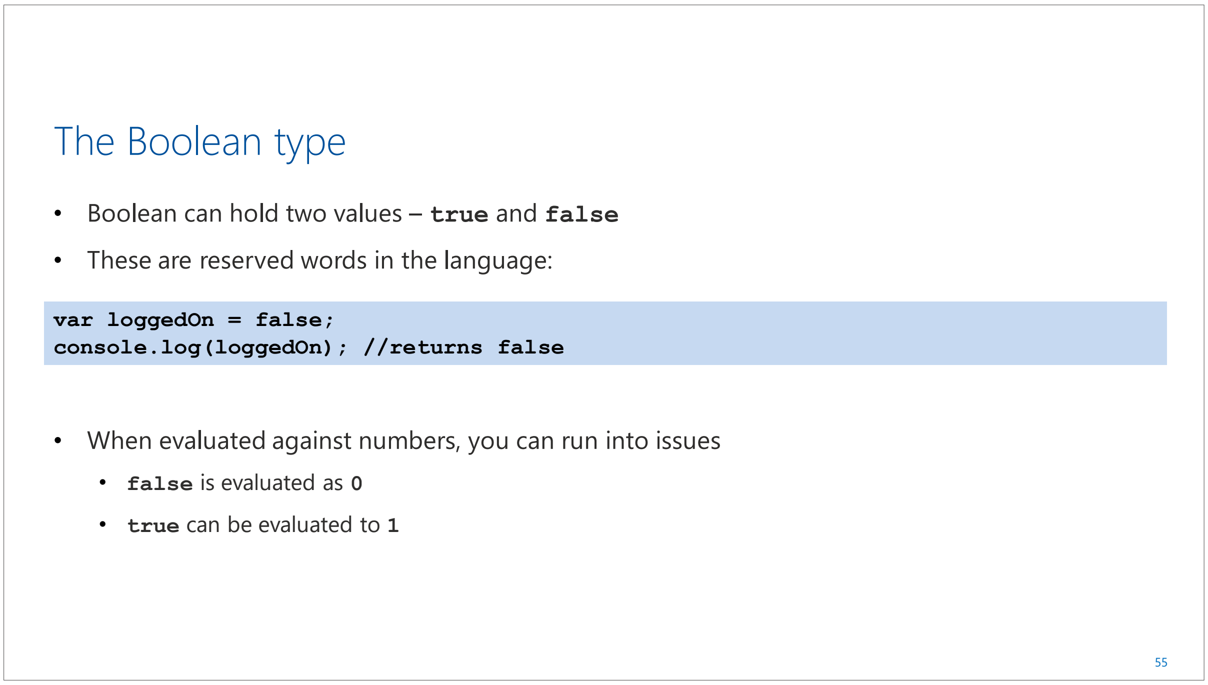
```
let userID = null;
console.log(userID); //returns null
```

- 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
```

54

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.



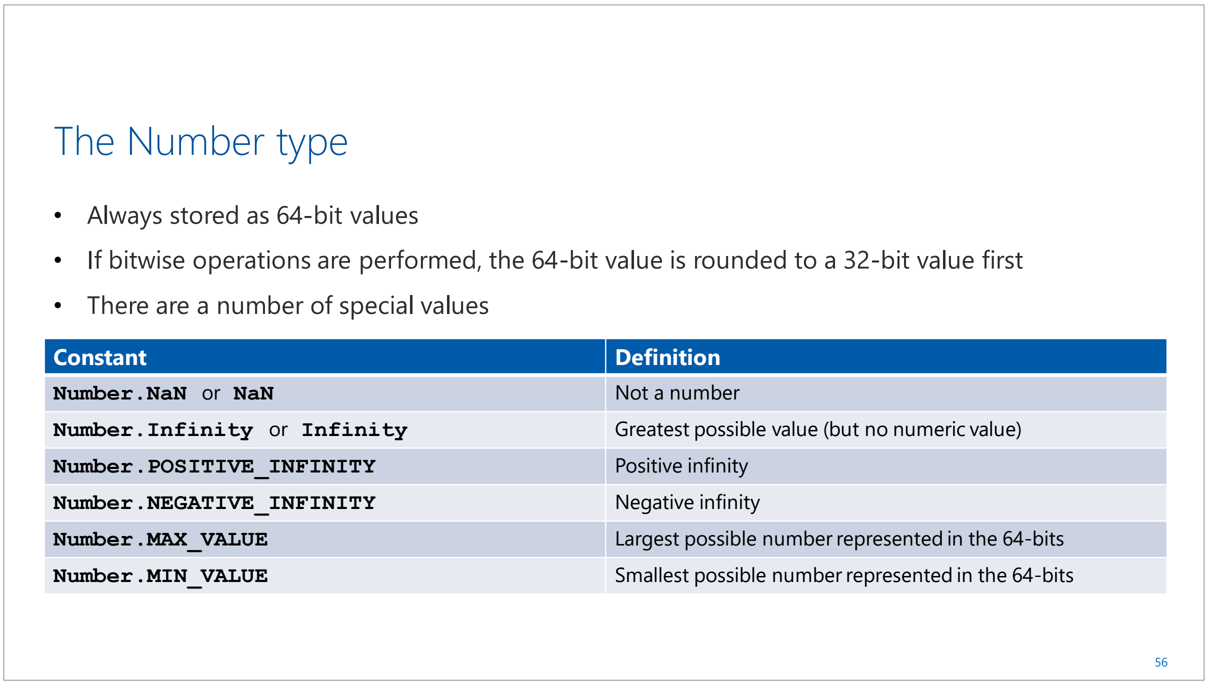
The Boolean type

- 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`**

55



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

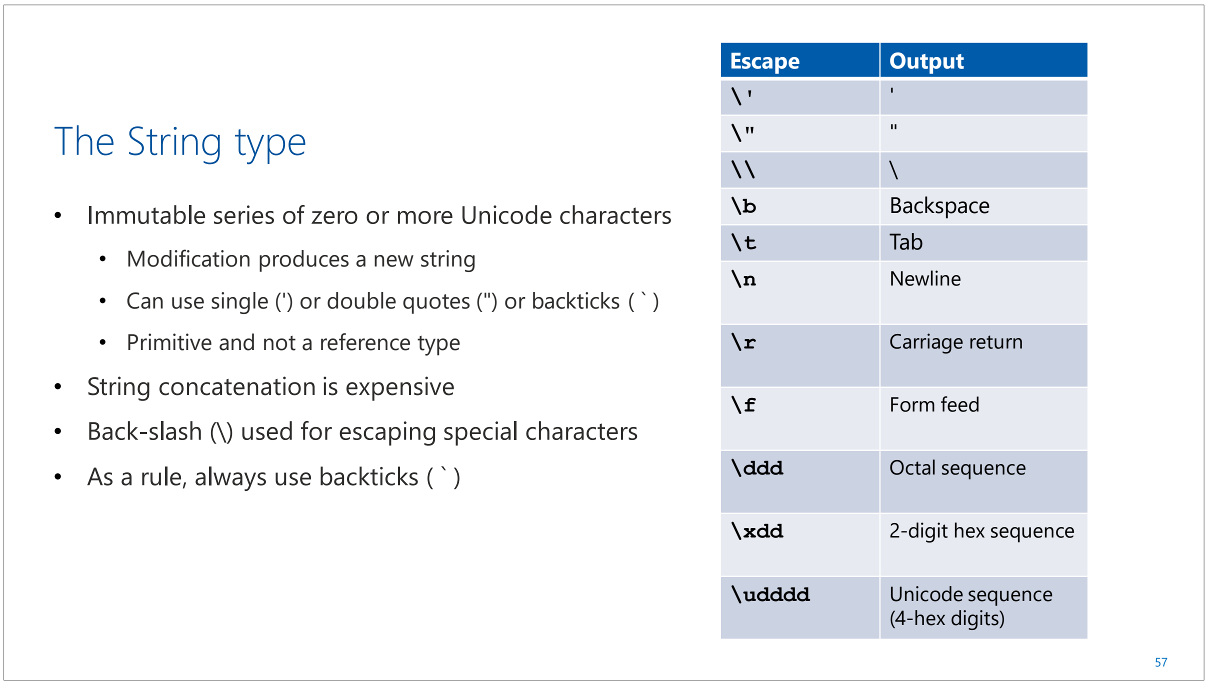
Constant	Definition
`Number.NaN` or `NaN`	Not a number
`Number.Infinity` or `Infinity`	Greatest possible value (but no numeric value)
`Number.POSITIVE_INFINITY`	Positive infinity
`Number.NEGATIVE_INFINITY`	Negative infinity
`Number.MAX_VALUE`	Largest possible number represented in the 64-bits
`Number.MIN_VALUE`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

56

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 noted below:

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 (`)
  - 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)

57

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

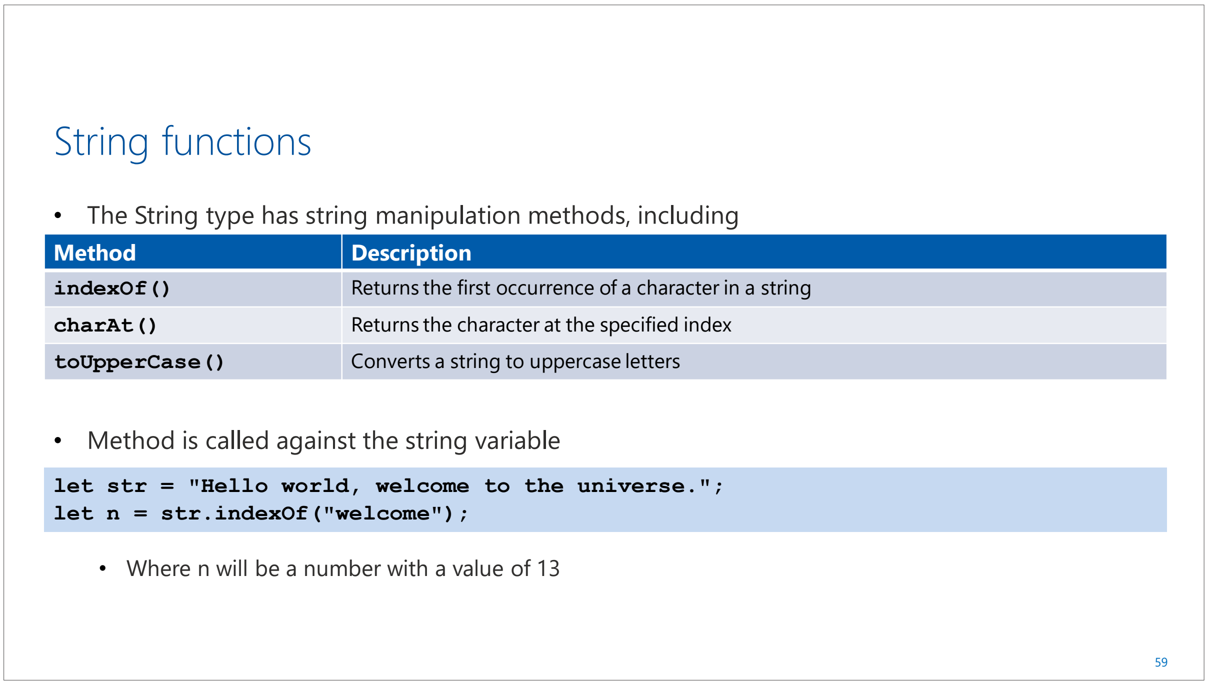
- Adding 2 (or more strings) is an expensive operation due to the memory manipulation required
- To concatenate a string the + operator is used

```
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
```

58



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

- Method is called against the string variable

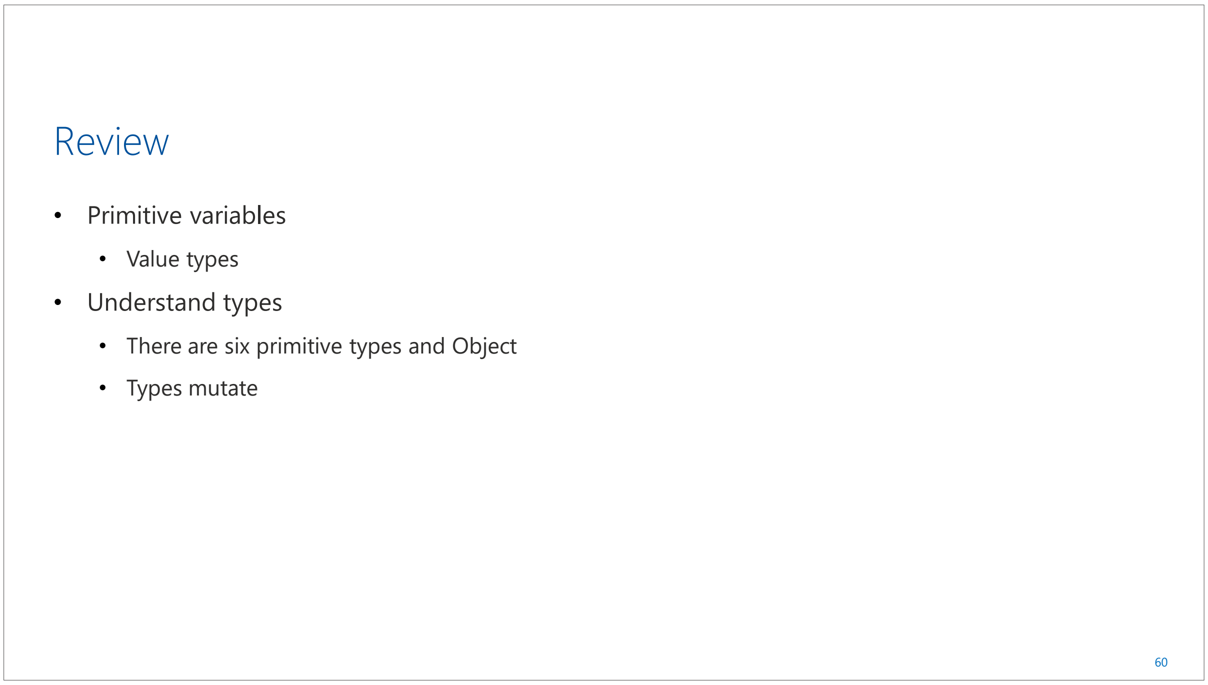
```
let str = "Hello world, welcome to the universe.";
let n = str.indexOf("welcome");
```

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

59

Further methods:

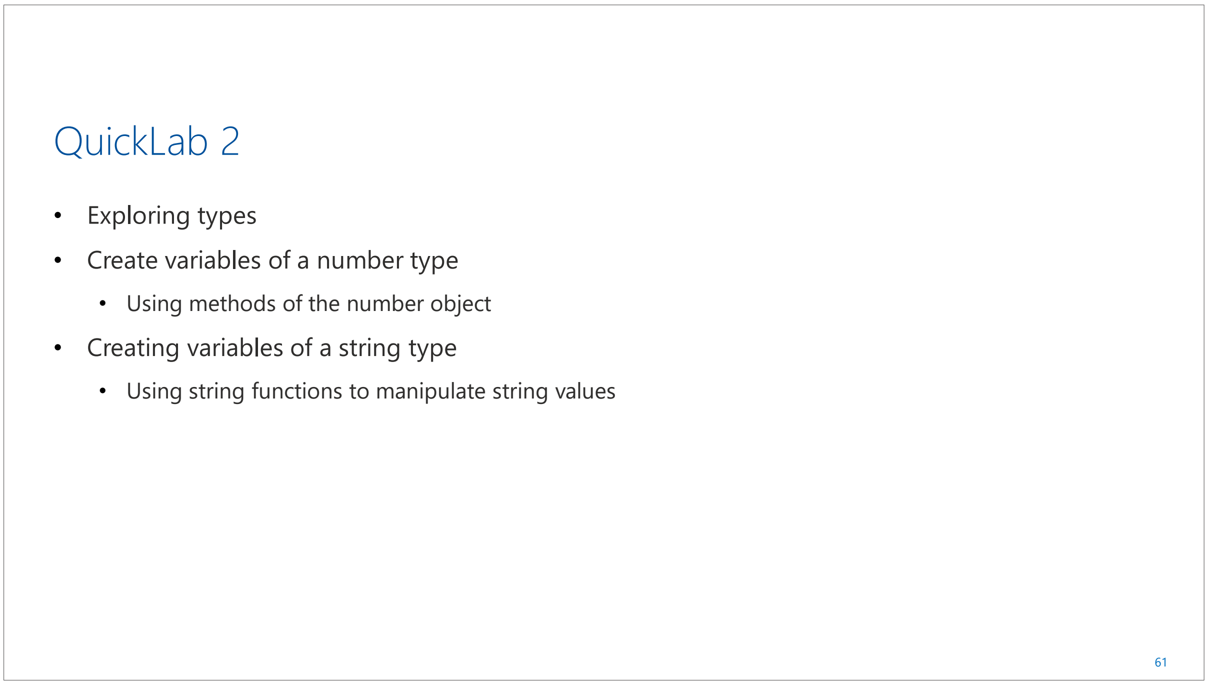
Method	Description
`charCodeAt()`	Returns the unicode of the character at the specified index
`concat()`	Joins strings and returns a copy of the joined string
`fromCharCode()`	Converts unicode values to characters
`lastIndexOf()`	Returns the position of last found occurrence from a string
`slice()`	Extracts 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 and Object
  - Types mutate

60



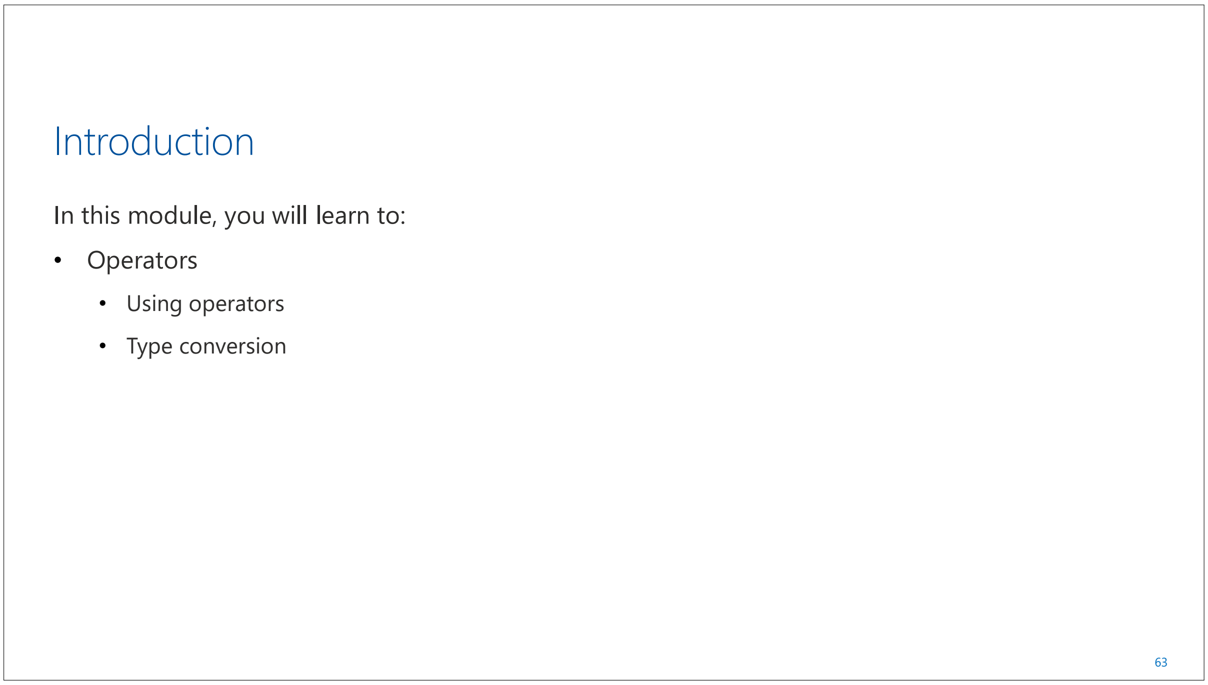
QuickLab 2

- Exploring types
- Create variables of a number type
  - Using methods of the number object
- Creating variables of a string type
  - Using string functions to manipulate string values

61

Operators

JAVASCRIPT FUNDAMENTALS

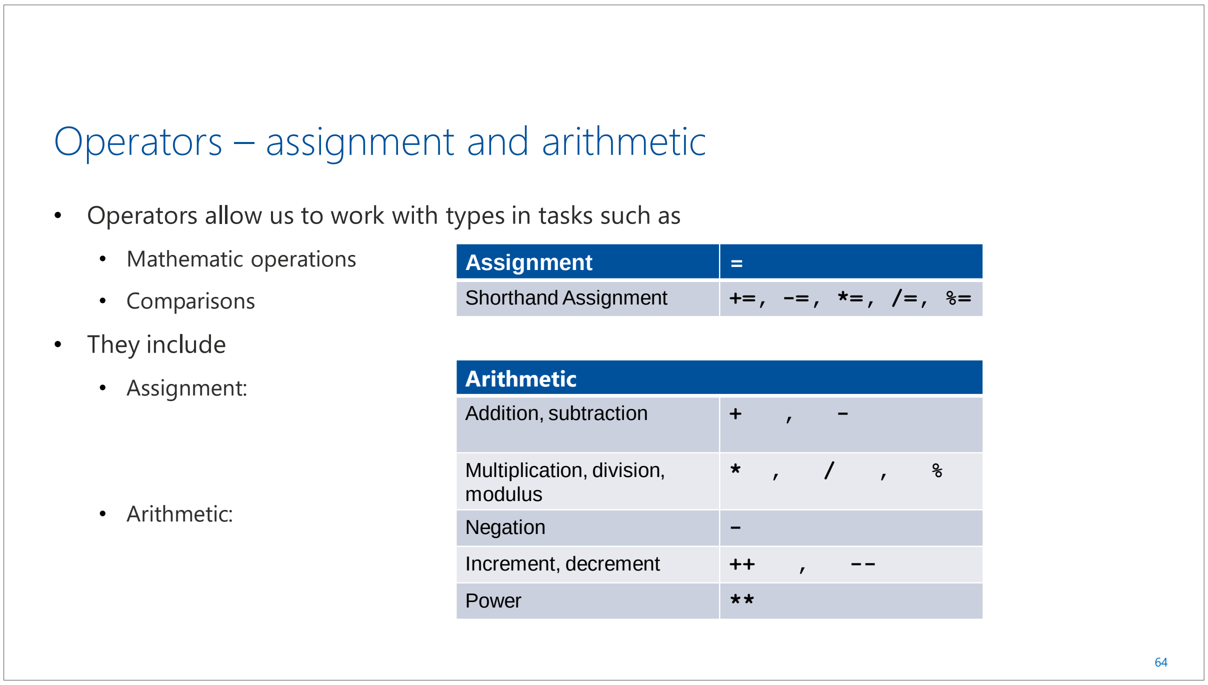


Introduction

In this module, you will learn to:

- Operators
  - Using operators
  - Type conversion

63



Operators – assignment and arithmetic

- Operators allow us to work with types in tasks such as
  - Mathematic operations
  - Comparisons
- They include
  - Assignment:
  - Arithmetic:

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

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

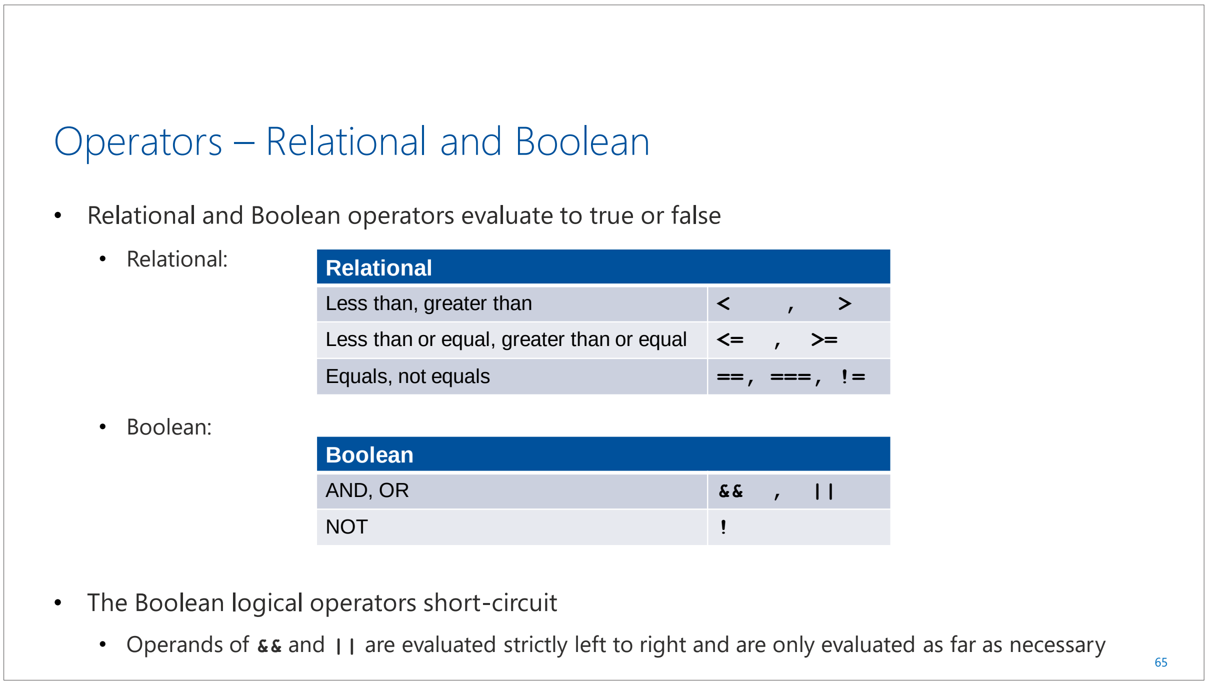
64

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	`!`

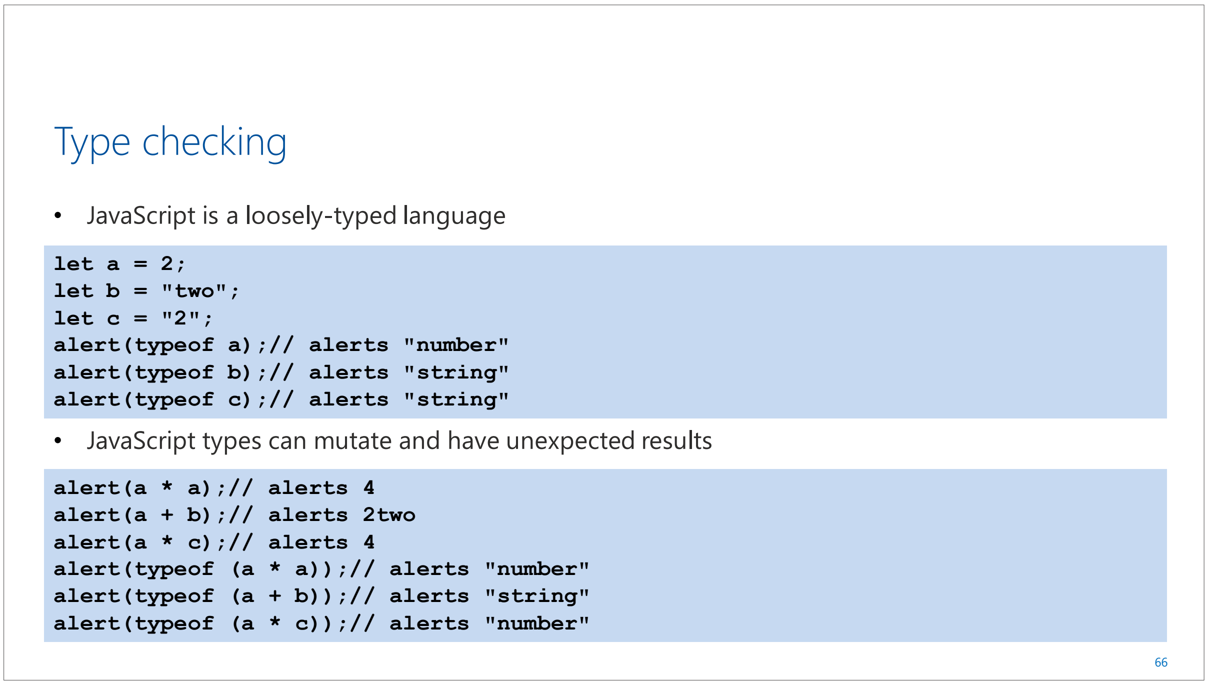
- The Boolean logical operators short-circuit
  - Operands of `&&` and `||` are evaluated strictly left to right and are only evaluated as far as necessary

65

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";
alert(typeof a);// alerts "number"
alert(typeof b);// alerts "string"
alert(typeof c);// alerts "string"
```

- JavaScript types can mutate and have unexpected results

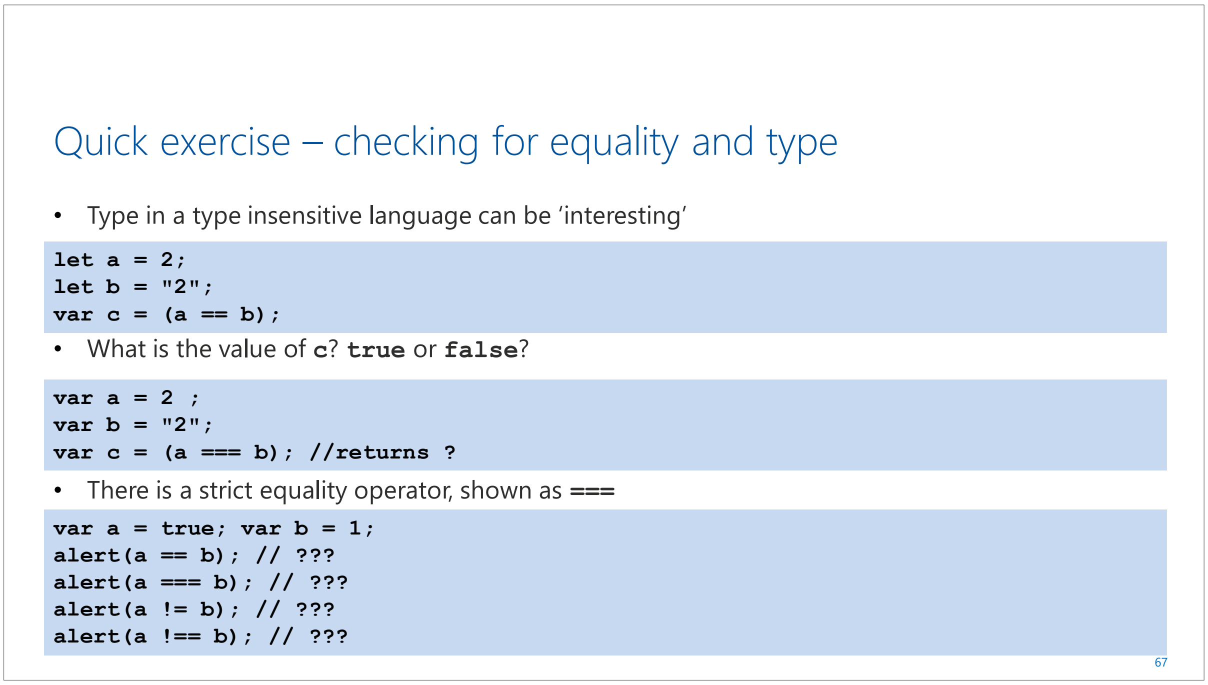
```
alert(a * a);// alerts 4
alert(a + b);// alerts 2two
alert(a * c);// alerts 4
alert(typeof (a * a));// alerts "number"
alert(typeof (a + b));// alerts "string"
alert(typeof (a * c));// alerts "number"
```

66

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";
var 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); // ???
```

67

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.

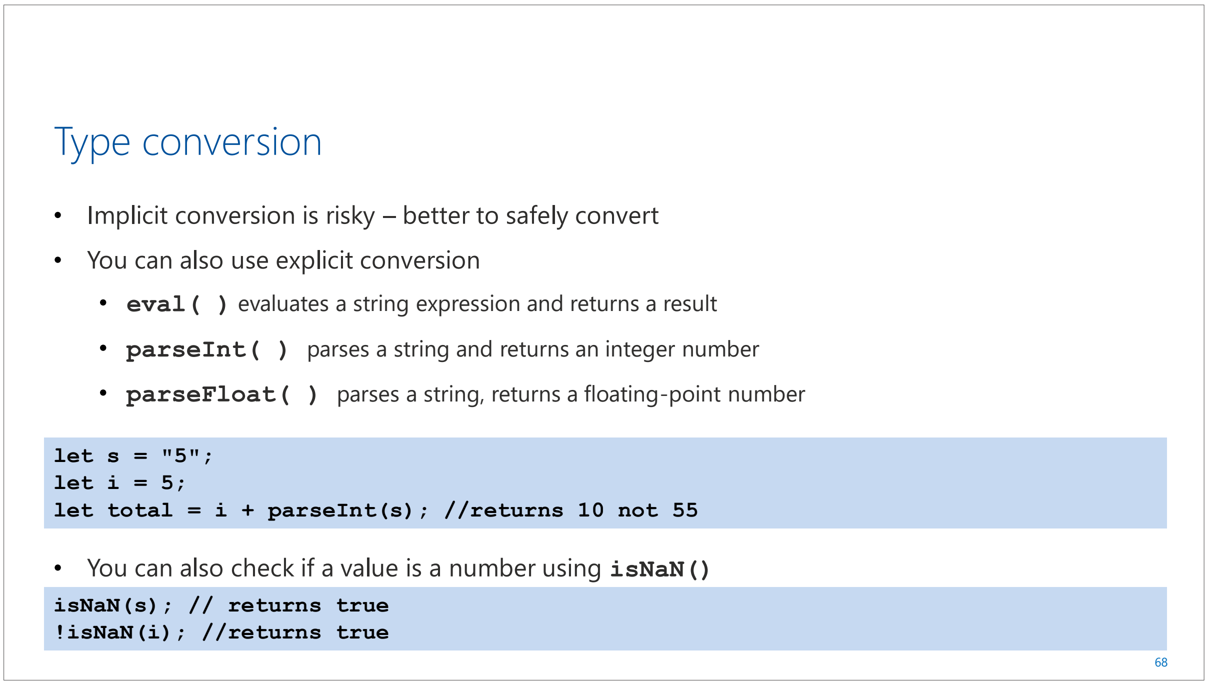
In JavaScript 1.3, the situation became even less simple, with the introduction of one further operator: the strict equality operator, shown as ===.

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
```

68

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.

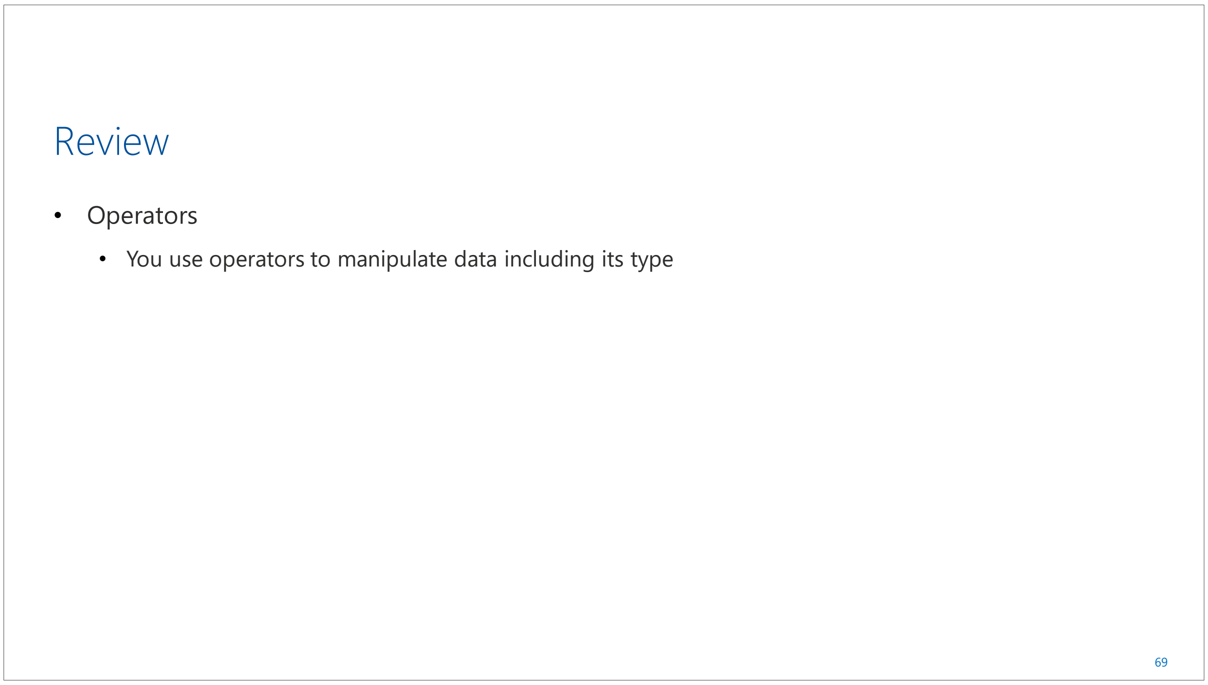
`parseInt()` – takes a value or variable that is not currently a number and tries to convert its value into a numeric type. Specifically, it is looking for numeric values and any decimal points. It preserves anything to the left of the decimal point, so:

> *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

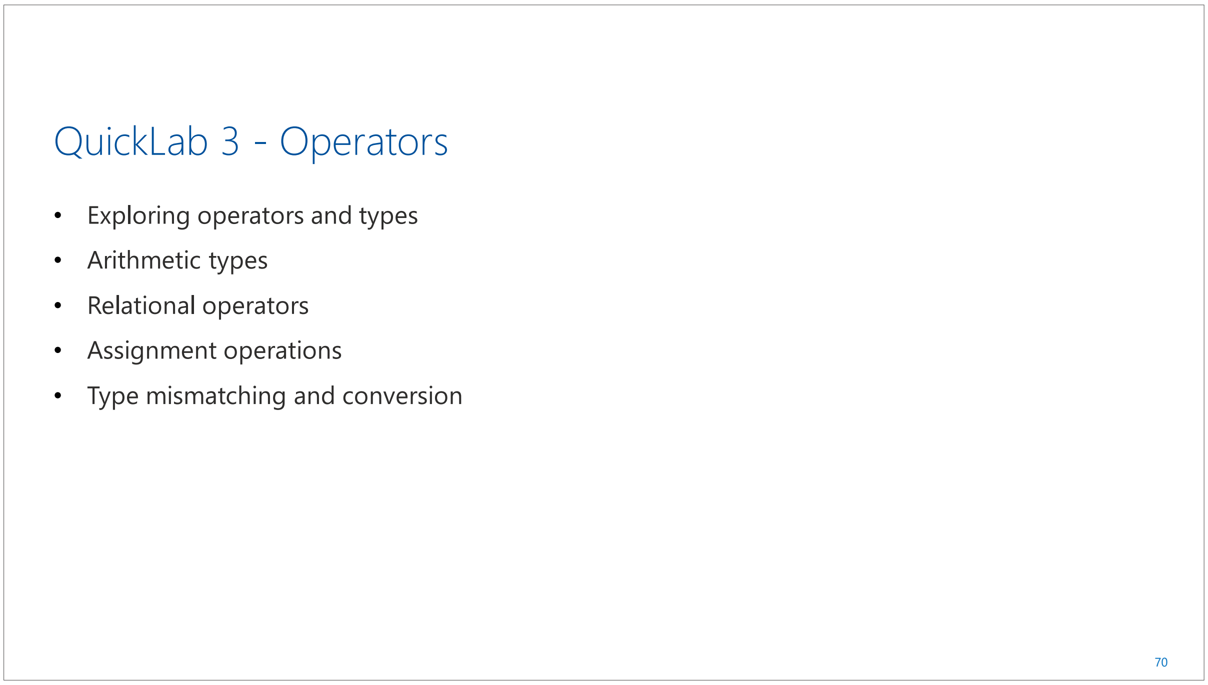
Both **parseInt** and **parseFloat** return a NaN error object if the conversion can not occur, which you can detect using the **isNan()** function.



Review

- Operators
  - You use operators to manipulate data including its type

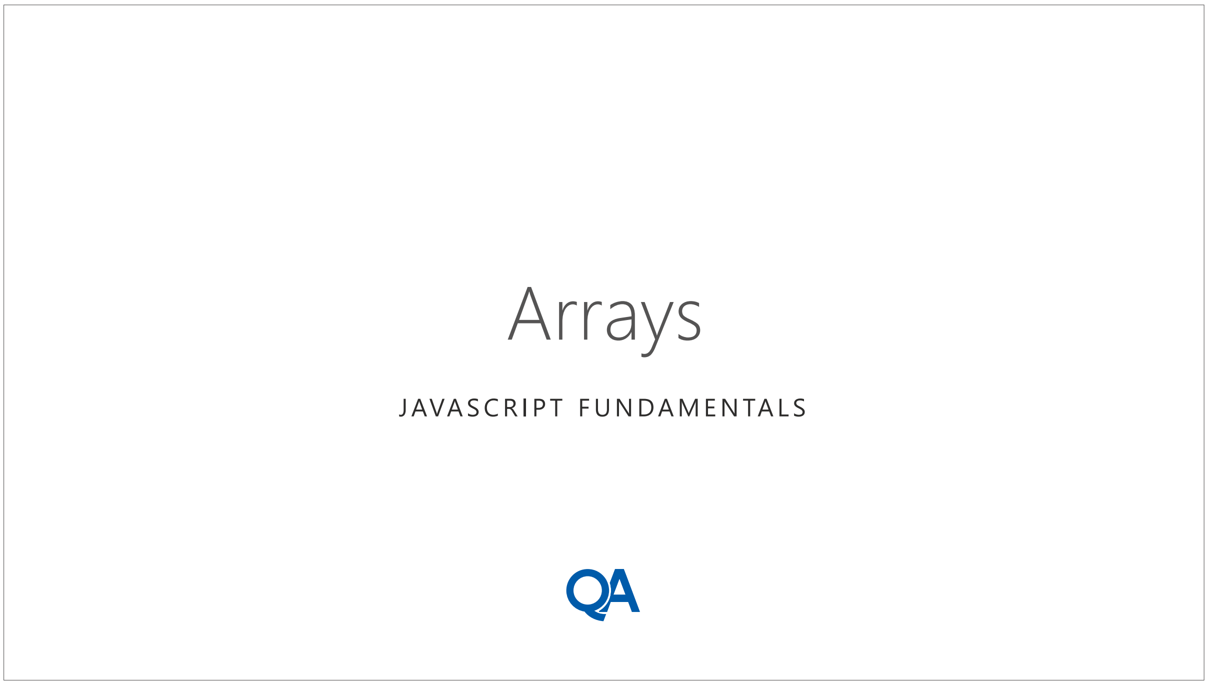
69



QuickLab 3 - Operators

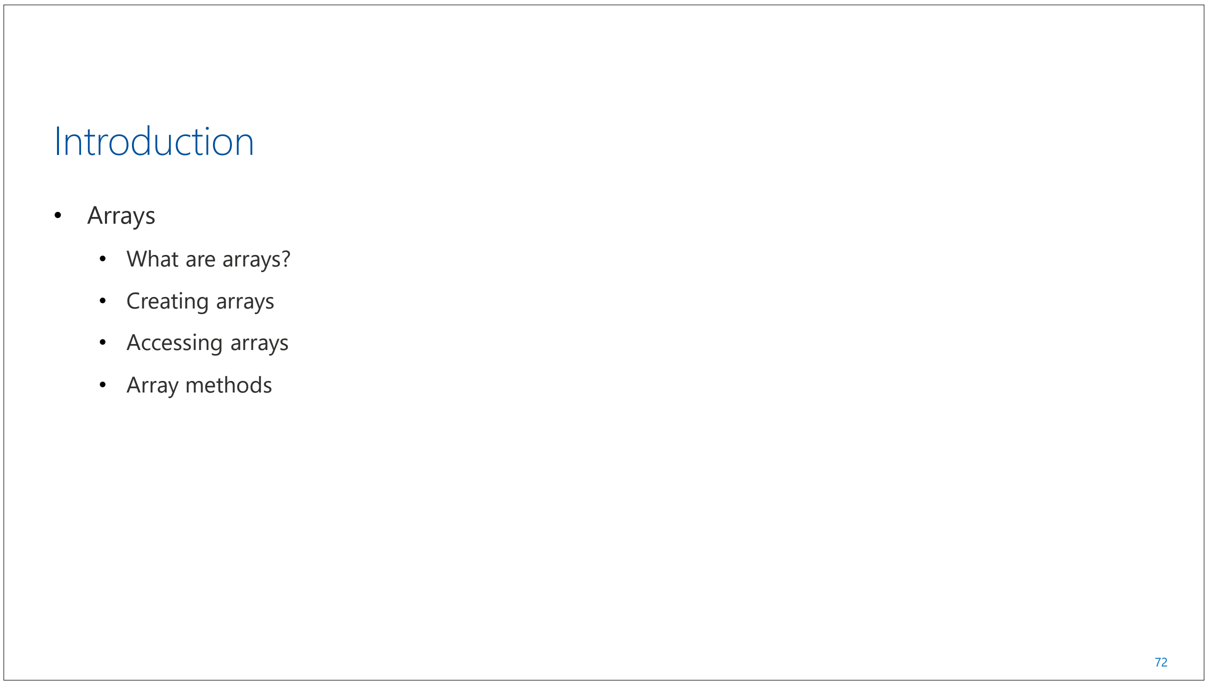
- Exploring operators and types
- Arithmetic types
- Relational operators
- Assignment operations
- Type mismatching and conversion

70



Arrays

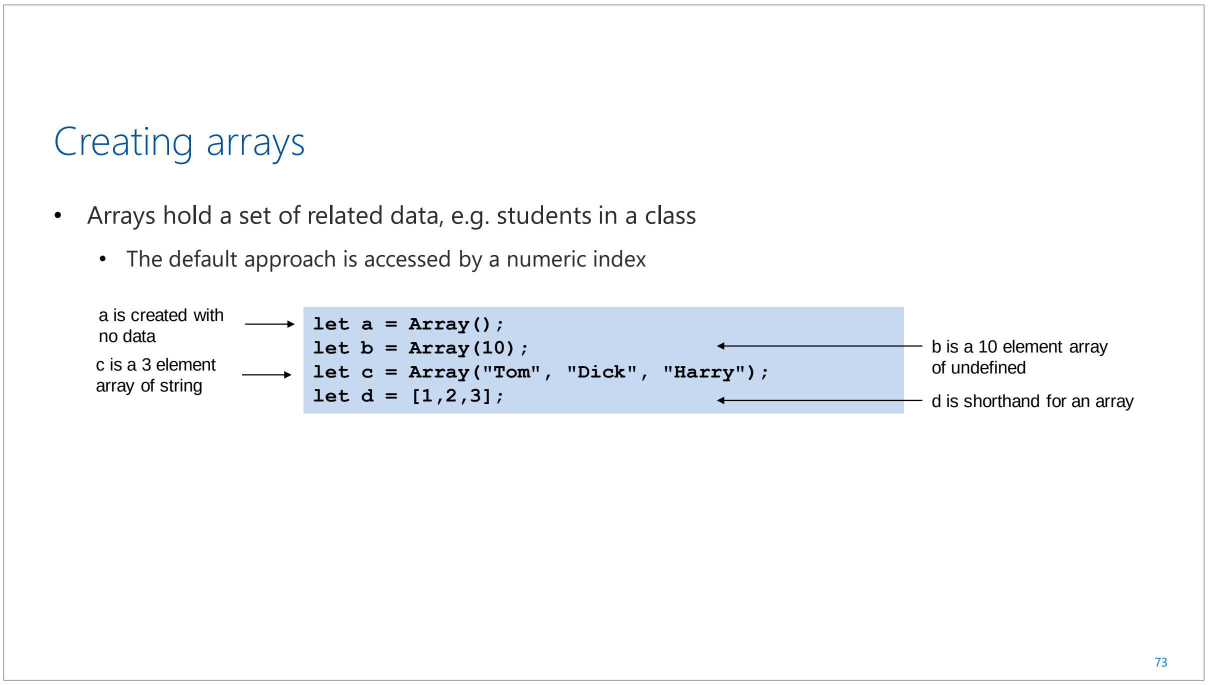
JAVASCRIPT FUNDAMENTALS



Introduction

- Arrays
  - What are arrays?
  - Creating arrays
  - Accessing arrays
  - Array methods

72



Creating arrays

- Arrays hold a set of related data, e.g. students in a class
  - The default approach is accessed by a numeric index

```
let a = Array();
let b = Array(10);
let c = Array("Tom", "Dick", "Harry");
let d = [1,2,3];
```

a is created with no data
c is a 3 element array of string
b is a 10 element array of undefined
d is shorthand for an array
73

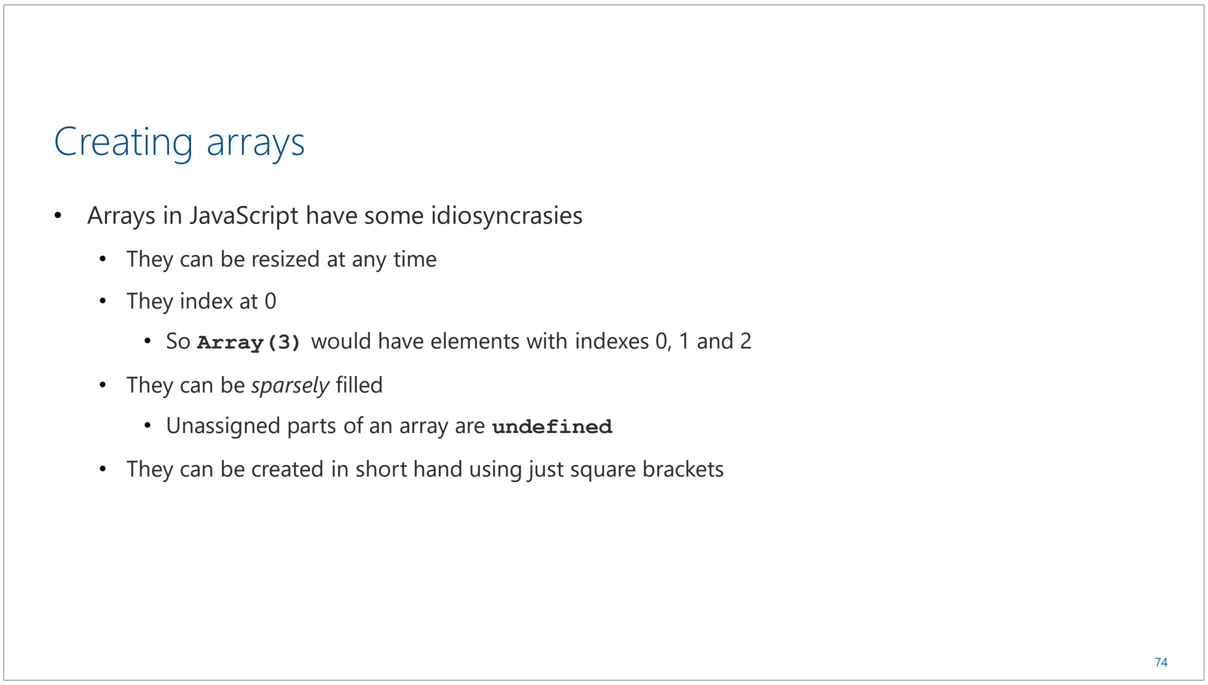
As with the rest of JavaScript, the elements in arrays are loosely-typed. This means that each separate element of an array can contain a different type. This is useful and dangerous at once and brings us back to the consideration that type matters.

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

74

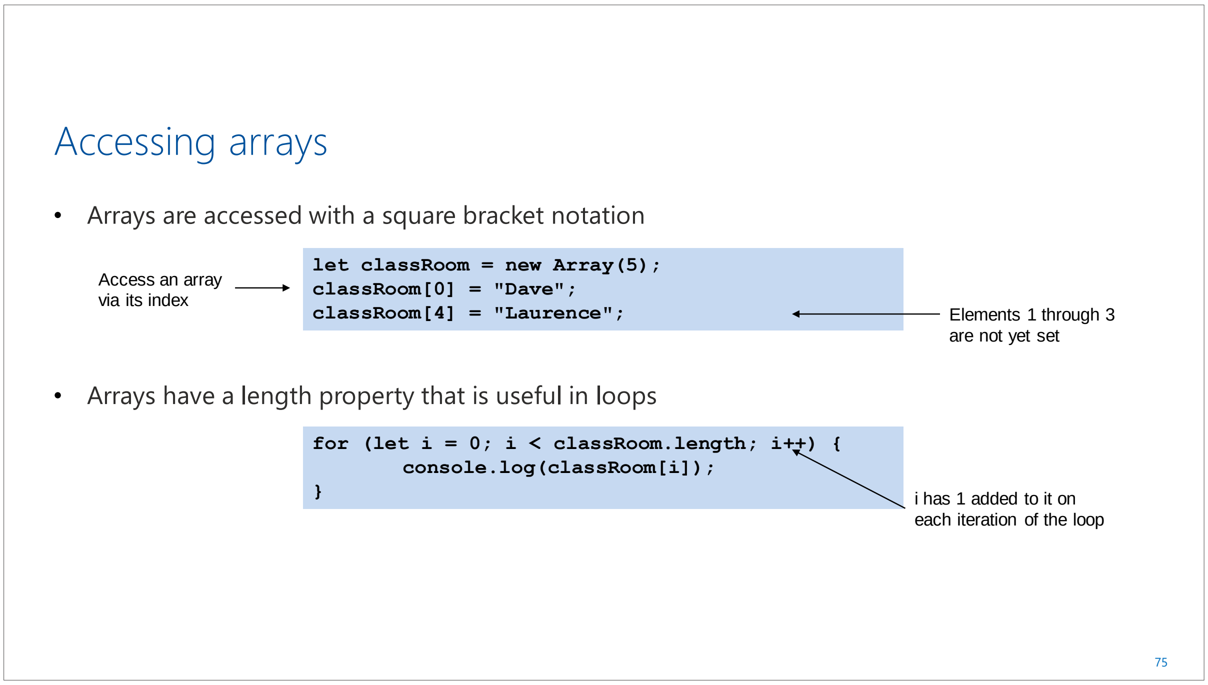
As with the rest of JavaScript, the elements in arrays are loosely-typed. This means that each separate element of an array can contain a different type. This is useful and dangerous at once and brings us back to the consideration that type matters.

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.



Accessing arrays

- Arrays are accessed with a square bracket notation

```
let classRoom = new Array(5);
classRoom[0] = "Dave";
classRoom[4] = "Laurence";
```

Access an array via its index
Elements 1 through 3 are not yet set

- Arrays have a length property that is useful in loops

```
for (let i = 0; i < classRoom.length; i++) {
        console.log(classRoom[i]);
}
```

i has 1 added to it on each iteration of the loop
75

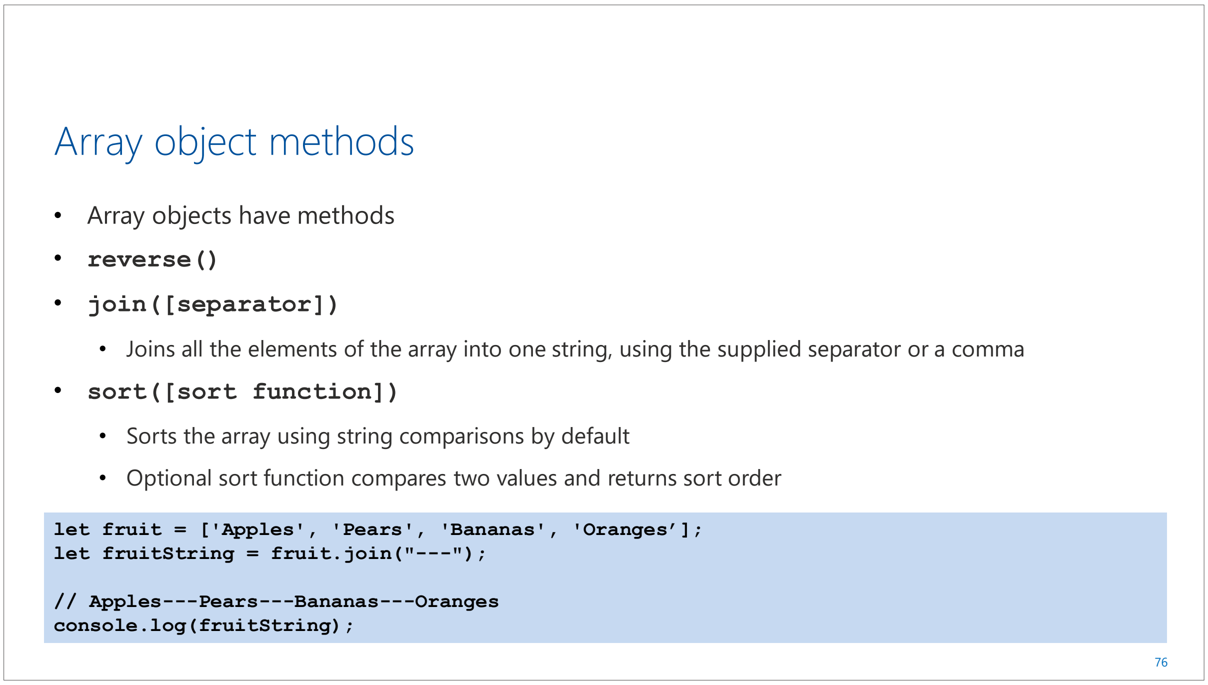
Arrays allow us to store a related set of data. Arrays store data in a list of elements; we access a particular elements by specifying the name of the array and the element index within square brackets, e.g.

```
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

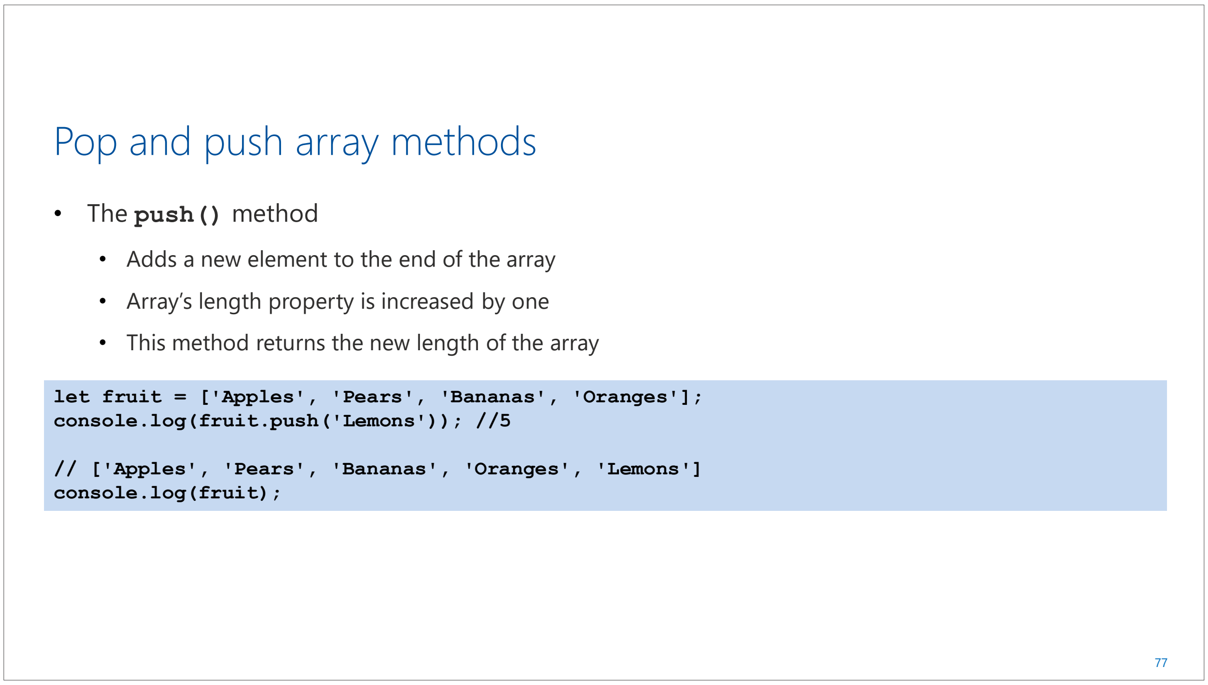
- Array objects have methods
- **`reverse()`**
- **`join([separator])`**
  - Joins all the elements of the array into one string, using the supplied separator or a comma
- **`sort([sort function])`**
  - Sorts the array using string comparisons by default
  - Optional sort function compares two values and returns sort order

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

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

76

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

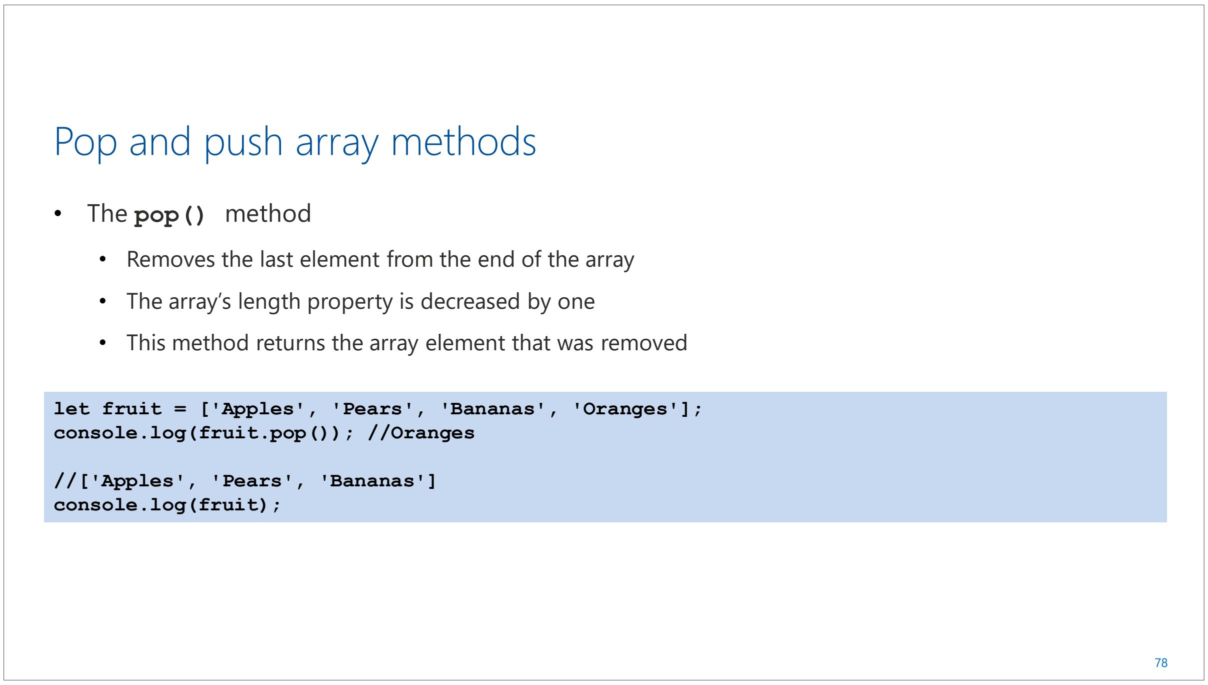
- The **`push()`** method
  - Adds a new element to the end of the array
  - Array's length property is increased by one
  - This method returns the new length of the array

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

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

77

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

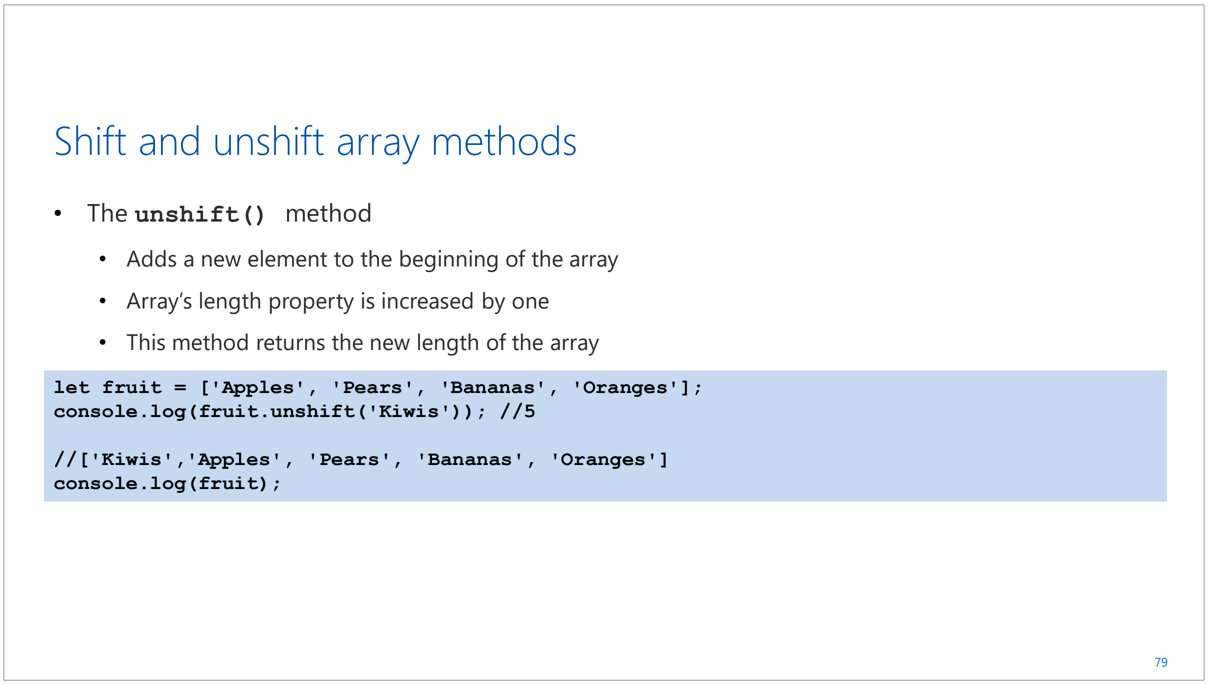
- The **`pop()`** method
  - Removes the last element from the end of the array
  - The array's length property is decreased by one
  - This method returns the array element that was removed

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

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

78

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

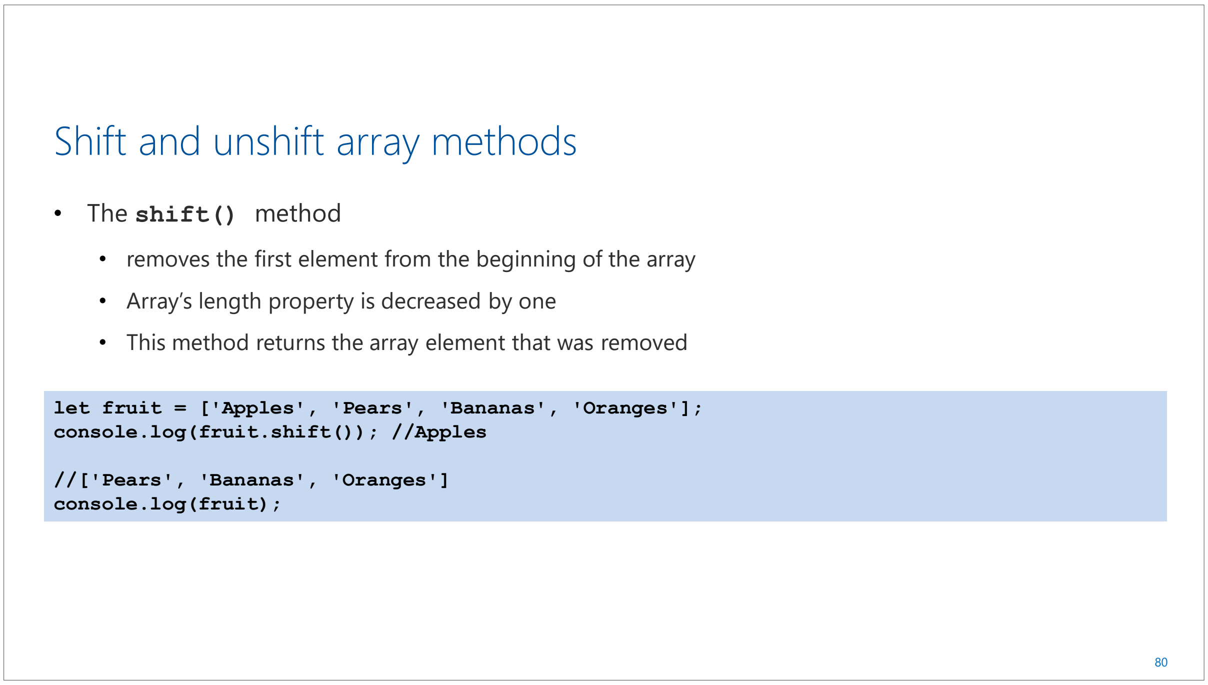
- The **`unshift()`** method
  - Adds a new element to the beginning of the array
  - Array's length property is increased by one
  - This method returns the new length of the array

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

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

79

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

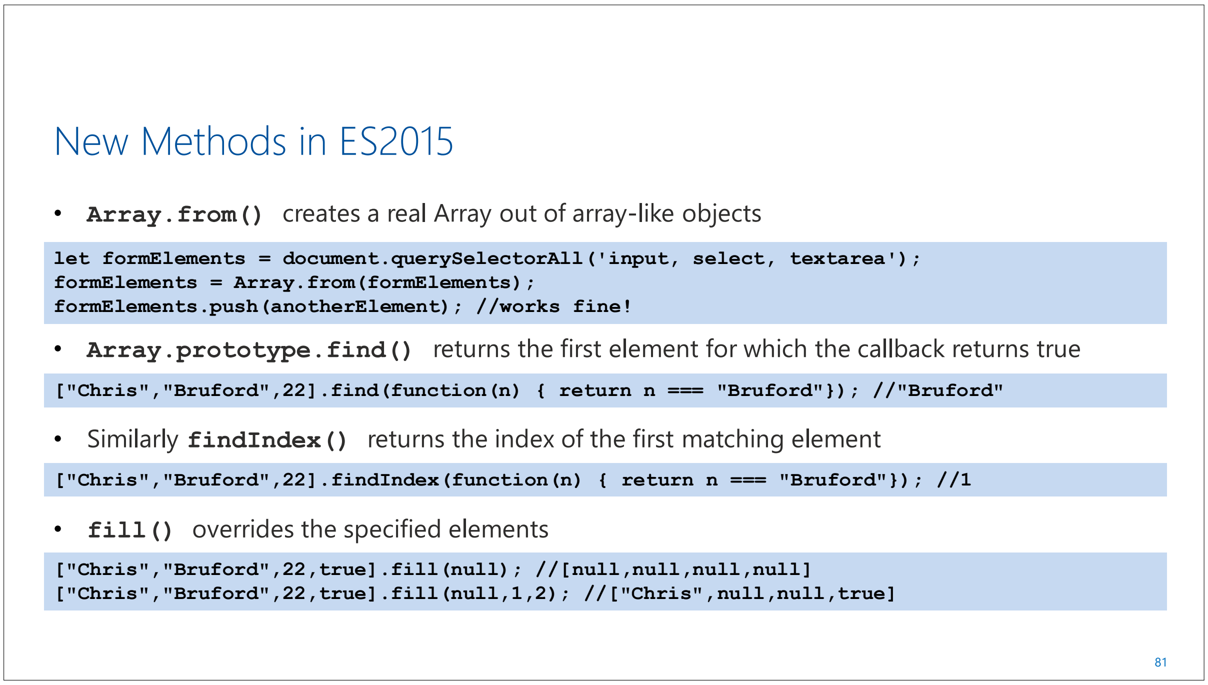
- The **`shift()`** method
  - removes the first element from the beginning of the array
  - Array’s length property is decreased by one
  - This method returns the array element that was removed

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

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

80

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.



New Methods 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]
```

81

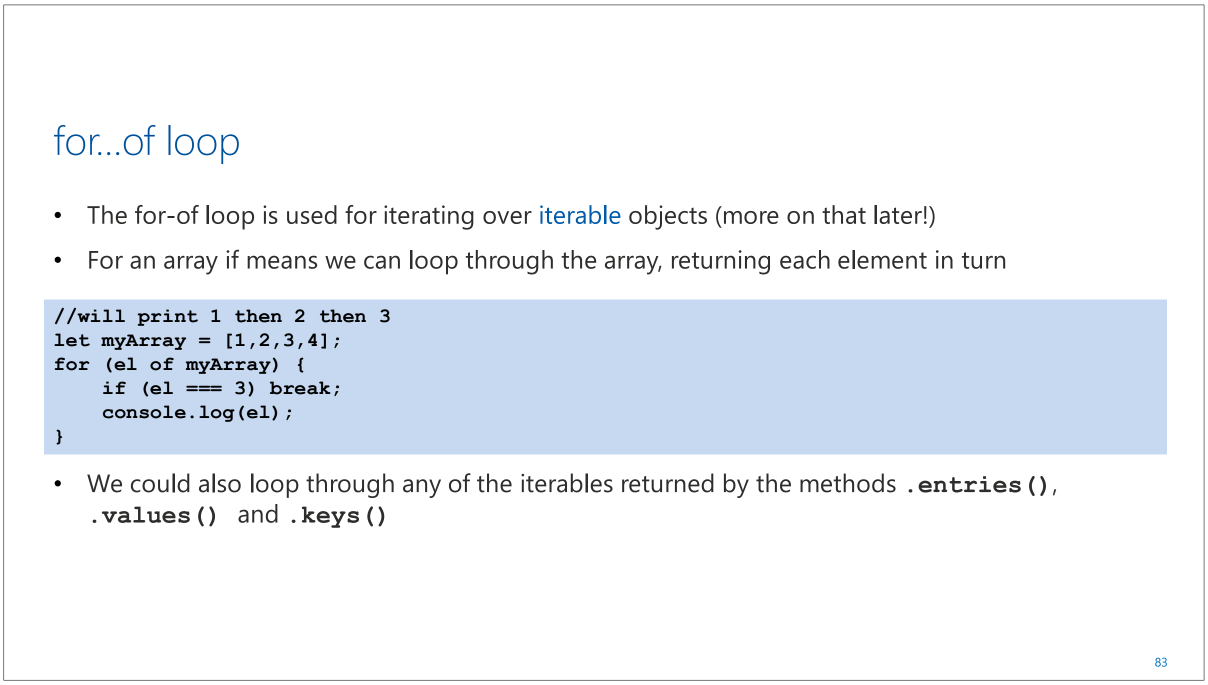
New Methods in ES2015

- `.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
```



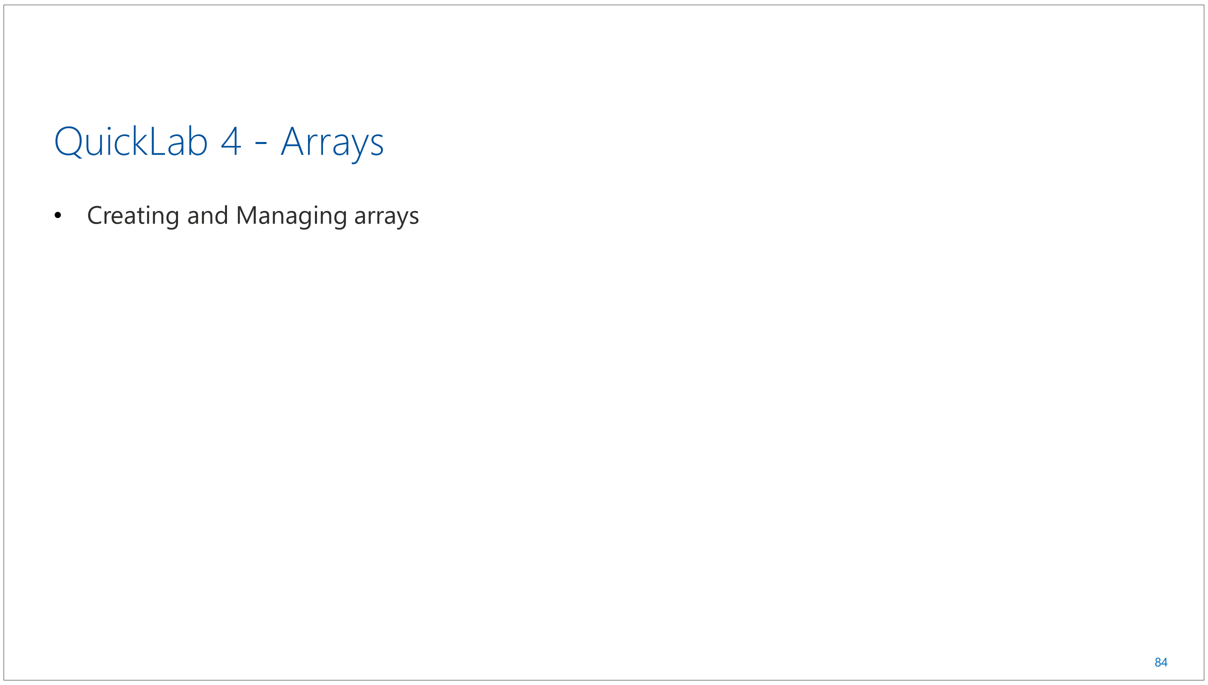
for...of loop

- 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 (el of myArray) {
    if (el === 3) break;
    console.log(el);
}
```

- We could also loop through any of the iterables returned by the methods **`.entries()`**, **`.values()`** and **`.keys()`**

83



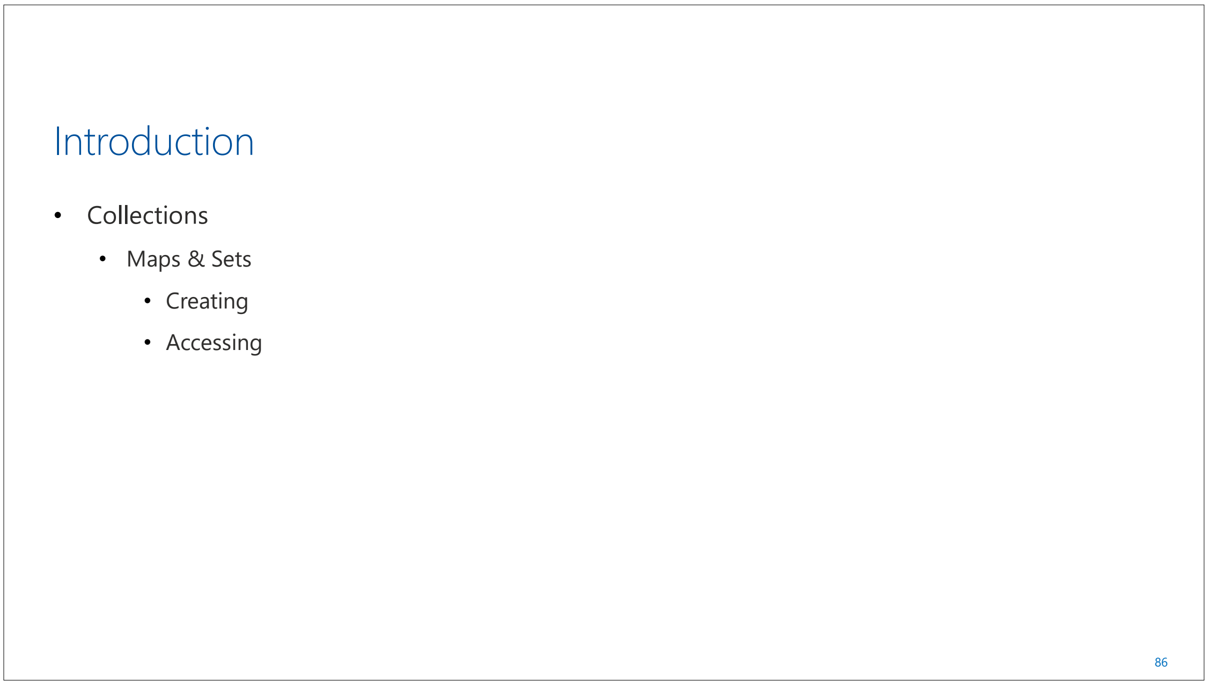
QuickLab 4 - Arrays

- Creating and Managing arrays

84

Collections

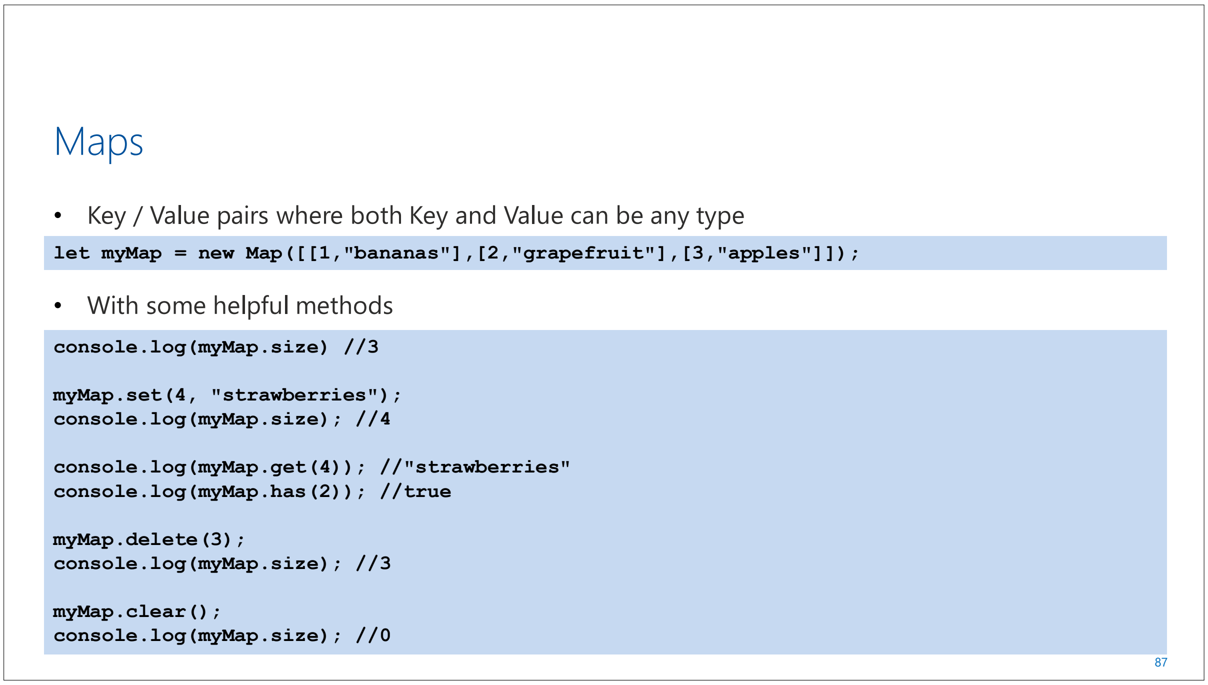
JAVASCRIPT FUNDAMENTALS



Introduction

- Collections
  - Maps & Sets
    - Creating
    - Accessing

86



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
```

87

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}`);
}
```

88

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
```

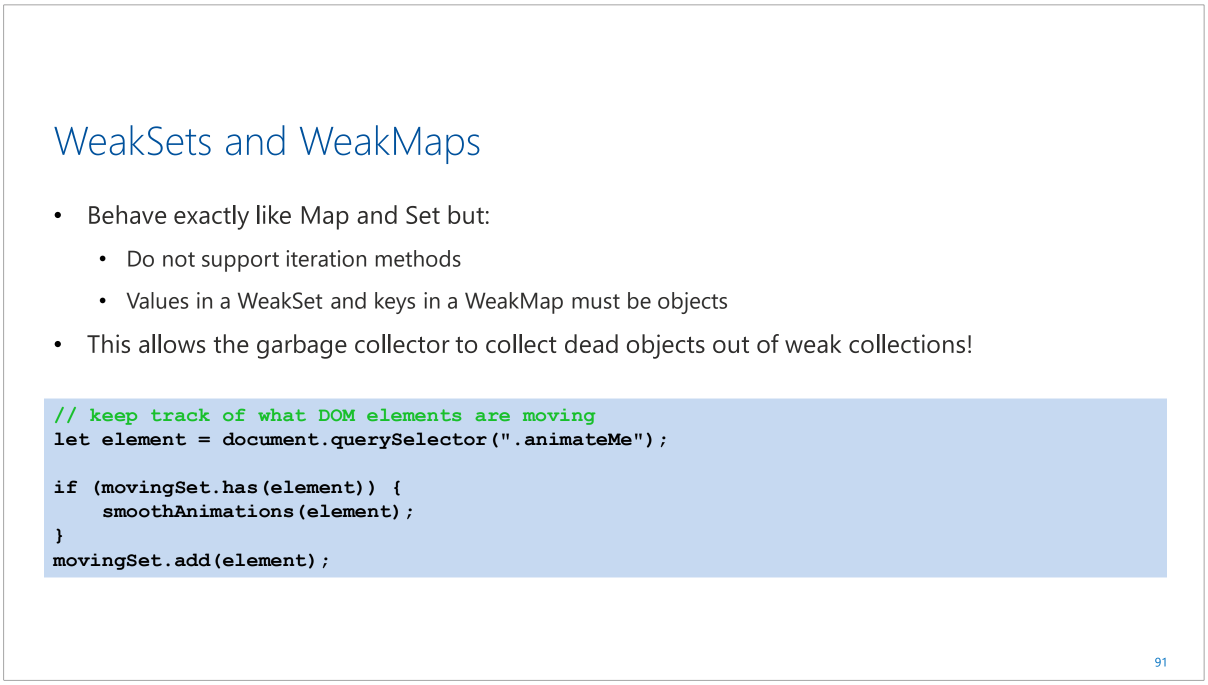
89

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}`);
}
```

90

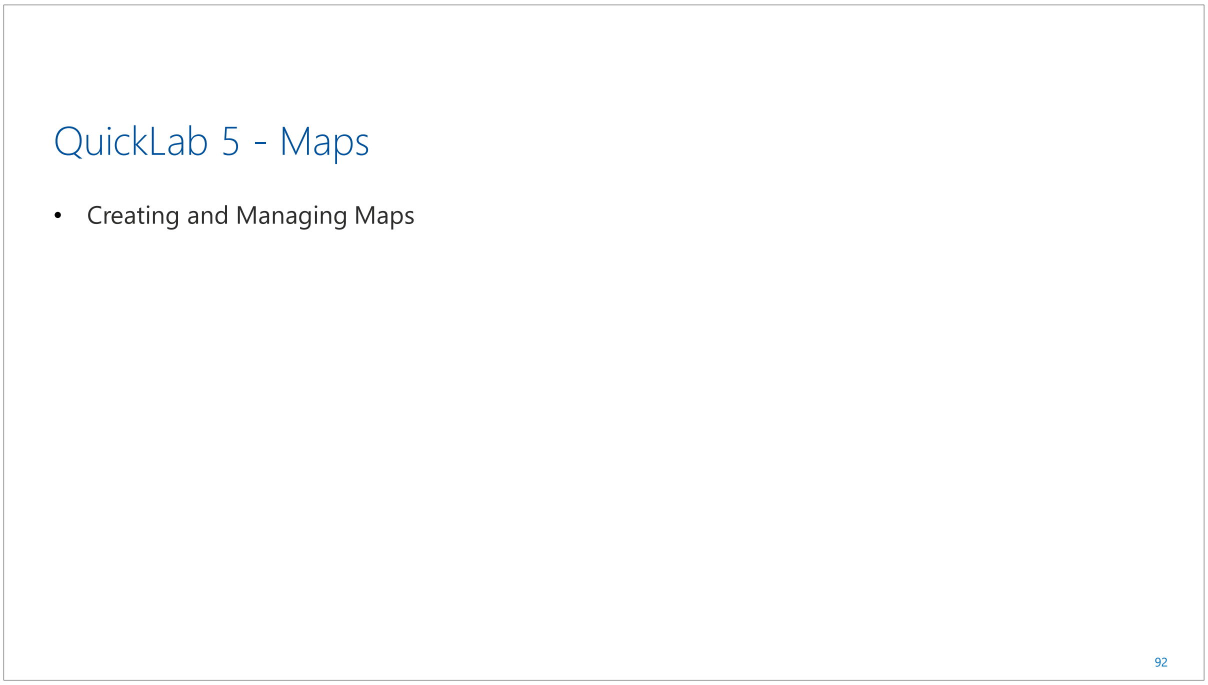


WeakSets and WeakMaps

- 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);
```

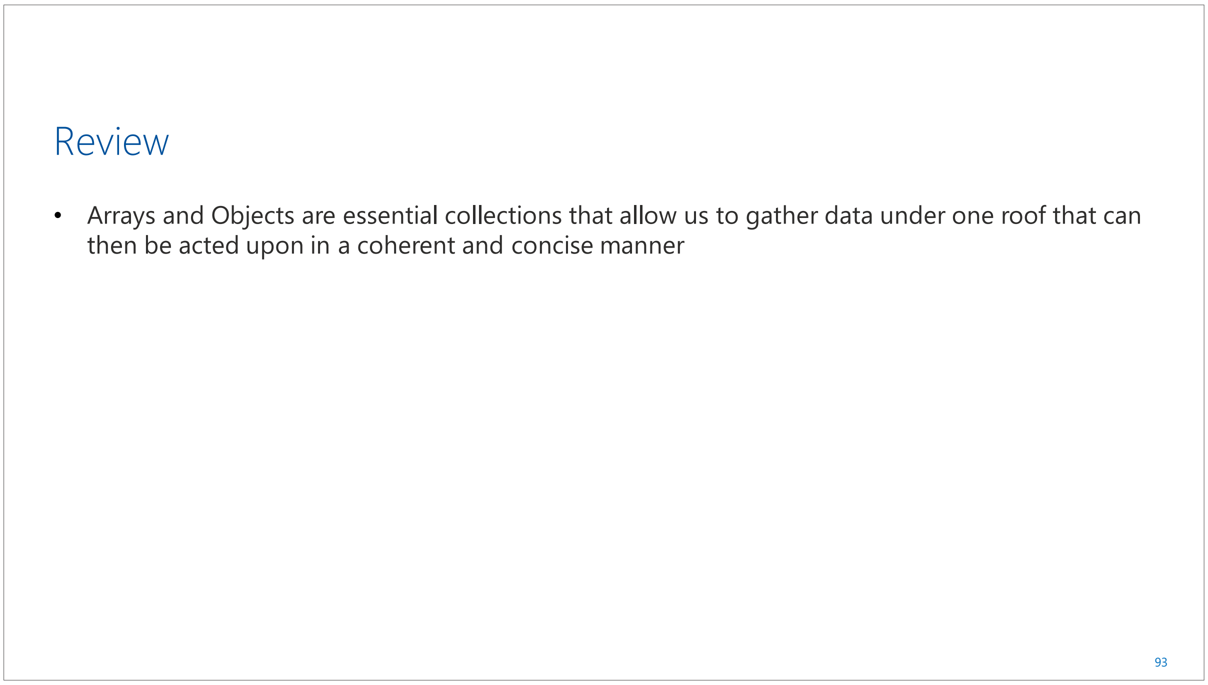
91



QuickLab 5 - Maps

- Creating and Managing Maps

92



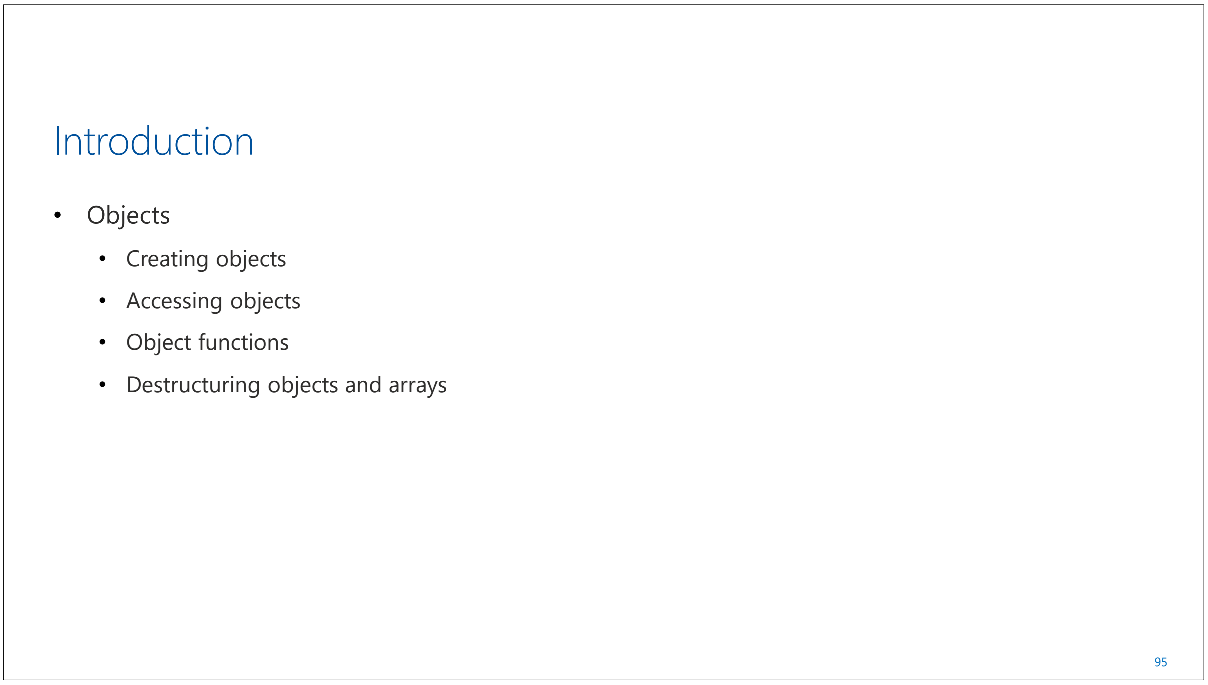
Review

- 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

93

Objects

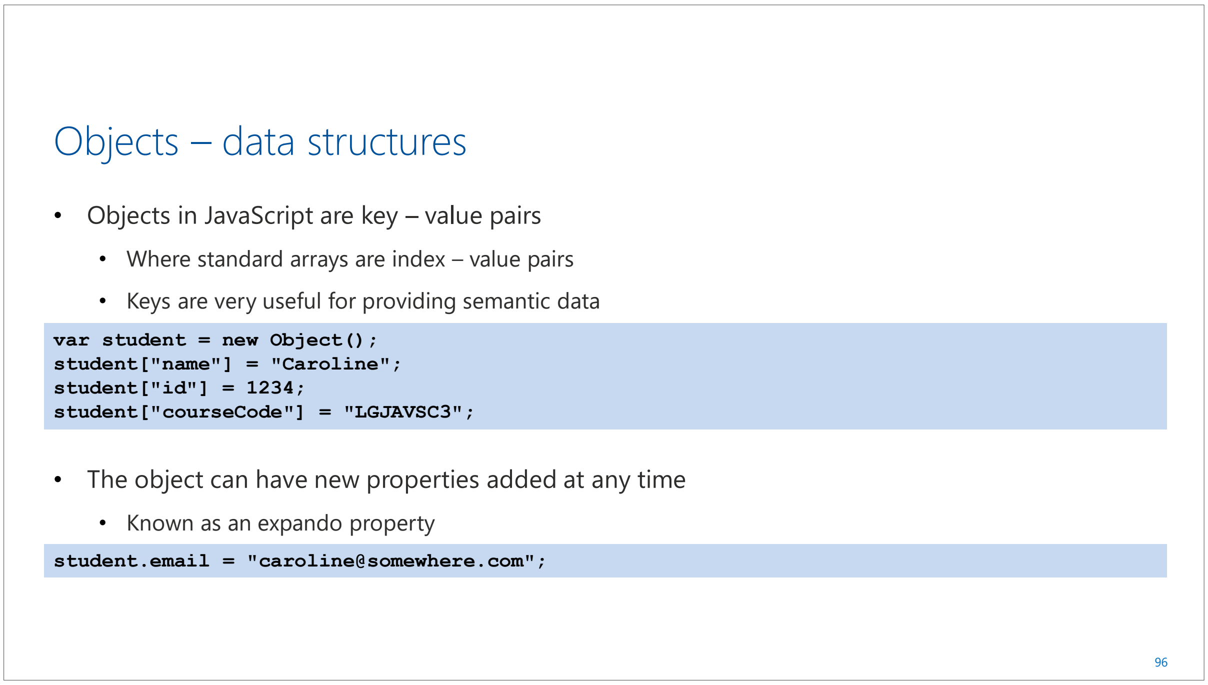
JAVASCRIPT FUNDAMENTALS



Introduction

- Objects
  - Creating objects
  - Accessing objects
  - Object functions
  - Destructuring objects and arrays

95



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";
```

96

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.

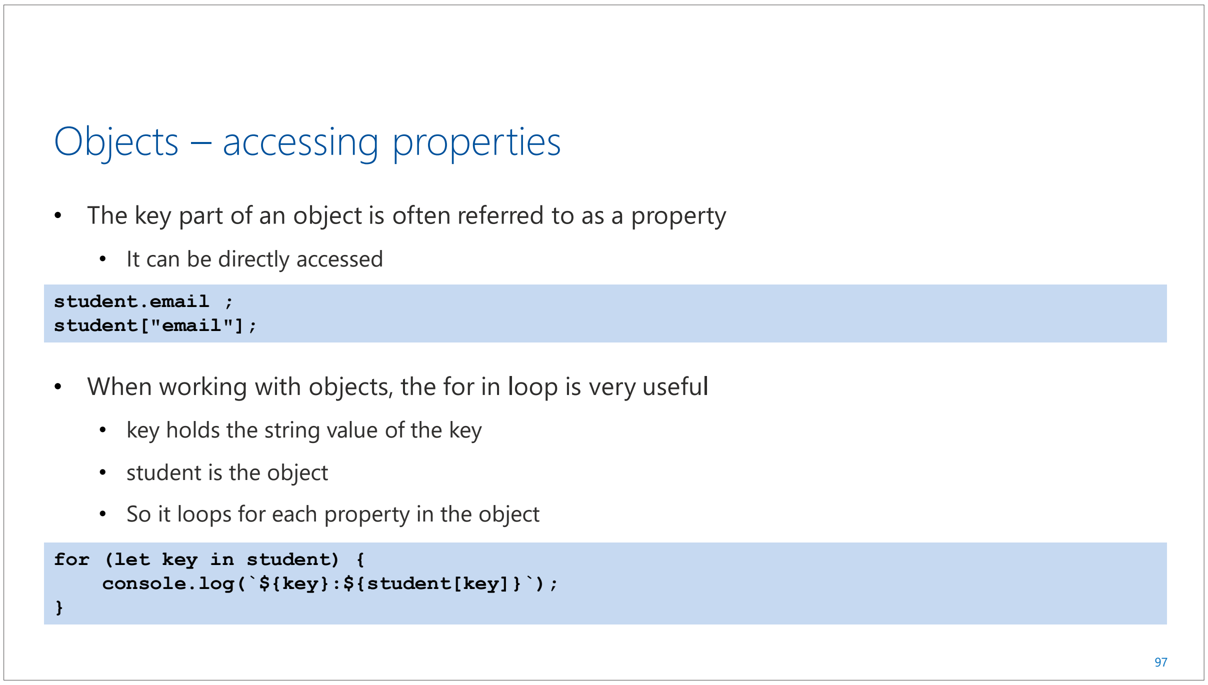
Objects are principal objects in JavaScript. If the variable does not refer to a primitive type, it is an object of some kind. In principal, they are very similar to the concepts of arrays but, instead of an indexed identifier, a string-based key is used (in fact – Arrays in JavaScript are nothing more 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"];
```

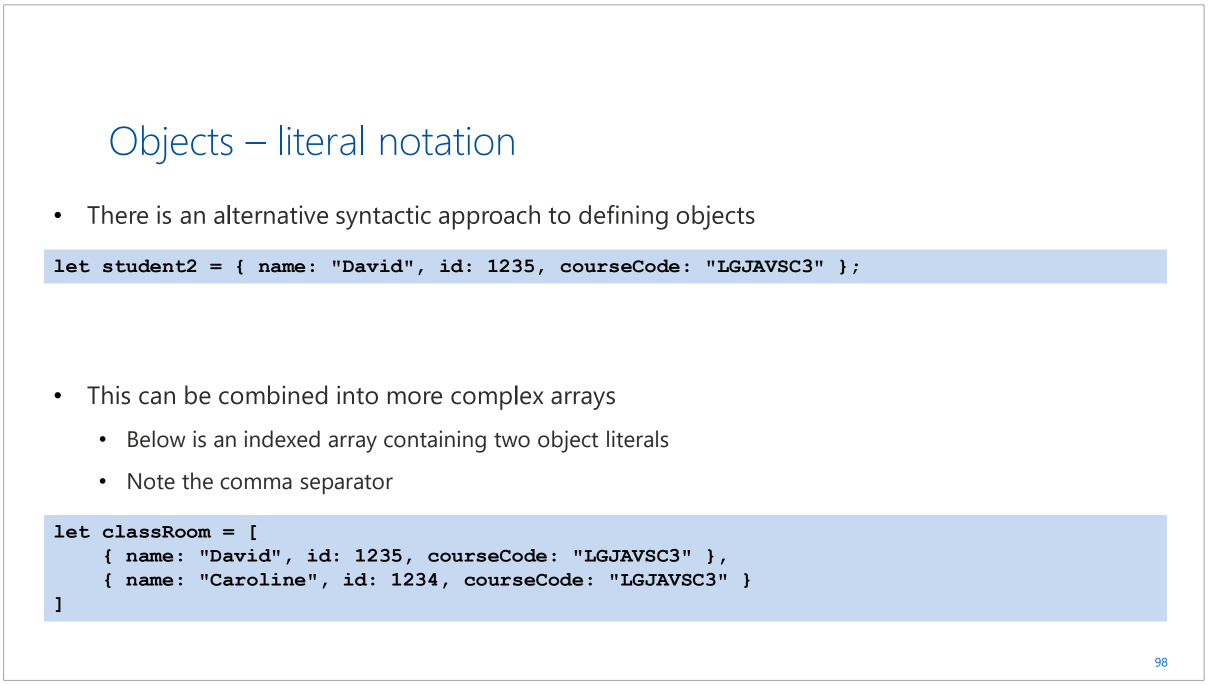
- When working with objects, the for in loop is very useful
  - key holds the string value of the key
  - student is the object
  - So it loops for each property in the object

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

97

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" };
```

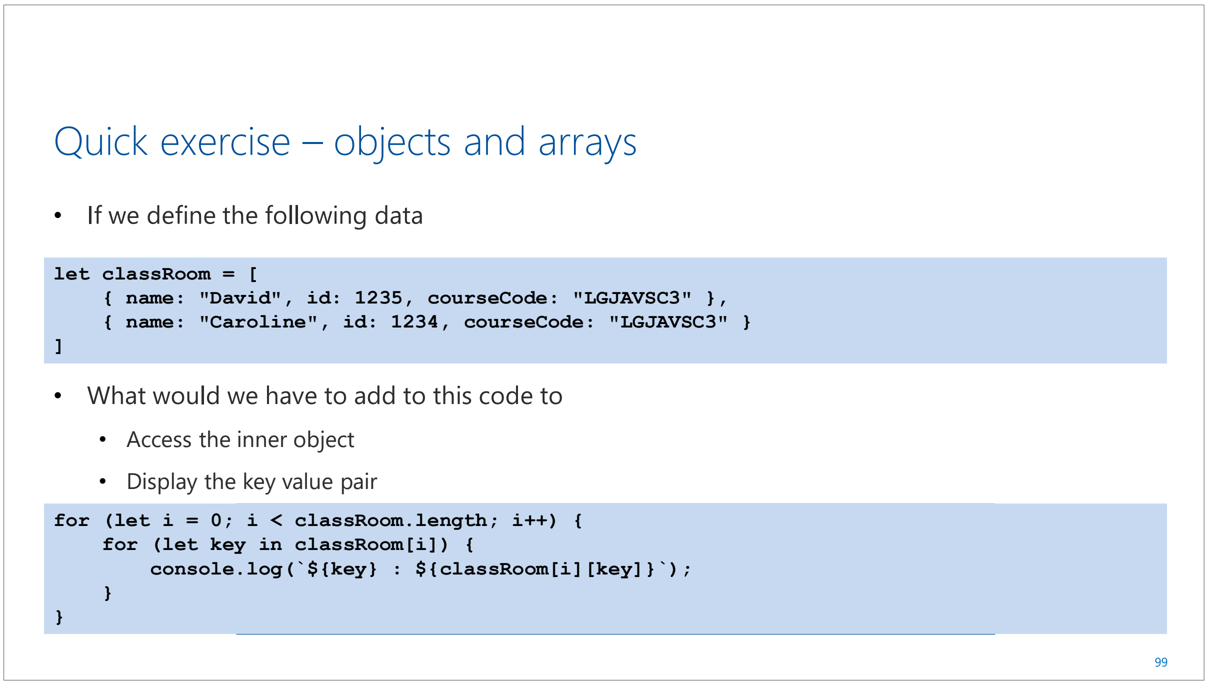
- This can be combined into more complex arrays
  - Below is an indexed array containing two object literals
  - Note the comma separator

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

98

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

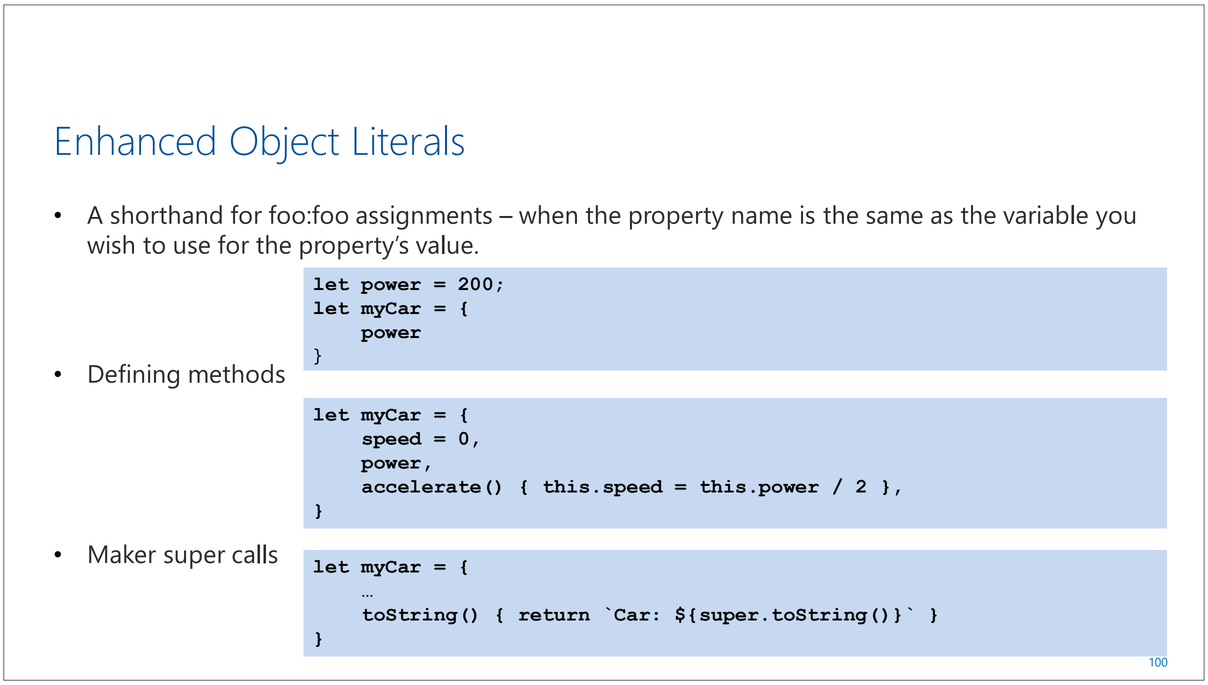
- If we define the following data

```
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 (let i = 0; i < classRoom.length; i++) {
    for (let key in classRoom[i]) {
        console.log(`${key} : ${classRoom[i][key]}`);
    }
}
```

99



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.

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

- Defining methods

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

- Maker super calls

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

100

Dynamic Property Names

- Dynamic property names

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

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

Object.assign()

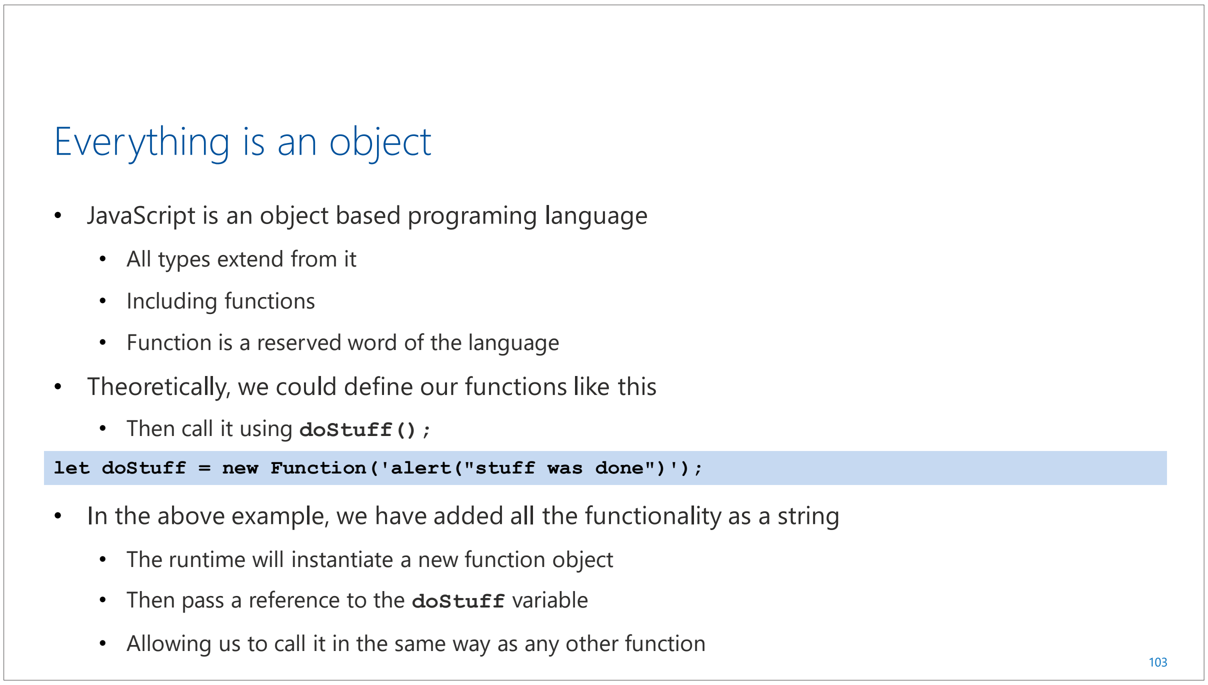
- The assign() method has been added to copy enumerable own properties to an object
- Can use this to merge objects

```
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);
```

102



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

103

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:

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

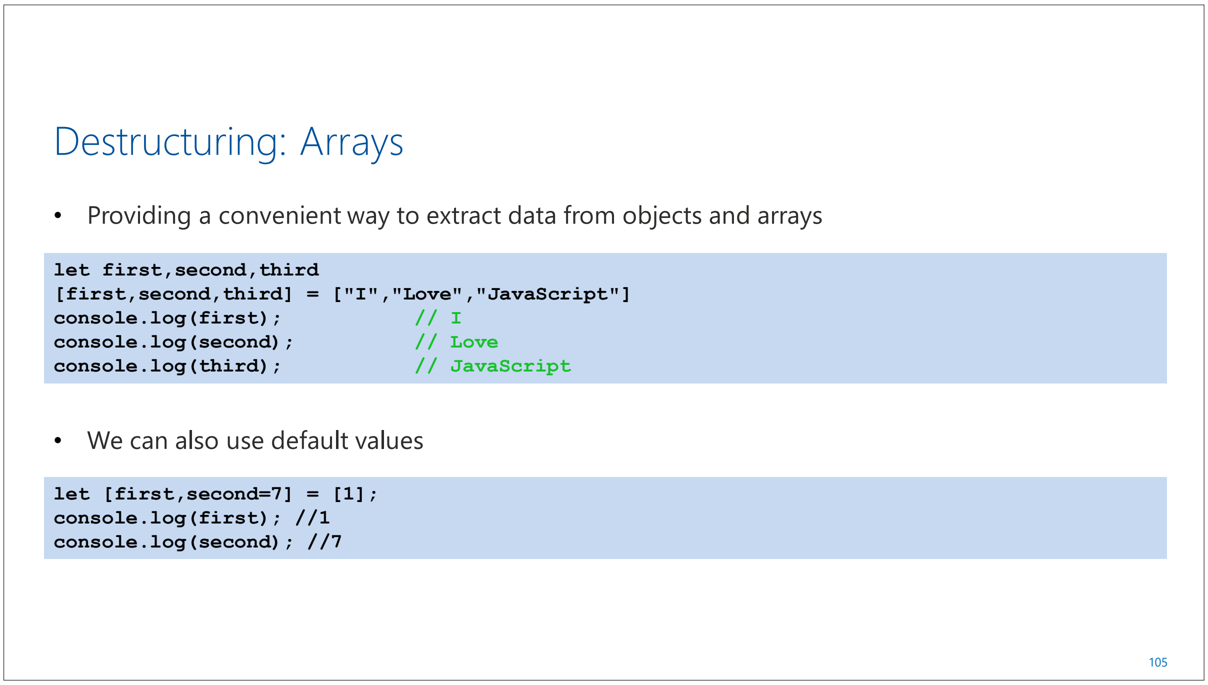
Both code implementations create an object of type number and the variable then receives a memory reference to that object. It is important to remember that JavaScript variables are simply pointers to this object.

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

JAVASCRIPT FUNDAMENTALS



Destructuring: Arrays

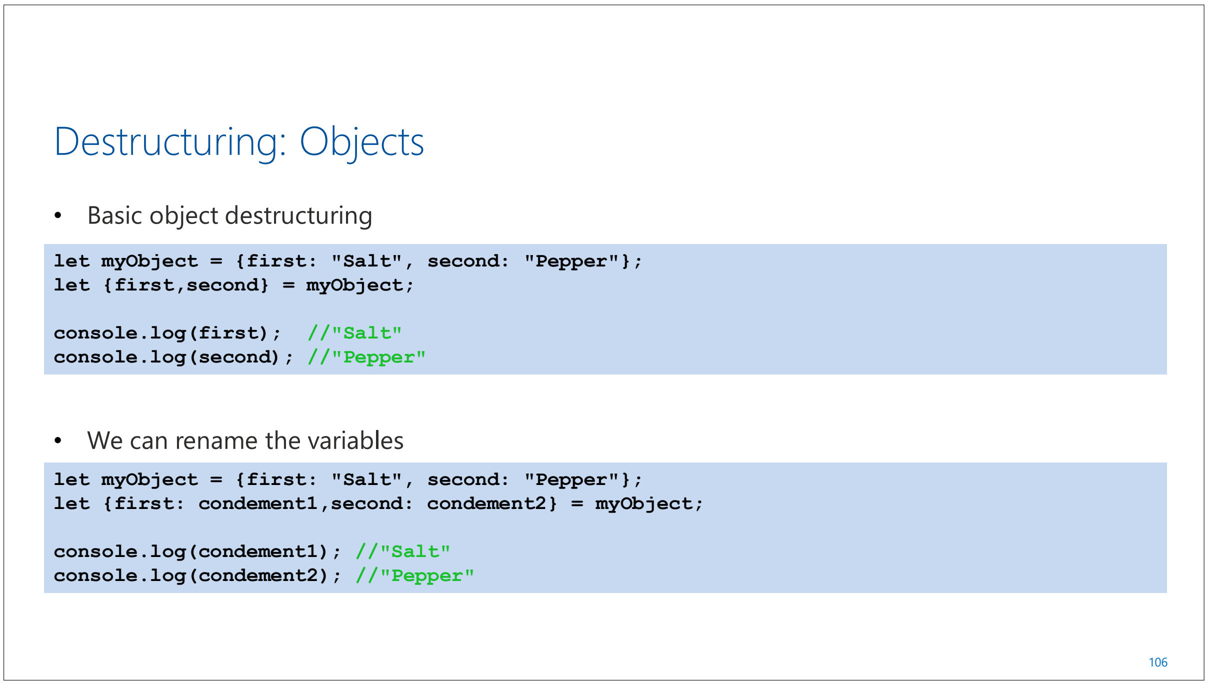
- Providing a convenient way to extract data from objects and arrays

```
let first,second,third
[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
```

105



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"
```

106

Destructuring: Objects

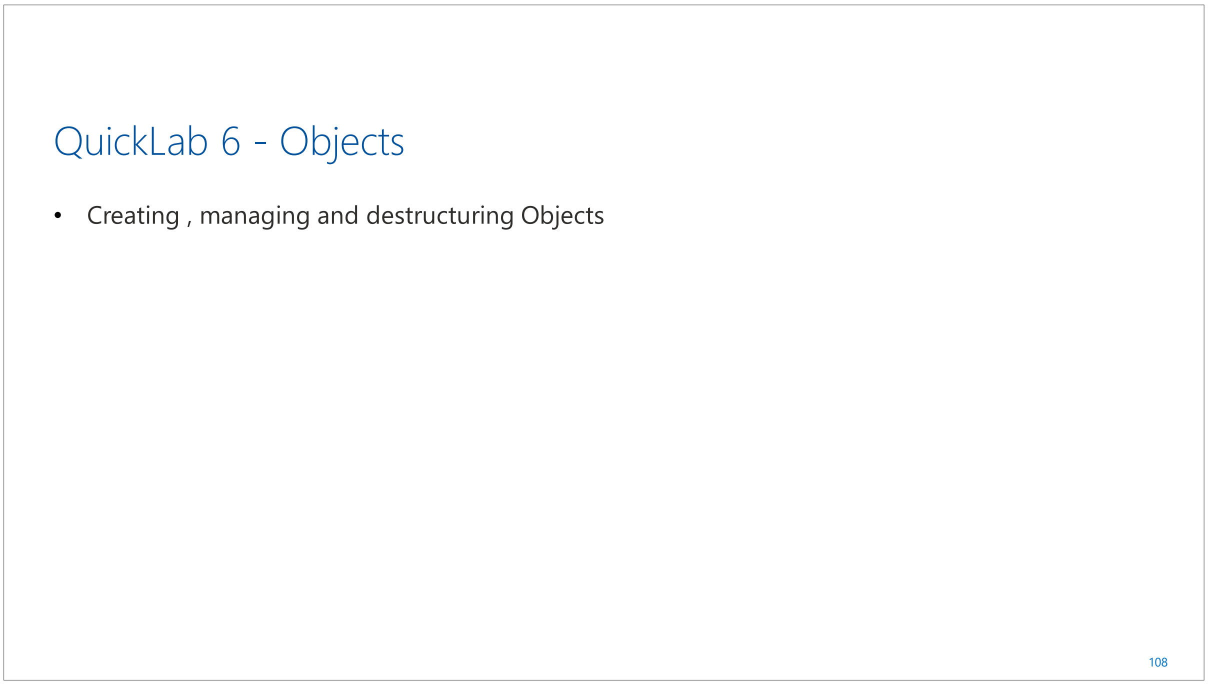
- Default values

```
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!
```

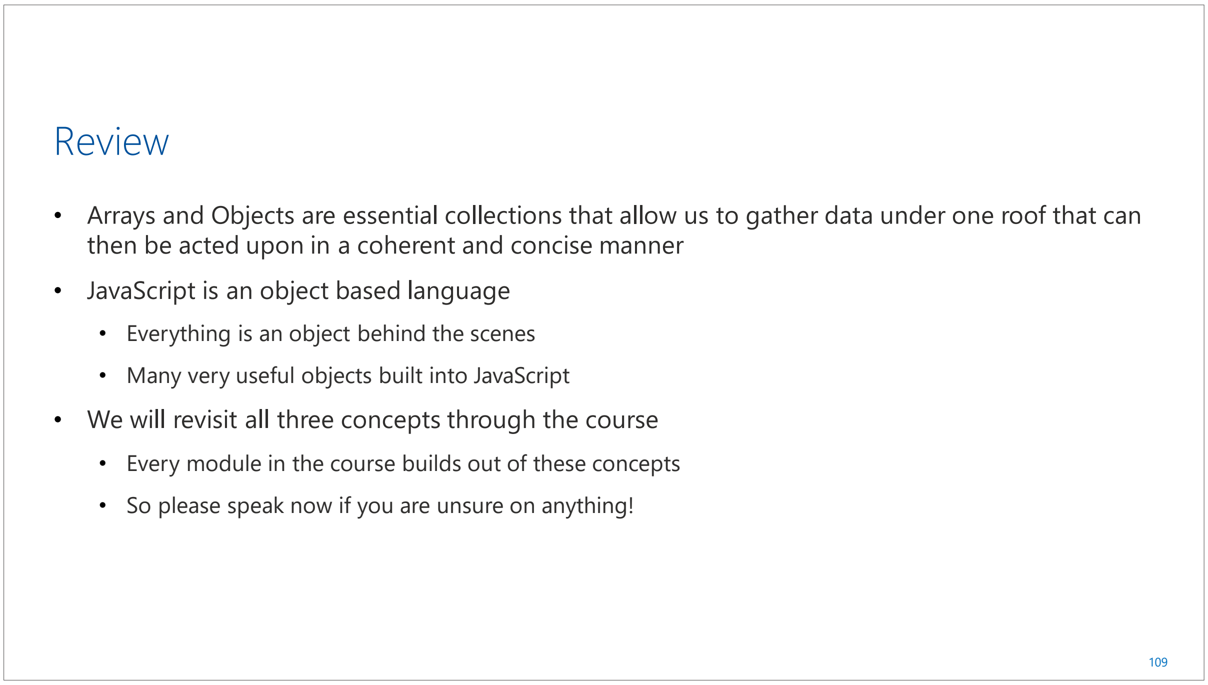
107



QuickLab 6 - Objects

- Creating , managing and destructuring Objects

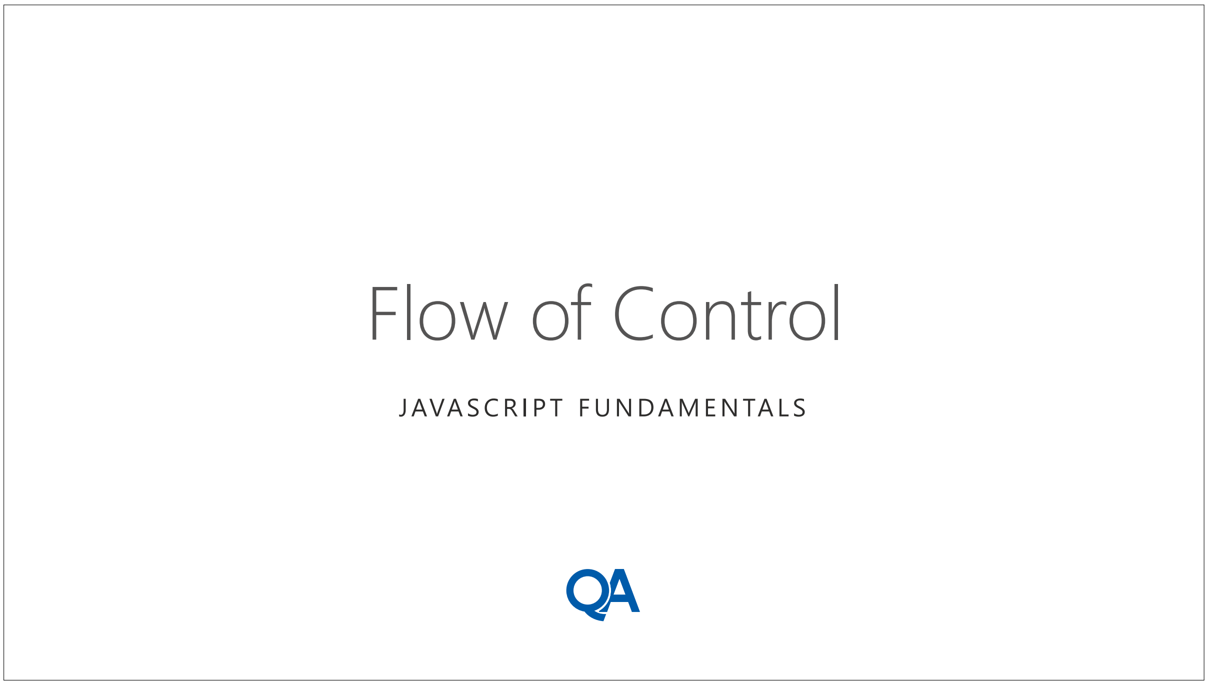
108



Review

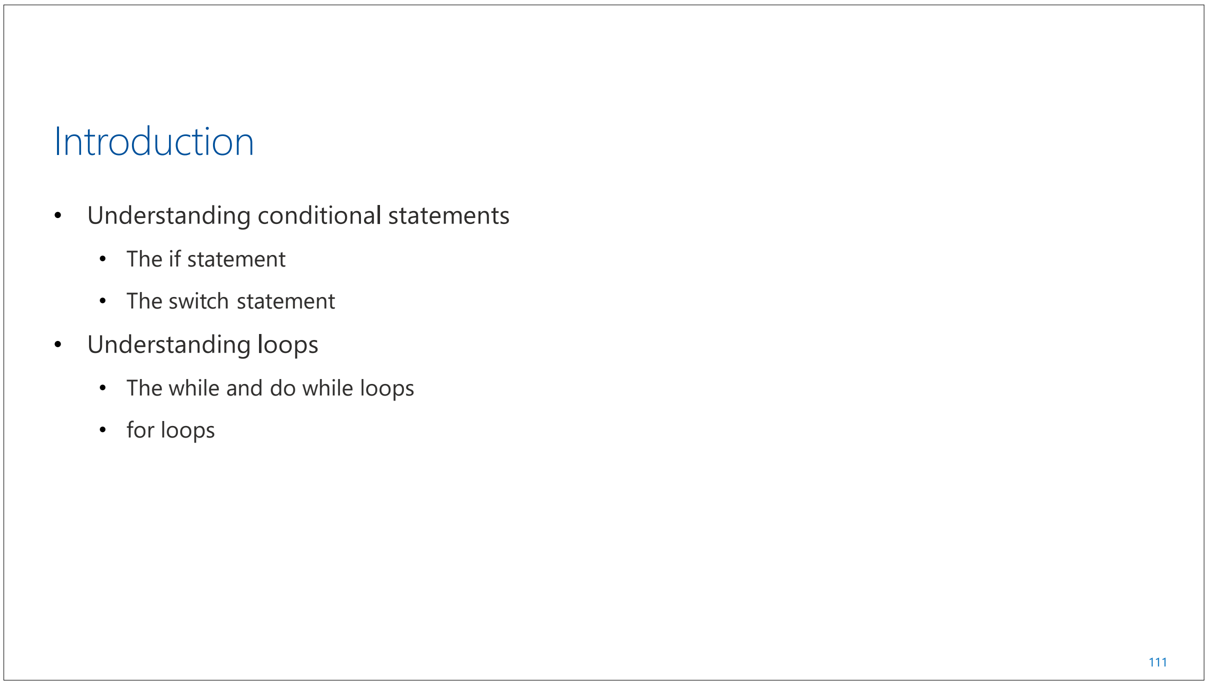
- Arrays and Objects are essential collections that allow us to gather data under one roof that can then be acted upon in a coherent and concise manner
- JavaScript is an object based language
  - Everything is an object behind the scenes
  - Many very useful objects built into JavaScript
- We will revisit 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!

109



Flow of Control

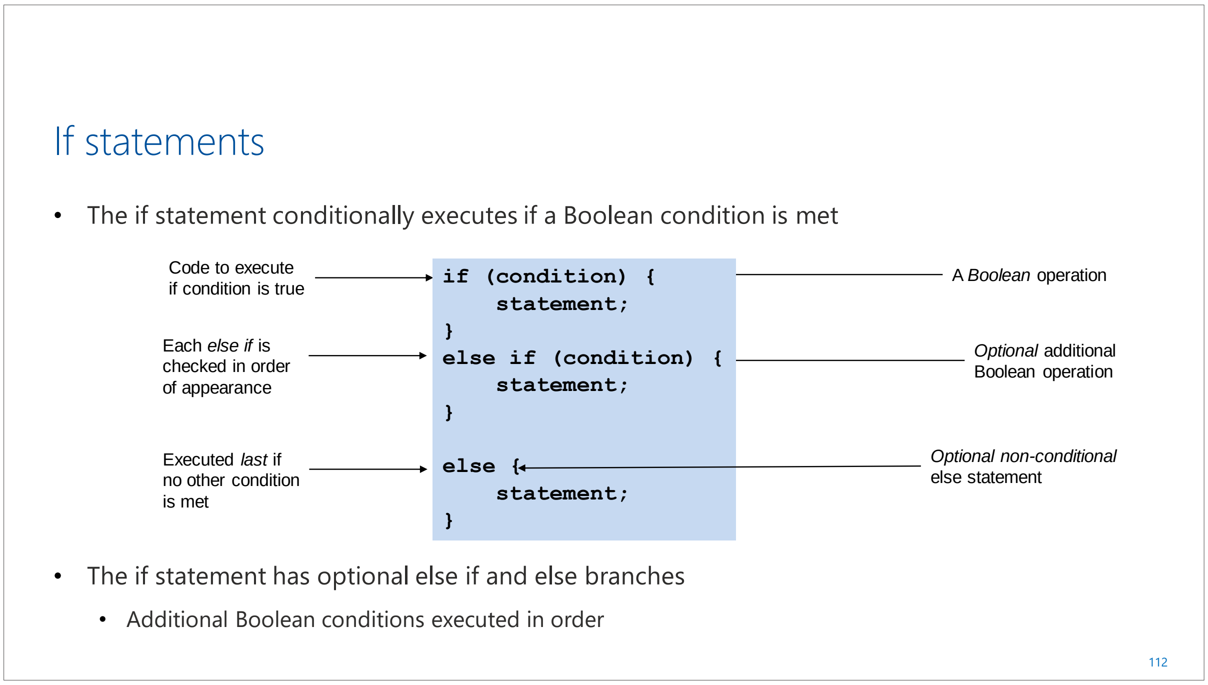
JAVASCRIPT FUNDAMENTALS



Introduction

- Understanding conditional statements
  - The if statement
  - The switch statement
- Understanding loops
  - The while and do while loops
  - for loops

111



If statements

- The if statement conditionally executes if a Boolean condition is met

Code to execute if condition is true

Each *else if* is checked in order of appearance

Executed *last* if no other condition is met

```
if (condition) {
    statement;
}
else if (condition) {
    statement;
}

else {
    statement;
}
```

A *Boolean* operation

Optional additional Boolean operation

Optional non-conditional else statement

- The if statement has optional else if and else branches
  - Additional Boolean conditions executed in order

112

JavaScript has a set of flow control statements, including `if`, `switch`, `while`, and `for`. The *condition* in each of these statements may be any expression that evaluates to either a Boolean or a number.

In the case of a numeric expression, the condition is considered to be `false` if the value of the expression is 0, and `true` in all other cases.

The *statement* shown in the syntax descriptions above may be any statement, including a statement block or a further flow control statement.

The curled braces are technically optional when there is only one line of code to be executed by the `if` statement. Therefore, the following code would be legal:

```
If (5 > 3)
        alert("true");
```

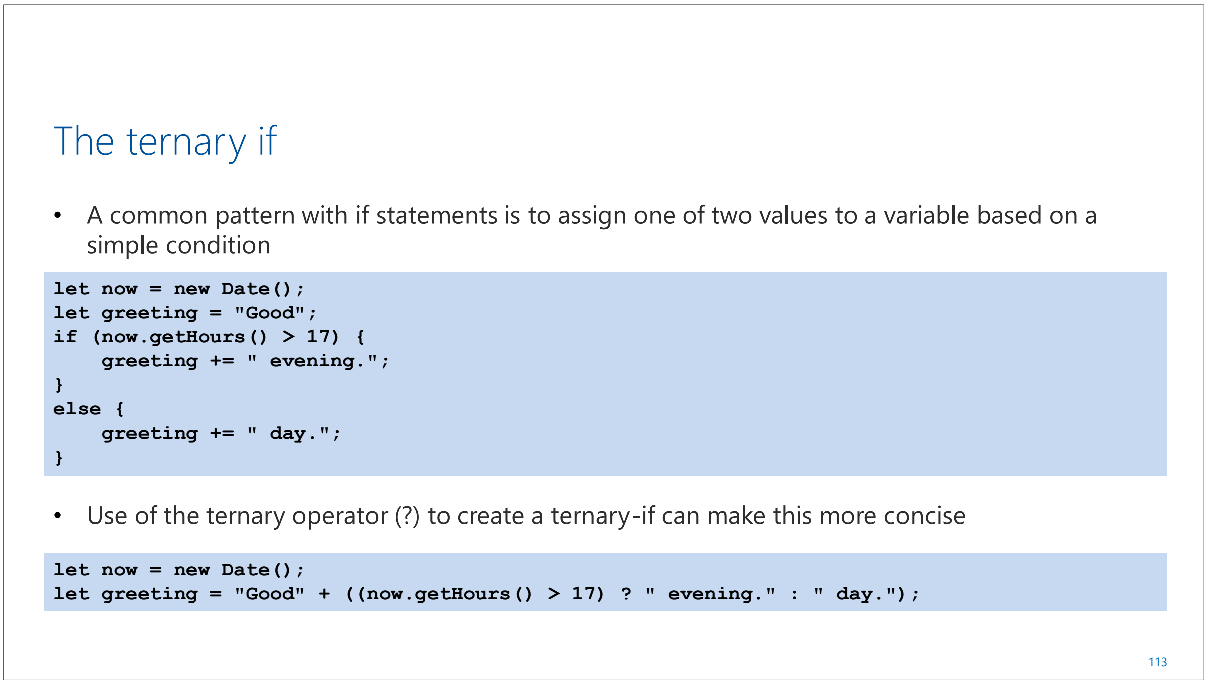
However, the following example would only consider the alert statement to be part of the conditional statements and not the variable declaration:

```
If (5 > 3)
        alert("true"); //conditional
        var x = 5; // not conditional
```

It is also possible to express an `if else` statement as a simple operator:

```
var result = (5 > 3) ? true : false;
```

This initialises the result variable to be a Boolean with a value of false.



The ternary if

- A common pattern with if statements is to assign one of two values to a variable based on a simple condition

```
let now = new Date();
let greeting = "Good";
if (now.getHours() > 17) {
    greeting += " evening.";
}
else {
    greeting += " day.";
}
```

- Use of the ternary operator (?) to create a ternary-if can make this more concise

```
let now = new Date();
let greeting = "Good" + ((now.getHours() > 17) ? " evening." : " day.");
```

113

The ternary operator is an alternative to the standard `if` statement. Primarily used for variable assignments or when a standard if is too unwieldy, it has the following syntax:

```
test ? expression1 : expression2
```

Test

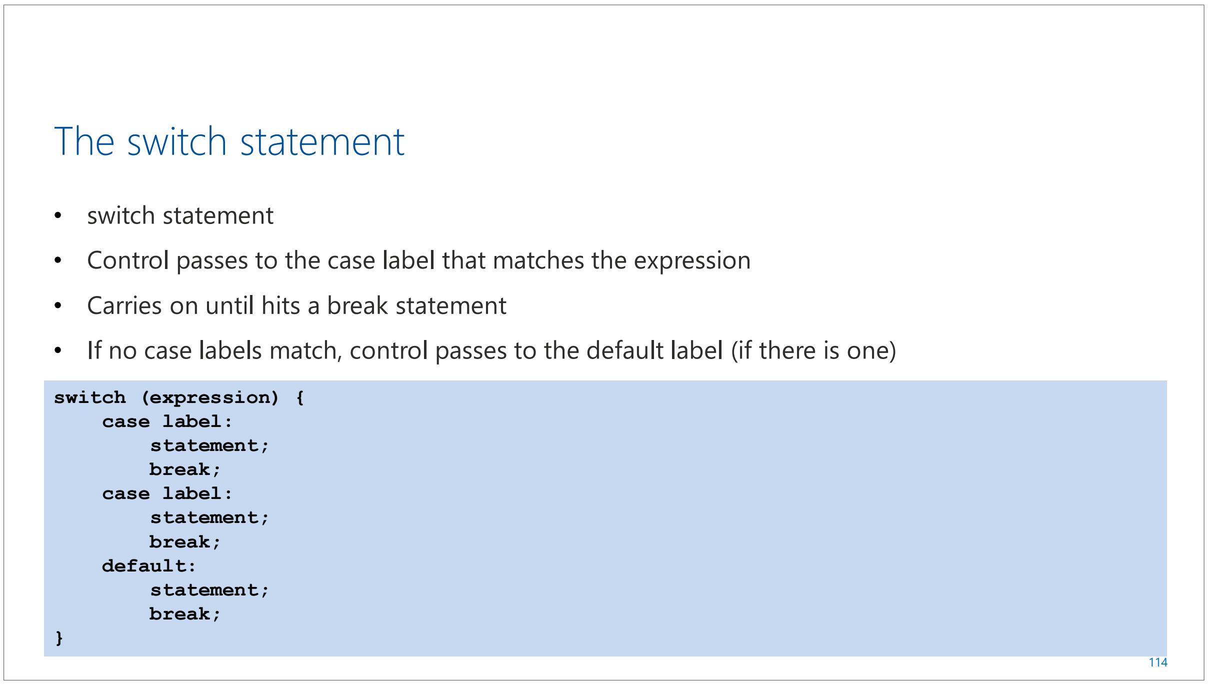
Any Boolean expression.

Expression1

An expression returned if *test* is **true**. May be a comma expression.

Expression2

An expression returned if *test* is **false**. More than one expression may be a linked by a comma expression.



The switch statement

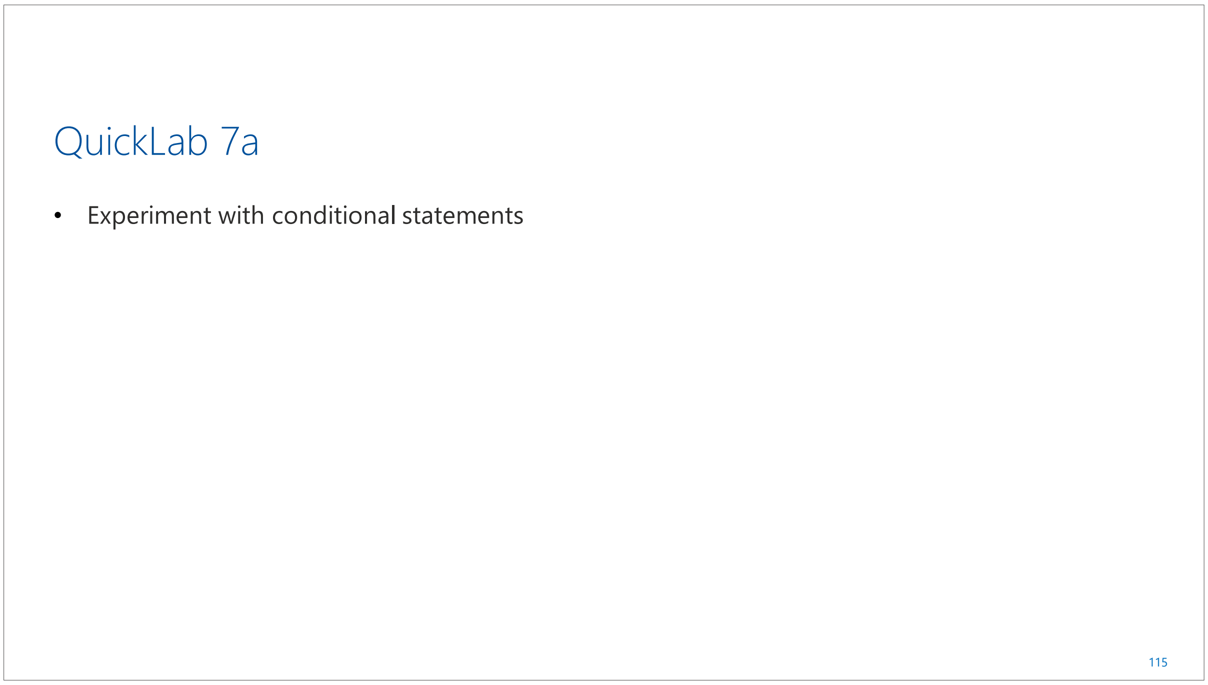
- switch statement
- Control passes to the case label that matches the expression
- Carries on until hits a break statement
- If no case labels match, control passes to the default label (if there is one)

```
switch (expression) {
    case label:
        statement;
        break;
    case label:
        statement;
        break;
    default:
        statement;
        break;
}
```

114

The switch statement is useful when selecting an action from a number of alternatives. However, it is rarely used where a set of separate if statements make things more manageable and often more efficient. `Switch` compares the value of an integer test-expression with each of the case labels in turn. If a match is found, the statement(s) following the label are executed. Execution continues either until the end of the entire `switch` statement or until a `break` is encountered. If no match is found, control is passed to the statement(s) following the `default` label. The default label is optional – if there is no match and no default, the switch does nothing.

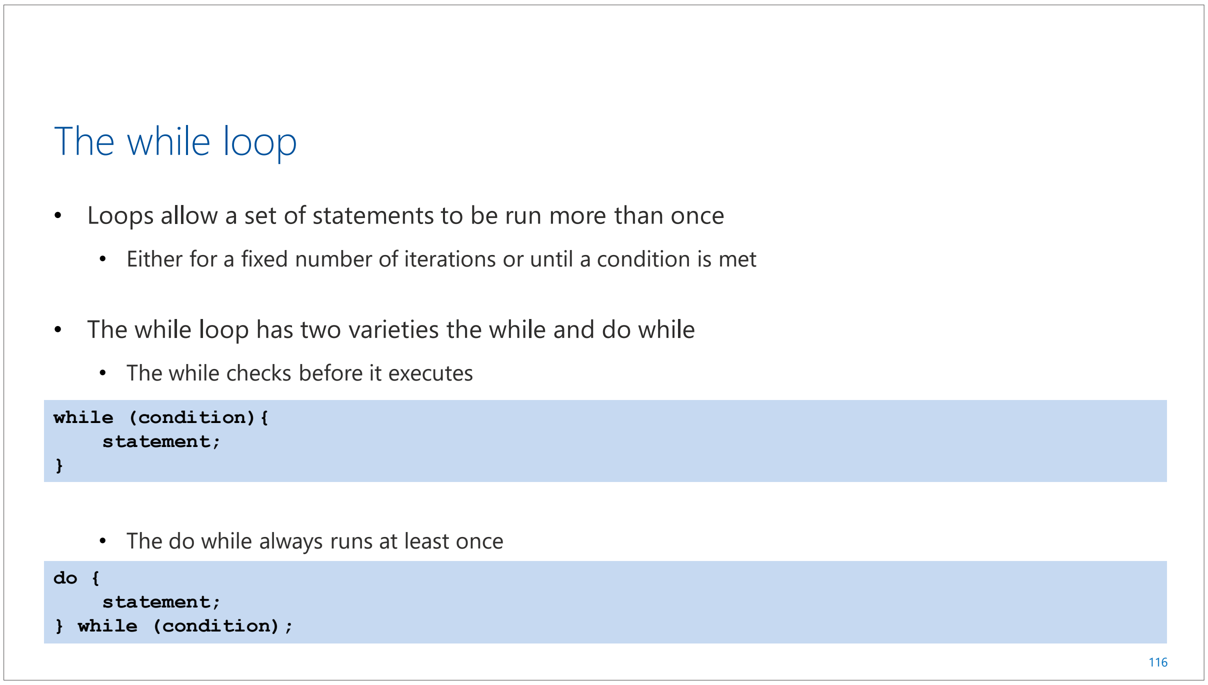
```
switch(weekDay)
{
        case "Mon":
        case "Tue":
        case "Wed":
        case "Thu":
        case "Fri":
            document.write("Go to work");
            break;
        case "Sat":
        case "Sun":
            document.write("Stay at home");
            break;
        default:
            document.write("Which planet are you on?");
            break;
}
```



QuickLab 7a

- Experiment with conditional statements

115



The while loop

- Loops allow a set of statements to be run more than once
  - Either for a fixed number of iterations or until a condition is met
- The while loop has two varieties the while and do while
  - The while checks before it executes

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

  - The do while always runs at least once

```
do {
    statement;
} while (condition);
```

116

All of the rules regarding brackets with the `if` statement apply to loops as well. Loops also provide two key loop control structures:

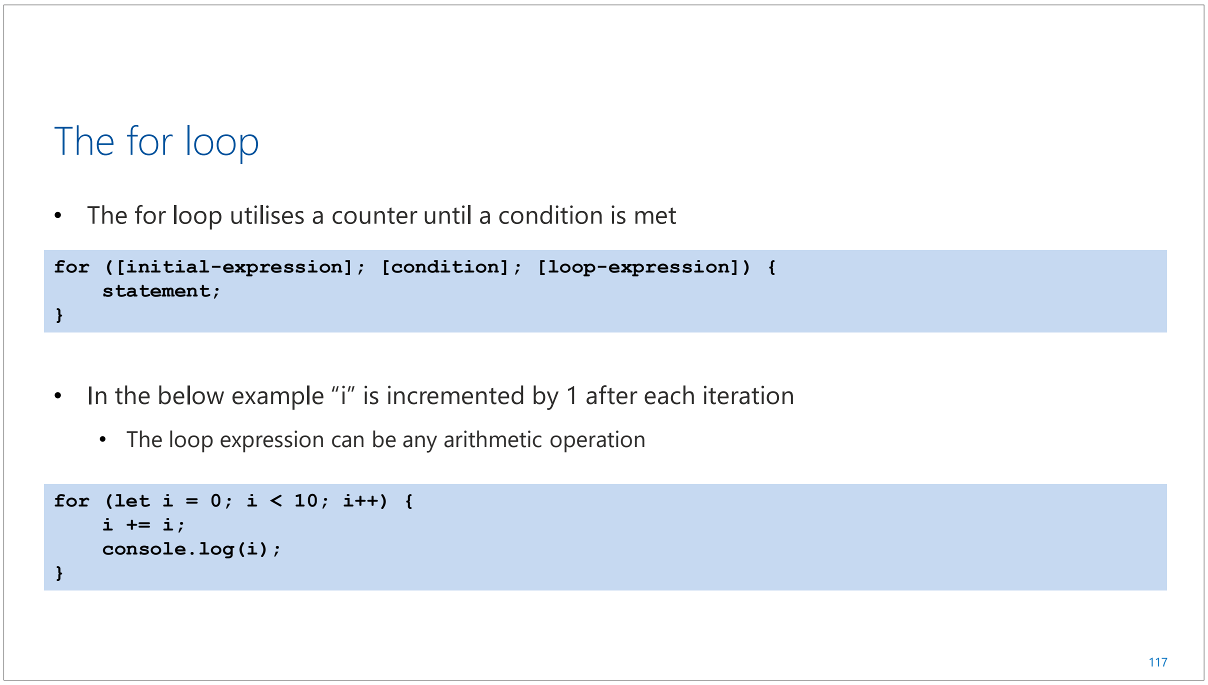
`break` – Exits the loop immediately.

`continue` – Jumps to the end of this iteration immediately.

So `while` and `do...while` loops repeat a statement (or statement block) repeatedly while the condition is `true.`

```
initial-expression;
while (condition)
{
      statement;
      loop-expression;
}
```

The `break` statement exits out of the loop immediately, and execution resumes at the statement immediately following the loop statement. The `continue` statement jumps to the end of the current iteration immediately; in the case of a `while` loop, execution will resume with evaluation of the *condition* expression at the top of the loop.



The for loop

- The for loop utilises a counter until a condition is met

```
for ([initial-expression]; [condition]; [loop-expression]) {
    statement;
}
```

- In the below example "i" is incremented by 1 after each iteration
  - The loop expression can be any arithmetic operation

```
for (let i = 0; i < 10; i++) {
    i += i;
    console.log(i);
}
```

117

The `for` loop is a specialised form of a `while` loop;

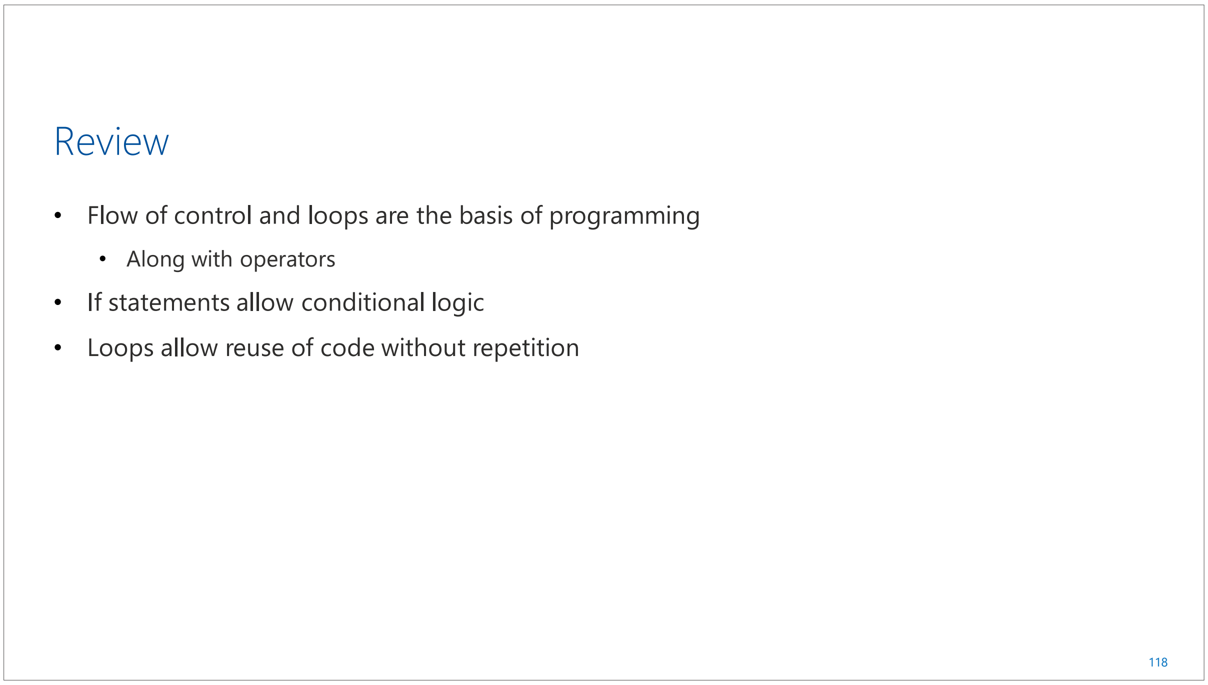
```
for (initial-expression; condition; loop-expression)
{
      statement;
}
```

it is equivalent to writing:

```
initial-expression;
while (condition)
{
      statement;
      loop-expression;
}
```

All three of the expressions in a `for` statement are optional, and may be used in any combination; a `for` statement with none of the expressions present
(i.e. `for (;;)`) creates an infinite loop.

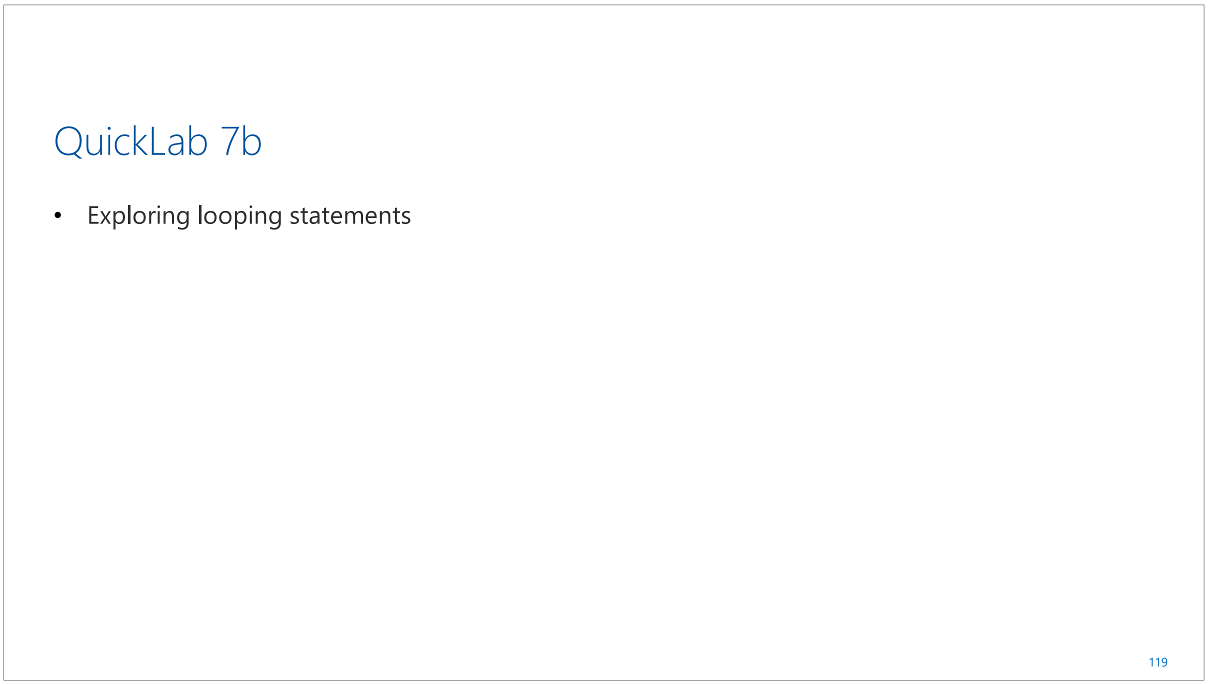
Using `continue` in a `for` loop causes execution to immediately jump to *loop expression* before then re-evaluating the *condition* and hence will effectively jump to the next iteration of the loop.



Review

- Flow of control and loops are the basis of programming
  - Along with operators
- If statements allow conditional logic
- Loops allow reuse of code without repetition

118



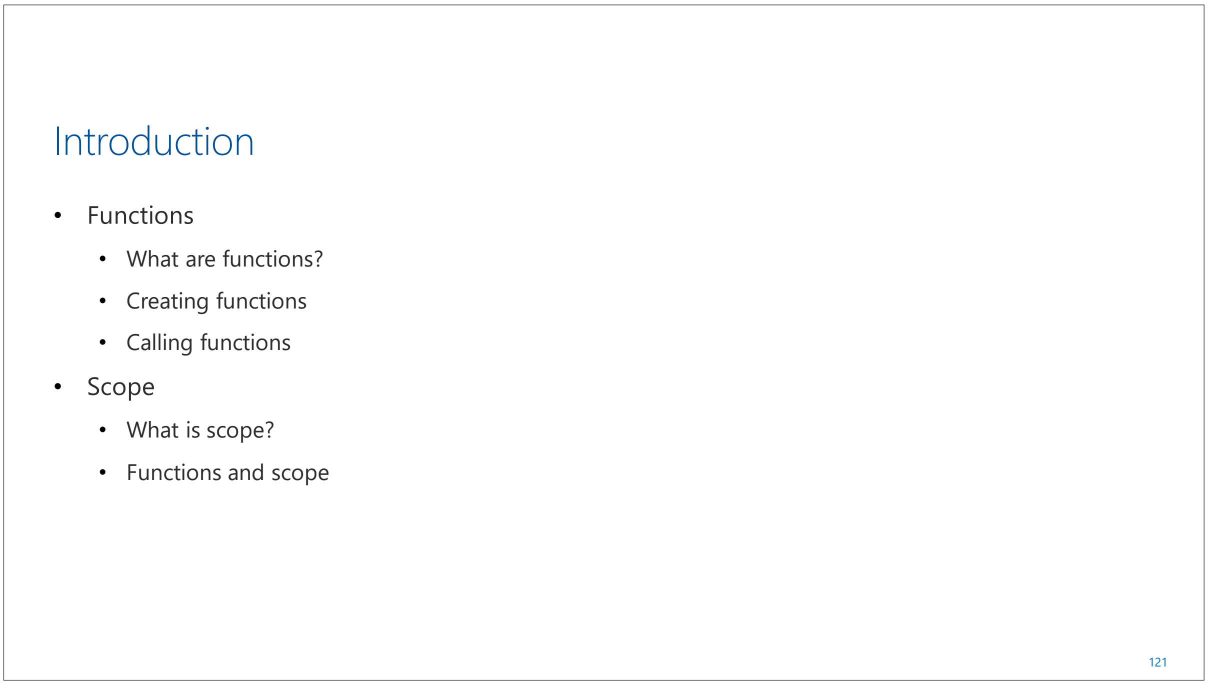
QuickLab 7b

- Exploring looping statements

119

Functions

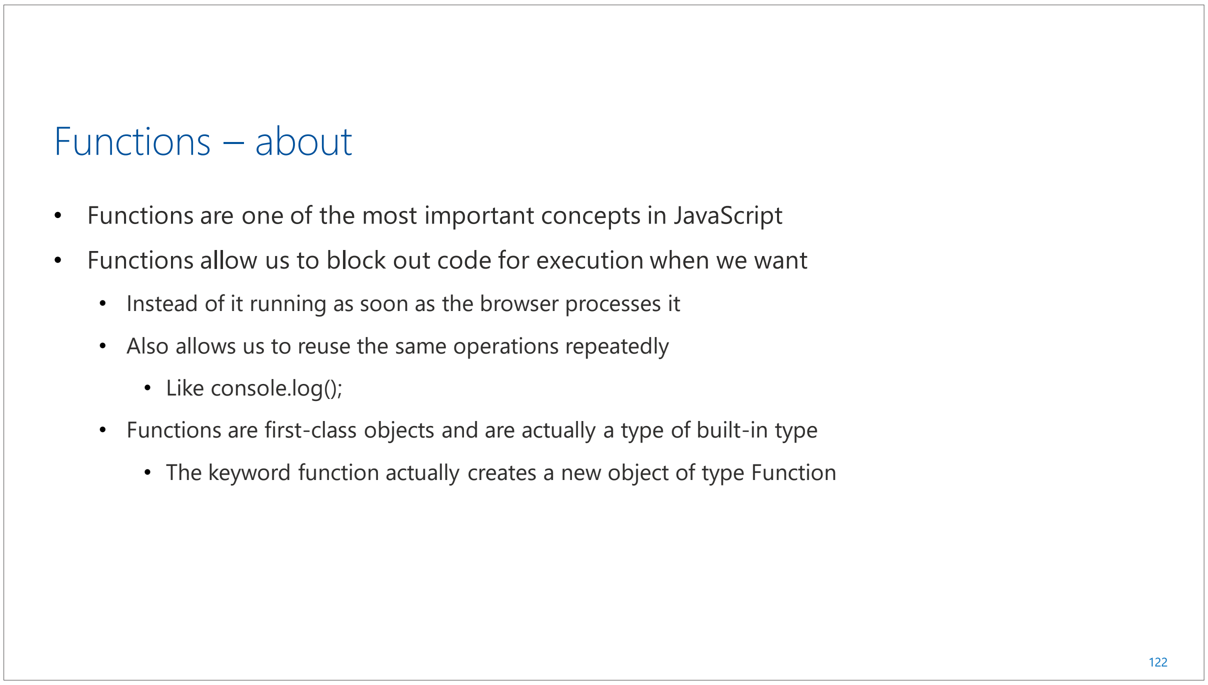
JAVASCRIPT FUNDAMENTALS



Introduction

- Functions
  - What are functions?
  - Creating functions
  - Calling functions
- Scope
  - What is scope?
  - Functions and scope

121

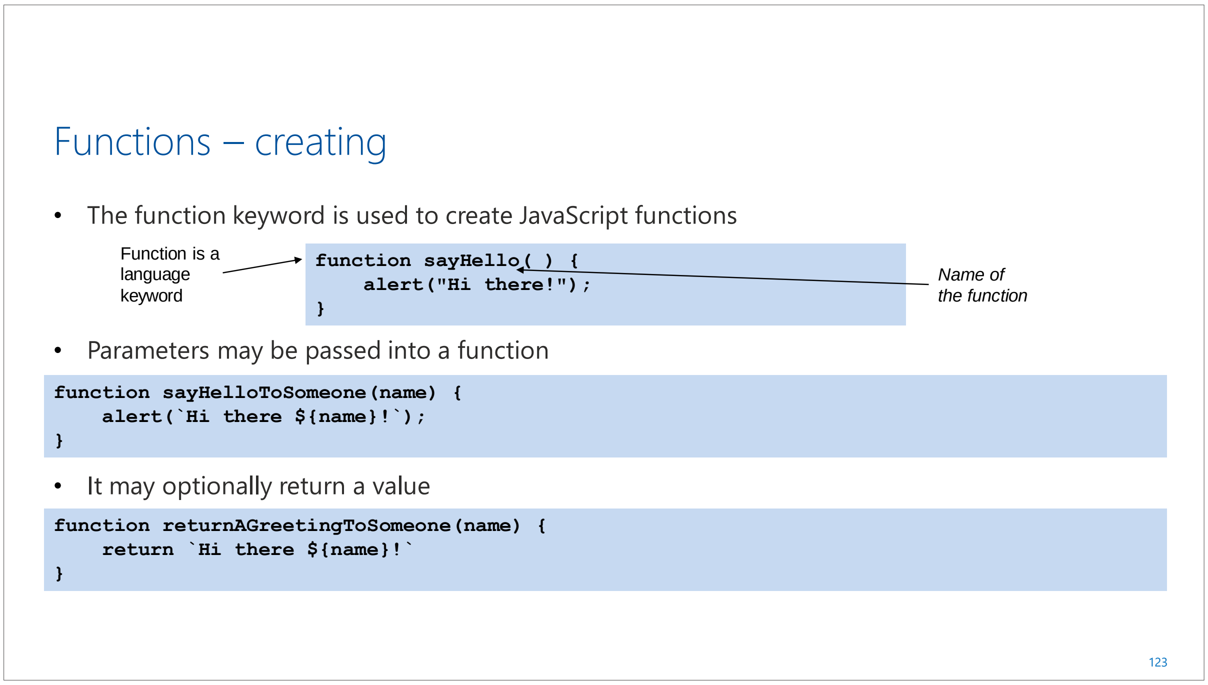


Functions – about

- 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

Function is a language keyword

```
function sayHello( ) {
    alert("Hi there!");
}
```

Name of the function

- 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}!`
}
```

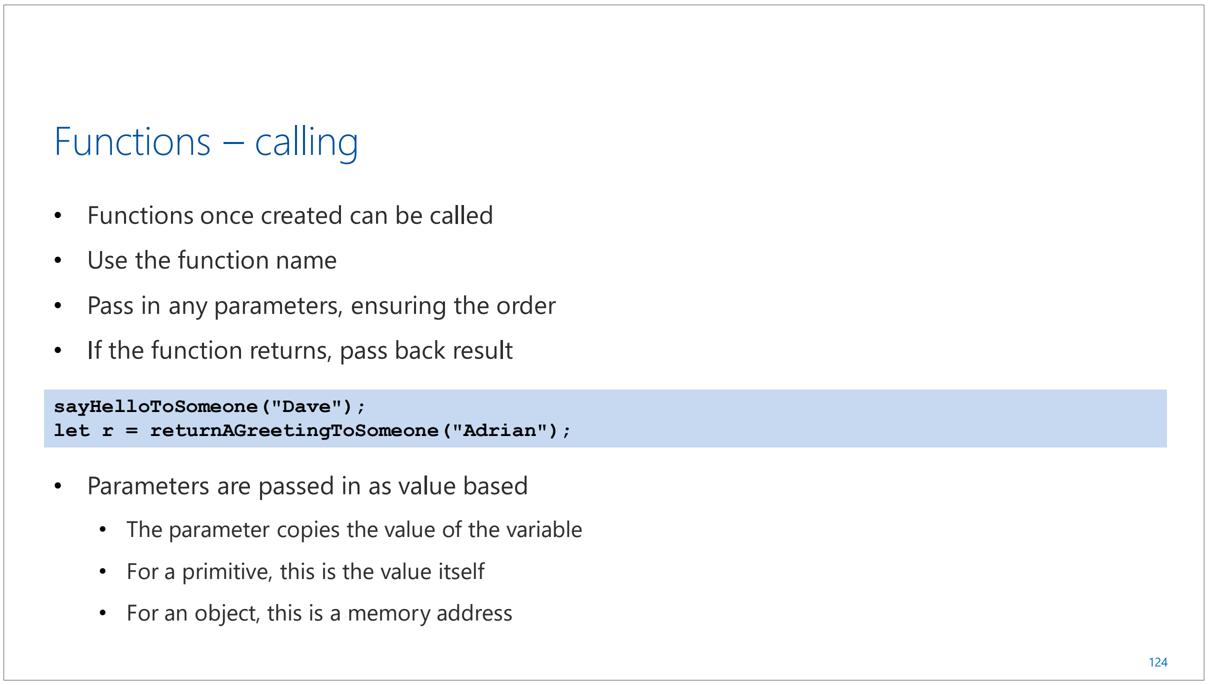
123

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

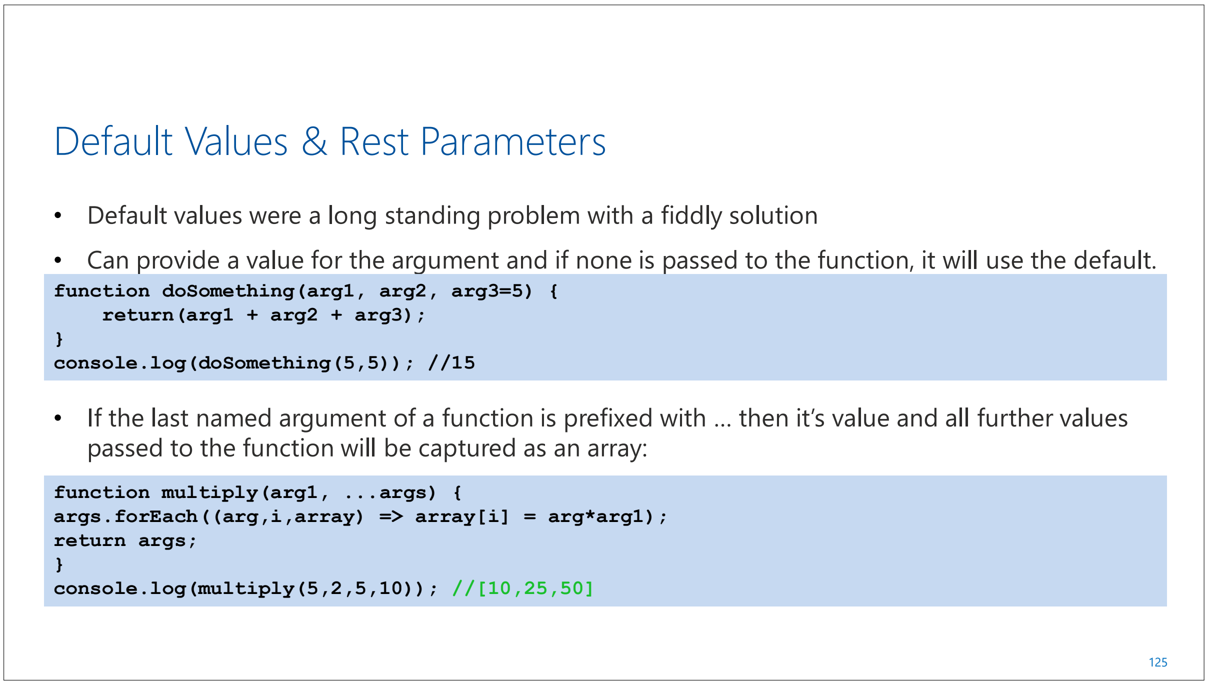
124

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:

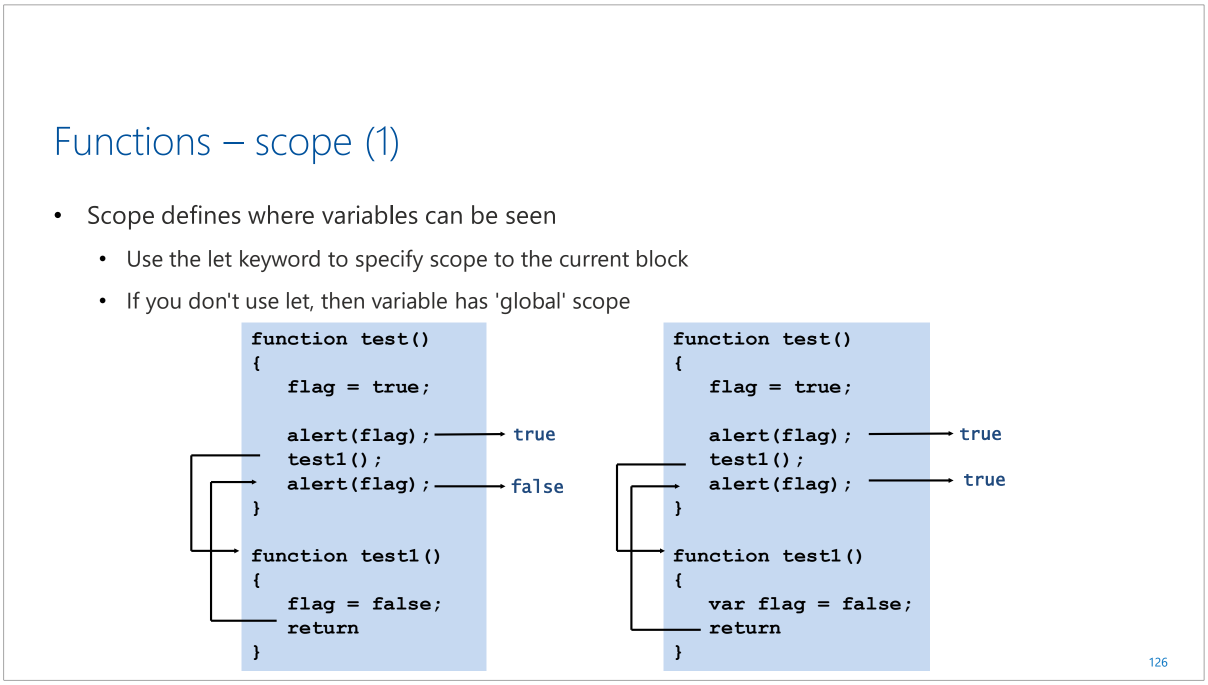
```
function multiply(arg1, ...args) {
args.forEach((arg,i,array) => array[i] = arg*arg1);
return args;
}
console.log(multiply(5,2,5,10)); //[10,25,50]
```

125

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

```
function test()
{
    flag = true;

    alert(flag);
    test1();
    alert(flag);
}

function test1()
{
    flag = false;
    return
}
```

true
false

```
function test()
{
    flag = true;

    alert(flag);
    test1();
    alert(flag);
}

function test1()
{
    var flag = false;
    return
}
```

true
true

126

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.

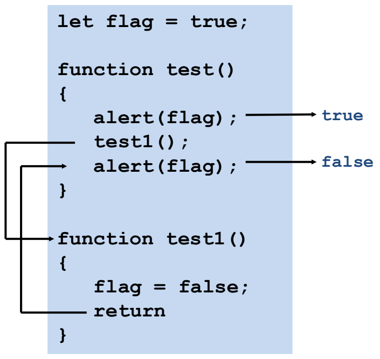
In the first example in the slide, setting the value of flag to false in function test1() has the effect of changing the value of the global variable and hence flag in function test().

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().

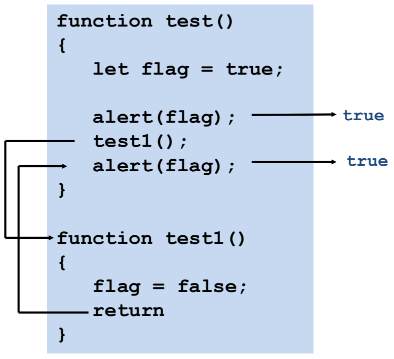
Improper use of scoping can be a source of major debugging headaches.

Functions – scope (2)

- In the code sample to the left the flag variable is explicitly defined at global level
- In the code sample to the right it is declared in the scope of test
    - Can test1 see it?

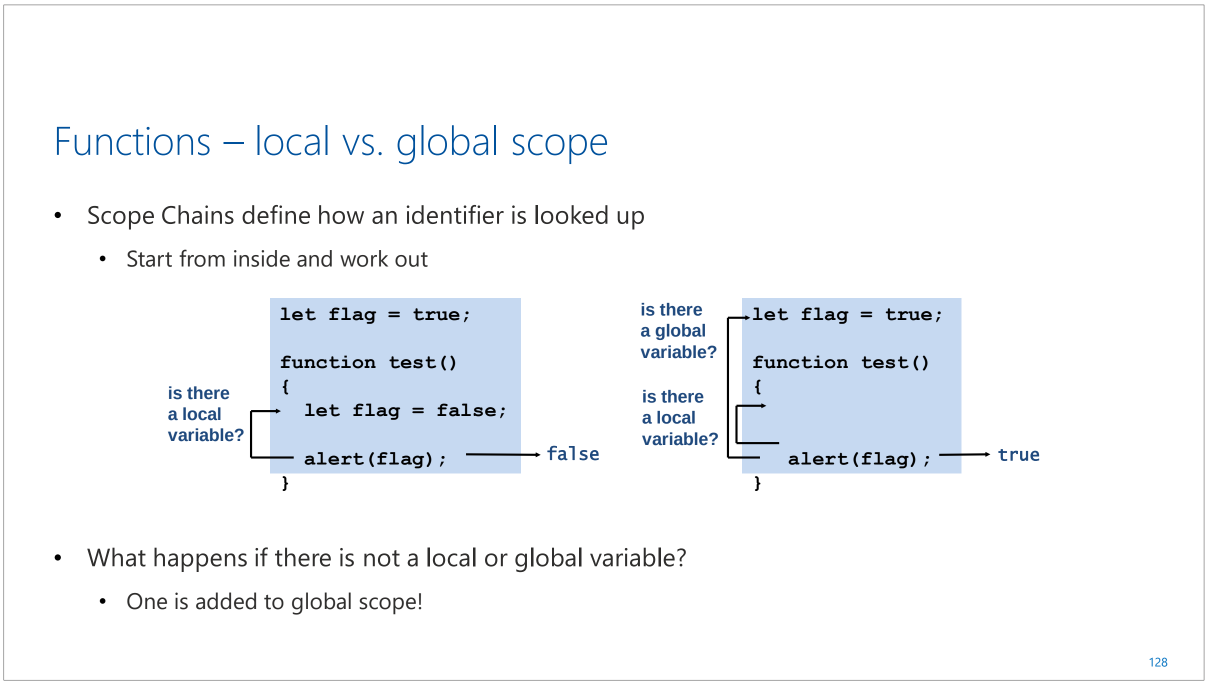


```
let flag = true;

function test()
{
    alert(flag);
    test1();
    alert(flag);
}

function test1()
{
    flag = false;
    return
}
```



```
function test()
{
    let flag = true;

    alert(flag);
    test1();
    alert(flag);
}

function test1()
{
    flag = false;
    return
}
```

127



Functions – local vs. global scope

- Scope Chains define how an identifier is looked up
  - Start from inside and work out

```
let flag = true;

function test()
{
  let flag = false;

  alert(flag);
}
```

is there a local variable?
false

```
let flag = true;

function test()
{


   alert(flag);
}
```

is there a global variable?
is there a local variable?
true

- What happens if there is not a local or global variable?
  - One is added to global scope!

128

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

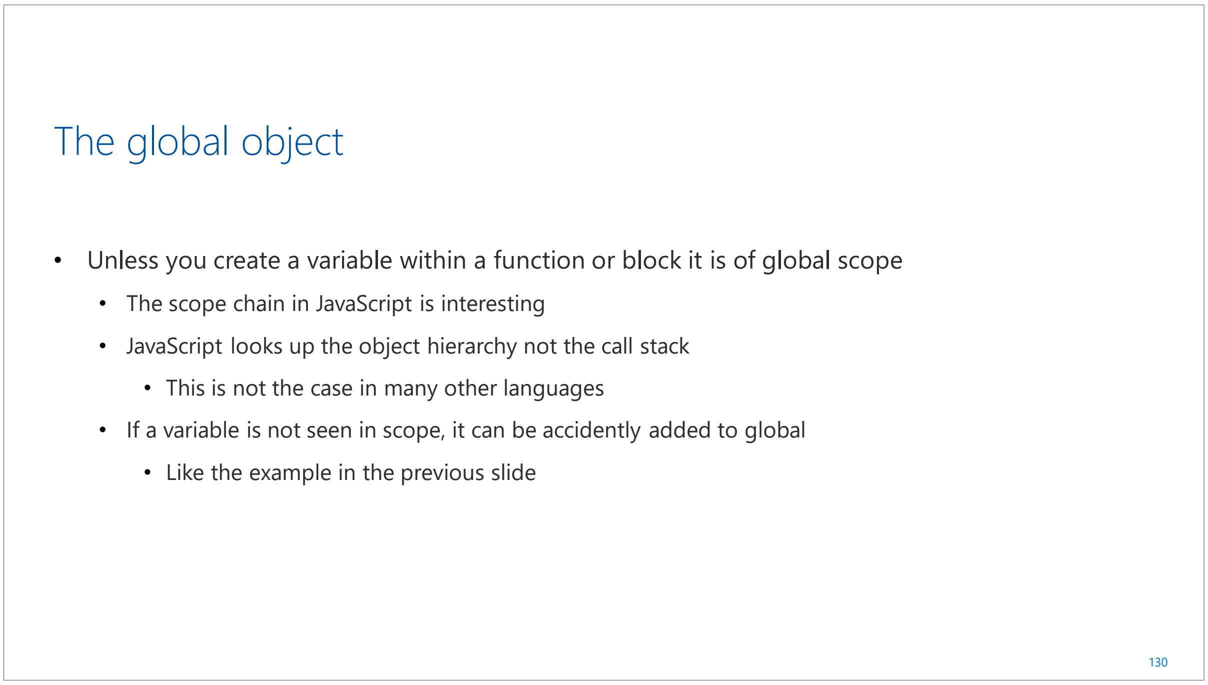
```
var a = 7;
alert(b);
```

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

← **These are equivalent**

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

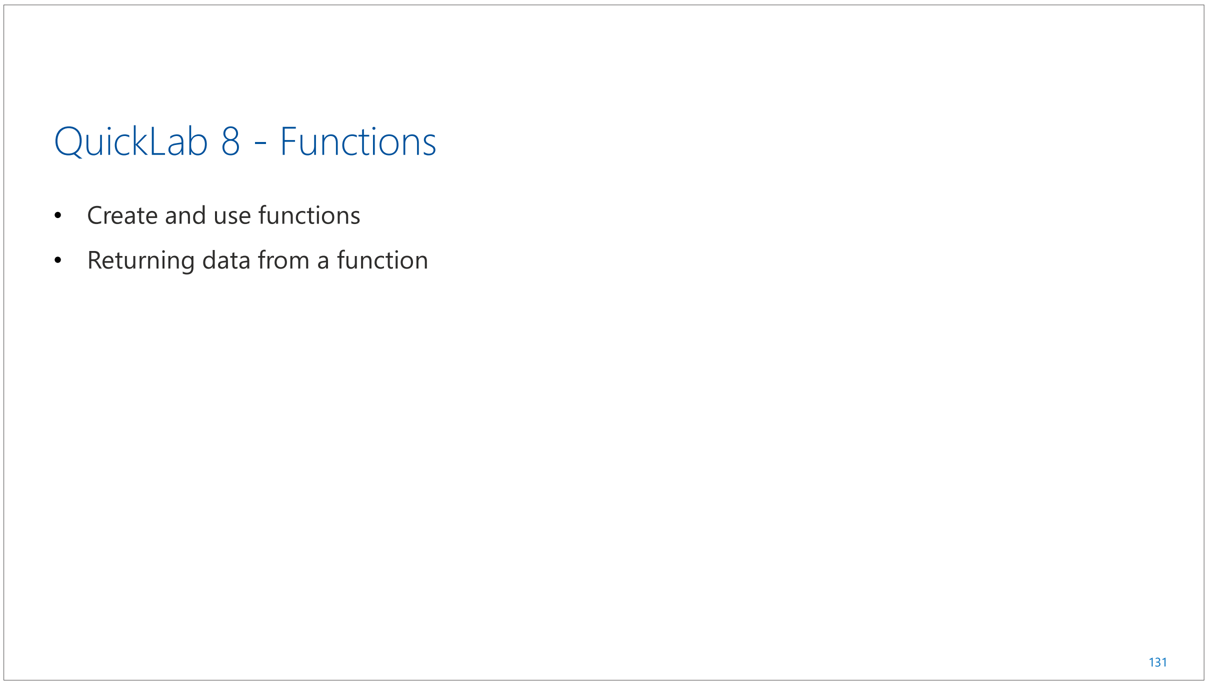
129



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

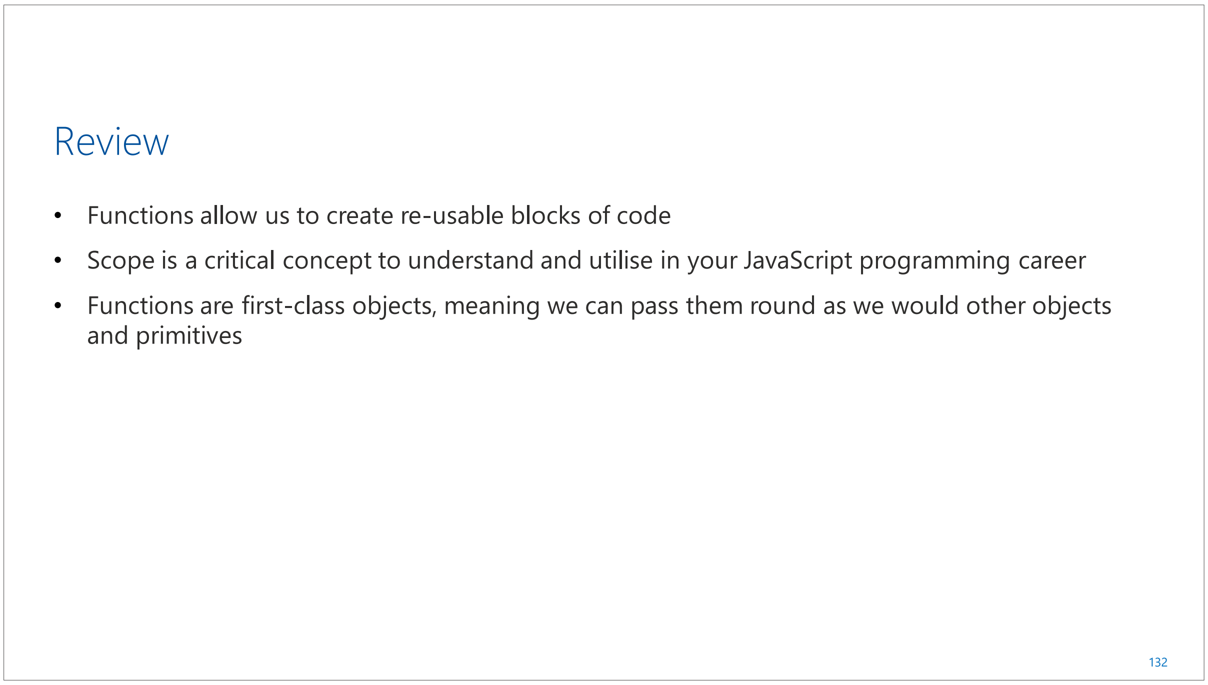
130



QuickLab 8 - Functions

- Create and use functions
- Returning data from a function

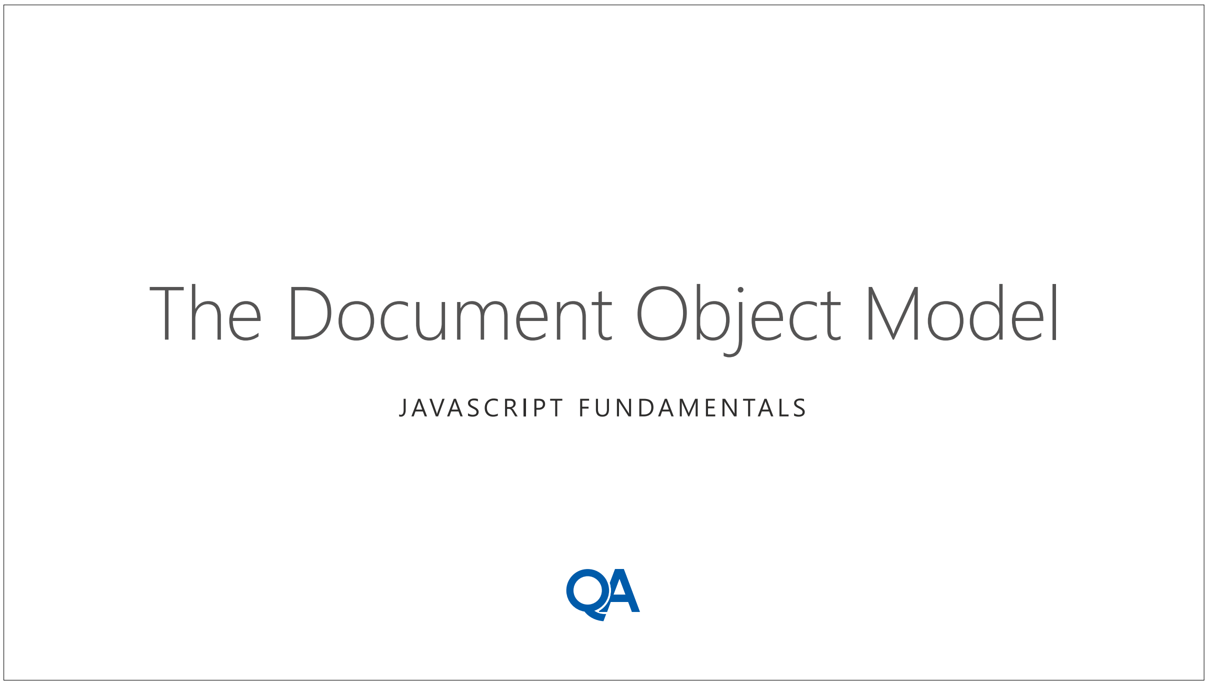
131



Review

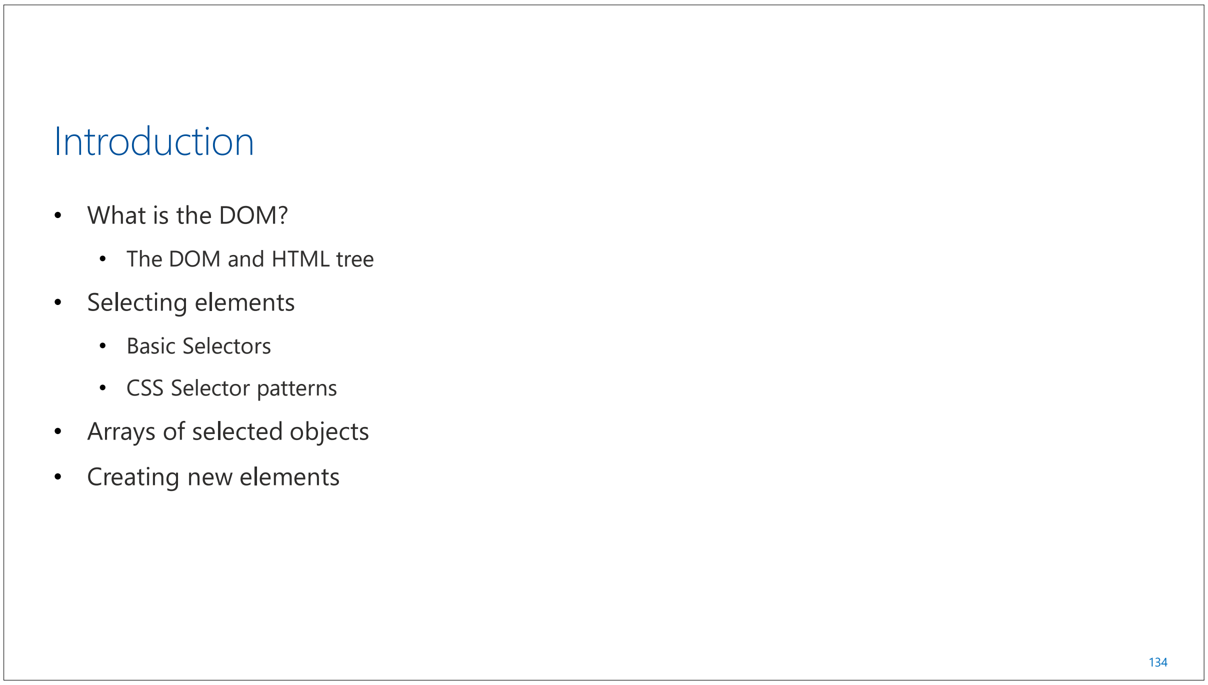
- Functions allow us to create re-usable blocks of code
- Scope is a critical concept to understand and utilise in your JavaScript programming career
- Functions are first-class objects, meaning we can pass them round as we would other objects and primitives

132



The Document Object Model

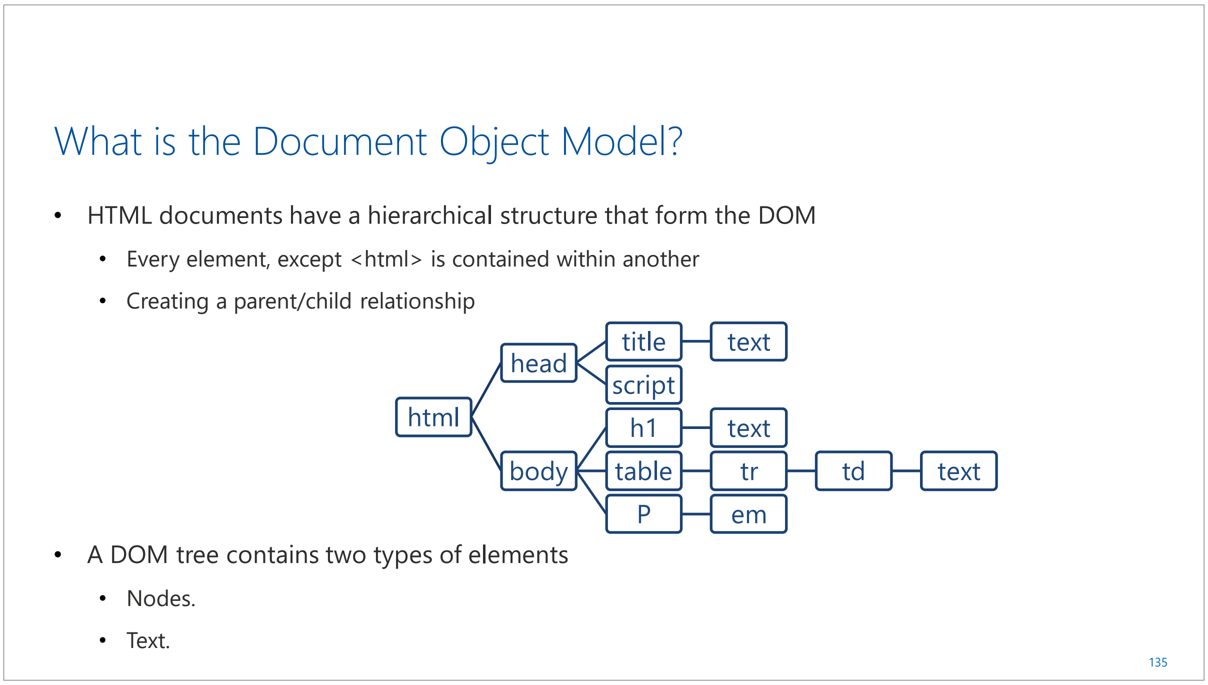
JAVASCRIPT FUNDAMENTALS



Introduction

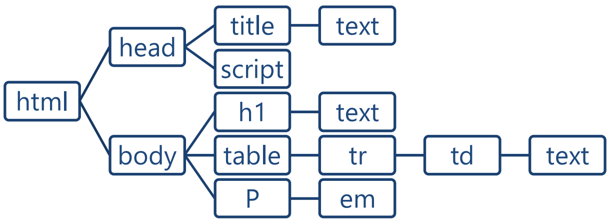
- What is the DOM?
  - The DOM and HTML tree
- Selecting elements
  - Basic Selectors
  - CSS Selector patterns
- Arrays of selected objects
- Creating new elements

134



What is the Document Object Model?

- HTML documents have a hierarchical structure that form the DOM
  - Every element, except <html> is contained within another
  - Creating a parent/child relationship



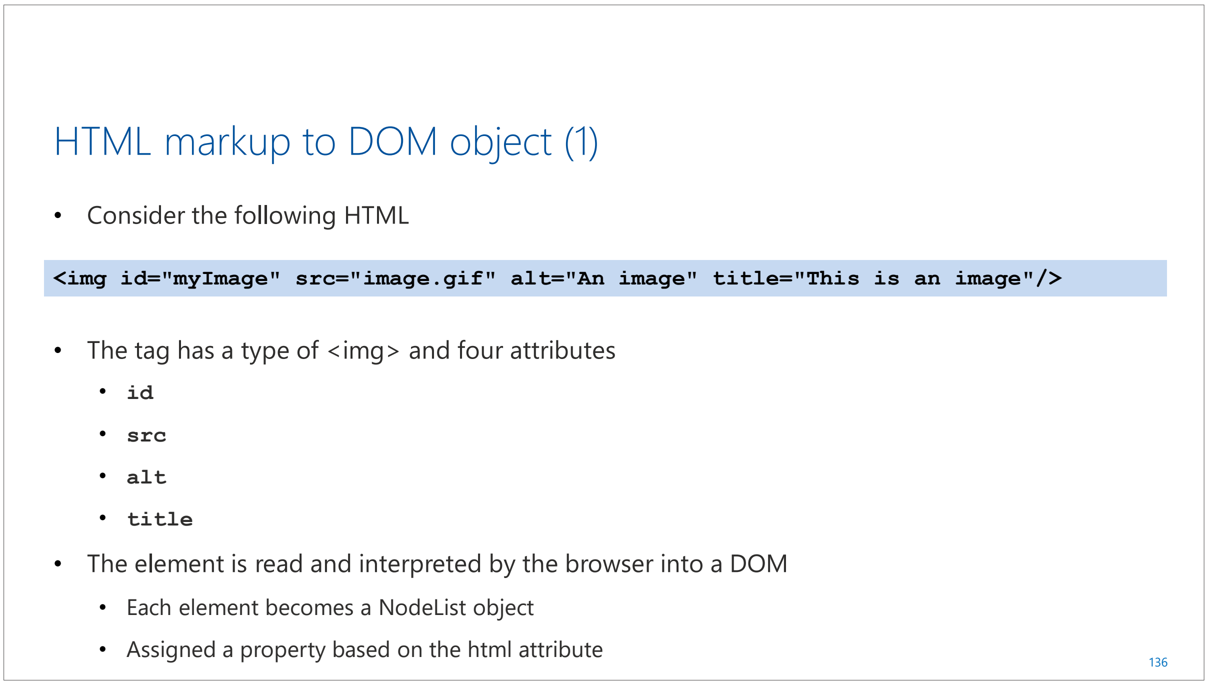
- A DOM tree contains two types of elements
  - Nodes.
  - Text.

135

HTML documents have what is called a hierarchical structure. Each element (or tag) except the top <html> tag is contained in another element, its parent. This element can in turn contain child elements. You can visualise this as a kind of organisational chart or family tree.

HTML has valid rules for how this DOM fits together and renders the markup delivered from the server into a client side DOM. For all intents and purposes, the DOM is a complex array.

JavaScript provides a series of inbuilt functions to manipulate the structure of the HTML in your browser. It is fair to say that nothing rendered in the page is safe from your grubby little mitts as you start to traverse the page with JavaScript. You can manipulate existing items and create new ones.



HTML markup to DOM object (1)

- Consider the following HTML

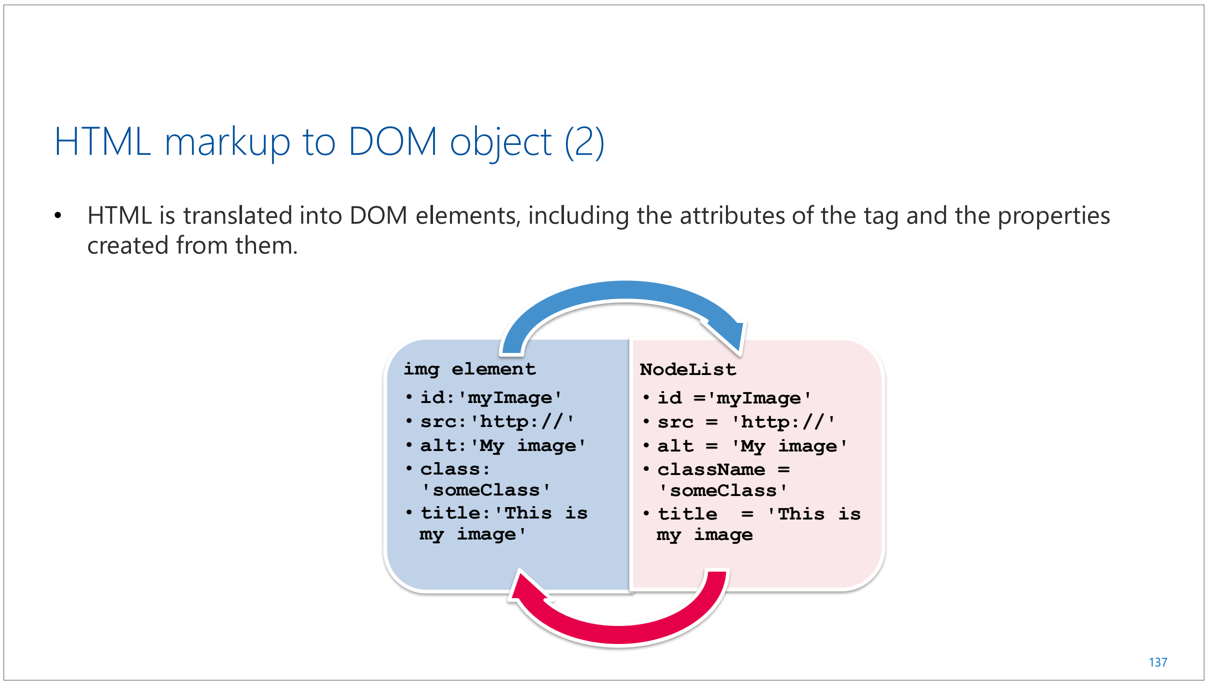
```
<img id="myImage" src="image.gif" alt="An image" title="This is an image"/>
```

- The tag has a type of <img> and four attributes
  - `id`
  - `src`
  - `alt`
  - `title`
- The element is read and interpreted by the browser into a DOM
  - Each element becomes a NodeList object
  - Assigned a property based on the html attribute

136

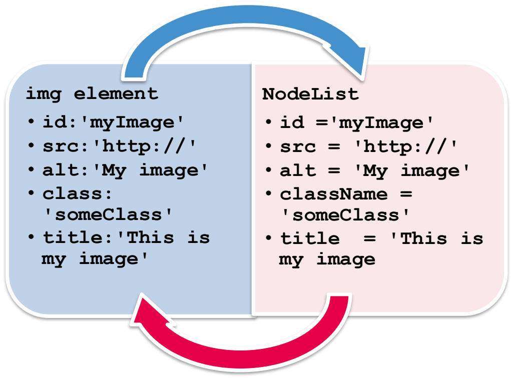
HTML gives the browser instructions on how to render the order and set the properties of the objects. Each element in HTML can have zero or more attributes assigned to it. So, in the example above each element has properties and attributes assigned to it.

These properties and attributes are initially assigned to the DOM elements as a result of parsing their HTML markup and can be changed dynamically under script control. To make sure we have our terminology and concepts straight, consider the following HTML markup for an image element:



HTML markup to DOM object (2)

- HTML is translated into DOM elements, including the attributes of the tag and the properties created from them.



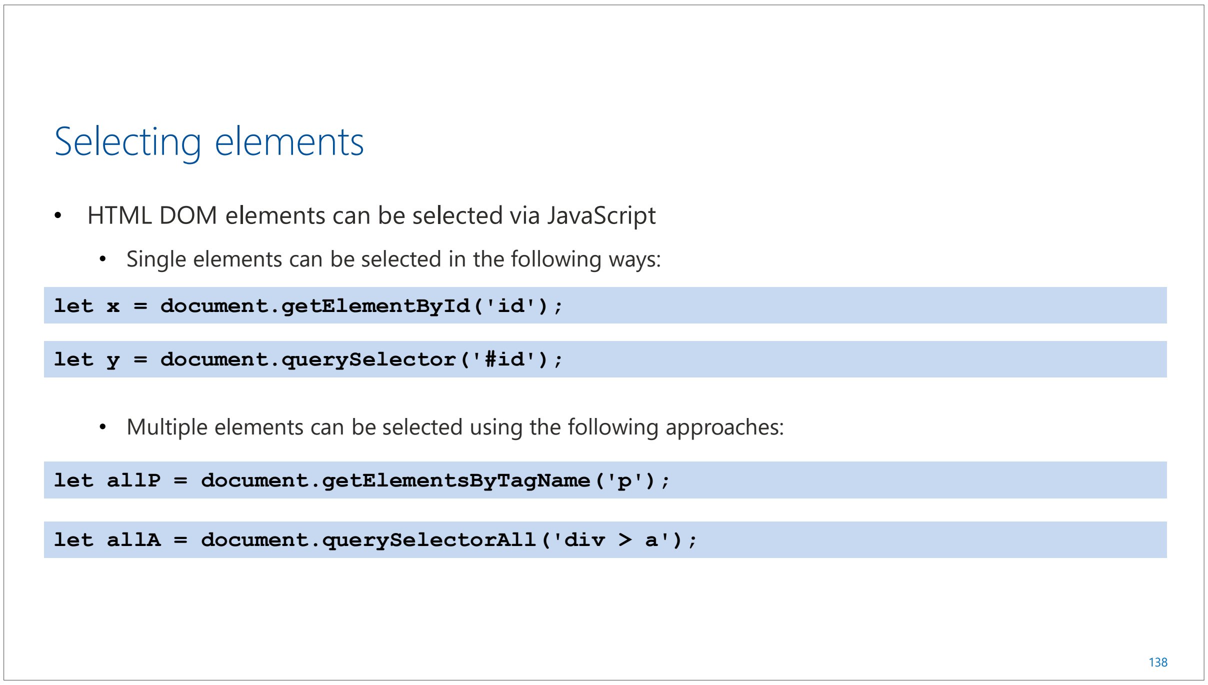
```
img element
• id:'myImage'
• src:'http://'
• alt:'My image'
• class:
  'someClass'
• title:'This is
  my image'
```

```
NodeList
• id ='myImage'
• src = 'http://'
• alt = 'My image'
• className =
  'someClass'
• title  = 'This is
  my image
```

137

For the most part, the name of a JavaScript attribute property matches that of any corresponding attribute, but there are some cases where they differ. For example, the class attribute in this example is represented by the className attribute property.

Every element mapped into the DOM is, in fact, an object literal of key value pairs as we discussed in the previous chapter. If we can select the element we want from the DOM, we can manipulate it in exactly the same way.



Selecting elements

- HTML DOM elements can be selected via JavaScript
  - Single elements can be selected in the following ways:

```
let x = document.getElementById('id');
```

```
let y = document.querySelector('#id');
```

  - Multiple elements can be selected using the following approaches:

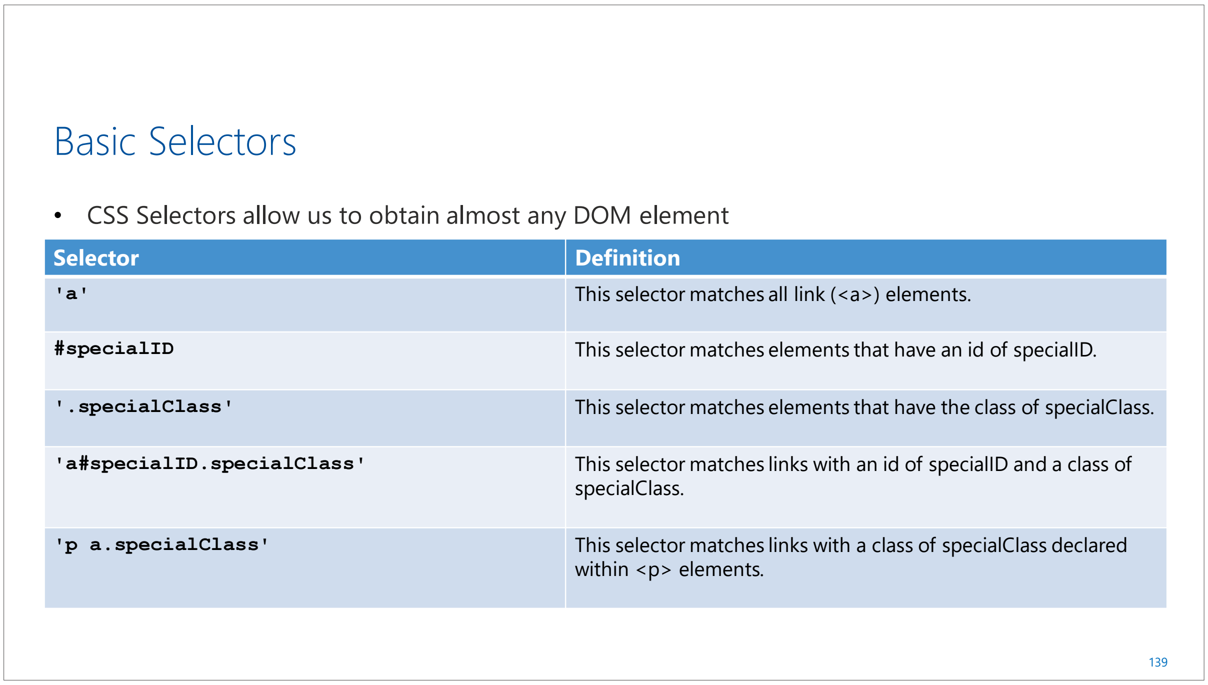
```
let allP = document.getElementsByTagName('p');
```

```
let allA = document.querySelectorAll('div > a');
```

138

To manipulate a DOM object, you need to gain access to it. To achieve this, we create a variable that points to the required object. Up until recently, this was achieved with a set of elements of the document object starting with getElement…

ECMAScript5 introduces a new way of doing things using the **querySelector** approach. These methods leverage the CSS3 selector engine. The power and flexibility of these selectors are astounding. However, legacy browsers (IE7<) do not support these methods and they will not work. We will look at strategies to deal with legacy browsers later in the course.



Basic Selectors

- CSS Selectors allow us to obtain almost any DOM element

Selector	Definition
`'a'`	This selector matches all link (<a>) elements.
`#specialID`	This selector matches elements that have an id of specialID.
`'.specialClass'`	This selector matches elements that have the class of specialClass.
`'a#specialID.specialClass'`	This selector matches links with an id of specialID and a class of specialClass.
`'p a.specialClass'`	This selector matches links with a class of specialClass declared within <p> elements.

139

For applying styles to page elements, web developers have become familiar with a small, but powerful and useful, group of selection methods that work across all browsers. Those methods include selection by an element's ID, CSS class name, tag name, and the DOM hierarchy of the page elements.

Here are some examples to give you a quick refresher:

- a – This selector matches all link (<a>) elements
- #specialID – This selector matches elements that have an id of specialID
- .specialClass – This selector matches elements that have the class of specialClass
- a#specialID.specialClass – This selector matches links with an id of specialID

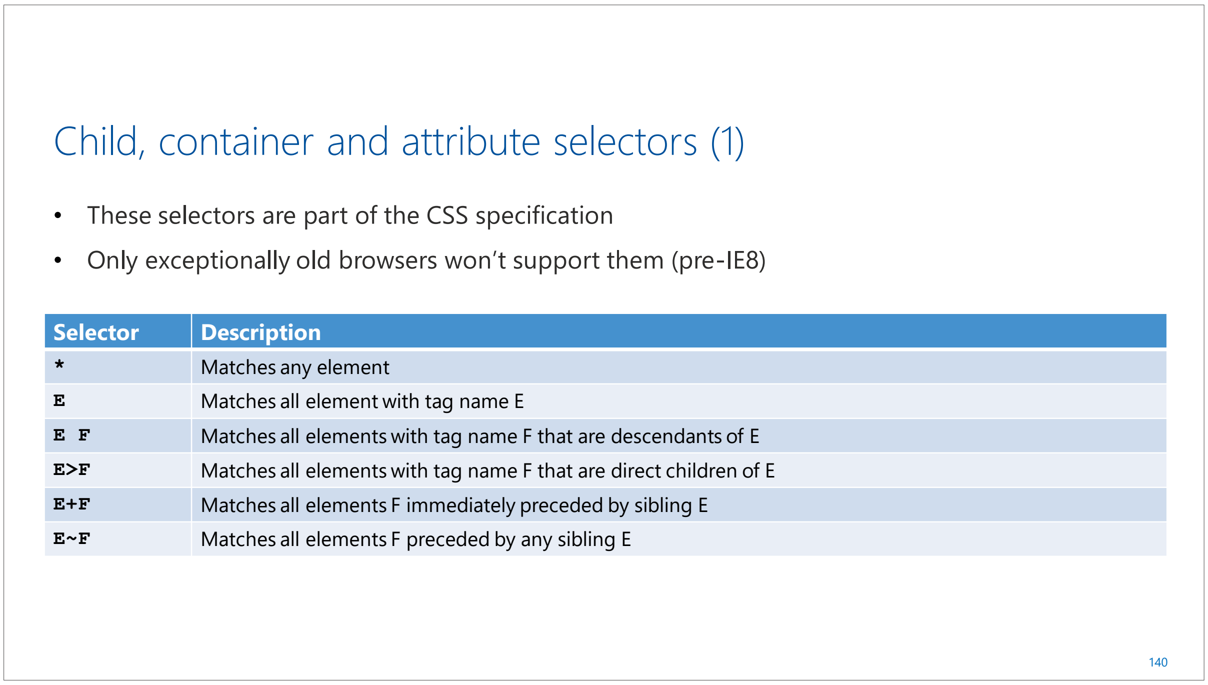
and a class of specialClass

- p a.specialClass – This selector matches links with a class of specialClass

declared within <p> elements

We can mix and match the basic selector types to select fairly fine-grained sets

of elements. In fact, the most fancy and creative websites use a combination of these basic options to create their dazzling displays.



Child, container and attribute selectors (1)

- These selectors are part of the CSS specification
- Only exceptionally old browsers won't support them (pre-IE8)

Selector	Description
`*`	Matches any element
`E`	Matches all element with tag name E
`E F`	Matches all elements with tag name F that are descendants of E
`E>F`	Matches all elements with tag name F that are direct children of E
`E+F`	Matches all elements F immediately preceded by sibling E
`E~F`	Matches all elements F preceded by any sibling E

140

For more advanced selectors, the **querySelector** and **querySelectorAll** methods use the next generation of CSS supported by Mozilla Firefox, Internet Explorer 8, Safari and other modern browsers. These advanced selectors include selecting the direct children of some elements, elements that occur after other elements in the DOM, and elements with attributes matching certain conditions.

Suppose we want to select an external hyperlink. Using basic CSS selectors, we might try something :

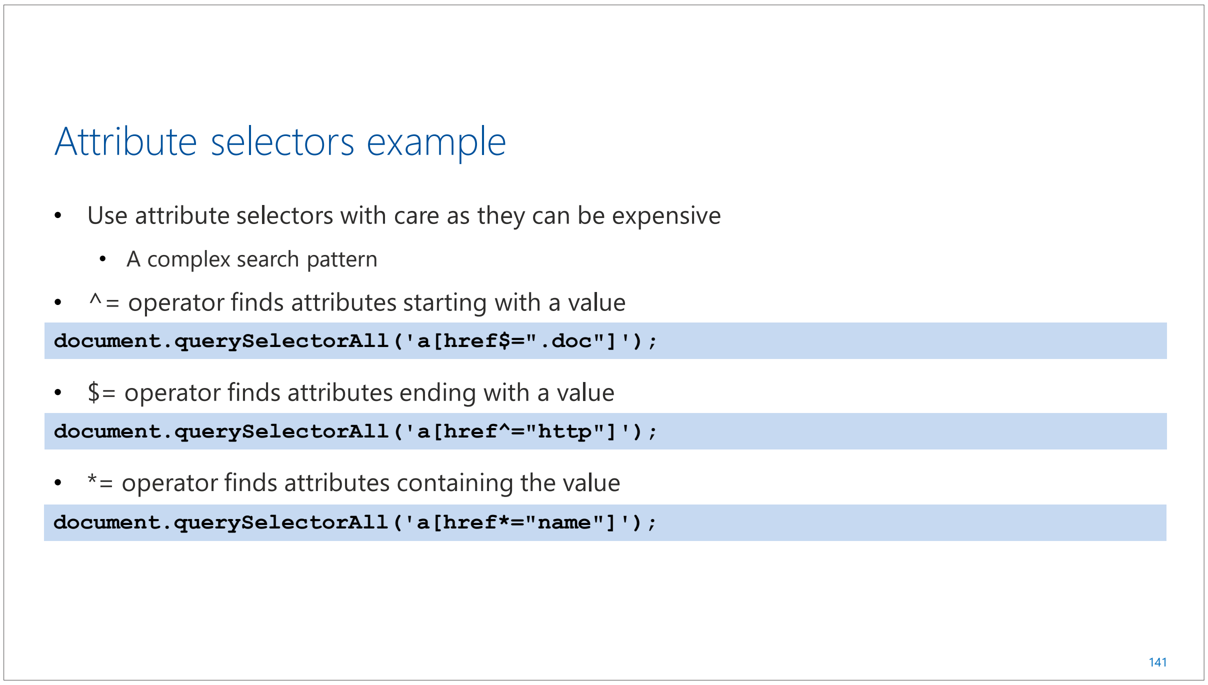
```
ul.myList li a.
```

Unfortunately, that selector would grab all links because they all descend from a list element.

A more advanced approach is to use child selectors, where a parent and its direct child are separated by the right angle bracket character (>). This selector matches only links that are direct children of an element. If a tag was further embedded, say within a <span> within a <p>, that tag would not be selected. Going back to our example, consider a selector such as:

```
ul.myList > li > a
```

This selector selects only links that are direct children of list elements, which are in turn direct children of <ul> elements that have the class myList.



Attribute selectors example

- Use attribute selectors with care as they can be expensive
  - A complex search pattern
- ^= operator finds attributes starting with a value

```
document.querySelectorAll('a[href$=".doc"]');
```

- $= operator finds attributes ending with a value

```
document.querySelectorAll('a[href^="http"]');
```

- *= operator finds attributes containing the value

```
document.querySelectorAll('a[href*="name"]');
```

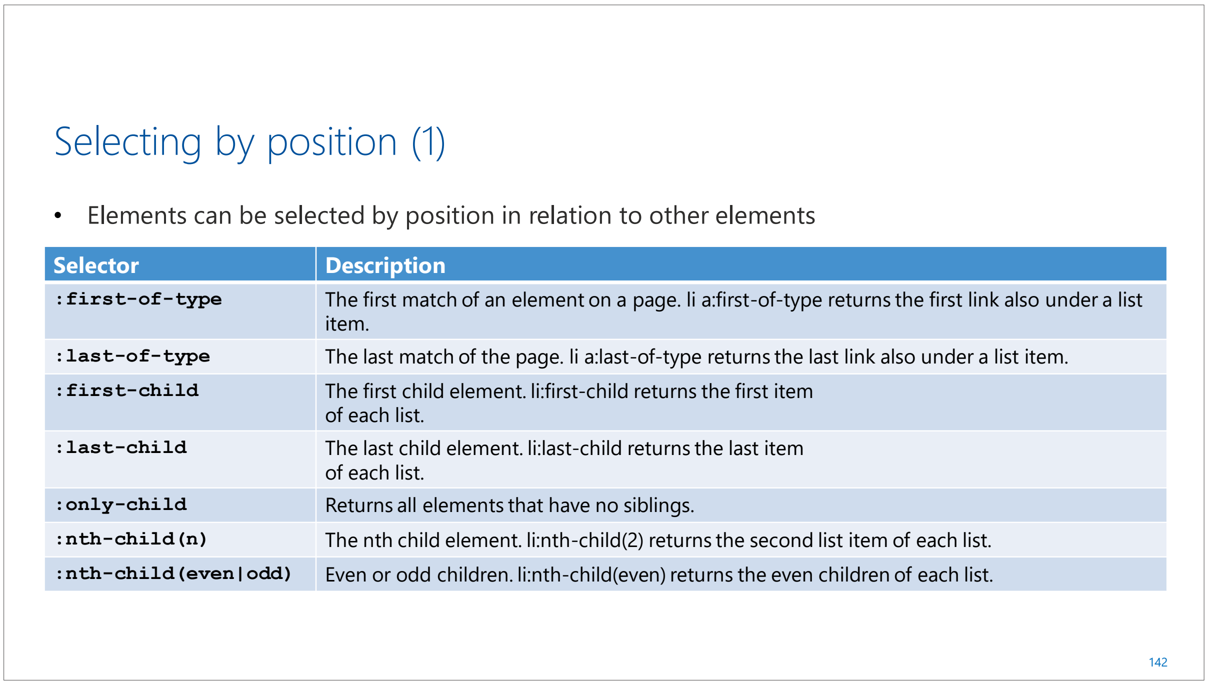
141

Attribute selectors are also extremely powerful. If we return to our example, where we wish to provide a different behaviour to links that point to external pages, we can easily achieve this.

Any external hyperlink will start with the prefix `http://` if we use the selector:

```
a[href^=http://]
```

The ^ symbol specifies that the match must occur at the beginning of the value.



Selecting by position (1)

- Elements can be selected by position in relation to other elements

Selector	Description
`:first-of-type`	The first match of an element on a page. li a:first-of-type returns the first link also under a list item.
`:last-of-type`	The last match of the page. li a:last-of-type returns the last link also under a list item.
`:first-child`	The first child element. li:first-child returns the first item of each list.
`:last-child`	The last child element. li:last-child returns the last item of each list.
`:only-child`	Returns all elements that have no siblings.
`:nth-child(n)`	The nth child element. li:nth-child(2) returns the second list item of each list.
`:nth-child(even|odd)`	Even or odd children. li:nth-child(even) returns the even children of each list.

142

Sometimes, we'll need to select elements by their position on the page or in relation to other elements. We might want to select the first link on the page, or every other paragraph, or the last list item of each list. jQuery supports mechanisms for achieving these specific selections, for example:

a:first-of-type

This format of selector matches the first <a> element on the page.

What about picking every other element?

p:nth-child(odd)

This selector matches every odd paragraph element. As we might expect, we can also specify that evenly ordered elements be selected with.

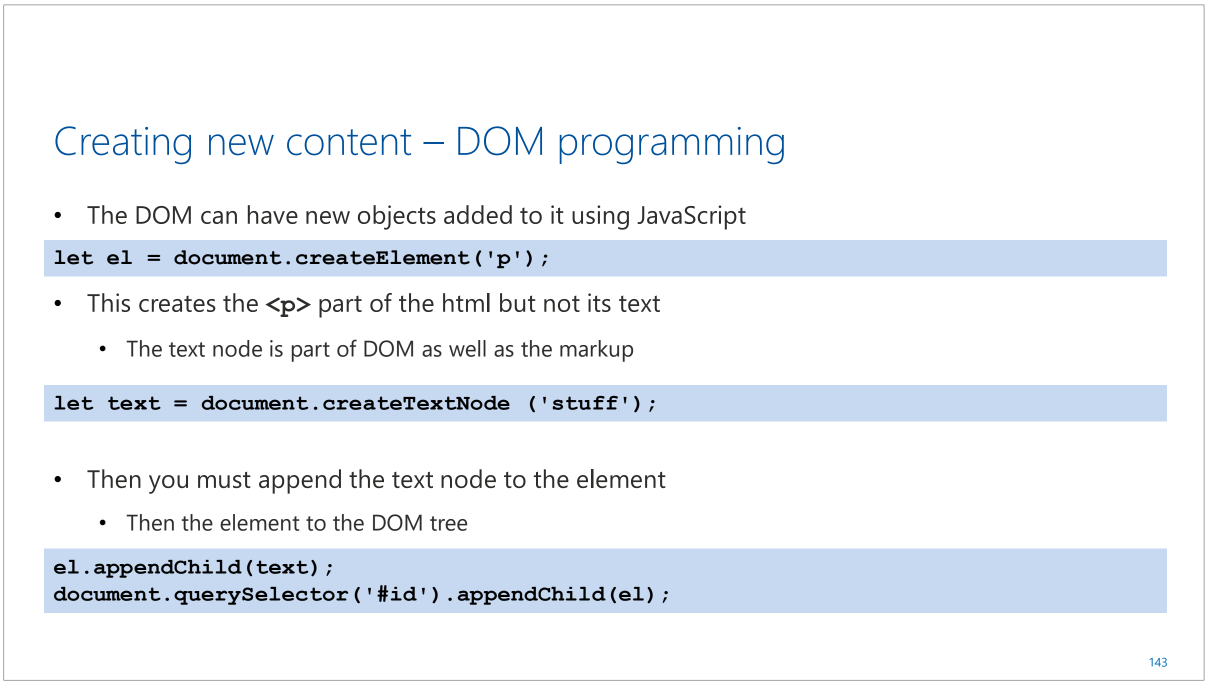
p:nth-child(even)

As above but even p elements.

li:last-child

chooses the last child of parent elements. In this example, the last <li> child of

each <ul> element is matched.



Creating new content – DOM programming

- The DOM can have new objects added to it using JavaScript

```
let el = document.createElement('p');
```

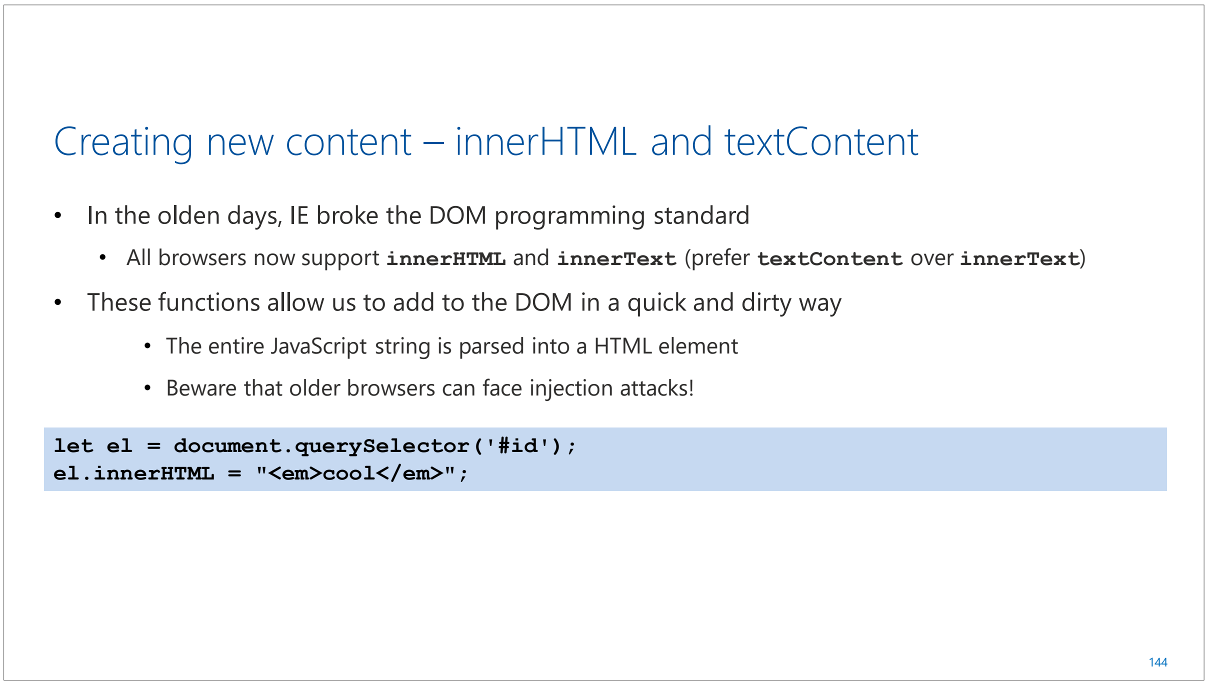
- This creates the **<p>** part of the html but not its text
  - The text node is part of DOM as well as the markup

```
let text = document.createTextNode ('stuff');
```

- Then you must append the text node to the element
  - Then the element to the DOM tree

```
el.appendChild(text);
document.querySelector('#id').appendChild(el);
```

143



Creating new content – innerHTML and textContent

- In the olden days, IE broke the DOM programming standard
  - All browsers now support **`innerHTML`** and **`innerText`** (prefer **`textContent`** over **`innerText`**)
- These functions allow us to add to the DOM in a quick and dirty way
  - The entire JavaScript string is parsed into a HTML element
  - Beware that older browsers can face injection attacks!

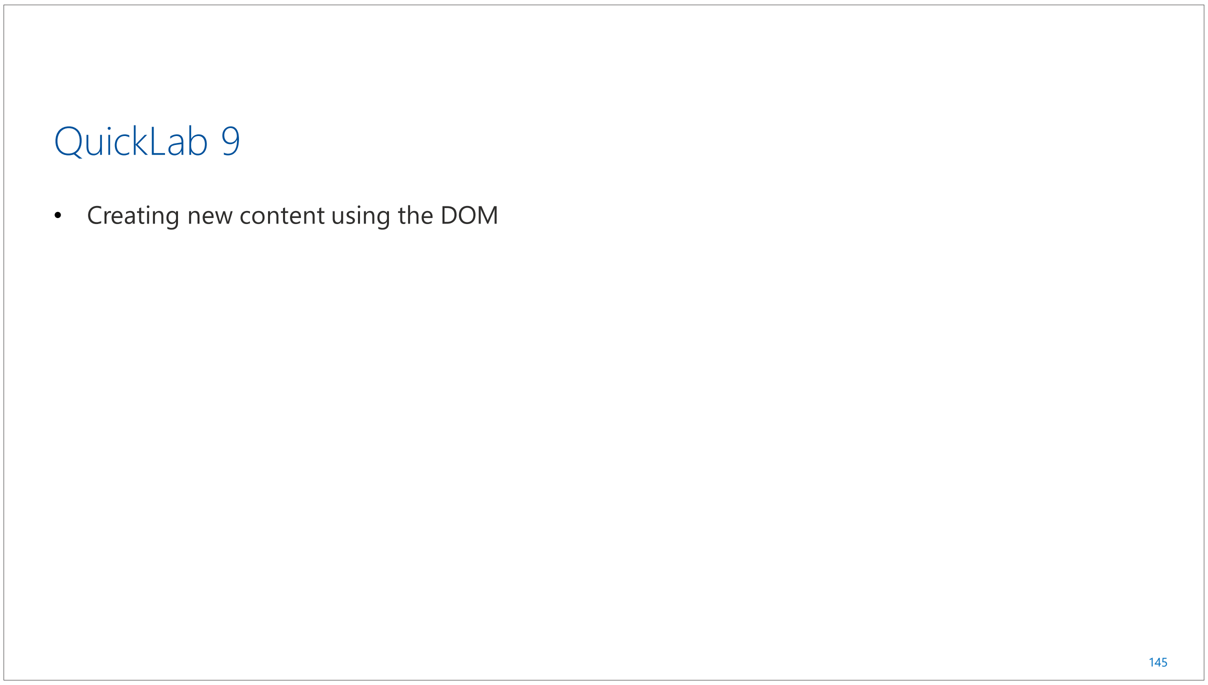
```
let el = document.querySelector('#id');
el.innerHTML = "<em>cool</em>";
```

144

From https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent

Internet Explorer introduced node.innerText. The intention is similar but with the following differences:

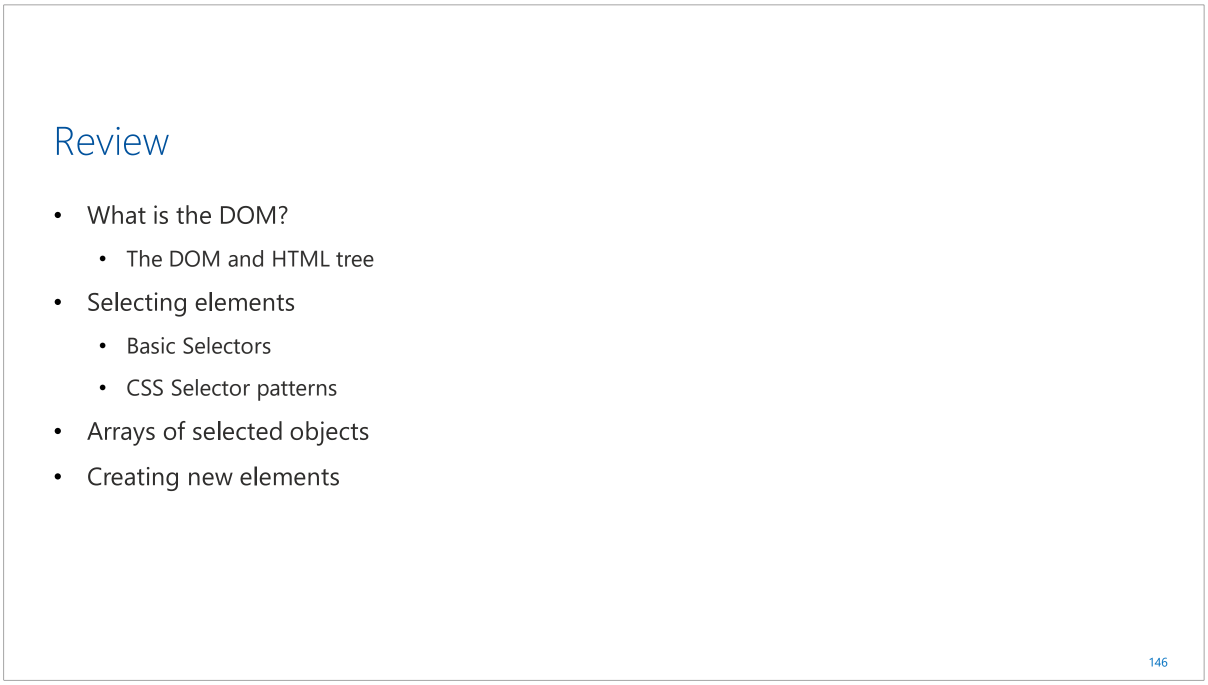
- While textContent gets the content of all elements, including <script> and <style> elements, innerText does not
- innerText is aware of style and will not return the text of hidden elements, whereas textContent will
- As innerText is aware of CSS styling, it will trigger a reflow, whereas textContentwill not
- Unlike textContent, altering innerText in Internet Explorer (up to version 11 inclusive) not only removes child nodes from the element, but also *permanently destroys* all descendant text nodes (so it is impossible to insert the nodes again into any other element or into the same element anymore)



QuickLab 9

- Creating new content using the DOM

145



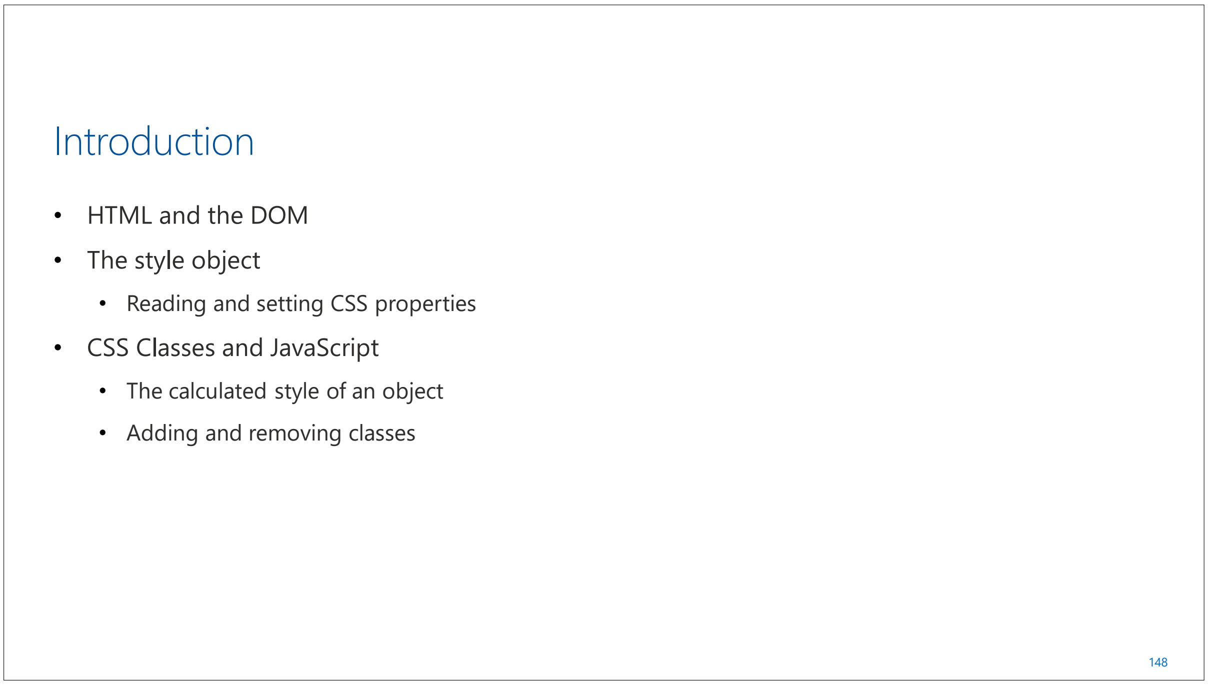
Review

- What is the DOM?
  - The DOM and HTML tree
- Selecting elements
  - Basic Selectors
  - CSS Selector patterns
- Arrays of selected objects
- Creating new elements

146

Manipulating Styles

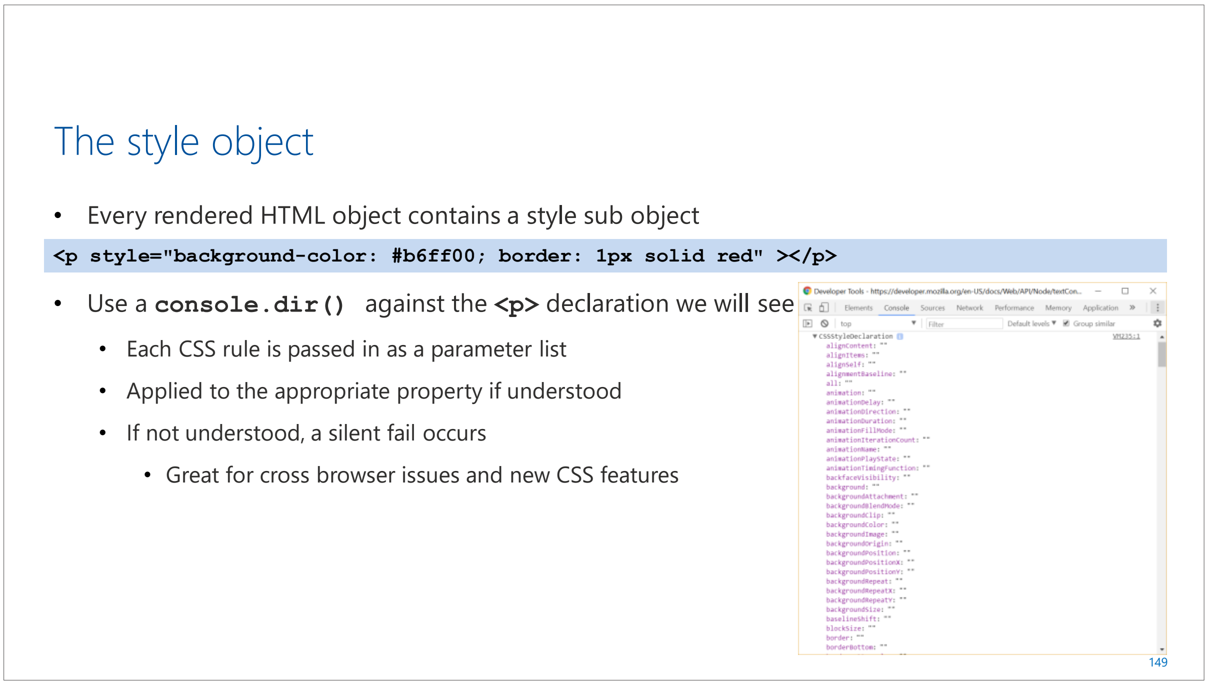
JAVASCRIPT FUNDAMENTALS



Introduction

- HTML and the DOM
- The style object
  - Reading and setting CSS properties
- CSS Classes and JavaScript
  - The calculated style of an object
  - Adding and removing classes

148

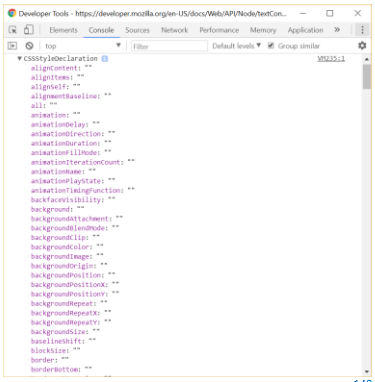


The style object

- Every rendered HTML object contains a style sub object

```
<p style="background-color: #b6ff00; border: 1px solid red" ></p>
```

- Use a **`console.dir()`** against the **`<p>`** declaration we will see
  - Each CSS rule is passed in as a parameter list
  - Applied to the appropriate property if understood
  - If not understood, a silent fail occurs
    - Great for cross browser issues and new CSS features

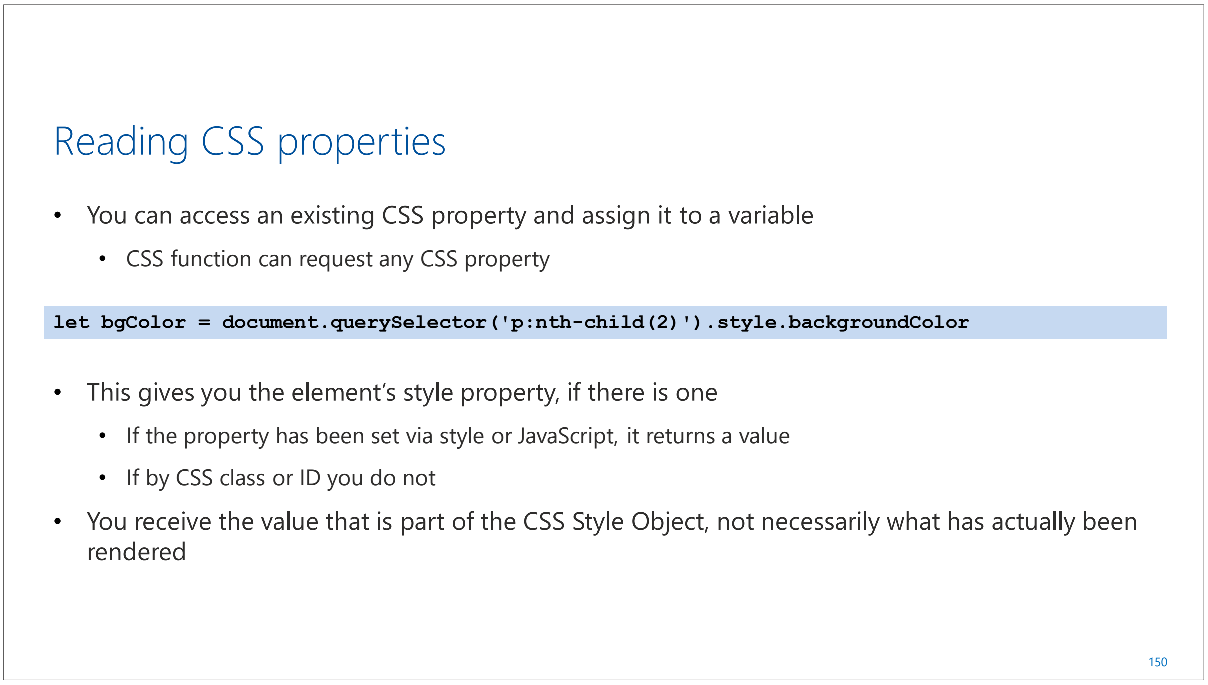


```
▼CSSStyleDeclaration                                    VM235:1
    alignContent: ""
    alignItems: ""
    alignSelf: ""
    alignmentBaseline: ""
    all: ""
    animation: ""
    animationDelay: ""
    animationDirection: ""
    animationDuration: ""
    animationFillMode: ""
    animationIterationCount: ""
    animationName: ""
    animationPlayState: ""
    animationTimingFunction: ""
    backfaceVisibility: ""
    background: ""
    backgroundAttachment: ""
    backgroundBlendMode: ""
    backgroundClip: ""
    backgroundColor: ""
    backgroundImage: ""
    backgroundOrigin: ""
    backgroundPosition: ""
    backgroundPositionX: ""
    backgroundPositionY: ""
    backgroundRepeat: ""
    backgroundRepeatX: ""
    backgroundRepeatY: ""
    backgroundSize: ""
    baselineShift: ""
    blockSize: ""
    border: ""
    borderBottom: ""
```

149

The style property of a HTML element allows us to apply an inline CSS style. The style object is an associative array of key properties and applied styles. As we will see in the next few slides, we can use JavaScript to manipulate the CSS style. Most CSS JavaScript programming occurs using this object and ensures the JavaScript manipulation will always take precedence over any CSS classes or IDs applied to the DOM object.

At execution, the CSS style or JavaScript is examined as a set of parameters and applied to the appropriate style keys. Via HTML rendering, any issues will fail silently. In our own JavaScript code, this is not the case and we should check for the property if we believe it may not be present for any reason.



Reading CSS properties

- You can access an existing CSS property and assign it to a variable
  - CSS function can request any CSS property

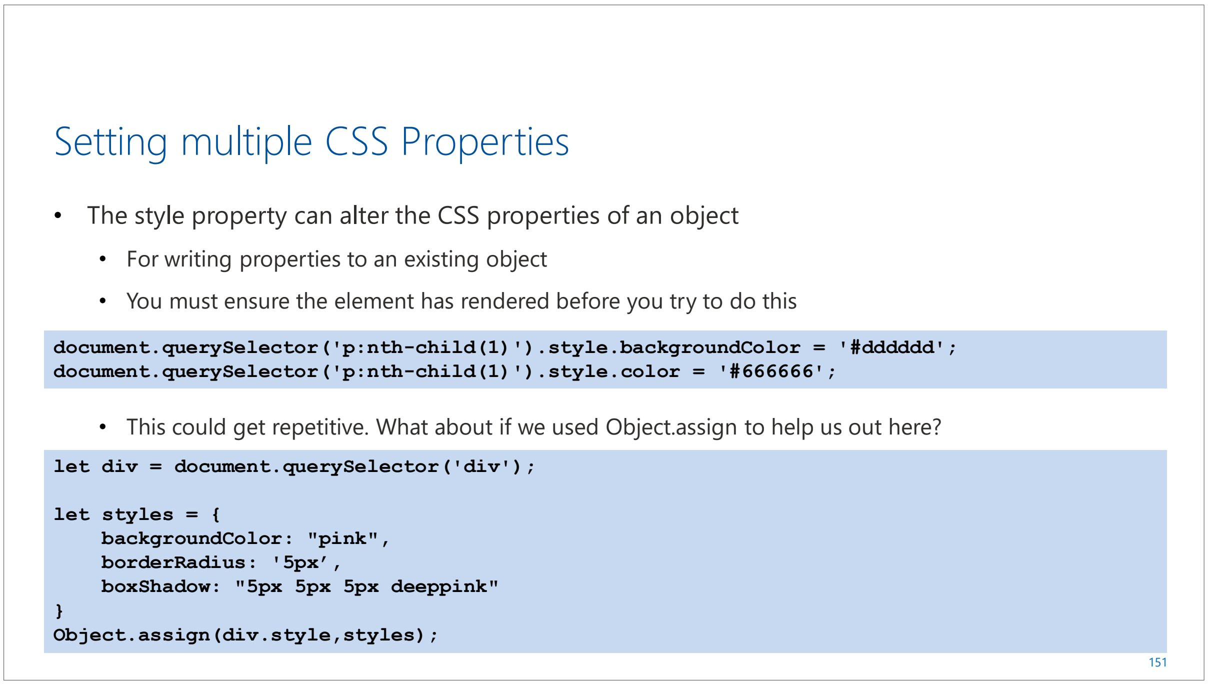
```
let bgColor = document.querySelector('p:nth-child(2)').style.backgroundColor
```

- This gives you the element’s style property, if there is one
  - If the property has been set via style or JavaScript, it returns a value
  - If by CSS class or ID you do not
- You receive the value that is part of the CSS Style Object, not necessarily what has actually been rendered

150

CSS properties of multiple elements

You can ask for a CSS property after selecting multiple elements, but this is almost always a bad idea: a function can only return a single result, so you’ll still only obtain the property for the first matched element.



Setting multiple CSS Properties

- The style property can alter the CSS properties of an object
    - For writing properties to an existing object
    - You must ensure the element has rendered before you try to do this

```
document.querySelector('p:nth-child(1)').style.backgroundColor = '#dddddd';
document.querySelector('p:nth-child(1)').style.color = '#666666';
```

- This could get repetitive. What about if we used Object.assign to help us out here?

```
let div = document.querySelector('div');

let styles = {
    backgroundColor: "pink",
    borderRadius: '5px’,
    boxShadow: "5px 5px 5px deeppink"
}
Object.assign(div.style,styles);
```

151

The advantage of using the style object is its authority in the hierarchy of CSS application – even if a CSS class is applied, this will override those rules. Each style rule must be applied as a separate function call, which can end up being very weighty in code.

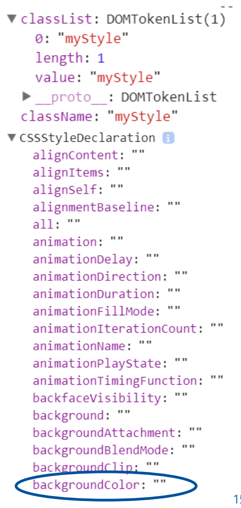
CSS classes and JavaScript

- Classes are attached to a DOM object in a slightly different way

```
.myStyle {
    background-color: #4cff00;
    border: 1px solid red;
}
```

```
<p class="myStyle">test</p>
```

- Classes render as part of a DOM element
  - Represented as an array
  - Multiple classes can be added to an element
  - Their styles do not become party of the style property



```
▼classList: DOMTokenList(1)
   0: "myStyle"
   length: 1
   value: "myStyle"
  ▶__proto__: DOMTokenList
 className: "myStyle"
▼CSSStyleDeclaration
   alignContent: ""
   alignItems: ""
   alignSelf: ""
   alignmentBaseline: ""
   all: ""
   animation: ""
   animationDelay: ""
   animationDirection: ""
   animationDuration: ""
   animationFillMode: ""
   animationIterationCount: ""
   animationName: ""
   animationPlayState: ""
   animationTimingFunction: ""
   backfaceVisibility: ""
   background: ""
   backgroundAttachment: ""
   backgroundBlendMode: ""
   backgroundClip: ""
   backgroundColor: ""
```

152

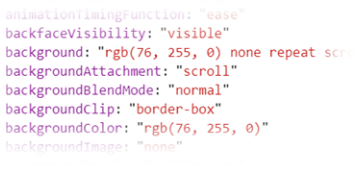
In all probability, we will be using CSS classes and in that situation, as outlined above. CSS classes apply to the DOM object, but we will not see it in the style property of the object.

Trying to retrieve the a CSS style property via JavaScript will only work when that style property has previously been set via JavaScript or when that style property was defined inline by using the style attribute in HTML. Now this doesn't do you much good if you haven't first set said CSS property in JavaScript and you still want to retrieve it via JavaScript.

Obtaining the calculated style of an object

- In most cases, we will read a CSS class from a style sheet
  - Use the **`window.getComputedStyle()`** method
  - Provides a read only final used values of the CSS
  - Returning a **`CSSStyleDeclaration`** object

```
let elem = document.querySelector('p:nth-child(1)');
let compStyle = getComputedStyle(elem).backgroundColor;
```

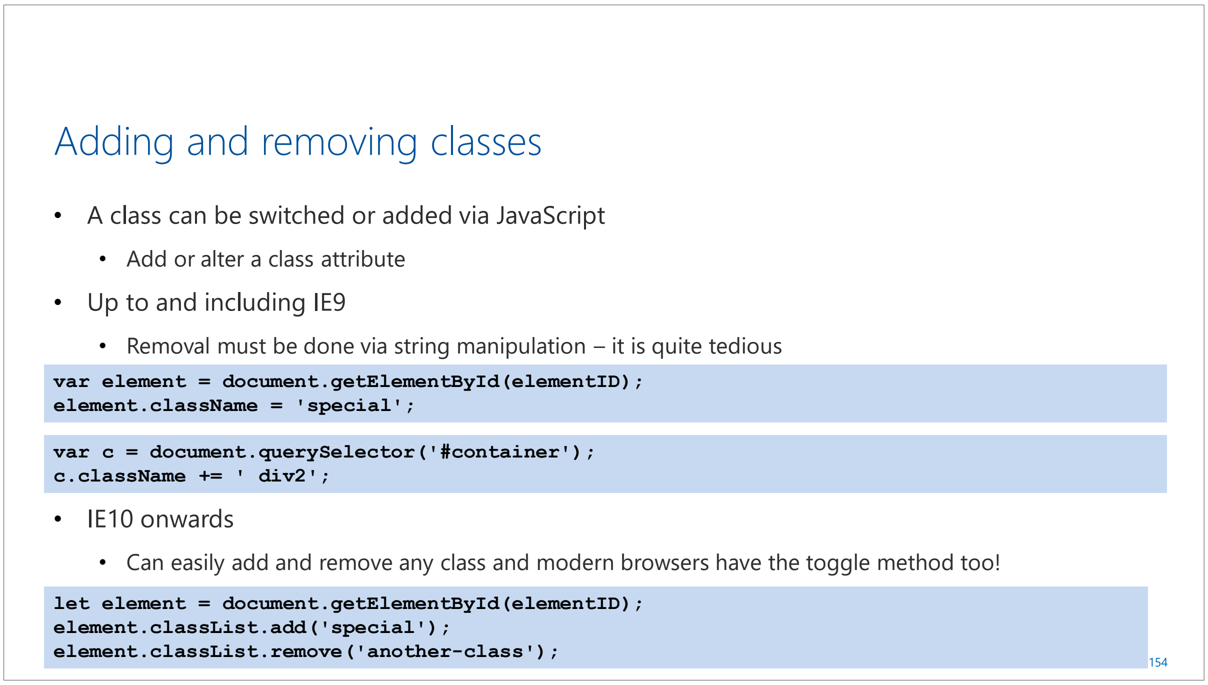


```
animationTimingFunction: "ease"
backfaceVisibility: "visible"
background: "rgb(76, 255, 0) none repeat scr
backgroundAttachment: "scroll"
backgroundBlendMode: "normal"
backgroundClip: "border-box"
backgroundColor: "rgb(76, 255, 0)"
backgroundImage: "none"
```

153

getComputedStyle() gives the final used values of all the CSS properties of an element, where the returned style is a **CSSStyleDeclaration** object.

The returned object is of the same type as that of the object returned from the element's style property; however, the two objects have different purposes. The object returned from **getComputedStyle** is read-only and can be used to inspect the element's style (including those set by a <style> element or an external stylesheet). The style object should be used to set styles on a specific element.



Adding and removing classes

- A class can be switched or added via JavaScript
  - Add or alter a class attribute
- Up to and including IE9
  - Removal must be done via string manipulation – it is quite tedious

```
var element = document.getElementById(elementID);
element.className = 'special';
```

```
var c = document.querySelector('#container');
c.className += ' div2';
```

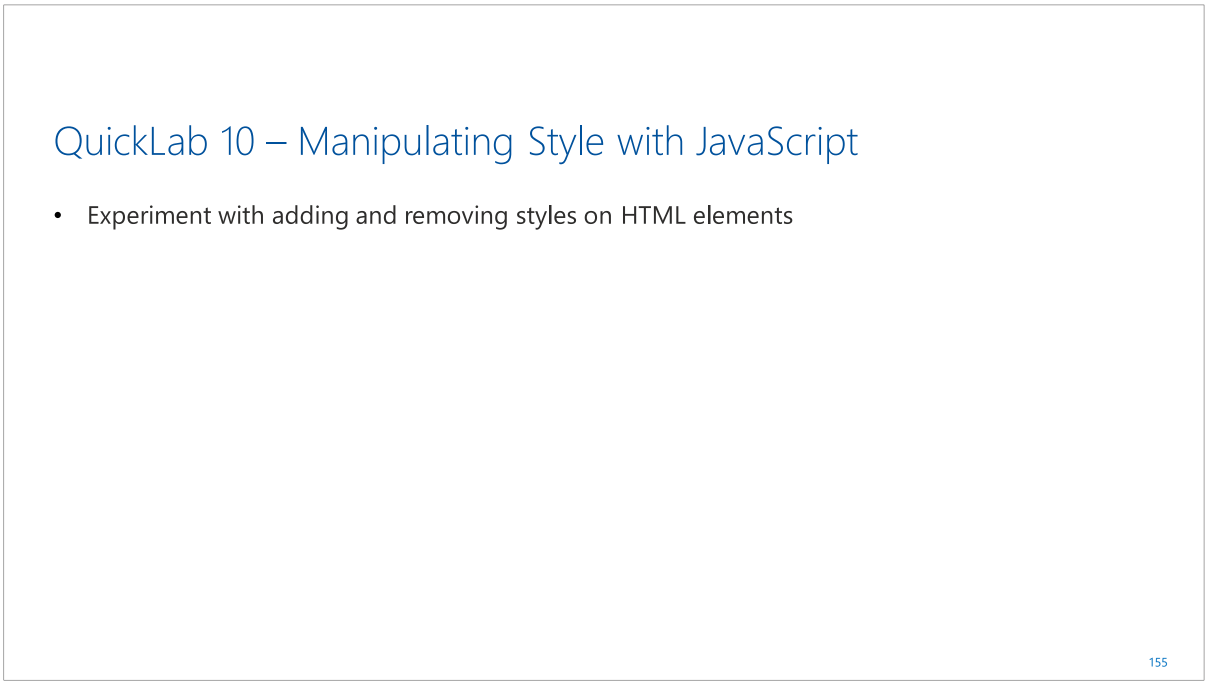
- IE10 onwards
  - Can easily add and remove any class and modern browsers have the toggle method too!

```
let element = document.getElementById(elementID);
element.classList.add('special');
element.classList.remove('another-class');
```

154

All elements can have attributes added or changed via JavaScript. The setAttribute method takes an attribute value and sets it to the new value. Any CSS class that is currently loaded into the page can be referenced in this way.

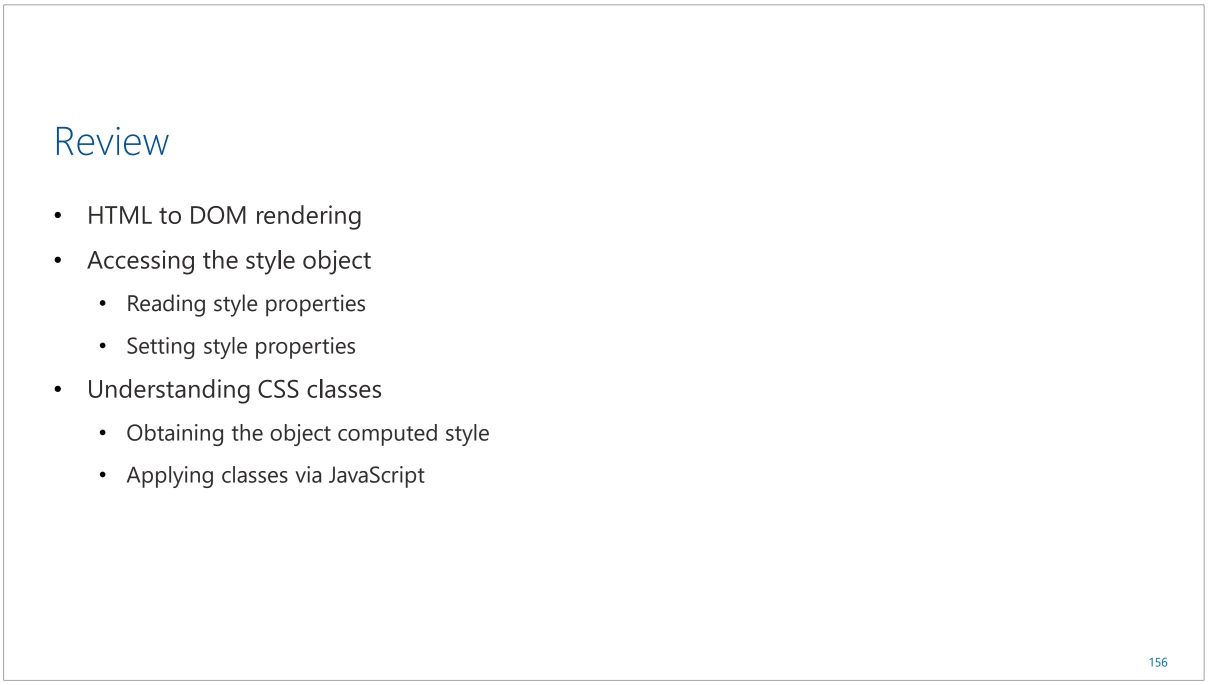
There is an alternative approach that can be used if you want to apply multiple classes. The DOM element holds all applied classes in the className property. It is simply a string value that we can append to.



QuickLab 10 – Manipulating Style with JavaScript

- Experiment with adding and removing styles on HTML elements

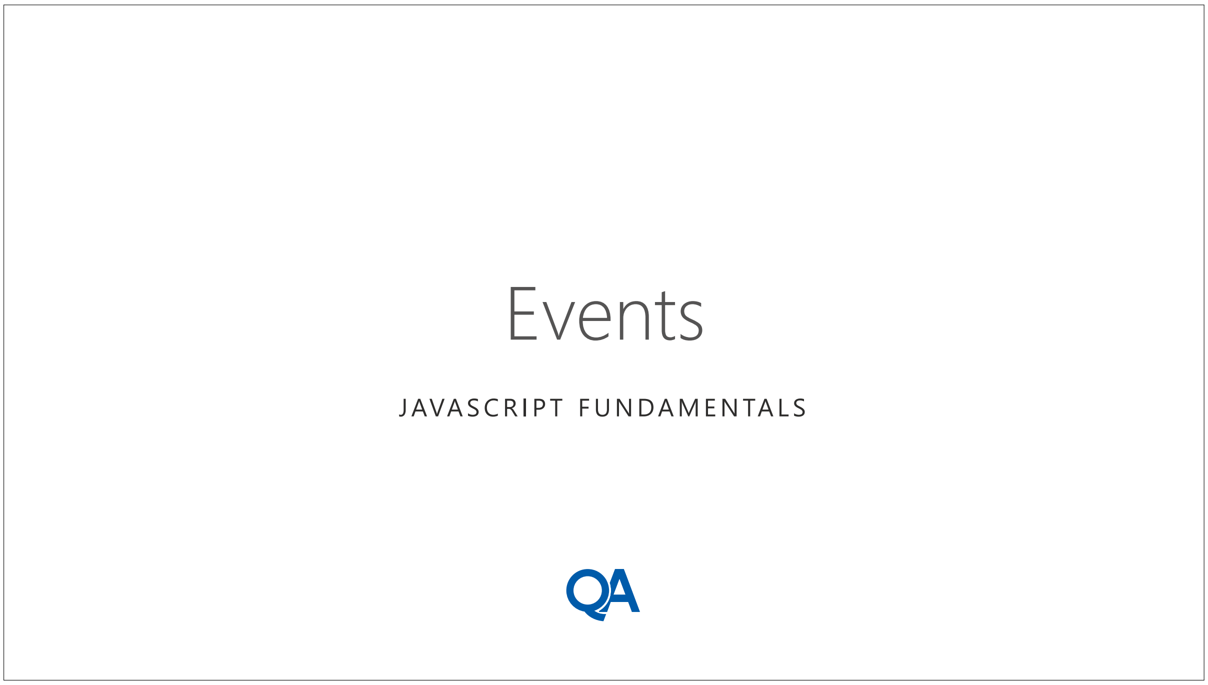
155



Review

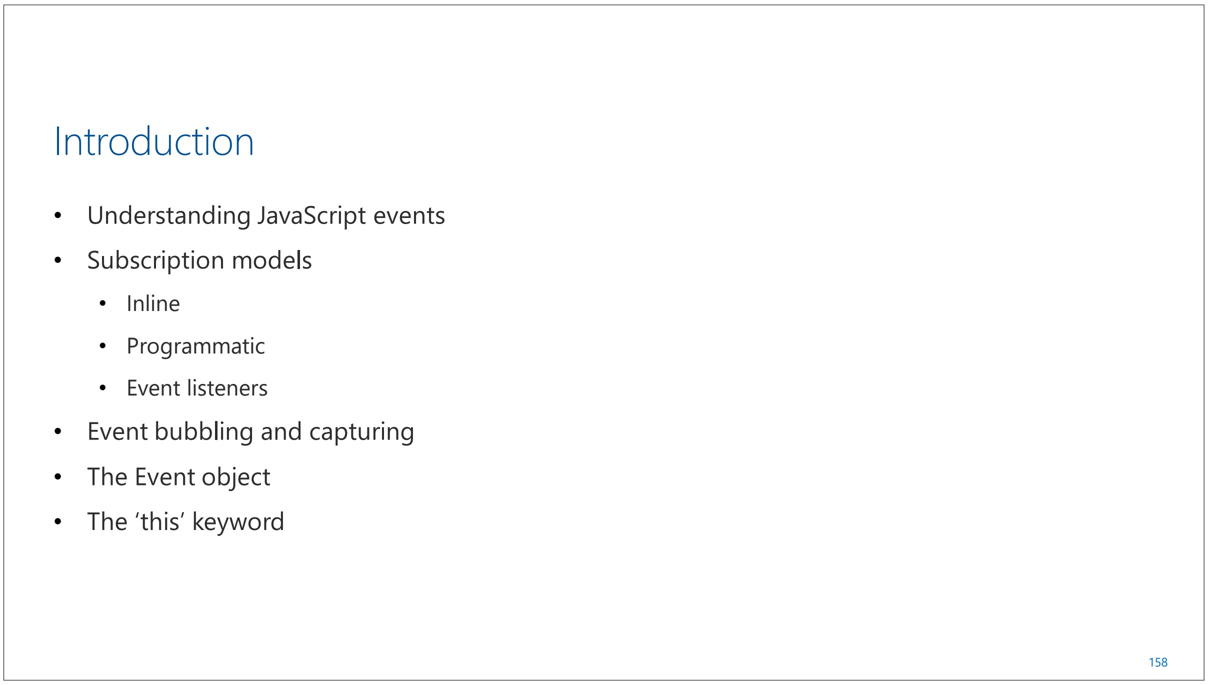
- HTML to DOM rendering
- Accessing the style object
  - Reading style properties
  - Setting style properties
- Understanding CSS classes
  - Obtaining the object computed style
  - Applying classes via JavaScript

156



Events

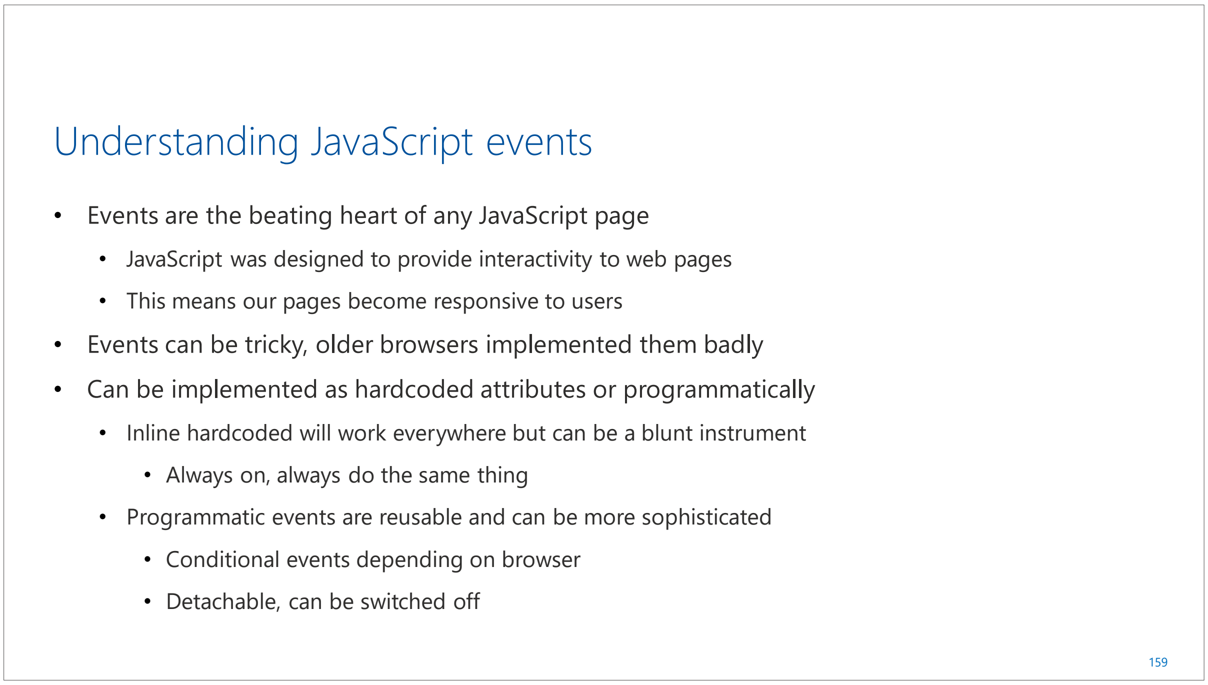
JAVASCRIPT FUNDAMENTALS



Introduction

- Understanding JavaScript events
- Subscription models
  - Inline
  - Programmatic
  - Event listeners
- Event bubbling and capturing
- The Event object
- The 'this' keyword

158



Understanding JavaScript events

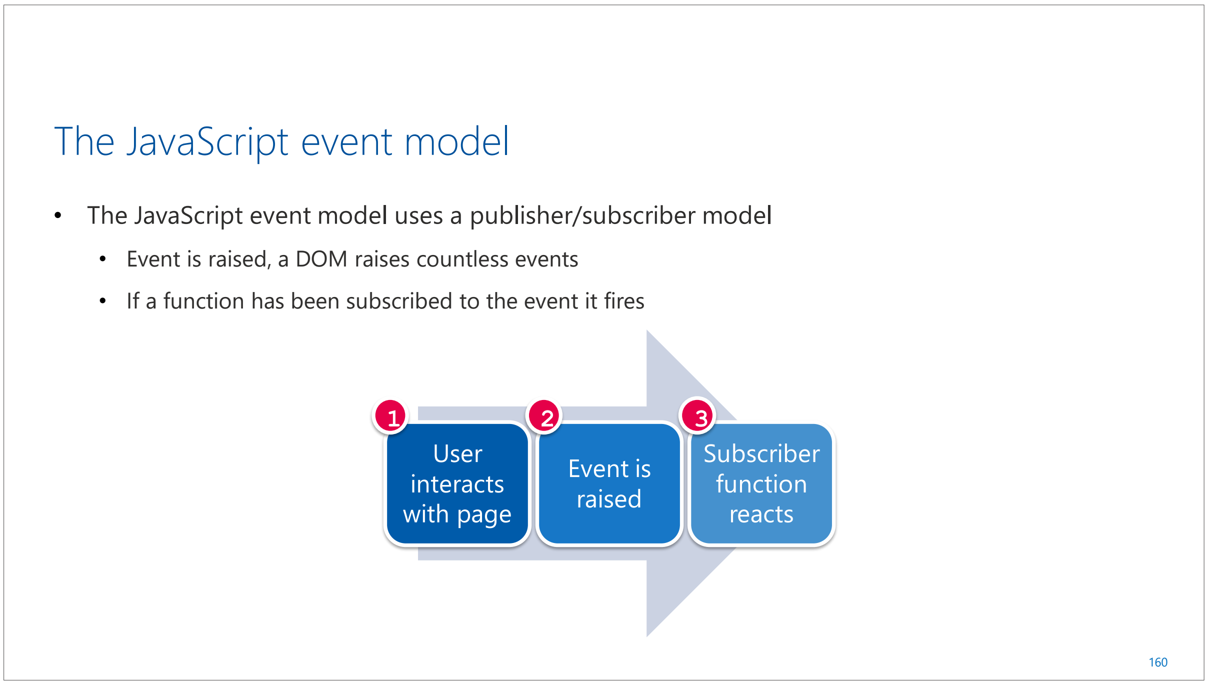
- Events are the beating heart of any JavaScript page
  - JavaScript was designed to provide interactivity to web pages
  - This means our pages become responsive to users
- Events can be tricky, older browsers implemented them badly
- Can be implemented as hardcoded attributes or programmatically
  - Inline hardcoded will work everywhere but can be a blunt instrument
    - Always on, always do the same thing
  - Programmatic events are reusable and can be more sophisticated
    - Conditional events depending on browser
    - Detachable, can be switched off

159

Without events there are no scripts. Take a look at any web page with JavaScript in it: in nearly all cases there will be an event that triggers the script. The reason is very simple. JavaScript is meant to add inter*activity* to your pages: the user does something and the page reacts.

Therefore JavaScript needs a way of detecting user actions, so that it knows when to react. It also needs to know which functions to execute, functions that do something that you, the web developer, have judged likely to increase the appeal of your pages. These pages describe the best way to write such scripts. It isn't easy, but it is very satisfying work.

When the user does something an *event* takes place. There are also some events that aren't directly caused by the user: the load event that fires when a page has been loaded, for instance.



The JavaScript event model

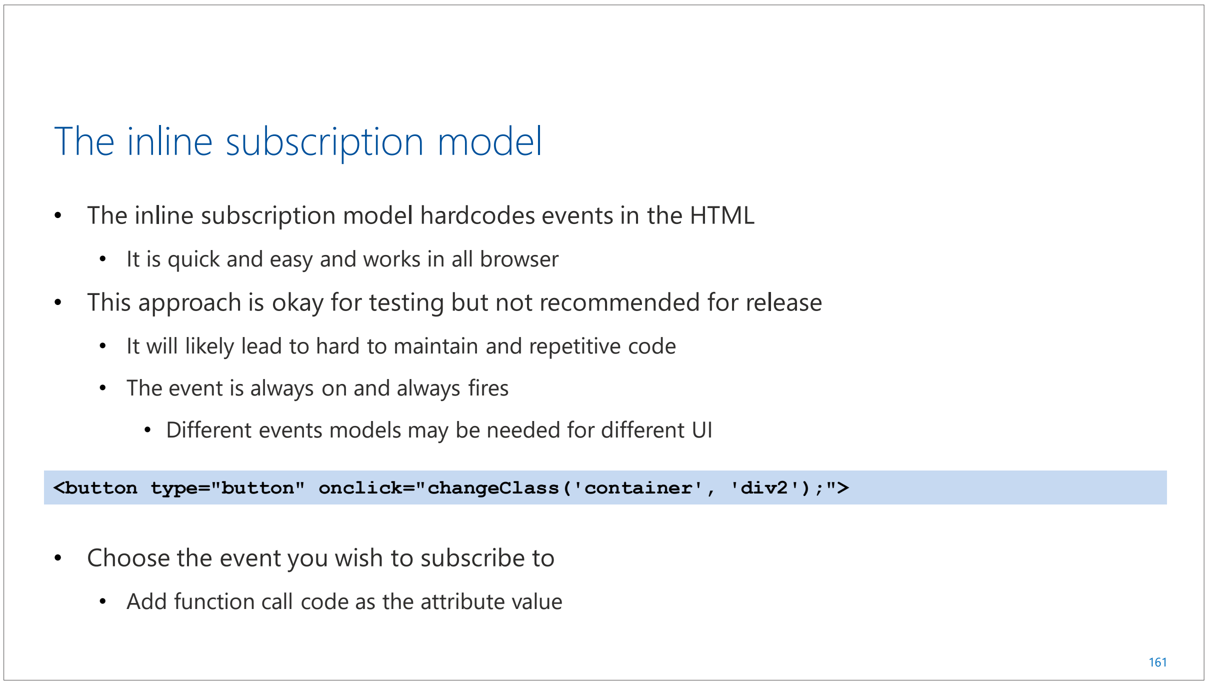
- The JavaScript event model uses a publisher/subscriber model
  - Event is raised, a DOM raises countless events
  - If a function has been subscribed to the event it fires

1
User interacts with page
2
Event is raised
3
Subscriber function reacts

160

The JavaScript event model has three stages. The user (or browser event) occurs. This could be through a mouse clicking a button or a key being pressed. Every DOM object has a list of events it can execute (we will investigate this shortly).

An event may be subscribed to by associating the event to a function. When the event is raised the subscribed function executes. We will examine a number of options for setting up events in the next few slides, the first two only allow a single subscriber function. The third allows multiple subscribers.



The inline subscription model

- The inline subscription model hardcodes events in the HTML
  - It is quick and easy and works in all browser
- This approach is okay for testing but not recommended for release
  - It will likely lead to hard to maintain and repetitive code
  - The event is always on and always fires
    - Different events models may be needed for different UI

```
<button type="button" onclick="changeClass('container', 'div2');">
```

- Choose the event you wish to subscribe to
  - Add function call code as the attribute value

161

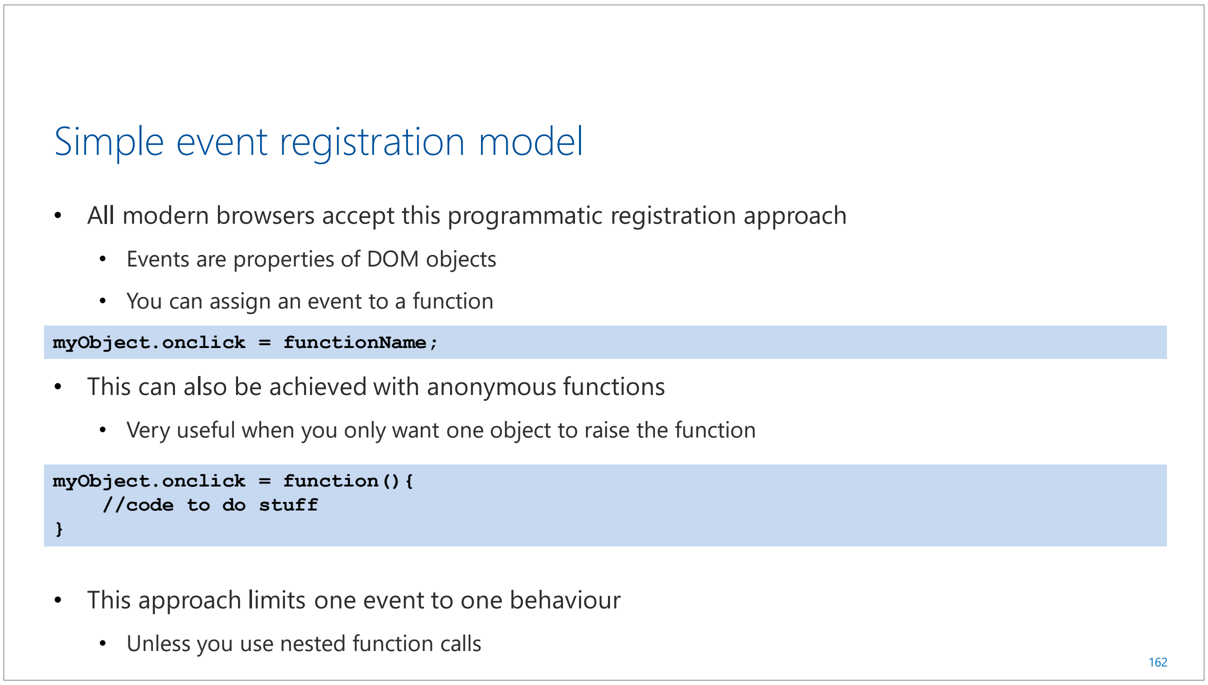
The oldest browsers support only one way of registering event handlers, the way invented by Netscape, works in all JavaScript browsers. In the *inline event registration model*, event handlers are added as attributes to the HTML elements they were working on.

Don't use inline registration

Although the inline event registration model is ancient and reliable, it has one serious drawback. It requires you to write JavaScript behaviour code in your XHTML structure layer, where it doesn't belong. In the world of touch, desktop and mobile devices the approach is not sufficient.

Default Actions

Consider an <a> element when the user clicks on a link the browser loads the page specified in its href attribute. This is the *default action* caused by a click event on a link. But what happens when you've also defined an onclick event handler? It should be executed. If the onclick event handler is to be executed at all, it must be done *before* the default action. This has become an important principle of event handling. If an event causes both a default action and execution of a event handling script the event handler script is executed first



Simple event registration model

- All modern browsers accept this programmatic registration approach
  - Events are properties of DOM objects
  - You can assign an event to a function

```
myObject.onclick = functionName;
```

- This can also be achieved with anonymous functions
  - Very useful when you only want one object to raise the function

```
myObject.onclick = function(){
    //code to do stuff
}
```

- This approach limits one event to one behaviour
  - Unless you use nested function calls

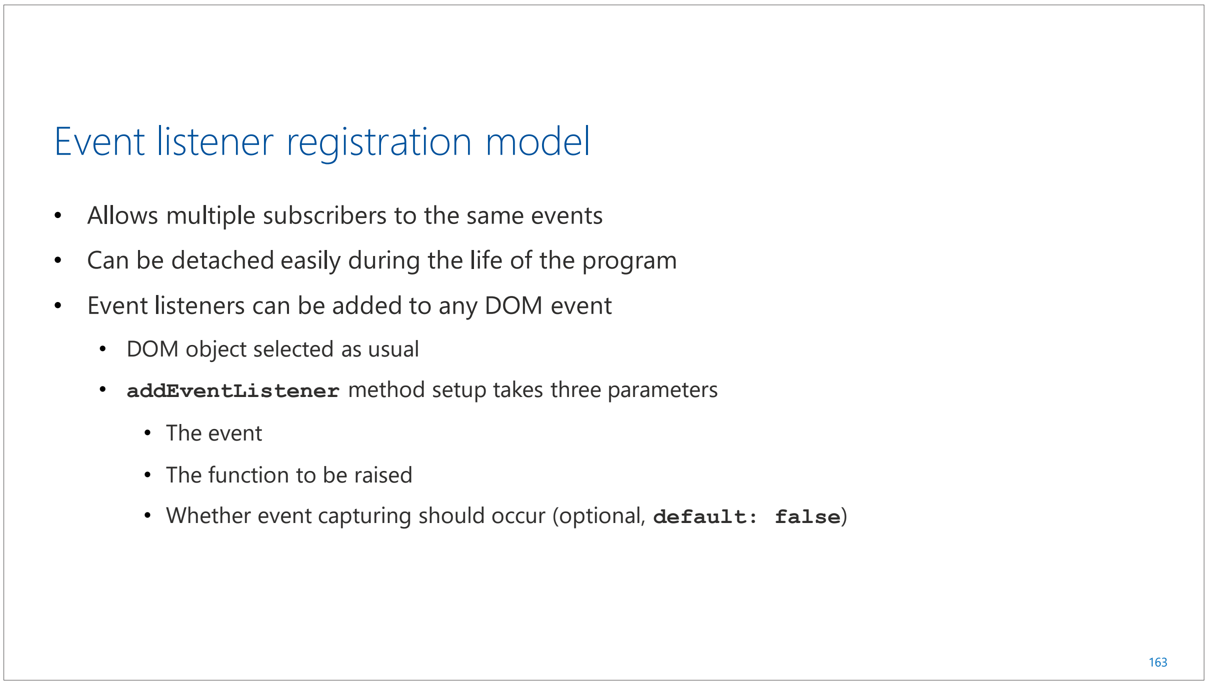
162

For simple function calls the simple event registration model is very easy and elegant to use. We already know that HTML elements become DOMElements and that they in turn are objects. Through that object, a series of null value properties are initialised by the browser. As you would of seen in the exercise you just completed, an inline event registration actually creates a function behind the scenes for us. By understanding this, we can leverage the functionality to improve our web development and remove functionality from the markup.

There are two common approaches to using this methodology. The first is to point (reference) the DOM object's event to separate function. This approach should be used when we want to use the same function for multiple elements. It allows code reusability, but means the function could be called by anyone, or used separate from event raising functionality. Obviously, this would cause catastrophic errors.

The alternative approach is to use anonymous functions. When an anonymous function is created, the only reference to the function object is held by the event itself. This creates a 1 to 1 relationship between publisher and subscriber.

One of the key issues with this approach is that only one function can be associated to one event. There is a small work around through creating an anonymous function that calls other functions in a nested hierarchy. This approach is an all or nothing approach and we always have to have the same functionality.



Event listener registration model

- Allows multiple subscribers to the same events
- Can be detached easily during the life of the program
- Event listeners can be added to any DOM event
    - DOM object selected as usual
    - **`addEventListener`** method setup takes three parameters
        - The event
        - The function to be raised
        - Whether event capturing should occur (optional, **`default: false`**)

163

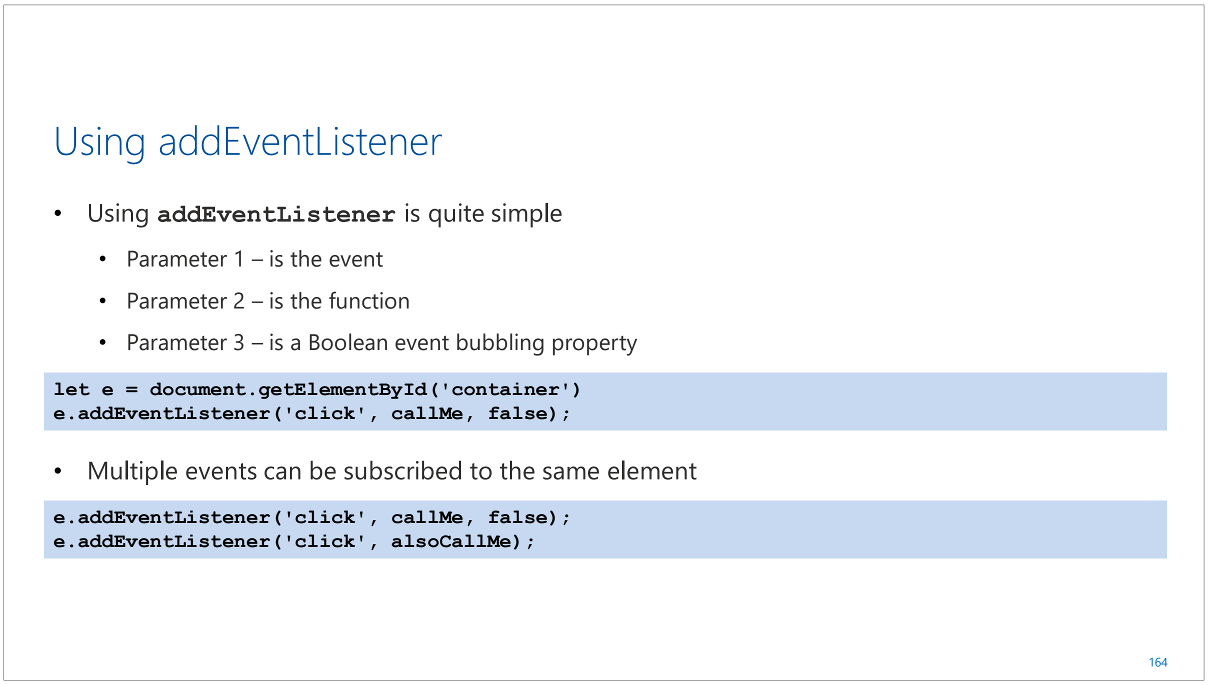
W3C's DOM Level 2 Event specification pays careful attention to the problems of the traditional model. It offers a simple way to register as many event handlers as you like for the same event on one element.

The W3C event registration model uses the method a**ddEventListener().** You give it three arguments:

- the event type
- the function to be executed
- a Boolean (true or false)

The third Boolean parameter is the trickiest to understand. It states whether the event handler should be executed in the capturing or in the bubbling phase. If you're not certain whether you want capturing or bubbling, use false (bubbling).

We will explore event bubbling in more depth shortly.



Using addEventListener

- Using **`addEventListener`** is quite simple
  - Parameter 1 – is the event
  - Parameter 2 – is the function
  - Parameter 3 – is a Boolean event bubbling property

```
let e = document.getElementById('container')
e.addEventListener('click', callMe, false);
```

- Multiple events can be subscribed to the same element

```
e.addEventListener('click', callMe, false);
e.addEventListener('click', alsoCallMe);
```

164

One of the best features of this event registration approach is that we can easily subscribe multiple functions to the same event. In the above example, we have associated the functions callMe and alsoCallMe to the same DOM object. When we click on the object associated to this event, we will fire off a call to both functions. Please note that the W3C model does not state which event handler is fired first and that can not be assumed.

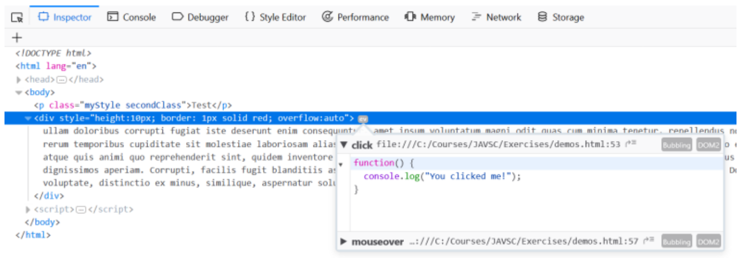
Subscribed functions can be removed from an event at any time using the removeEventListener function:

```
element.removeEventListener('event',funcName,false)
```

This level of granularity allows us to choose which method will be unsubscribed.

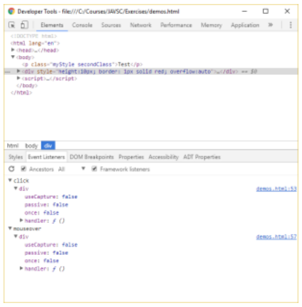
Debugging event listeners

- Previously, we have examined a DOM object to see the event
  - This will not work with event listeners
  - The 'hooking up' occurs behind the scenes
- Browser debug tools can help us out here



```
<!DOCTYPE html>
<html lang="en">
▸ <head>…</head>
▾ <body>
    <p class="myStyle secondClass">Test</p>
  ▾ <div style="height:10px; border: 1px solid red; overflow:auto"> ev
      ullam doloribus corrupti fugiat iste deserunt enim consequunt  amet ipsum voluptatum magni odit quas cum minima tenetur, repellendus n
      rerum temporibus cupiditate sit molestiae laboriosam alia
      atque quis animi quo reprehenderit sint, quidem inventore
      dignissimos aperiam. Corrupti, facilis fugit blanditiis a
      voluptate, distinctio ex minus, similique, aspernatur sol
    </div>
  ▸ <script>…</script>
  </body>
</html>
```

```
▼ click file:///C:/Courses/JAVSC/Exercises/demos.html:53
function() {
  console.log("You clicked me!");
}
▶ mouseover …:///C:/Courses/JAVSC/Exercises/demos.html:57
```



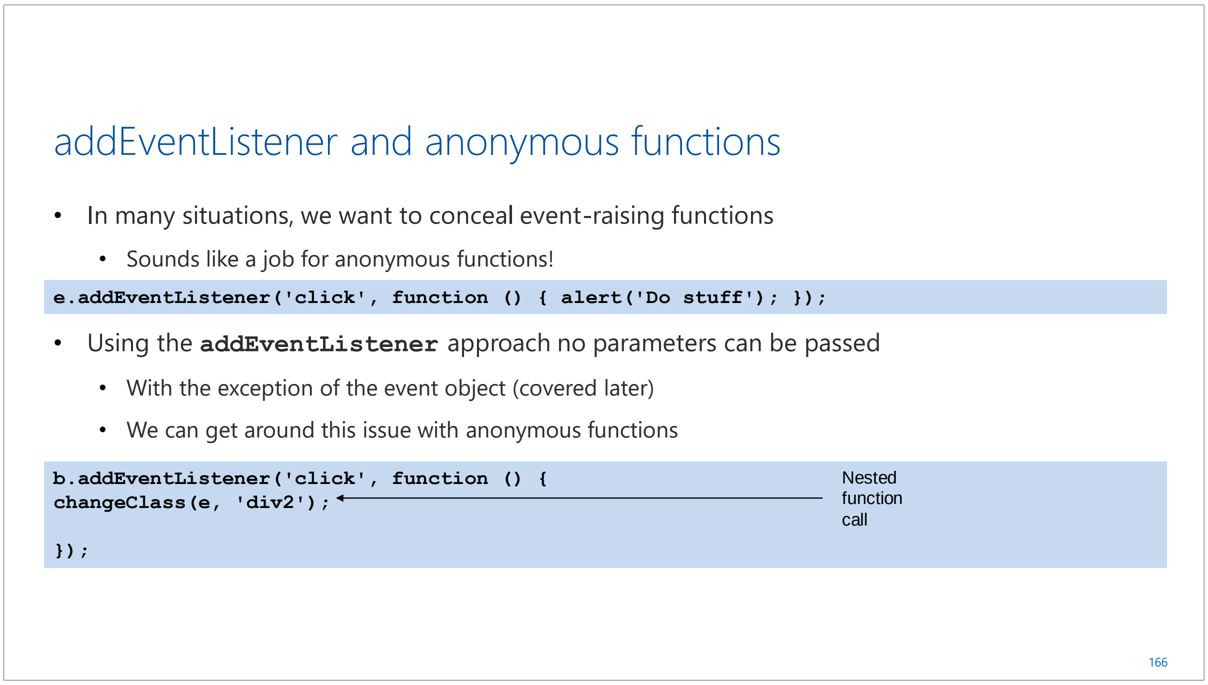
```
<!DOCTYPE html>
<html lang="en">
▶ <head>…</head>
▼ <body>
    <p class="myStyle secondClass">Test</p>
... ▶ <div style="height:10px; border: 1px solid red; overflow:auto">…</div> == $0
  ▶ <script>…</script>
  </body>
</html>
```

```
▼ click
  ▼ div                                   demos.html:53
      useCapture: false
      passive: false
      once: false
    ▶ handler: ƒ ()
▼ mouseover
  ▼ div                                   demos.html:57
      useCapture: false
      passive: false
      once: false
    ▶ handler: ƒ ()
```

165

If we were to examine the onclick property of our DOM element, we would find that there is no function assigned to it. This is because the event listener approach follows an observer or delegate pattern. Its role is to hold references to the interested functions of that event.

Chrome provides two views to this. In the Elements panel, it shows the DOM elements that raise specific events (note the browser may attach listeners to events for its internal functionality). The Sources panel provides the ability to attach to mouse, keyboard or other types of listeners and when they fire the debugger will automatically capture the event.



addEventListener and anonymous functions

- In many situations, we want to conceal event-raising functions
  - Sounds like a job for anonymous functions!

```
e.addEventListener('click', function () { alert('Do stuff'); });
```

- Using the **`addEventListener`** approach no parameters can be passed
  - With the exception of the event object (covered later)
  - We can get around this issue with anonymous functions

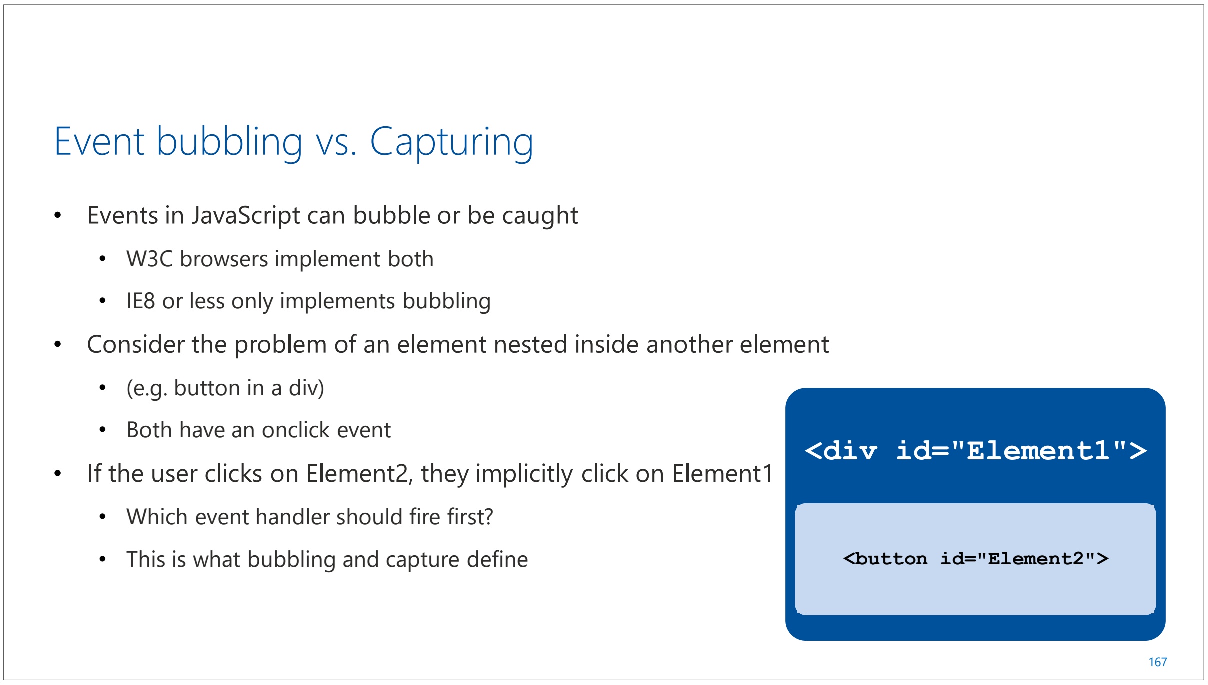
```
b.addEventListener('click', function () {
changeClass(e, 'div2');

});
```

Nested function call

166

As we discussed earlier, in this chapter it is not at all uncommon that we want to conceal event raising functions, so only the event handler can raise them. The event listener approach also allows us to continue this approach.

This can be extremely useful when we need to pass parameters to the listener. Other than the event itself, no parameter is passed when the event is raised (although, it does maintain scope) instead, we can wrap the functions we want to call inside an anonymous function block and pass the parameter from the DOM element that raises the event.



Event bubbling vs. Capturing

- Events in JavaScript can bubble or be caught
  - W3C browsers implement both
  - IE8 or less only implements bubbling
- Consider the problem of an element nested inside another element
  - (e.g. button in a div)
  - Both have an onclick event
- If the user clicks on Element2, they implicitly click on Element1
  - Which event handler should fire first?
  - This is what bubbling and capture define

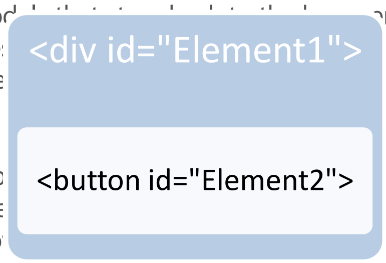
```
<div id="Element1">
    <button id="Element2">
```

167

The basic problem is very simple. Suppose you have a element inside an element.

Both have an onClick event handler. If the user clicks on element2, they cause a click event in both element1 and element2. But which event fires first? Which event handler should be executed first? What, in other words, is the event order?

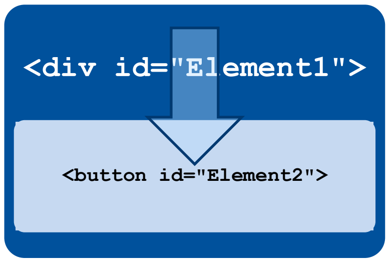
Web programming provides two mod r wars of the late 1990's. Netscape
said that the event on element1 take pturing. Microsoft maintained that
the event on element2 takes precede



The two event orders are radically op ides the ability to do both. IE8 and
below, followed their approach and e ough, I caused incredible difficulty
to web developers. IE9 onwards follo will cease to be a concern in the
future.

Event Capturing

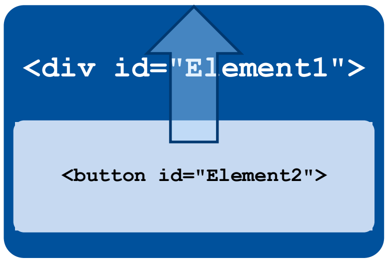
- With Event Capturing the event handler of Element1 fires first



168

Event Bubbling

- With Event Bubbling the event handler of Element2 fires first

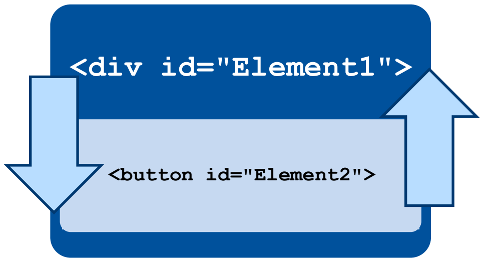


- Legacy Note: IE8 and below could only use bubbling

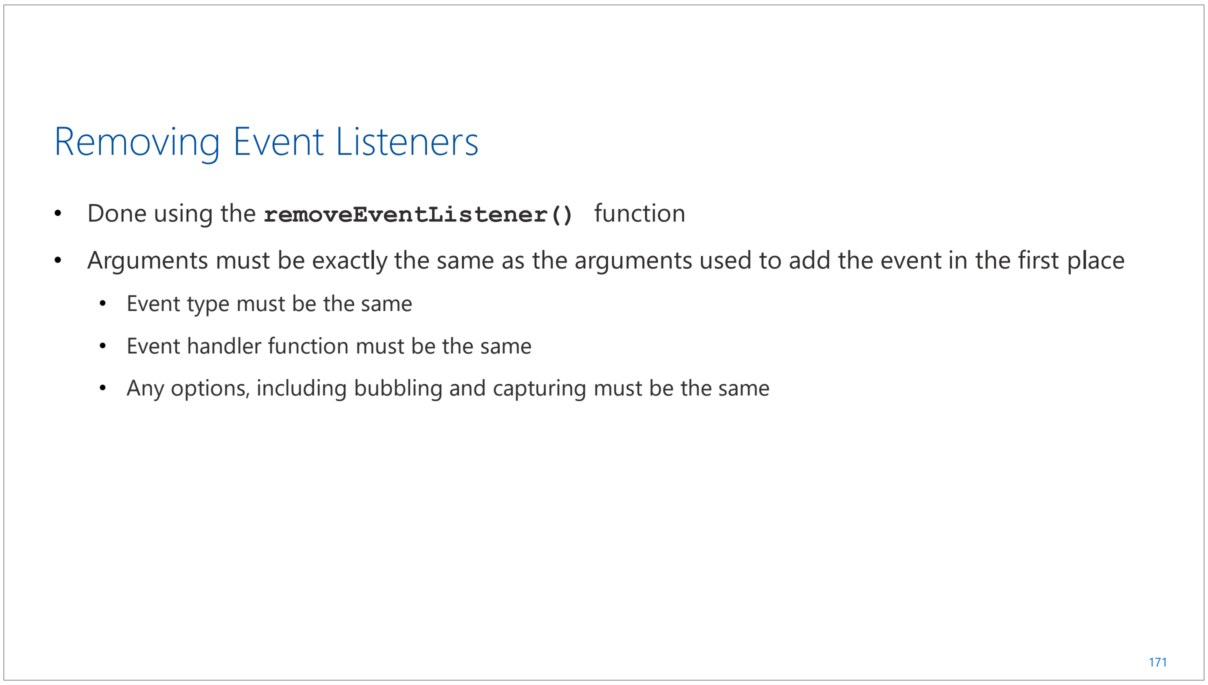
169

W3C model

- The W3C model can use either approach
  - The third parameter in **`addEventListener`** sets how the event works
  - You can mix both in the same page if appropriate



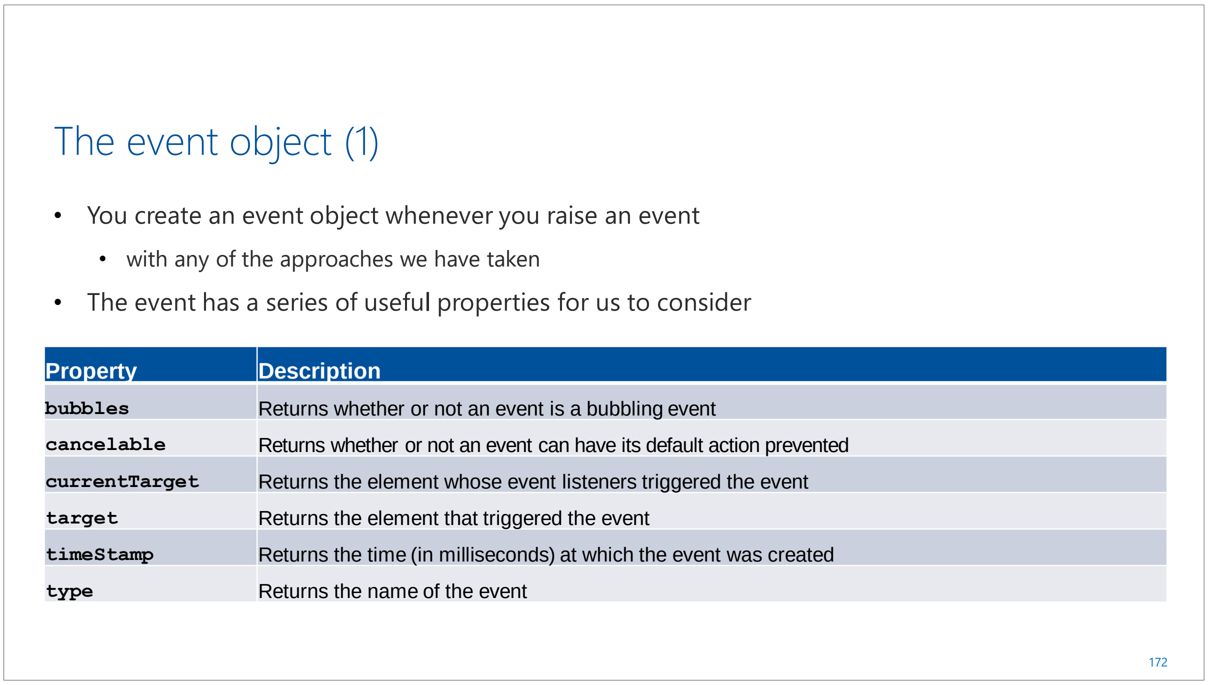
170



Removing Event Listeners

- Done using the **`removeEventListener()`** function
- Arguments must be exactly the same as the arguments used to add the event in the first place
  - Event type must be the same
  - Event handler function must be the same
  - Any options, including bubbling and capturing must be the same

171



The event object (1)

- You create an event object whenever you raise an event
  - with any of the approaches we have taken
- The event has a series of useful properties for us to consider

Property	Description
`bubbles`	Returns whether or not an event is a bubbling event
`cancelable`	Returns whether or not an event can have its default action prevented
`currentTarget`	Returns the element whose event listeners triggered the event
`target`	Returns the element that triggered the event
`timeStamp`	Returns the time (in milliseconds) at which the event was created
`type`	Returns the name of the event

172

The event object also exposes a series of methods:

event.initEvent

Initialises the value of an Event created through the DocumentEvent interface.

event.preventDefault

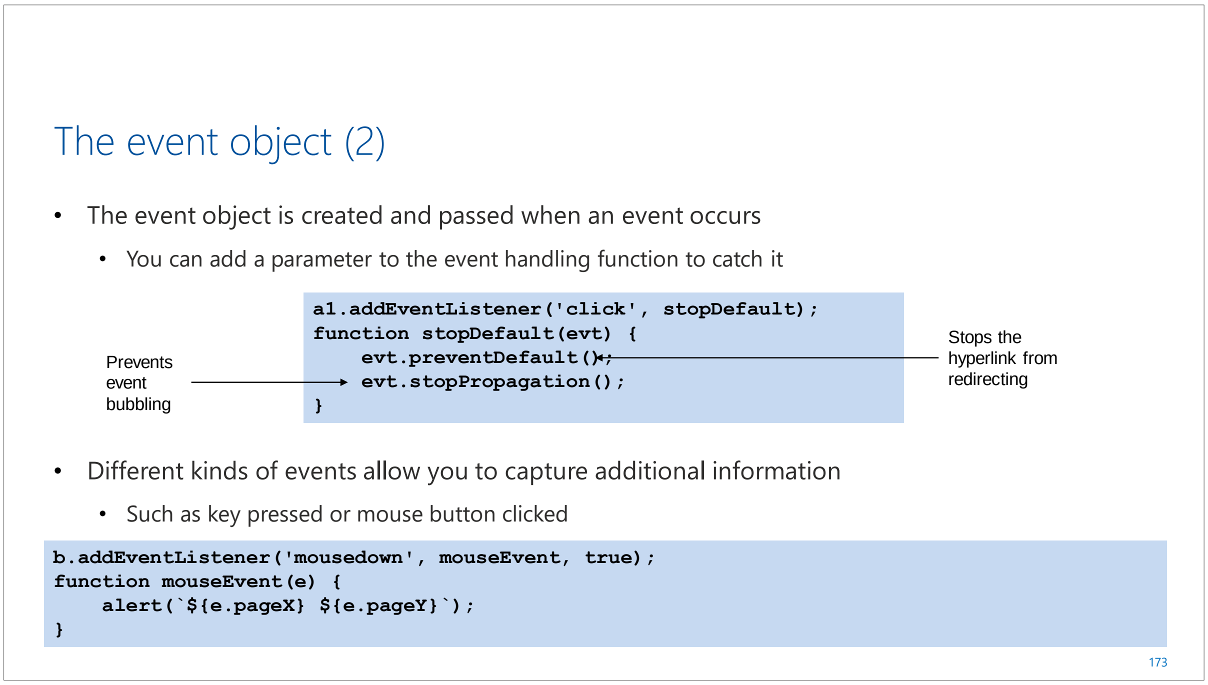
Cancels the event (if it is cancellable).

event.stopImmediatePropagation

For this particular event, no other listener will be called – neither those attached on the same element, nor those attached on elements that will be traversed later (in capture phase, for instance).

event.stopPropagation

Stops the propagation of events further along in the DOM.



The event object (2)

- The event object is created and passed when an event occurs
  - You can add a parameter to the event handling function to catch it

```
a1.addEventListener('click', stopDefault);
function stopDefault(evt) {
    evt.preventDefault();
    evt.stopPropagation();
}
```

Prevents event bubbling

Stops the hyperlink from redirecting

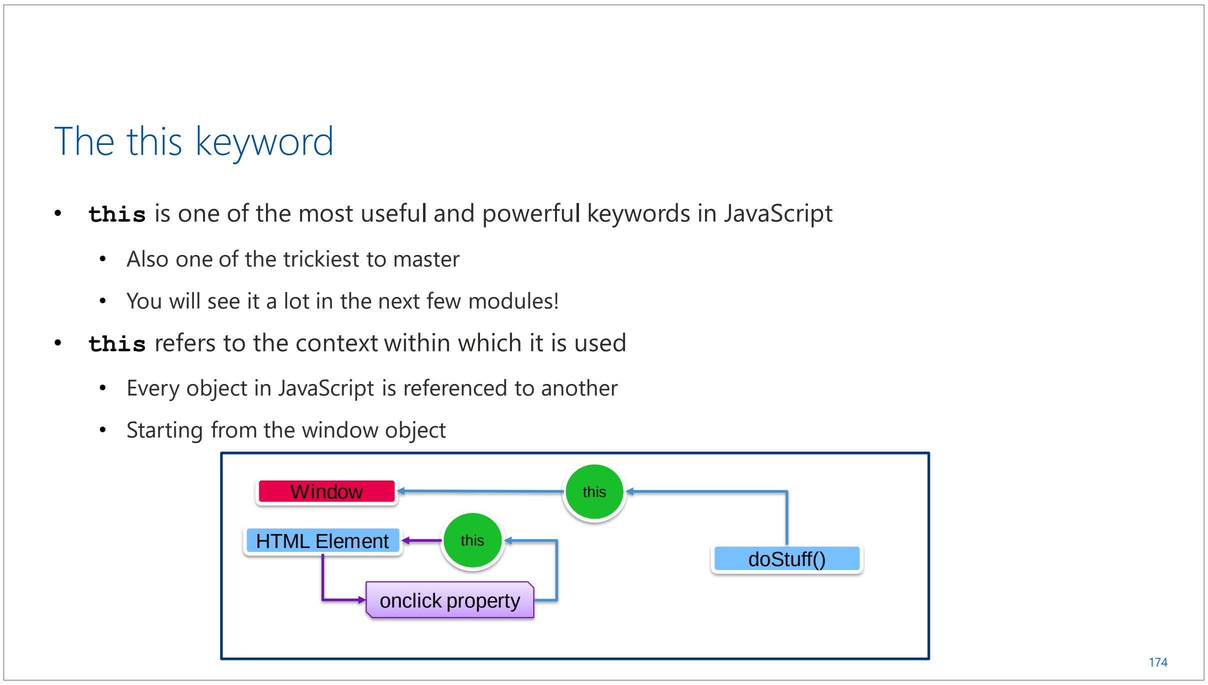
- Different kinds of events allow you to capture additional information
  - Such as key pressed or mouse button clicked

```
b.addEventListener('mousedown', mouseEvent, true);
function mouseEvent(e) {
    alert(`${e.pageX} ${e.pageY}`);
}
```

173

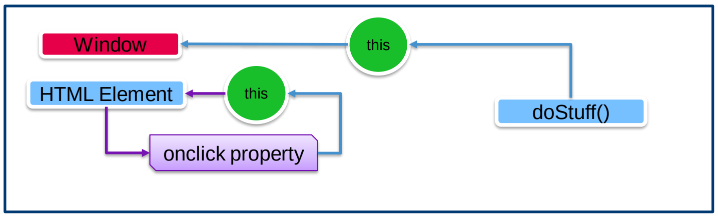
Event handlers may be attached to various objects, including DOM elements, document, the window object, etc. When an event occurs, an event object is created and passed sequentially to the event listeners.

The DOM Event interface is accessible from within the handler function, via the event object passed as the first argument. The following simple example shows how an event object is passed to the event handler function, and can be used from within one such function.



The this keyword

- `this` is one of the most useful and powerful keywords in JavaScript
  - Also one of the trickiest to master
  - You will see it a lot in the next few modules!
- `this` refers to the context within which it is used
  - Every object in JavaScript is referenced to another
  - Starting from the window object



174

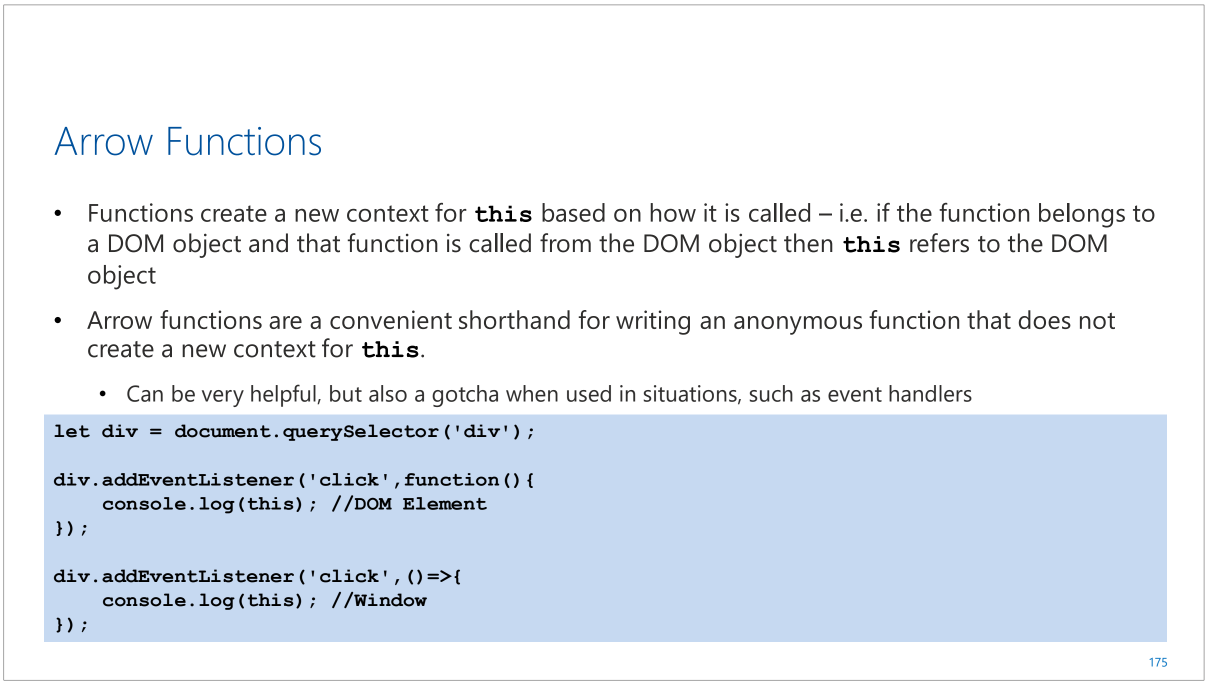
In JavaScript, the **this** keyword usually refers to the “owner” of a function. In the case of event handlers, it is very useful if this refers to the HTML element the event is handled by, so that you have easy access to it.

Unfortunately, the **this** keyword, though very powerful, is hard to use if you don’t know exactly how it works

The question that we'll discuss for the remainder of the page is: What does **this** refer to in the function **doStuff()**? In a page, its owner is the window object.

If we execute **doStuff()** without any more preparation, the **this** keyword refers to the **window** and the function tries to change the **style.color** of the **window**. Since the **window** is a **BOM** and not a **DOM** object, it does not have a **style** object and this fails, with errors.

There is an important difference between inline and programmatic event registration. Inline references the function object, (so there is only ever 1), programmatic event association copies the function object. If we use programmatic event registration, **this** becomes a reference to the calling DOM object. If we use inline registration, **this** refers to the **window** object.



Arrow Functions

- Functions create a new context for **`this`** based on how it is called – i.e. if the function belongs to a DOM object and that function is called from the DOM object then **`this`** refers to the DOM object
- Arrow functions are a convenient shorthand for writing an anonymous function that does not create a new context for **`this`**.
  - Can be very helpful, but also a gotcha when used in situations, such as event handlers

```
let div = document.querySelector('div');

div.addEventListener('click',function(){
    console.log(this); //DOM Element
});

div.addEventListener('click',()=>{
    console.log(this); //Window
});
```

175

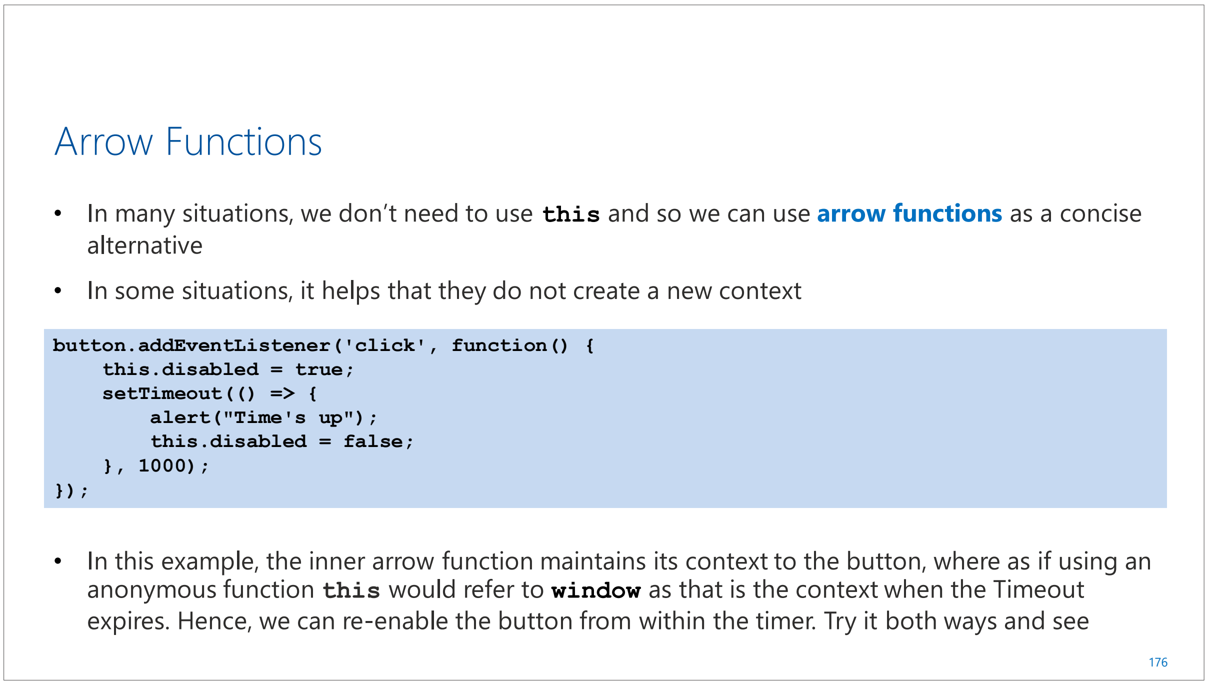
In JavaScript, the **this** keyword usually refers to the “owner” of a function. In the case of event handlers, it is very useful if this refers to the HTML element the event is handled by, so that you have easy access to it.

Unfortunately, the **this** keyword, though very powerful, is hard to use if you don’t know exactly how it works

The question that we'll discuss for the remainder of the page is: What does **this** refer to in the function **doStuff()**? In a page, its owner is the window object.

If we execute **doStuff()** without any more preparation the **this** keyword refers to the **window** and the function tries to change the **style.color** of the **window**. Since the **window** is a **BOM** and not a **DOM** object, it does not have a **style** object and this fails, with errors.

There is an important difference between inline and programmatic event registration. Inline references the function object, (so there is only ever 1), programmatic event association copies the function object. If we use programmatic event registration, **this** becomes a reference to the calling DOM object. If we use inline registration, **this** refers to the **window** object.



Arrow Functions

- In many situations, we don’t need to use `this` and so we can use **arrow functions** as a concise alternative
- In some situations, it helps that they do not create a new context

```
button.addEventListener('click', function() {
    this.disabled = true;
    setTimeout(() => {
        alert("Time's up");
        this.disabled = false;
    }, 1000);
});
```

- In this example, the inner arrow function maintains its context to the button, where as if using an anonymous function `this` would refer to `window` as that is the context when the Timeout expires. Hence, we can re-enable the button from within the timer. Try it both ways and see

176

In JavaScript, the **this** keyword usually refers to the “owner” of a function. In the case of event handlers, it is very useful if this refers to the HTML element the event is handled by, so that you have easy access to it.

Unfortunately, the **this** keyword, though very powerful, is hard to use if you don’t know exactly how it works

The question that we'll discuss for the remainder of the page is: What does **this** refer to in the function **doStuff()**? In a page, its owner is the window object.

If we execute **doStuff()** without any more preparation, the **this** keyword refers to the **window** and the function tries to change the **style.color** of the **window**. Since the **window** is a **BOM** and not a **DOM** object, it does not have a **style** object and this fails, with errors.

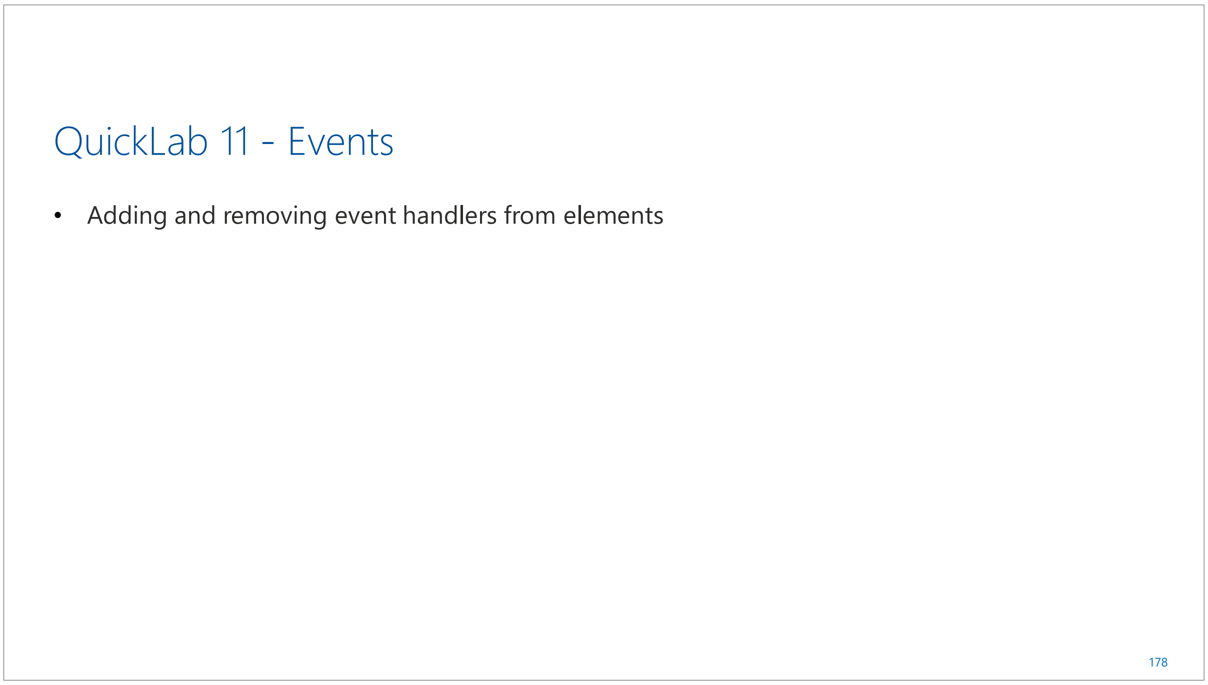
There is an important difference between inline and programmatic event registration. Inline references the function object, (so there is only ever 1), programmatic event association copies the function object. If we use programmatic event registration, **this** becomes a reference to the calling DOM object. If we use inline registration, **this** refers to the **window** object.

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;                                          // outputs value of
                                                                    someArgument

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

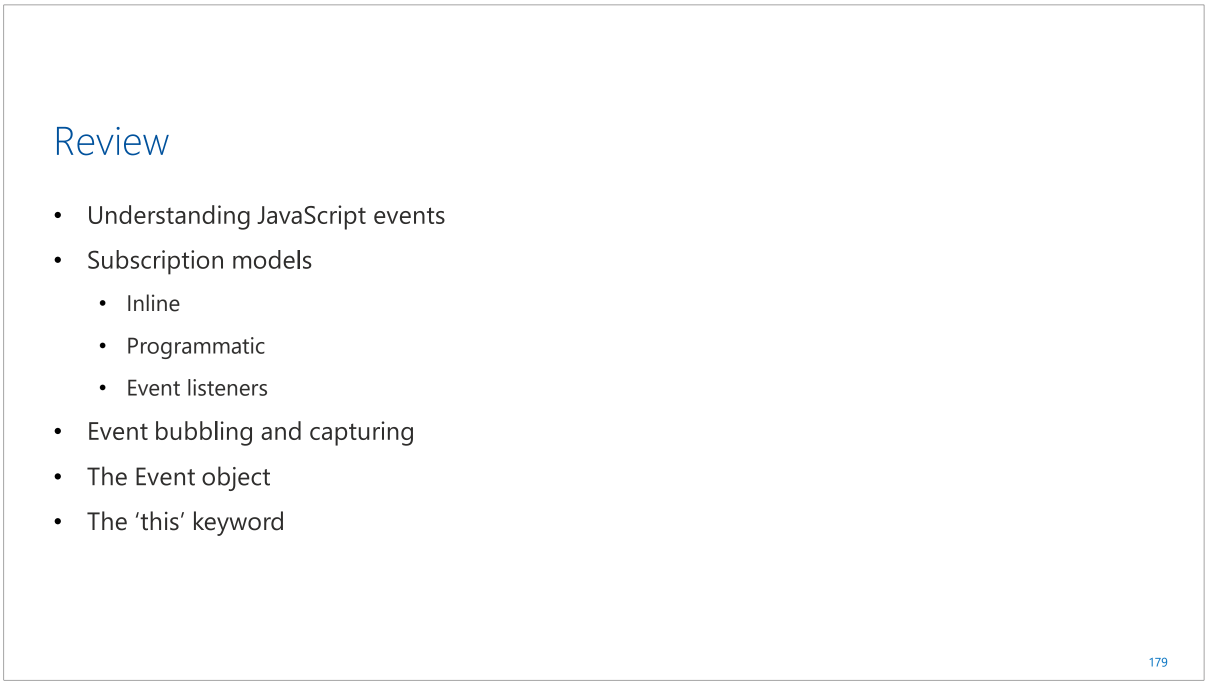
177



QuickLab 11 - Events

- Adding and removing event handlers from elements

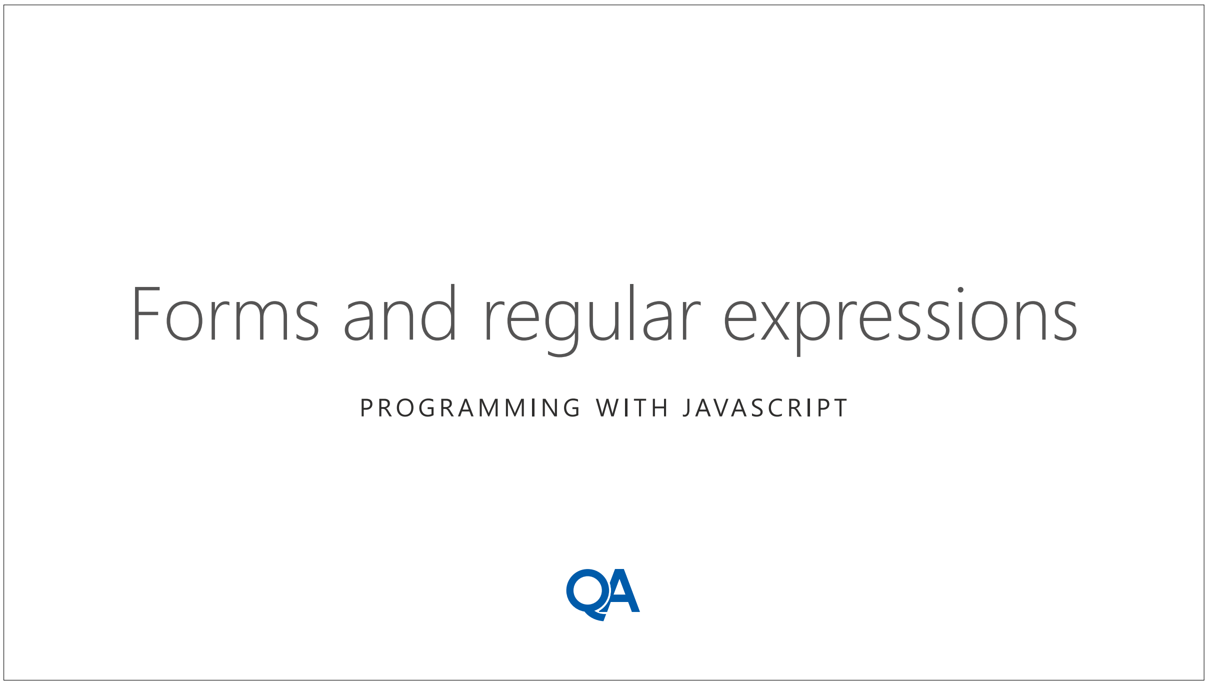
178



Review

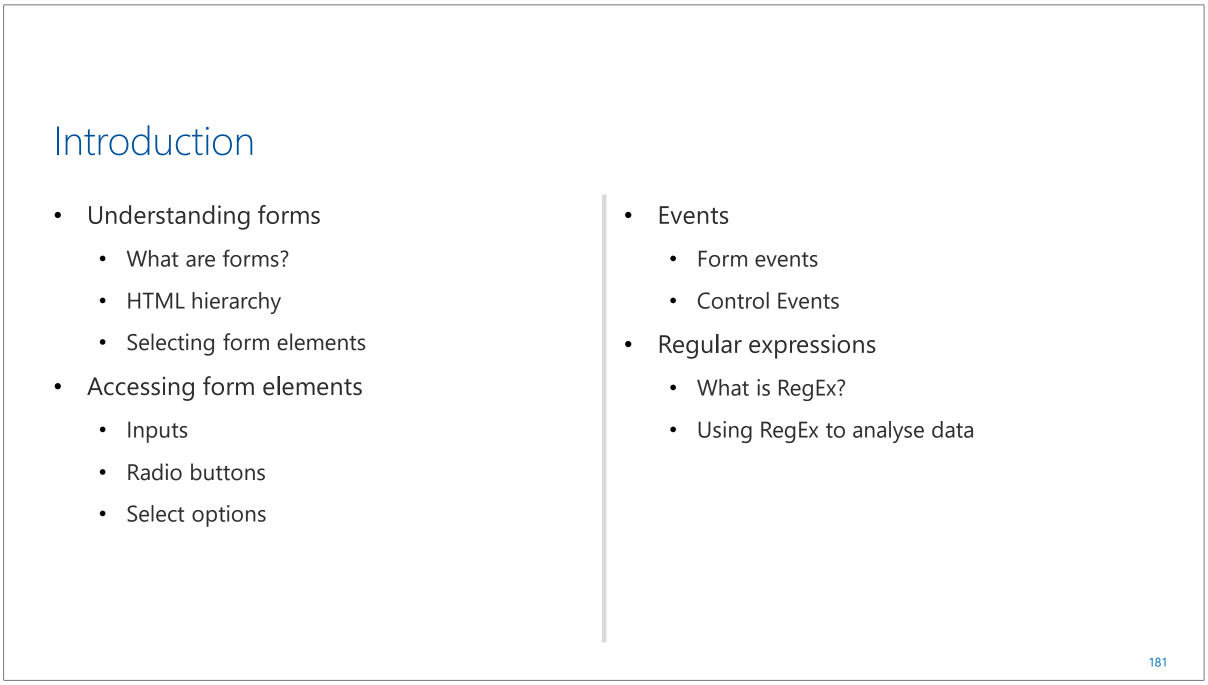
- Understanding JavaScript events
- Subscription models
  - Inline
  - Programmatic
  - Event listeners
- Event bubbling and capturing
- The Event object
- The ‘this’ keyword

179



Forms and regular expressions

PROGRAMMING WITH JAVASCRIPT



Introduction

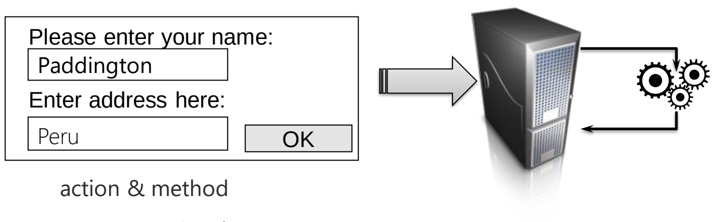
- Understanding forms
  - What are forms?
  - HTML hierarchy
  - Selecting form elements
- Accessing form elements
  - Inputs
  - Radio buttons
  - Select options
- Events
  - Form events
  - Control Events
- Regular expressions
  - What is RegEx?
  - Using RegEx to analyse data

181

Understanding forms – What are forms?

- Forms allow us to send data to the server for processing

```
<form action="/folder/enrol.aspx" method="POST">
    <input type="text" name="username" />
    <input type="text" name="address" />
    <input type="submit" value="OK" />
</form>
```



Please enter your name:
Paddington
Enter address here:
Peru
OK

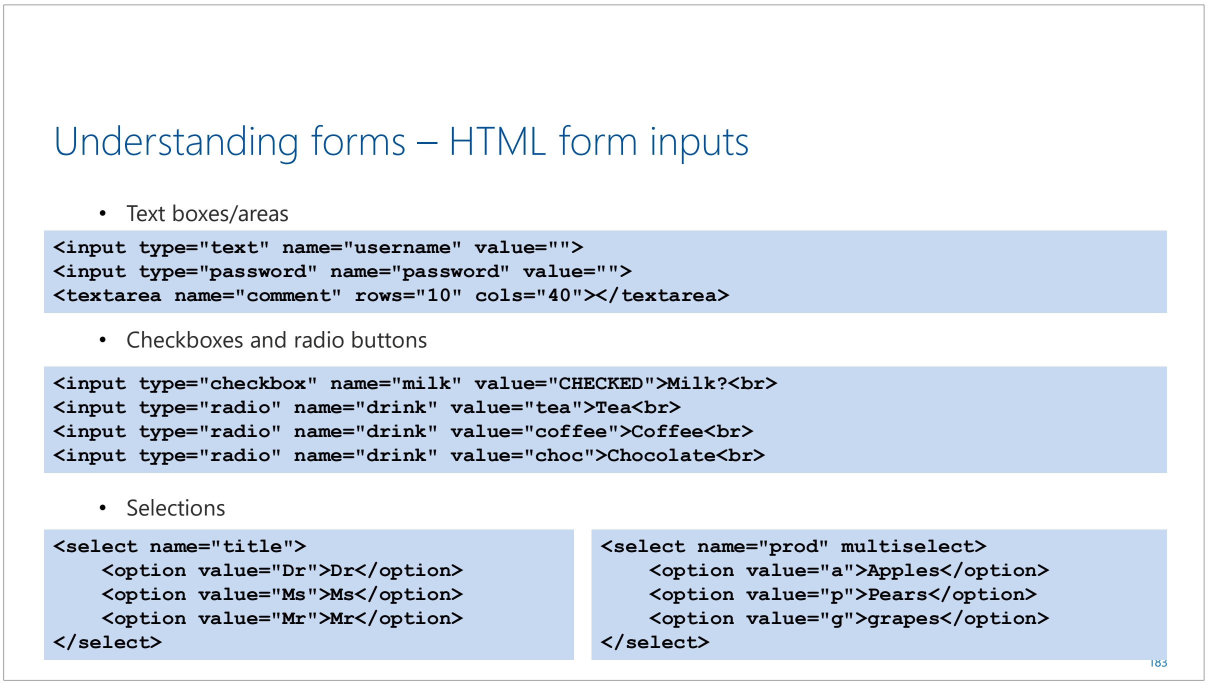
- Define:
  - Form – action & method
  - Inputs – name, type & value

182

Users can interact with programs running on the server by using HTML forms; in this case, the web server acts as nothing more than a gateway to forward the form's data to the back-end application and to return any generated response. Forms allow the web to be used for a variety of purposes, including user inquiries, ordering and booking systems, and front ends to databases.

A form is described using the `<form>` tag; this will contain other elements and text, including the elements describing the form's input fields. The FORM tag takes two attributes: the `method` states how the data is to be sent to the web server, and the `action` gives the URL of a resource used to process the form. This is normally a server-side program or script, but can be a `mailto: URL`.

The values for the `method` attribute determine how the data is sent to the server and takes the value POST or GET. Depending on which you use, the server application needs to be programmed slightly differently, but the details of this are beyond the scope of this course.



Understanding forms – HTML form inputs

- Text boxes/areas

```
<input type="text" name="username" value="">
<input type="password" name="password" value="">
<textarea name="comment" rows="10" cols="40"></textarea>
```

- Checkboxes and radio buttons

```
<input type="checkbox" name="milk" value="CHECKED">Milk?<br>
<input type="radio" name="drink" value="tea">Tea<br>
<input type="radio" name="drink" value="coffee">Coffee<br>
<input type="radio" name="drink" value="choc">Chocolate<br>
```

- Selections

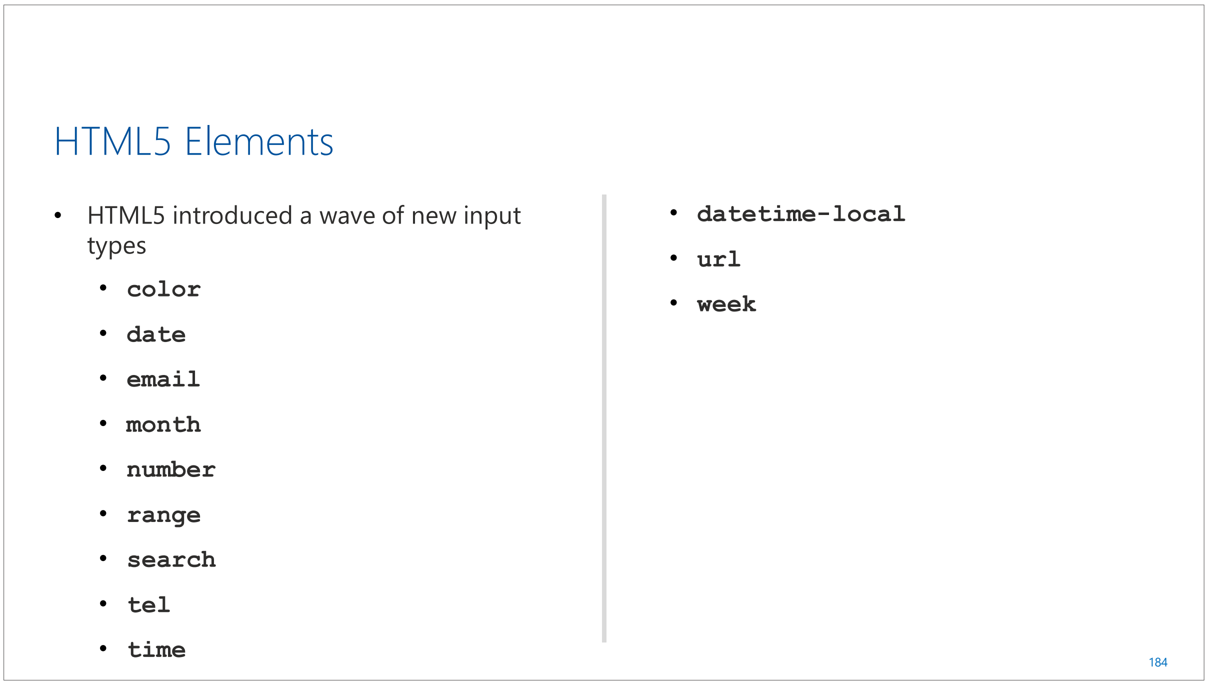
```
<select name="title">
    <option value="Dr">Dr</option>
    <option value="Ms">Ms</option>
    <option value="Mr">Mr</option>
</select>
```

```
<select name="prod" multiselect>
    <option value="a">Apples</option>
    <option value="p">Pears</option>
    <option value="g">grapes</option>
</select>
```

183

Most form fields are defined using an `<input>` tag; the `type` attribute specifies whether this is a checkbox, radio button, etc. A few field types have their own specific tags; for example, `textarea` and `select`. These are shown above.

Each input element has name and a value. The name is specified in the `name` attribute, the value may be specified in the `value` attribute (except for text areas when it is specified by placing text within the `<textarea>…</textarea> tags)` or left for the user to enter (e.g. with text box input). The `name-value` pairs are how back-end code accesses the data from the form.



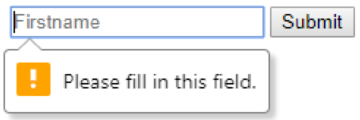
HTML5 Elements

- HTML5 introduced a wave of new input types
  - `color`
  - `date`
  - `email`
  - `month`
  - `number`
  - `range`
  - `search`
  - `tel`
  - `time`
- `datetime-local`
- `url`
- `week`

184

Required fields

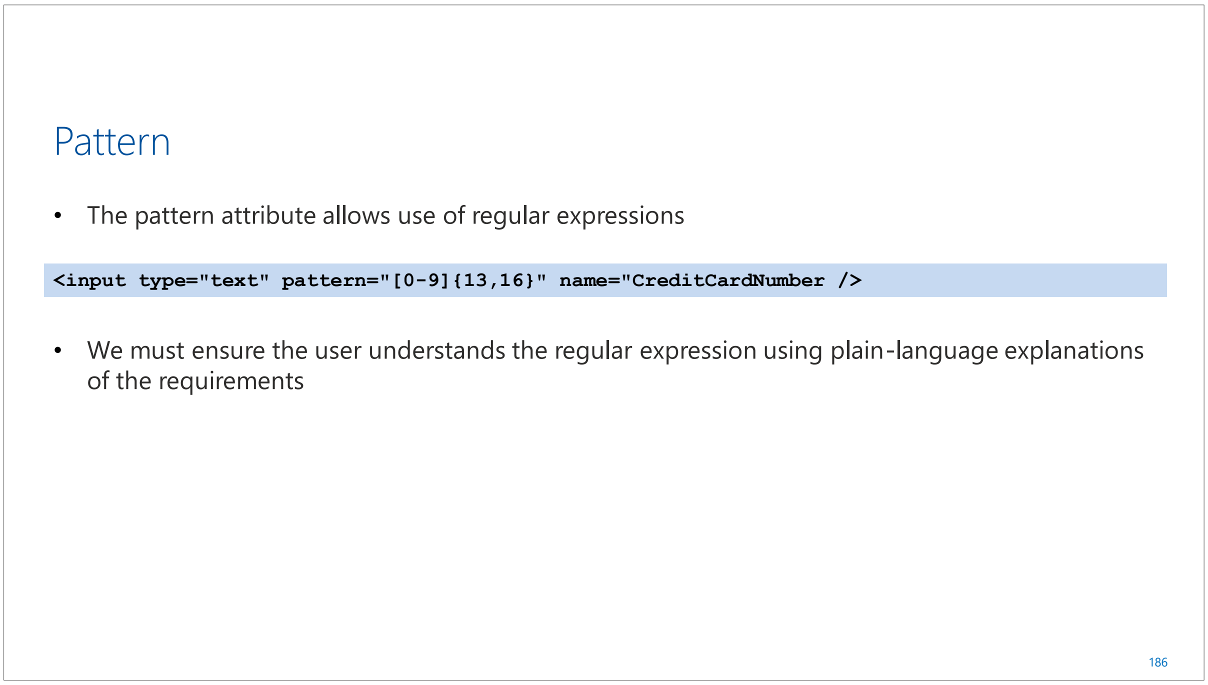
- You can force a field to be mandatory on the client



- On a submit action an error message may appear:

```
<input type="text" autofocus="true" required/>
```

185



Pattern

- The pattern attribute allows use of regular expressions

```
<input type="text" pattern="[0-9]{13,16}" name="CreditCardNumber />
```

- We must ensure the user understands the regular expression using plain-language explanations of the requirements

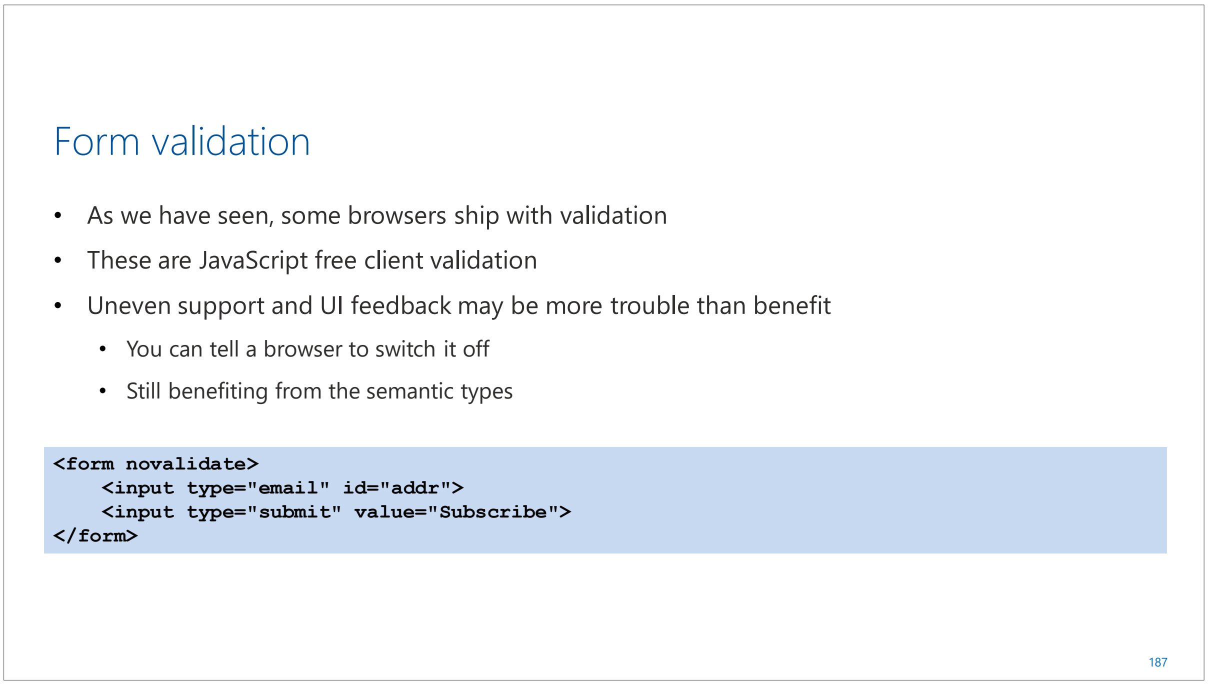
186

The pattern attribute specifies a regular expression that the <input> element's value is checked against.

Note: The pattern attribute works with the following input types: text, search, url, tel, email, and password.

Useful pattern generation website:

http://html5pattern.com/



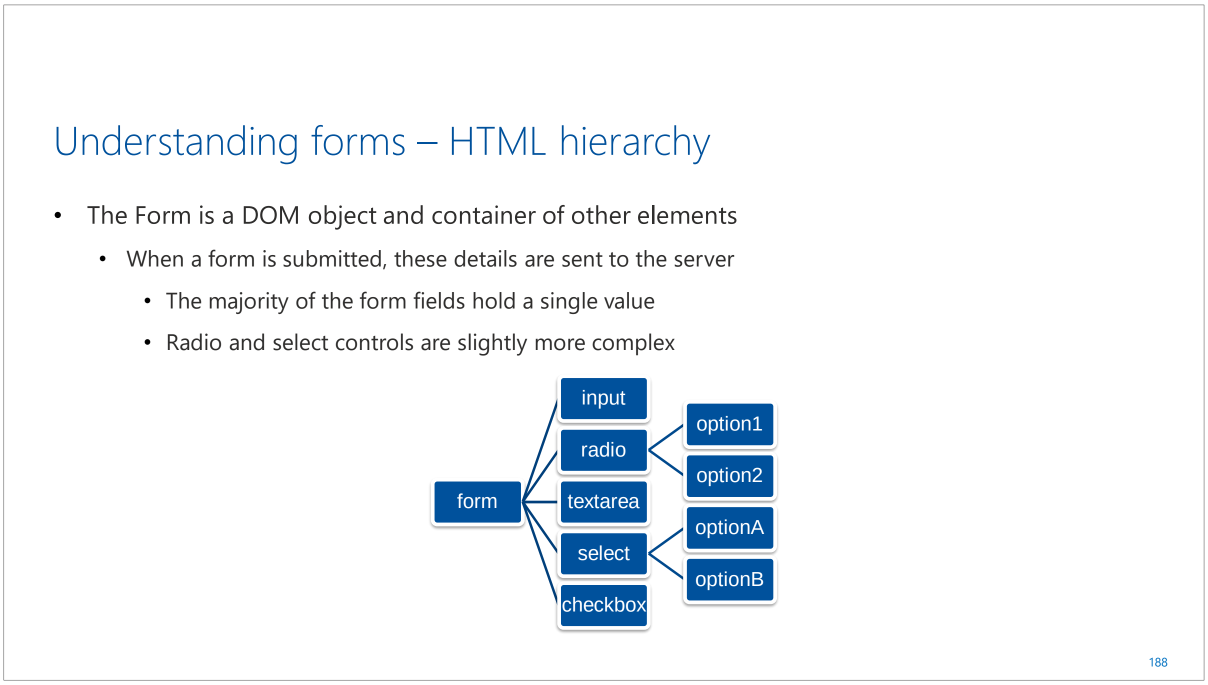
Form validation

- As we have seen, some browsers ship with validation
- These are JavaScript free client validation
- Uneven support and UI feedback may be more trouble than benefit
  - You can tell a browser to switch it off
  - Still benefiting from the semantic types

```
<form novalidate>
    <input type="email" id="addr">
    <input type="submit" value="Subscribe">
</form>
```

187

Even if you choose to disable the JavaScript free validation and go for your own controls, the new semantic types are useful. With them, we can define a clearer sense of meaning to the form creators intentions



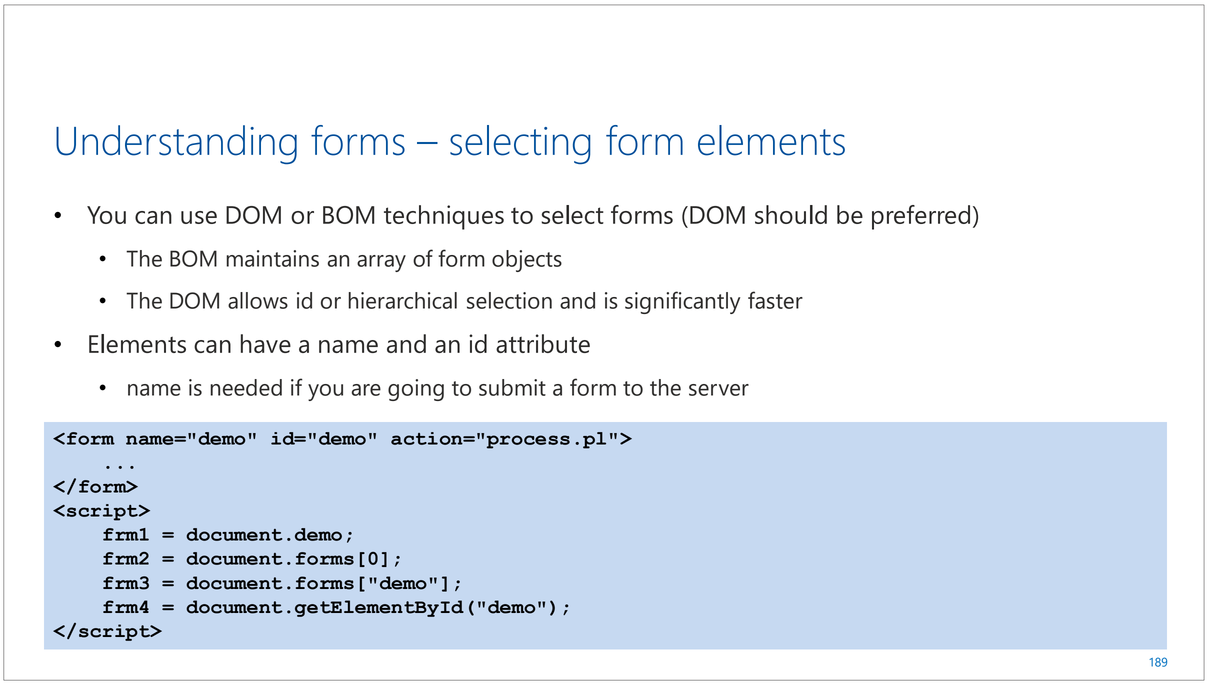
Understanding forms – HTML hierarchy

- The Form is a DOM object and container of other elements
  - When a form is submitted, these details are sent to the server
    - The majority of the form fields hold a single value
    - Radio and select controls are slightly more complex

form
input
radio
textarea
select
checkbox
option1
option2
optionA
optionB

188

When you submit data to a server, you send all form DOM objects contained within it. Most of these controls are simple to work with, they have a value attribute that holds the current user entry. Certain multi-option controls, like select and radio, are slightly more complex to work with.



Understanding forms – selecting form elements

- You can use DOM or BOM techniques to select forms (DOM should be preferred)
  - The BOM maintains an array of form objects
  - The DOM allows id or hierarchical selection and is significantly faster
- Elements can have a name and an id attribute
  - name is needed if you are going to submit a form to the server

```
<form name="demo" id="demo" action="process.pl">
    ...
</form>
<script>
    frm1 = document.demo;
    frm2 = document.forms[0];
    frm3 = document.forms["demo"];
    frm4 = document.getElementById("demo");
</script>
```

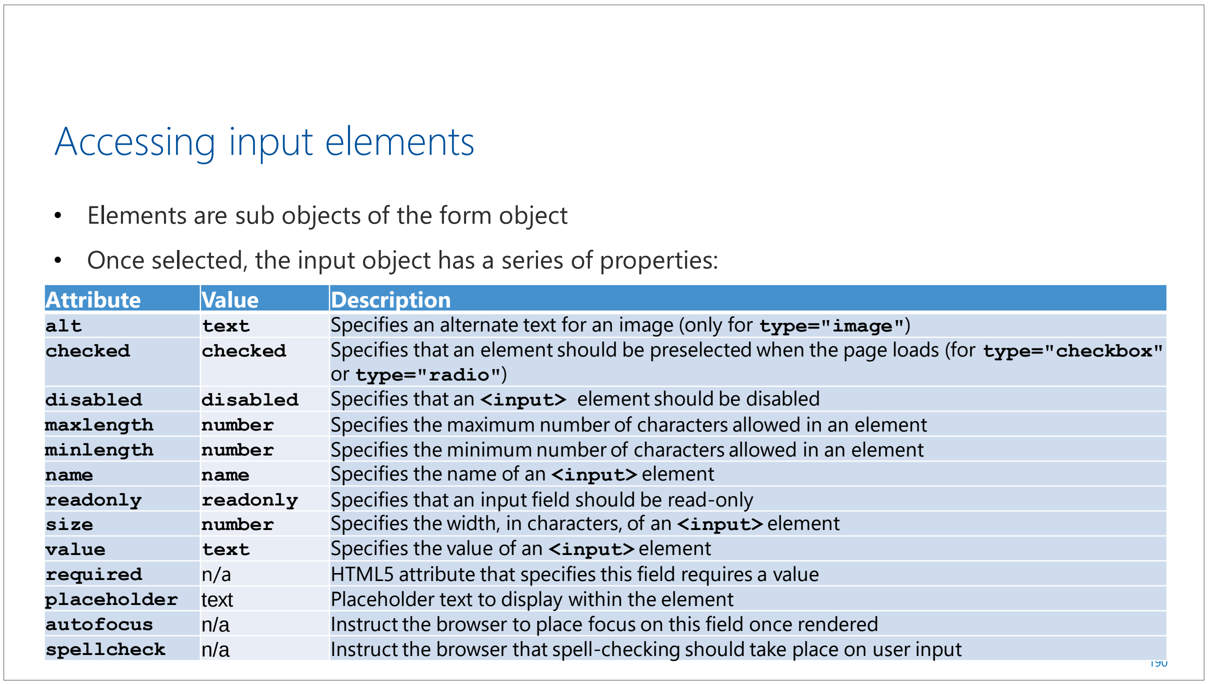
189

Selecting forms is interesting, there are two distinct approaches that could be used. This was one of the original JavaScript functionalities originating back in Netscape in pre DOM Programming. The BOM approach still remains, where the document object maintains an array of form objects upon the page.

Since the advent of DOM based Programming, we have been able to reference by the ID attribute. This is the preferred approach as the selector engine is a faster way of retrieving elements.

HTML has always been designed to be backwards compatible and as a result standard form submission builds up a string of name value pairs. However, an ID will need to be provided as an additional attribute. In most situation, the id and the name will be the same to create a consistent model for Programming.

In certain scenarios the form’s submit action will be prevented in its entirety, this commonly happens in AJAX based activity and will be discussed later in the course.



Accessing input elements

- Elements are sub objects of the form object
- Once selected, the input object has a series of properties:

Attribute	Value	Description
`alt`	**`text`**	Specifies an alternate text for an image (only for **`type="image"`**)
`checked`	**`checked`**	Specifies that an element should be preselected when the page loads (for **`type="checkbox"`** or **`type="radio"`**)
`disabled`	**`disabled`**	Specifies that an **`<input>`** element should be disabled
`maxlength`	**`number`**	Specifies the maximum number of characters allowed in an element
`minlength`	**`number`**	Specifies the minimum number of characters allowed in an element
`name`	**`name`**	Specifies the name of an **`<input>`** element
`readonly`	**`readonly`**	Specifies that an input field should be read-only
`size`	**`number`**	Specifies the width, in characters, of an **`<input>`** element
`value`	**`text`**	Specifies the value of an **`<input>`** element
`required`	n/a	HTML5 attribute that specifies this field requires a value
`placeholder`	text	Placeholder text to display within the element
`autofocus`	n/a	Instruct the browser to place focus on this field once rendered
`spellcheck`	n/a	Instruct the browser that spell-checking should take place on user input

190

The majority of form controls are input-based. Only the early Netscape breaks of the HTML 3 standard with tags such as <textarea> and select are different. This helps us a great deal to have a common interface for programming-style behaviour.

Once you reference the DOM element, you can access the appropriate property. The most important being value:

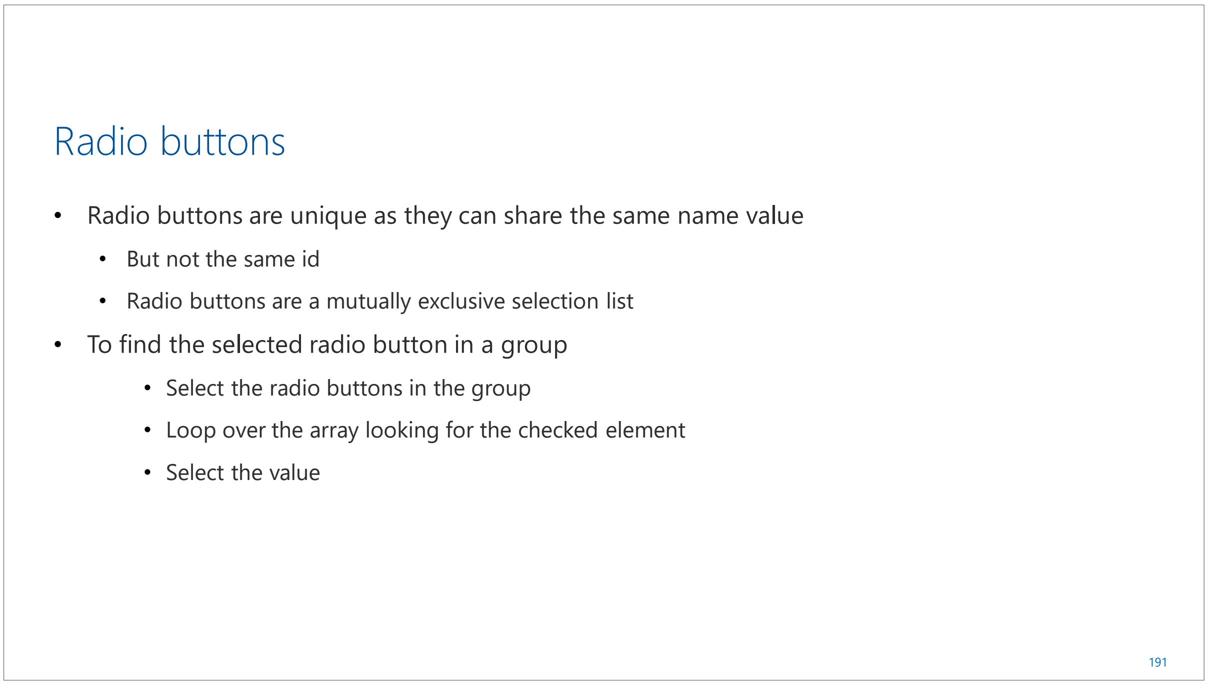
```
document.getElementById('uname').value
```

The following methods are also available:

The `focus()` method moves the input focus to the particular element. This is useful, for example, when a particular field has failed validation and you want the user to change its contents.

The `blur()` method moves the focus away from the element.

Elements that allow the user to enter text (textarea, text, and password) also support a `select()` method. This highlights and selects the contents of the field. Note that `select()` does not imply `focus();` if you want to select an element's contents and allow the user's typing to appear in the element, you need to use both `focus()` and `select()`.



Radio buttons

- Radio buttons are unique as they can share the same name value
  - But not the same id
  - Radio buttons are a mutually exclusive selection list
- To find the selected radio button in a group
  - Select the radio buttons in the group
  - Loop over the array looking for the checked element
  - Select the value

191

Radio buttons are a very useful tool in gathering user-input without them typing in user data. Their shared property name makes life very simple for us with modern selector techniques. Within the radio button group, only one radio button can be selected at one time. This is located with the checked property.

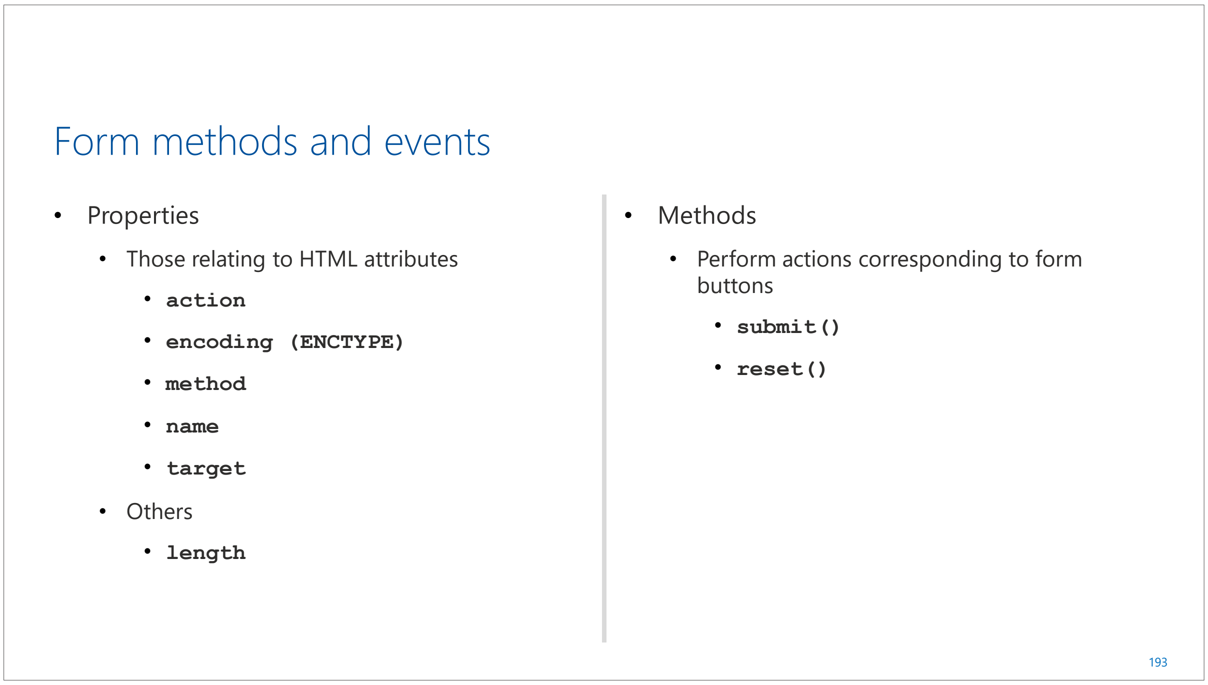
With older selectors, the following code will achieve the same result:

```
var paymentMethod = document.getElementsByName('cardtype');
for ( var i = 0; i &lt; paymentMethod.length; i++) {
    if(paymentMethod[i].checked) {
        console.log(paymentMethod[i].value);
    }
}
```

Accessing select options

- The selected element holds several useful properties to know what options have been selected
  - **`selectedIndex`**: the index of the first selected item
  - **`selectedOptions`**: An **`HTMLCollection`** representing the set of **`<option>`** elements that are selected
  - **`value`**: a string representing the value of the first selected item

```
let select = document.querySelector('select');
console.log(select.value);
```



Form methods and events

- Properties
  - Those relating to HTML attributes
    - **`action`**
    - **`encoding (ENCTYPE)`**
    - **`method`**
    - **`name`**
    - **`target`**
  - Others
    - **`length`**
- Methods
  - Perform actions corresponding to form buttons
    - **`submit()`**
    - **`reset()`**

193

The elements array and its objects are the most useful properties of the form object, but it does have some others:

The action, encoding, method, name, and target properties are all reflections of the HTML attributes of the form. The encoding property reflects the ENCTYPE attribute, and the others all reflect the named attributes.

The length attribute specifies the number of elements in the form, and, as such, has the same value as the length property of the elements array.

The form object has two methods:

`submit()`, which submits the form immediately, without waiting for the user to press the submit button.

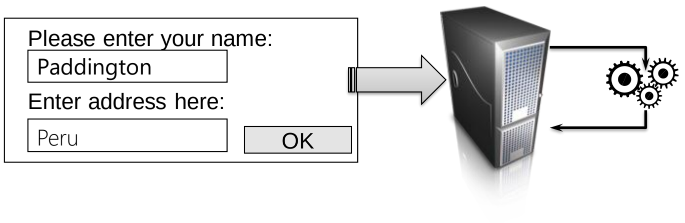
`reset()`, which clears the form contents and sets all fields back to their default values.

Understanding forms – Form submission (GET)

- A form is submitted when a button is pressed
  - type="submit" or type="image"
- Form data is sent to server along with URL request

```
GET /…/enrol.aspx?username=Paddington&address=Peru HTTP/1.0
```

- JavaScript can intercept form submission for validation

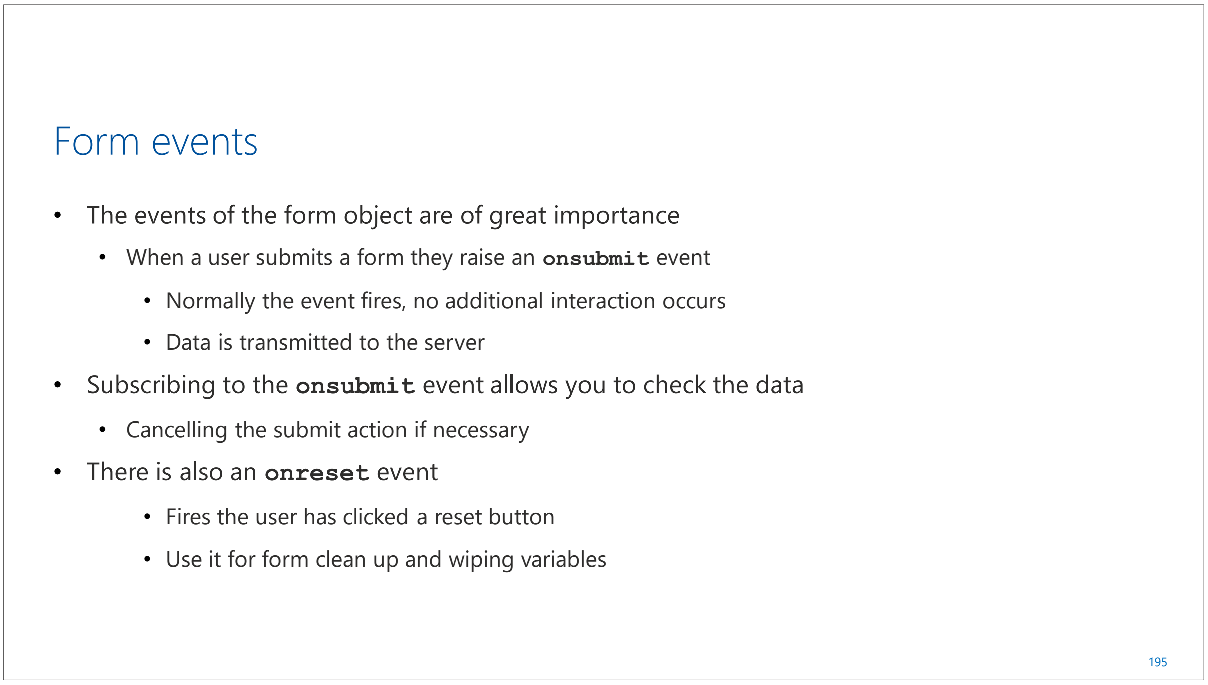


Please enter your name: Paddington

Enter address here: Peru

All data entry into a form is managed entirely by the client browser – there is no interaction with the server at this point. Eventually, the user takes some action that causes the browser to send the data to the server. This happens when they click on either a TYPE=SUBMIT button or on a TYPE=IMAGE element. In either case, the browser sends a request to the resource named in the ACTION attribute of the form, and passes the form data as a parameter with this request.

Using JavaScript, it is possible to intervene between this submit request and the data actually being sent to the server. This gives you the opportunity to validate the data of the form locally, and to prevent the form from being submitted, if the data is not valid. Doing so can substantially reduce the load on the server, reduce the amount of network traffic and reduce the time the user has to wait to discover his input was invalid.

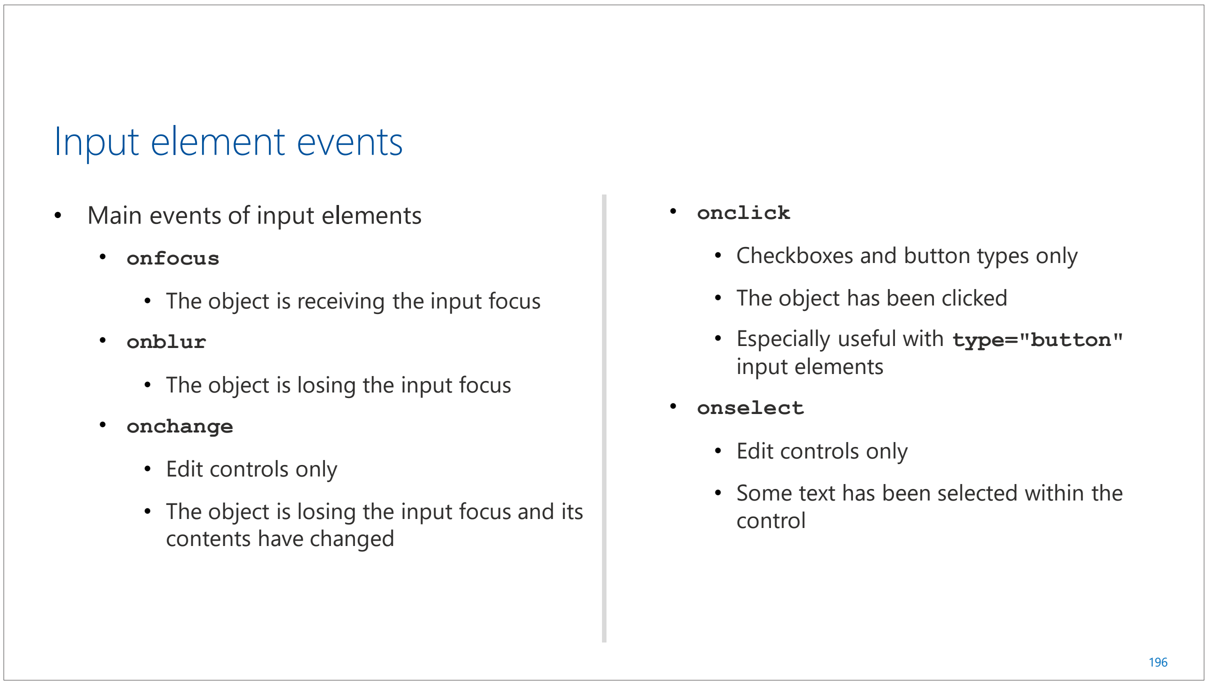


Form events

- The events of the form object are of great importance
  - When a user submits a form they raise an **`onsubmit`** event
    - Normally the event fires, no additional interaction occurs
    - Data is transmitted to the server
- Subscribing to the **`onsubmit`** event allows you to check the data
  - Cancelling the submit action if necessary
- There is also an **`onreset`** event
    - Fires the user has clicked a reset button
    - Use it for form clean up and wiping variables

195

Forms have just two event handlers: `onsubmit` and `onreset`. `onsubmit` is called if the user presses a submit button, and `onreset` is called if they press a reset button. The `onsubmit` event handler can prevent the form from being submitted.



Input element events

- Main events of input elements
  - **`onfocus`**
    - The object is receiving the input focus
  - **`onblur`**
    - The object is losing the input focus
  - **`onchange`**
    - Edit controls only
    - The object is losing the input focus and its contents have changed
- **`onclick`**
  - Checkboxes and button types only
  - The object has been clicked
  - Especially useful with **`type="button"`** input elements
- **`onselect`**
  - Edit controls only
  - Some text has been selected within the control

196

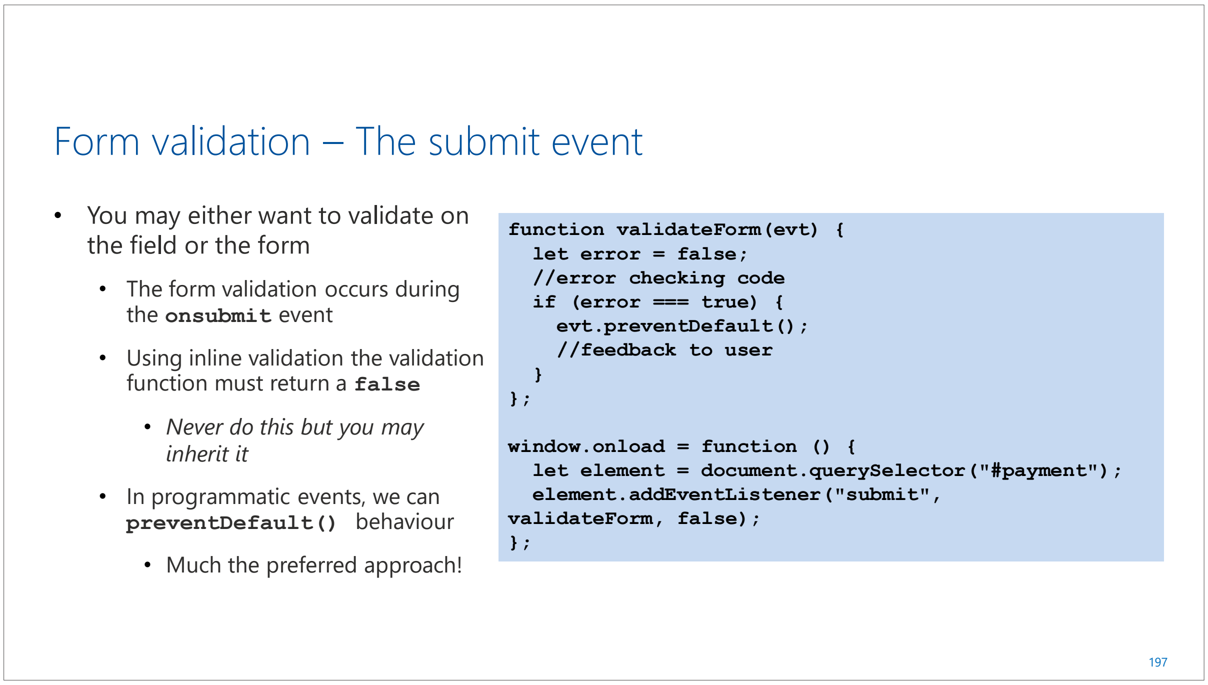
Input elements have the following event handlers:

- `onfocus` is called when the element receives the focus
- `onblur` is called when the element loses the focus

In addition, editable elements have the following event handlers:

- `onchange` – The element is losing the focus and its contents have been altered
- `onselect` – The selection has been altered in the control

Finally, checkboxes, radio buttons, and normal (push) buttons also have an `onclick` event handler. The `type="button"` input element is not ordinarily useful in a form as the button has not HTML usage (unlike submit & reset). However, with the aid of JavaScript and the `onclick` event handler, it is possible to give generic push buttons some behaviour or actions to perform. In fact, this makes it possible to write a simple interactive client-side application, and there is no need for a submit button or an `action` attribute on the form at all. A calculator is an example of a possible application.



Form validation – The submit event

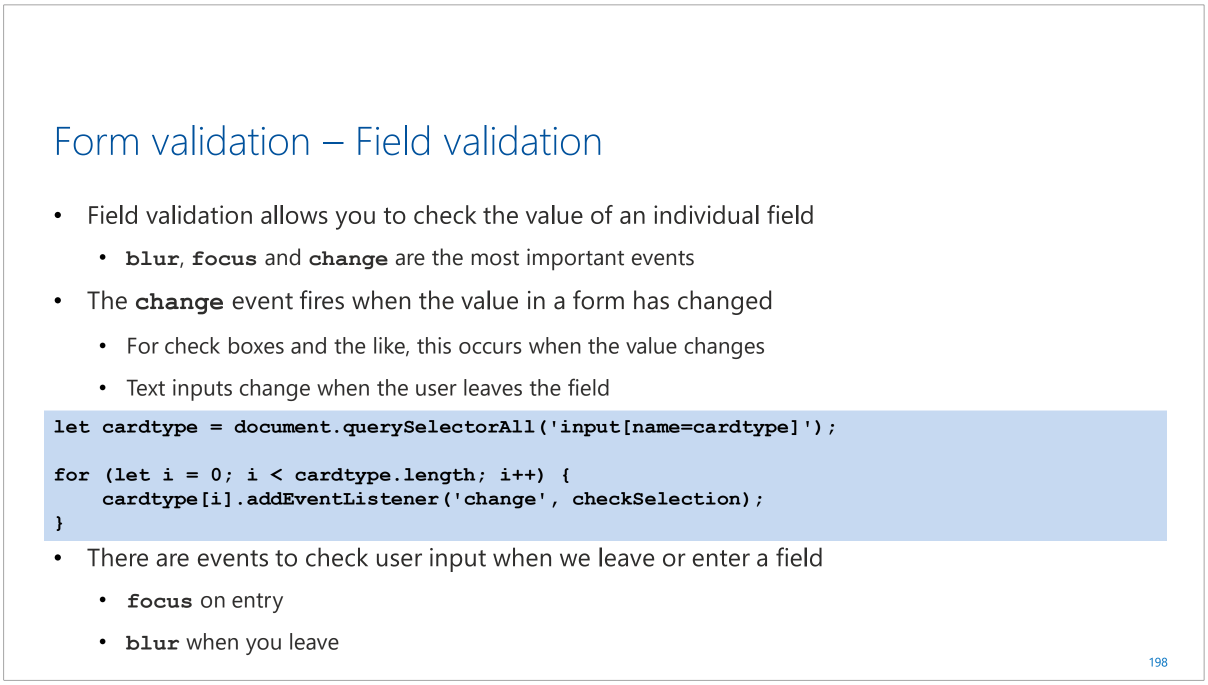
- You may either want to validate on the field or the form
  - The form validation occurs during the **`onsubmit`** event
  - Using inline validation the validation function must return a **`false`**
    - *Never do this but you may inherit it*
  - In programmatic events, we can **`preventDefault()`** behaviour
    - Much the preferred approach!

```
function validateForm(evt) {
  let error = false;
  //error checking code
  if (error === true) {
    evt.preventDefault();
    //feedback to user
  }
};

window.onload = function () {
  let element = document.querySelector("#payment");
  element.addEventListener("submit",
validateForm, false);
};
```

197

A great deal of the code above should, by now, feel familiar to you. The **submit** event needs to be interrupted if we are to stop the form just sending its data to the server. In contemporary JavaScript, all browsers understand **preventDefault** and as a result, we can simply catch the raised event and prevent the default action, i.e. the form submission from occurring.

Note the error variable above. This variable works as a Boolean flag within the function. Within the code, we would check each submission issue and if an error was found set the flag to be true. Before we leave the function, we simply check if the error has been switched to true. If it has, we prevent the submission and feedback to the user.



Form validation – Field validation

- Field validation allows you to check the value of an individual field
  - **`blur`**, **`focus`** and **`change`** are the most important events
- The **`change`** event fires when the value in a form has changed
  - For check boxes and the like, this occurs when the value changes
  - Text inputs change when the user leaves the field

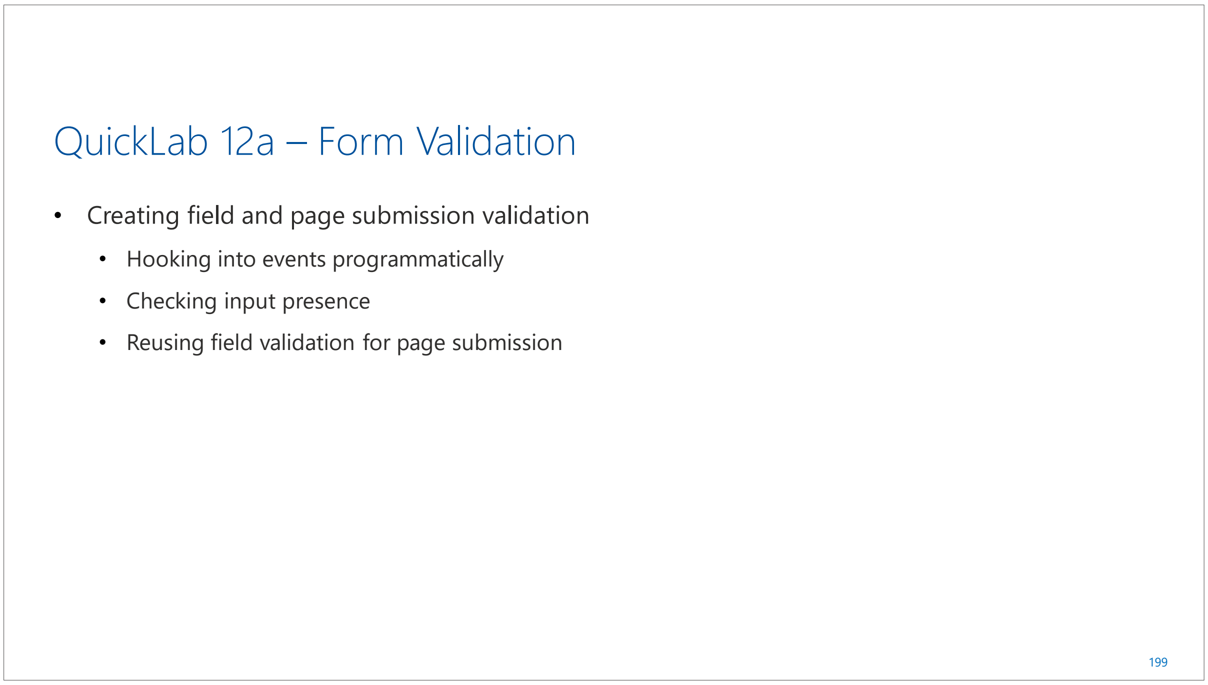
```
let cardtype = document.querySelectorAll('input[name=cardtype]');

for (let i = 0; i < cardtype.length; i++) {
    cardtype[i].addEventListener('change', checkSelection);
}
```

- There are events to check user input when we leave or enter a field
  - **`focus`** on entry
  - **`blur`** when you leave

198

We can check individual fields using blur, focus and change events. The majority of change-based events will occur on the change event. When a user moves out of a currently focused field and the value has changed, the event is raised. Focus can be very useful in applying some CSS functionality to the element to highlight it for the user.

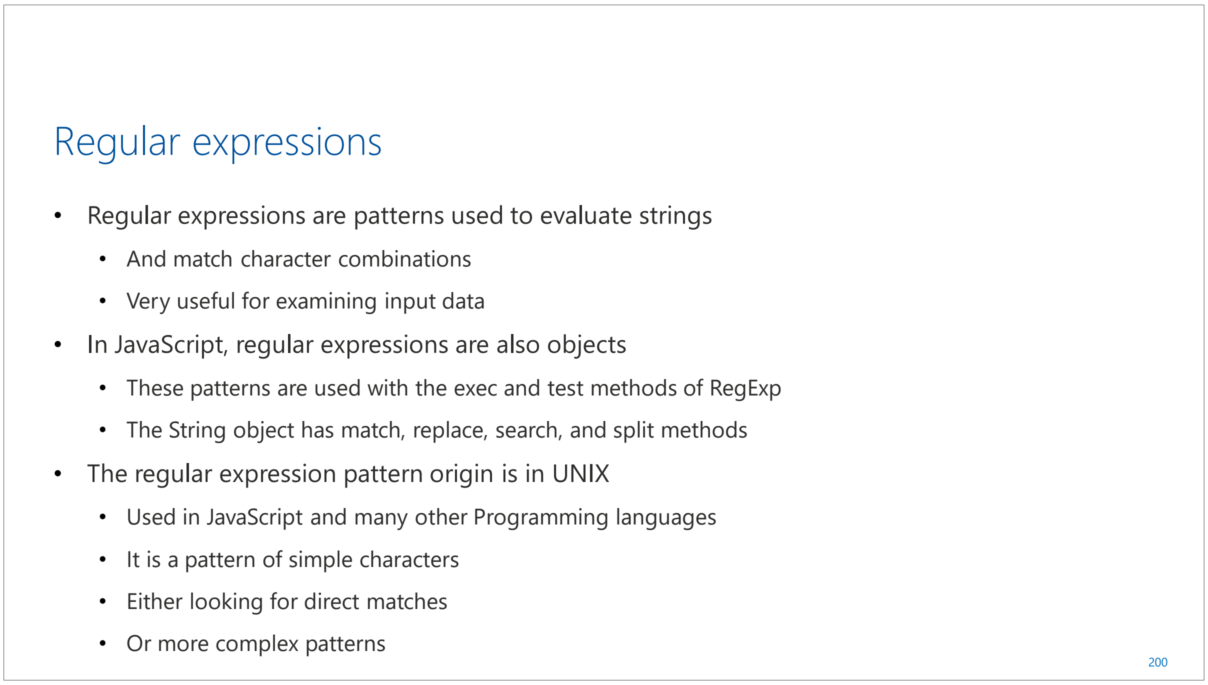
A few critically important notes an usability. Never use prevent default as part of any change, or blur event, and never redirect a user back to a previously focused field. This is frustrating to the end user and terrible in terms of accessibility for users with disabilities.



QuickLab 12a – Form Validation

- Creating field and page submission validation
  - Hooking into events programmatically
  - Checking input presence
  - Reusing field validation for page submission

199



Regular expressions

- Regular expressions are patterns used to evaluate strings
  - And match character combinations
  - Very useful for examining input data
- In JavaScript, regular expressions are also objects
  - These patterns are used with the exec and test methods of RegExp
  - The String object has match, replace, search, and split methods
- The regular expression pattern origin is in UNIX
  - Used in JavaScript and many other Programming languages
  - It is a pattern of simple characters
  - Either looking for direct matches
  - Or more complex patterns

200

Regular expressions use special (and, at first, somewhat confusing) codes to detect **patterns** in strings of text. For example, if you're presenting your visitors with an HTML form to enter their details, you might have one field for their phone number. Now let's face it: some site visitors are better at following instructions than others. Even if you put a little hint next to the text field indicating the required format of the phone number (e.g.: "(XXX) XXX-XXXX" for North American numbers), some people are going to get it wrong. Writing a script to check every character of the entered string to ensure that all the numbers are where they belong, with parentheses and a dash in just the right spots, would be a pretty tedious bit of code to write. And a telephone number is a relatively simple case! What if you had to check that a user had indeed entered an email address or, worse yet, a URL?

Regular expressions provide a quick and easy way of matching a string to a pattern. In our phone number example, we could write a simple regular expression and use it to check in one quick step whether or not any given string is a properly formatted phone number. We'll explore this example a little further, once we've taken care of a few technical details.

In JavaScript source code, a regular expression is written in the form of /pattern/modifiers where "pattern" is the regular expression itself, and "modifiers" are a series of characters indicating various options. The "modifiers" part is optional. This syntax is borrowed from Perl.

Creating a regular expression

- Regular expressions are objects and must be instantiated
  - There are two ways to do this

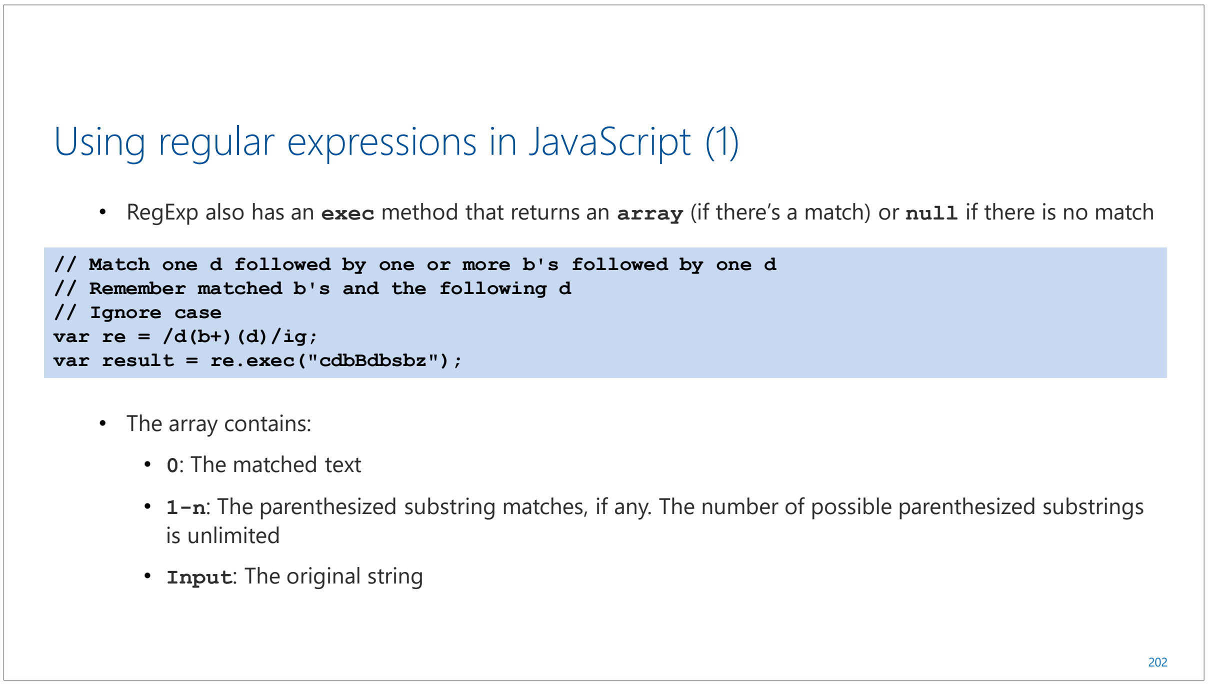
```
let re = new RegExp("ab+c");
let quickRe = /ab+c/;
```

- The regular expression object can be used to evaluate a string
  - The example below checks a string and returns a Boolean

```
var str = "bus";
/bus/.test(str); //only matches on bus
```

- Excellent online testing tool: http://regexr.com/

201



Using regular expressions in JavaScript (1)

- RegExp also has an **`exec`** method that returns an **`array`** (if there's a match) or **`null`** if there is no match

```
// Match one d followed by one or more b's followed by one d
// Remember matched b's and the following d
// Ignore case
var re = /d(b+)(d)/ig;
var result = re.exec("cdbBdbsbz");
```

- The array contains:
  - **`0`**: The matched text
  - **`1-n`**: The parenthesized substring matches, if any. The number of possible parenthesized substrings is unlimited
  - **`Input`**: The original string

202

By default, JavaScript regular expressions are case sensitive and only search for the first match in any given string. By adding the g (for global) and i (for ignore case) modifiers after the second /, you can make a regular expression search for all matches in the string and ignore case, respectively.

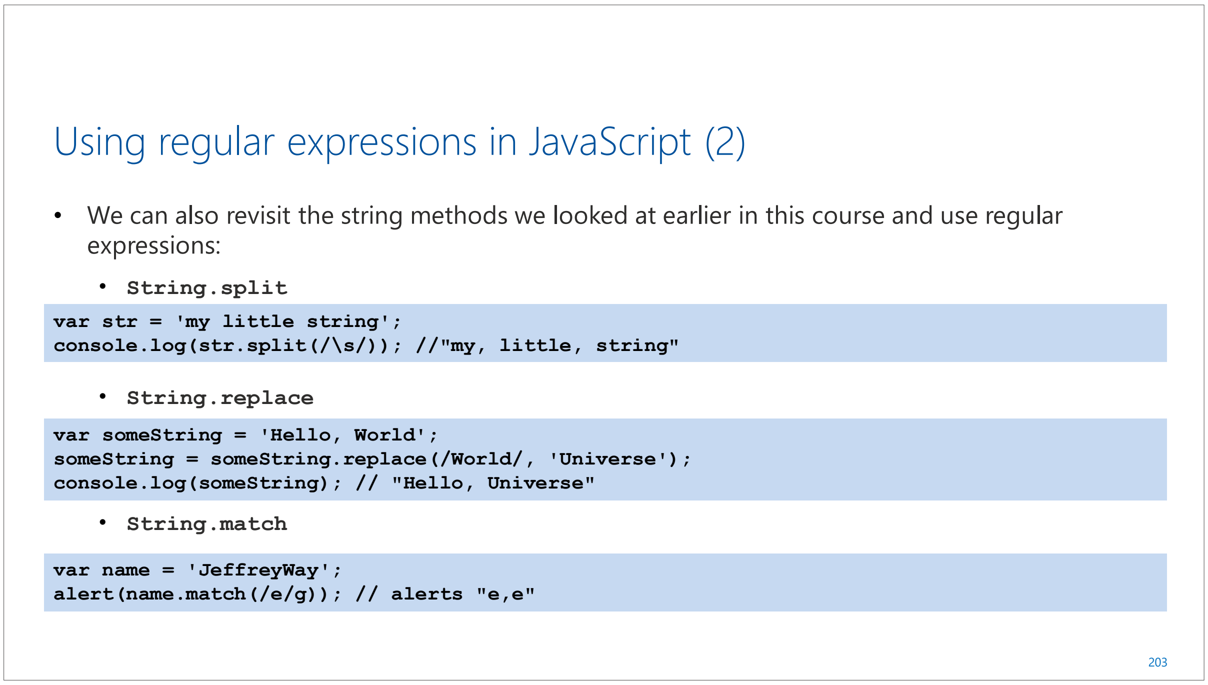
The following key methods are of use to us:

`Test()`

This one accepts a single string parameter and returns a Boolean indicating whether or not a match has been found. If you don't necessarily need to perform an operation with the a specific matched result – for instance, when validating a username – "test" will do the job just fine.

`Exec()`

Executes a search for a match in a specified string. Returns a result array, or null. If the match succeeds, the exec method returns an array and updates properties of the regular expression object. The returned array has the matched text as the first item, and then one item for each capturing parenthesis that matched containing the text that was captured. If the match fails, the exec method returns null. If you are executing a match simply to find true or false, use the test method or the string search method.



Using regular expressions in JavaScript (2)

- We can also revisit the string methods we looked at earlier in this course and use regular expressions:
  - `String.split`

```
var str = 'my little string';
console.log(str.split(/\s/)); //"my, little, string"
```

  - `String.replace`

```
var someString = 'Hello, World';
someString = someString.replace(/World/, 'Universe');
console.log(someString); // "Hello, Universe"
```

  - `String.match`

```
var name = 'JeffreyWay';
alert(name.match(/e/g)); // alerts "e,e"
```

203

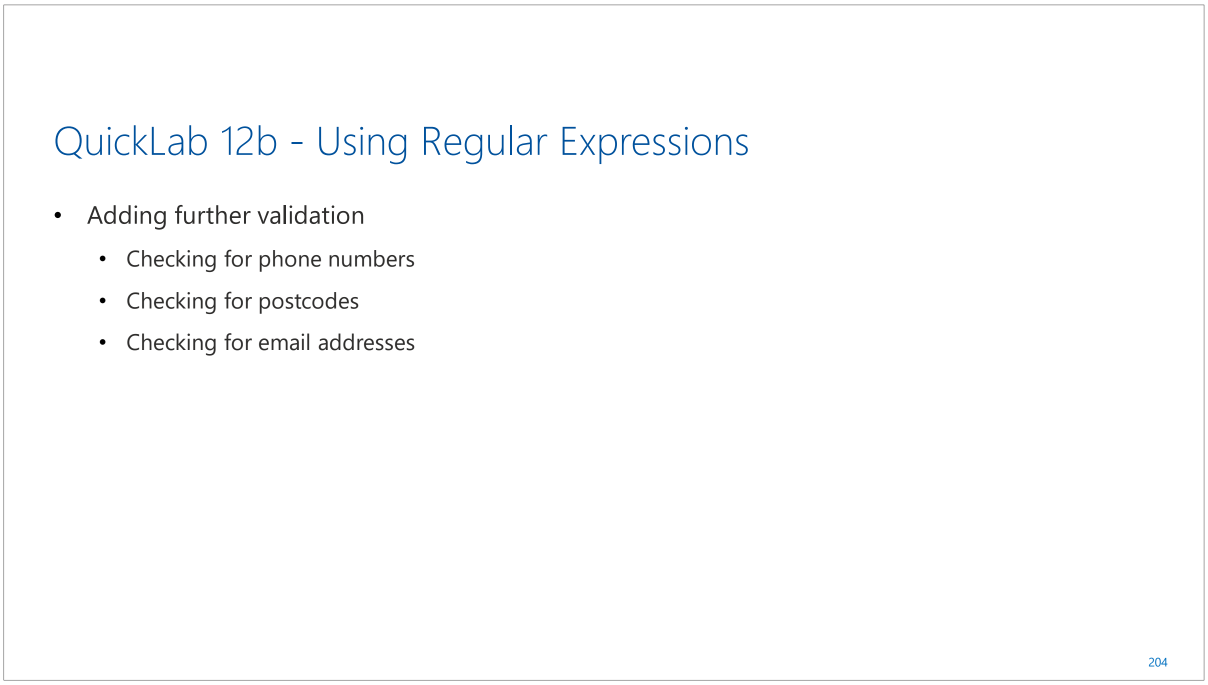
As you might expect, the “replace” method allows you to replace a certain block of text, represented by a string or regular expression, with a different string. You’re most likely already familiar with the split method. It accepts a single regular expression that represents where the “split” should occur. Please note that we can also use a string if we’d prefer. In the example above, by passing “\s” – representing a single space – we’ve now split our string into an array. If you need to access one particular value, just append the desired index.

`Replace()`

As you might expect, the “replace” method allows you to replace a certain block of text, represented by a string or regular expression, with a different string. Using the replace method does not automatically overwrite the value the variable, we must reassign the returned value back to the variable.

`Match()`

Unlike the “test” method, “match()” will return an array containing each match found. In the code above, there are actually two e’s in the string “JeffreyWay.” Once again, we must use the “g” modifier to declare a “global search”.



QuickLab 12b - Using Regular Expressions

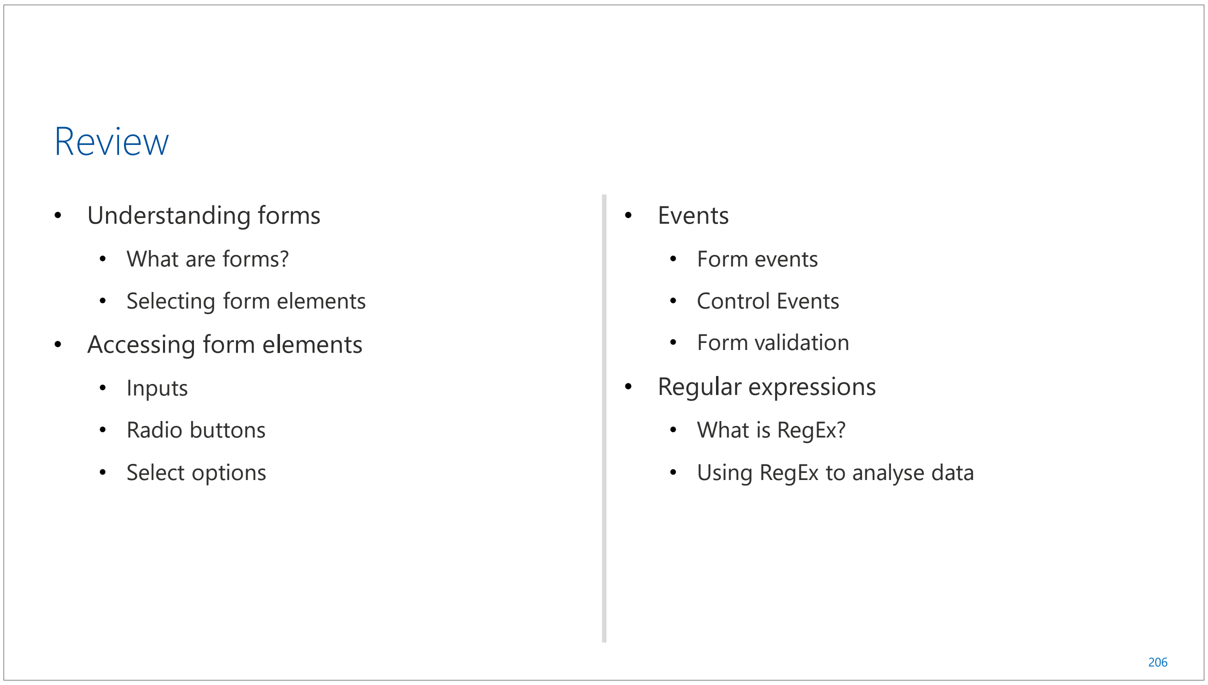
- Adding further validation
  - Checking for phone numbers
  - Checking for postcodes
  - Checking for email addresses

204

Hackathon Part 1

- In this part Hackathon, you will build on a partially developed solution (whether that be your previous iteration or the provided starting point) for QA Cinemas' website by adding validation to the form. All the necessary tools, knowledge and techniques have been covered in the course so far.
- This part of the Hackathon is intended to help you develop your skills and knowledge to be able to use JavaScript to validate a 'Sign-Up' form for users of the QA Cinemas website before this is sent to be held on the server.

205



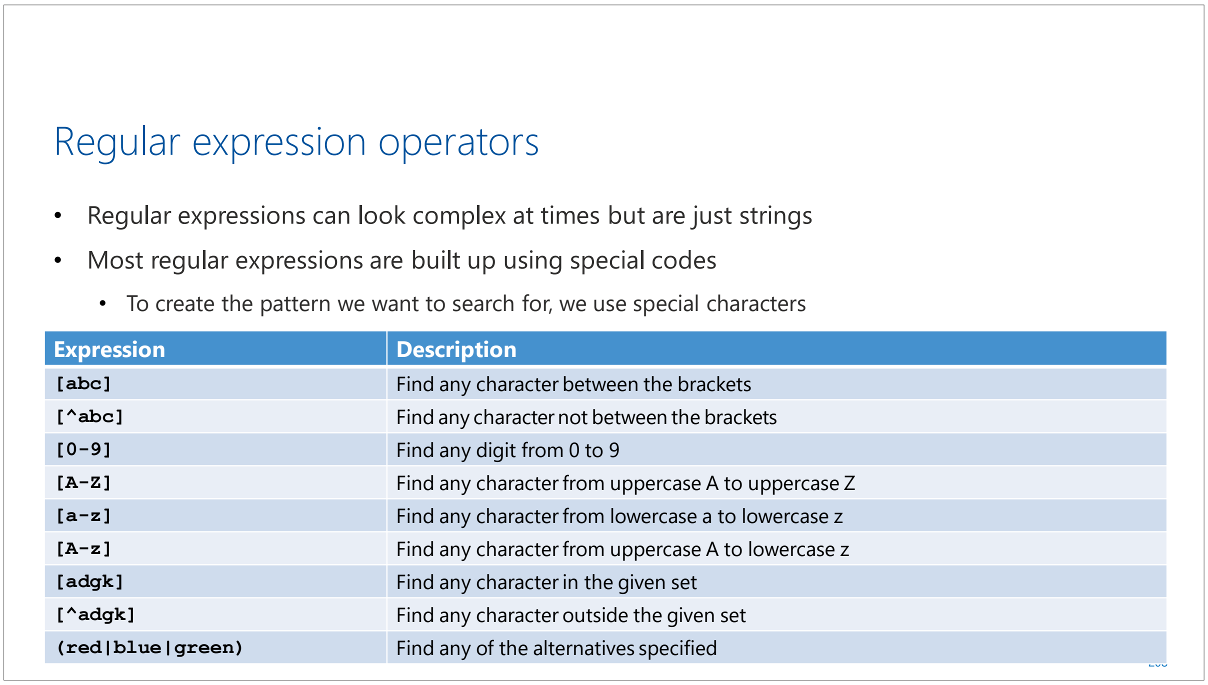
Review

- Understanding forms
  - What are forms?
  - Selecting form elements
- Accessing form elements
  - Inputs
  - Radio buttons
  - Select options
- Events
  - Form events
  - Control Events
  - Form validation
- Regular expressions
  - What is RegEx?
  - Using RegEx to analyse data

206

Appendix – More on Regular Expressions

JAVASCRIPT FUNDAMENTALS - EXTRA



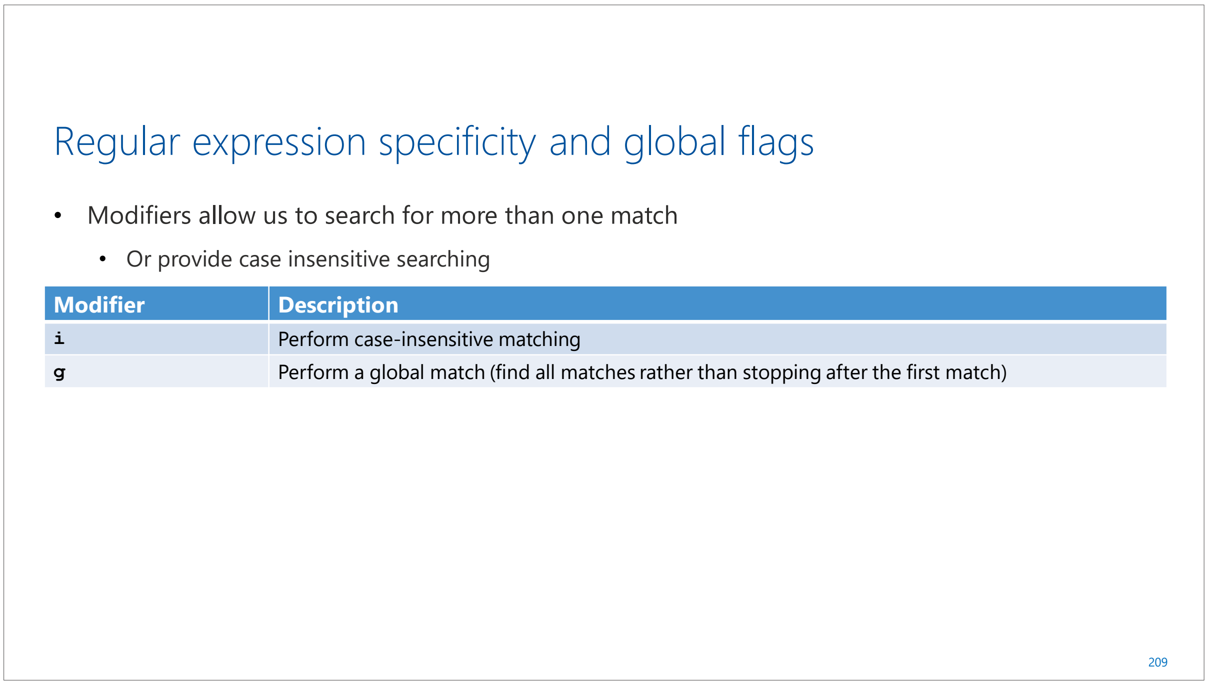
Regular expression operators

- Regular expressions can look complex at times but are just strings
- Most regular expressions are built up using special codes
  - To create the pattern we want to search for, we use special characters

Expression	Description
`[abc]`	Find any character between the brackets
`[^abc]`	Find any character not between the brackets
`[0-9]`	Find any digit from 0 to 9
`[A-Z]`	Find any character from uppercase A to uppercase Z
`[a-z]`	Find any character from lowercase a to lowercase z
`[A-z]`	Find any character from uppercase A to lowercase z
`[adgk]`	Find any character in the given set
`[^adgk]`	Find any character outside the given set
`(red|blue|green)`	Find any of the alternatives specified

Metacharacters provide special meaning to a search.

Metacharacter	Description
.	Find a single character, except newline or line terminator
\w	Find a word character
\W	Find a non-word character
\d	Find a digit
\D	Find a non-digit character
\s	Find a whitespace character
\S	Find a non-whitespace character
\b	Find a match at the beginning/end of a word
\B	Find a match not at the beginning/end of a word
\0	Find a NUL character
\n	Find a new line character
\f	Find a form feed character
\r	Find a carriage return character
\t	Find a tab character
\v	Find a vertical tab character
\xxx	Find the character specified by an octal number xxx
\xdd	Find the character specified by a hexadecimal number dd
\uxxxx	Find the Unicode character specified by a hexadecimal number xxxx



Regular expression specificity and global flags

- Modifiers allow us to search for more than one match
  - Or provide case insensitive searching

Modifier	Description
`i`	Perform case-insensitive matching
`g`	Perform a global match (find all matches rather than stopping after the first match)

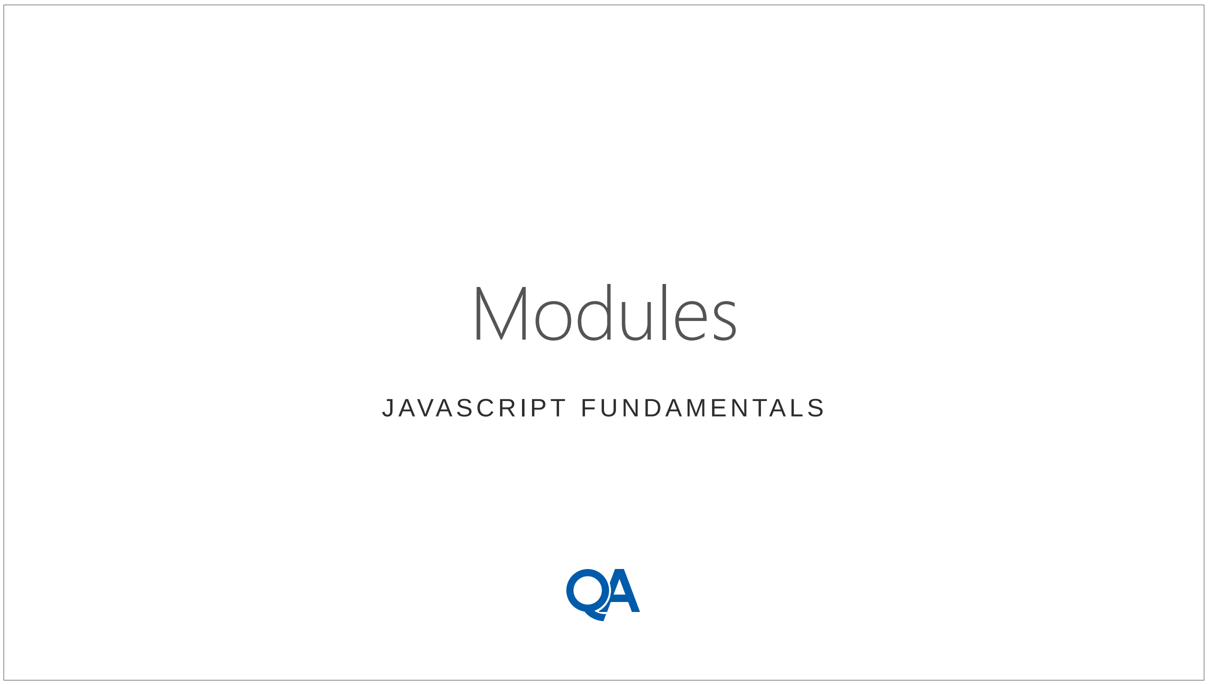
209

Regular expression specificity and global flags

- Quantifiers allow you to refine the search or sub-query a result

Quantifier	Description
`n+`	Matches any string with at least one n
`n*`	Matches any string with no occurrences of n
`n?`	Strings with zero or more occurrences of n
`n{x}`	Matches any string with a sequence of n's
`n$`	Matches any string with n at the end of it
`^n`	Matches any string with n at the beginning of it

210



Modules

JAVASCRIPT FUNDAMENTALS

Modules

- Any serious, sizable JavaScript project will benefit from modularization
  - Previously implemented using 3rd party libraries such as RequireJS
- ECMAScript 2015 Modules are built upon the export and import keywords
- Automatically "**`use strict`**"
- Have their own scope
- Can export any top-level **`function`**, **`class`**, **`var`**, **`let`** or **`const`**
- **`export`** and **`import`** must be used at the top-level, so no conditional loading

Modules

- Export using the export keyword

```
//circle-functions.js
export function area(r) {
  return Math.PI * Math.pow(r, 2)
}
```

- Import using the import keyword

```
//app.js
import {area} from "circle-functions";

console.log(area(5)); //78.53981633974483
```

Modules

- Import many items.

```
//circle-functions.js
export function area(r) {
  return Math.PI * Math.pow(r, 2)
}

export function circumference(r) {
  return 2 * Math.PI * r
}
```

```
//app.js
import {area, circumference} from "circle-functions";

console.log(area(5));                 //78.53981633974483
console.log(circumference(5));        //31.41592653589793
```

Modules

- Import many items.

```
//circle-functions.js
export function area(r) {
  return Math.PI * Math.pow(r, 2)
}

export function circumference(r) {
  return 2 * Math.PI * r
}
```

```
//app.js
import * as circle from "circle-functions";

console.log(circle.area(5));                 //78.53981633974483
console.log(circle.circumference(5));        //31.41592653589793
```

Modules

Export many items.

```
//circle-functions.js
export {area, circumference};

function area(r) {
  return Math.PI * Math.pow(r, 2)
}

function circumference(r) {
  return 2 * Math.PI * r
}
```

```
//app.js
import * as circle from "circle-functions";

console.log(circle.area(5));                //78.53981633974483
console.log(circle.circumference(5));       //31.41592653589793
```

```
//circle-functions.js
export {area, circumference};

function area(r) {
  return Math.PI * Math.pow(r, 2)
}

function circumference(r) {
  return 2 * Math.PI * r
}

export default {
  radius: 5
  centre: [0,0]
}
```

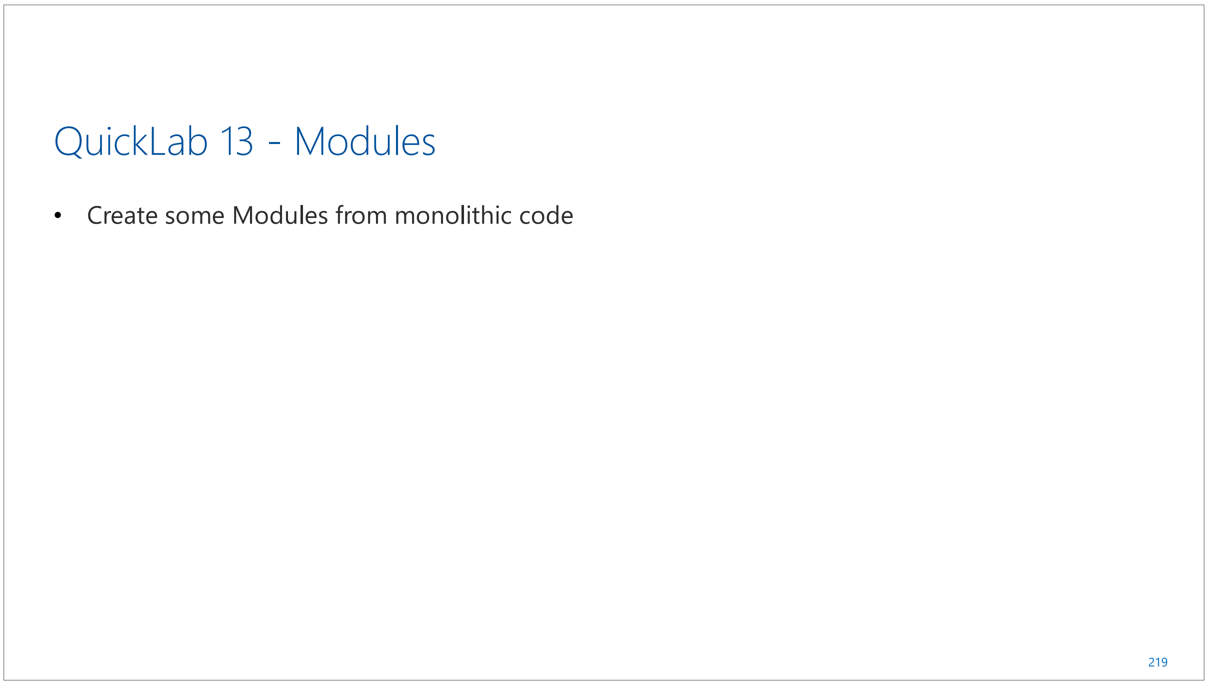
```
//app.js
import circle from "circle-functions";

console.log(circle); //{radius: 5, centre: [0,0]}
```

Export and Import default.

```
//circle-functions.js
export {area, circumference};

function area(r) {
  return Math.PI * Math.pow(r, 2)
}

function circumference(r) {
  return 2 * Math.PI * r
}

export default {
  radius: 5
  centre: [0,0]
}
```

```
//app.js
import circle, {area, circumference} from "circle-functions";

console.log(circle); //{radius: 5, centre: [0,0]}
console.log(area(circle.radius)); //78.53981633974483
```

Export and Import default.



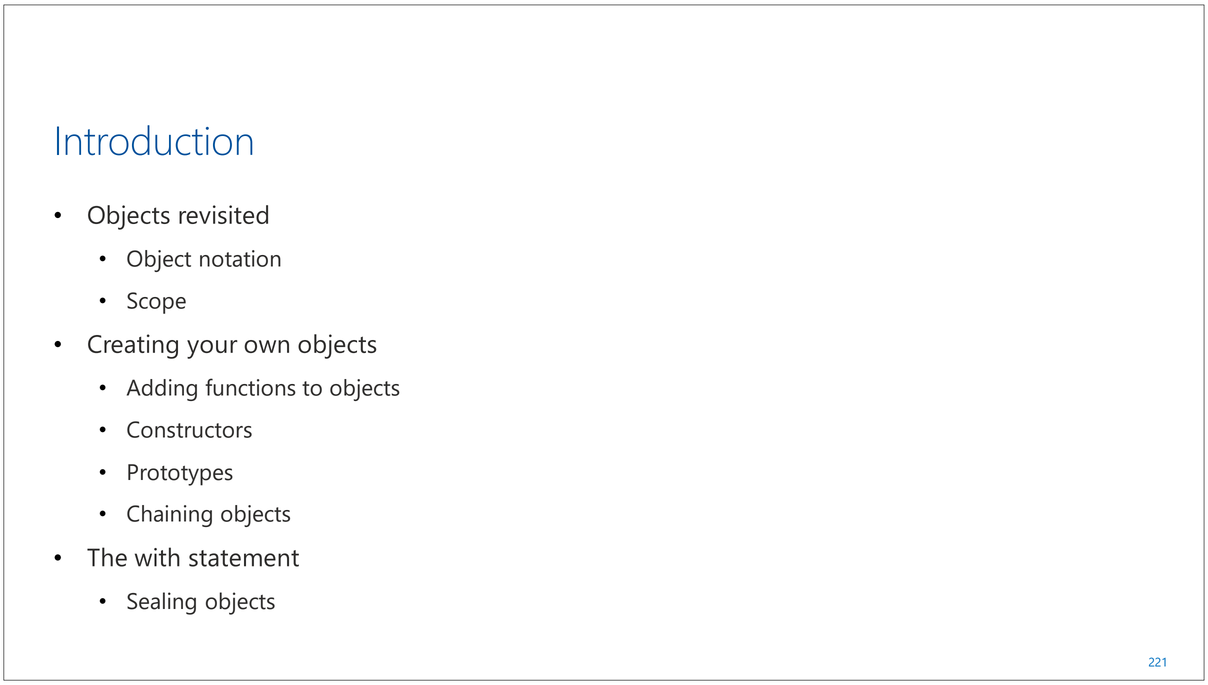
QuickLab 13 - Modules

- Create some Modules from monolithic code

219

Object Orientated JavaScript

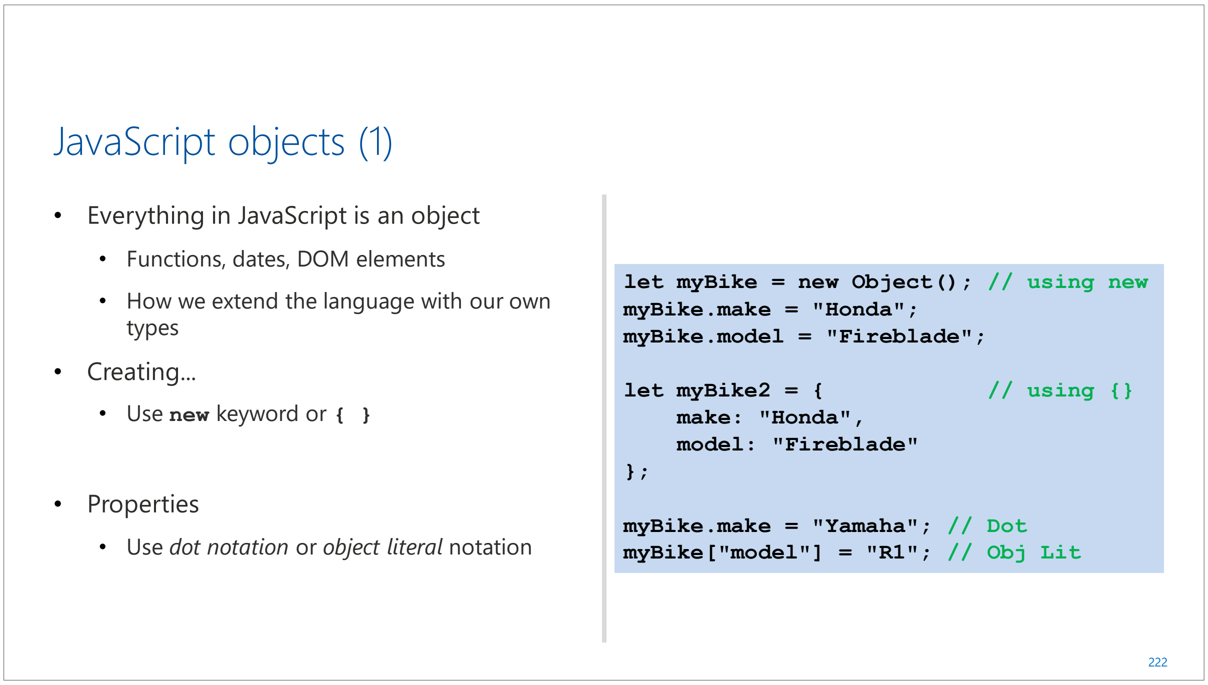
JAVASCRIPT FUNDAMENTALS



Introduction

- Objects revisited
  - Object notation
  - Scope
- Creating your own objects
  - Adding functions to objects
  - Constructors
  - Prototypes
  - Chaining objects
- The with statement
  - Sealing objects

221



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
```

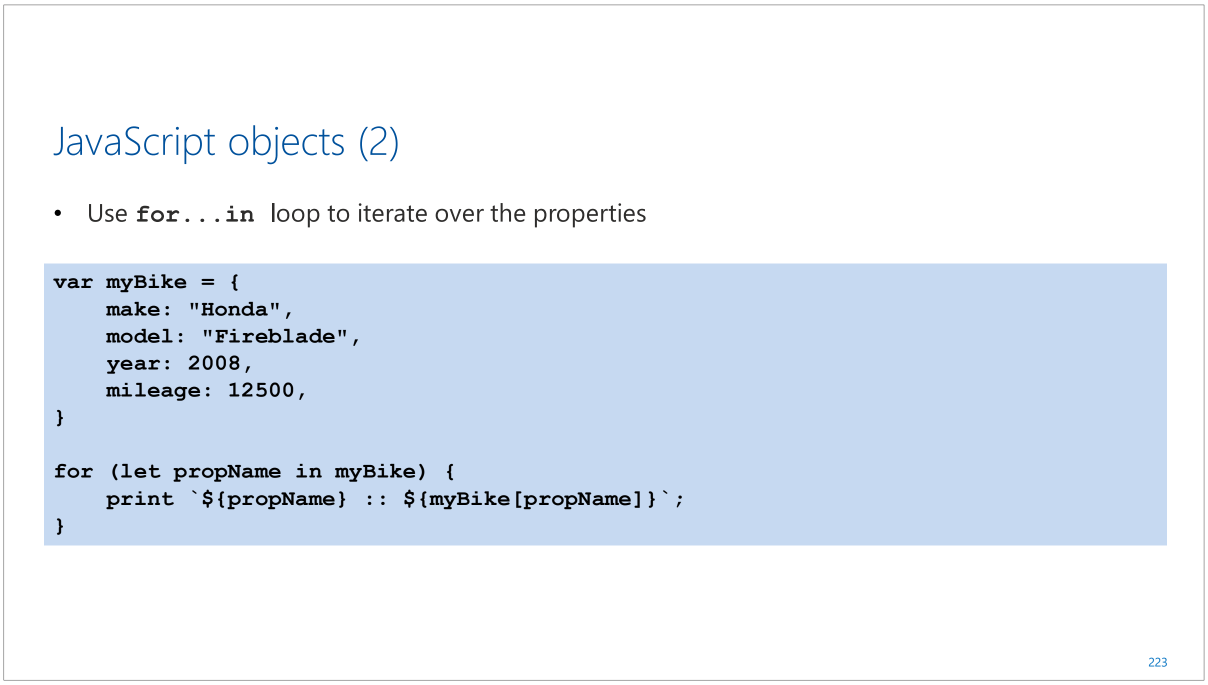
222

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)

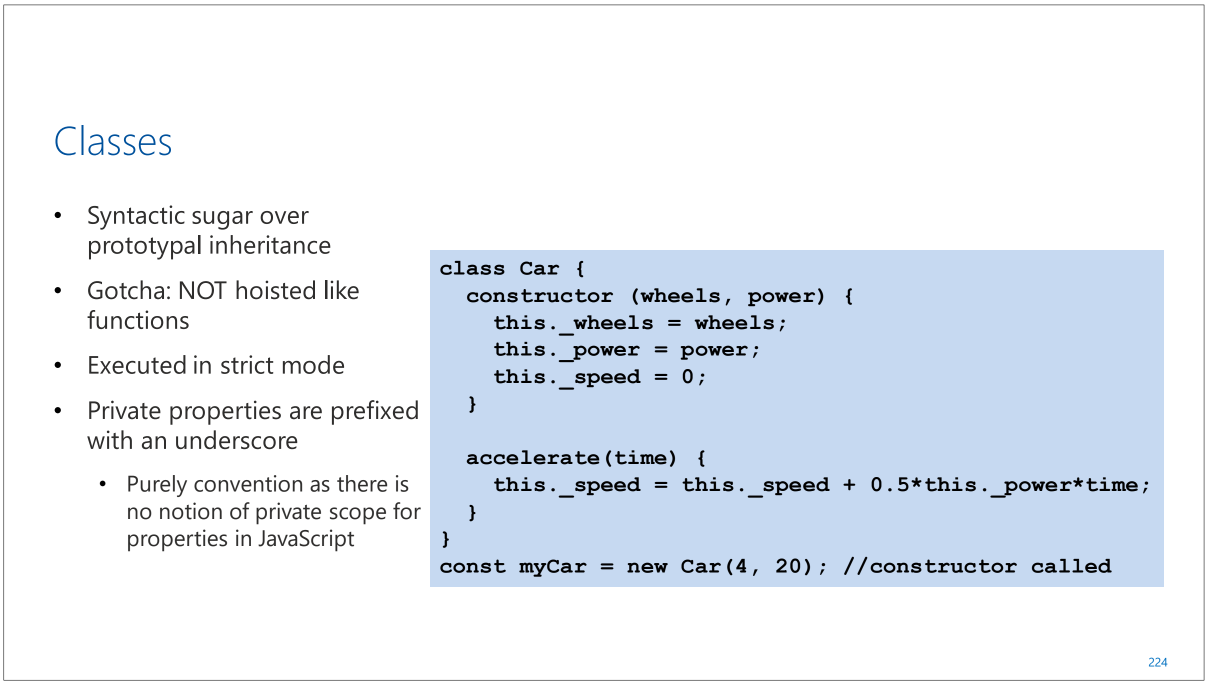
- Use **`for...in`** loop to iterate over the properties

```
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
```

224

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

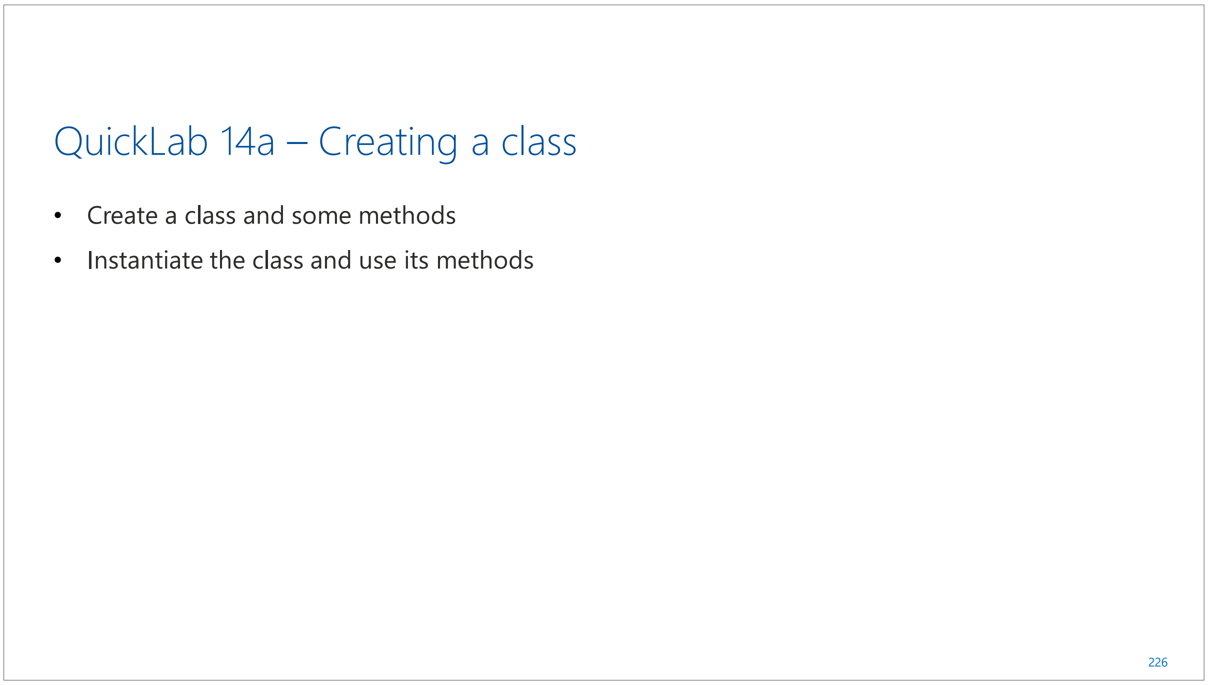
Methods can be created inside the class definition without using the function keyword or assigning to “this”.

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

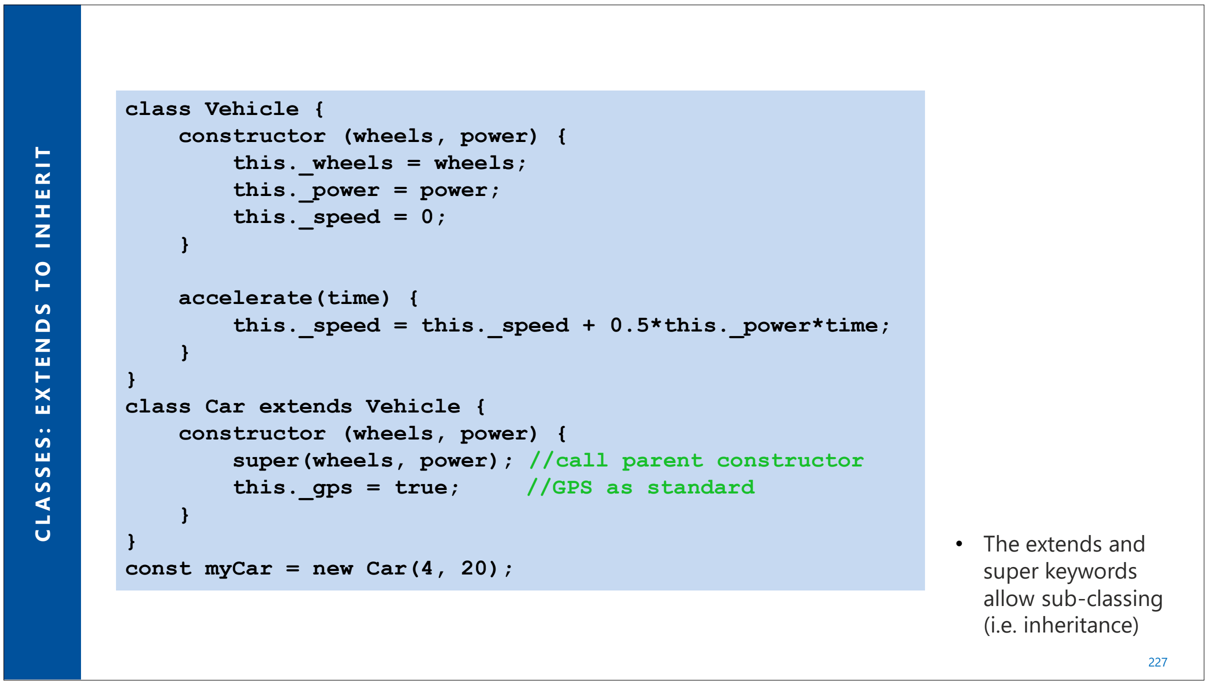
225



QuickLab 14a – Creating a class

- Create a class and some methods
- Instantiate the class and use its methods

226



CLASSES: EXTENDS TO INHERIT

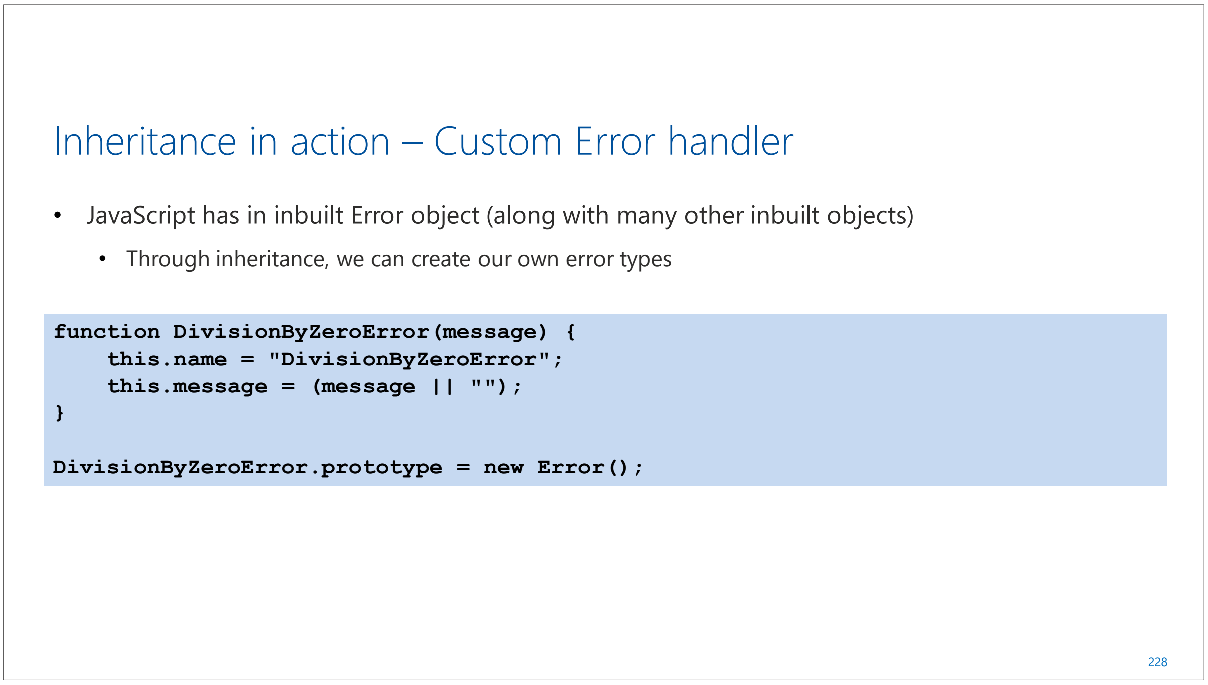
```
class Vehicle {
    constructor (wheels, power) {
        this._wheels = wheels;
        this._power = power;
        this._speed = 0;
    }

    accelerate(time) {
        this._speed = this._speed + 0.5*this._power*time;
    }
}
class Car extends Vehicle {
    constructor (wheels, power) {
        super(wheels, power); //call parent constructor
        this._gps = true;     //GPS as standard
    }
}
const myCar = new Car(4, 20);
```

- The extends and super keywords allow sub-classing (i.e. inheritance)

227

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();
```

228

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.

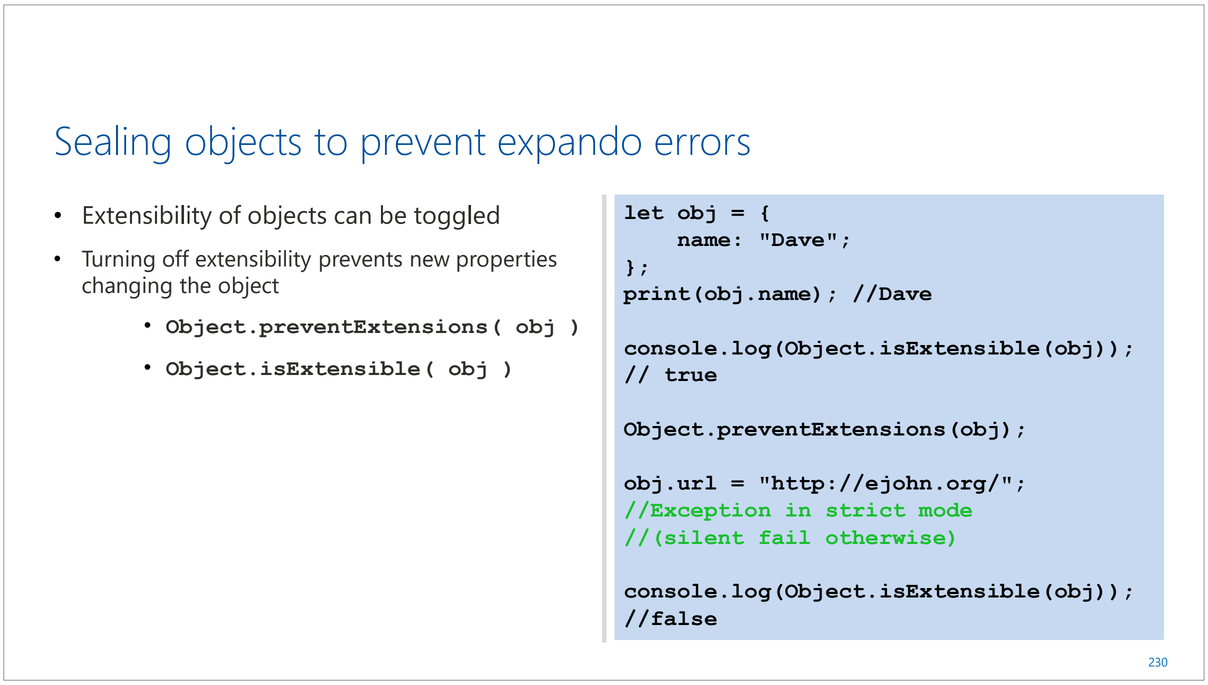
In this example, we are creating the “DivisionByZeroError” exception type. The function in the example acts as the constructor for our new type. The constructor takes care of assigning the “name” and “message” properties.

CLASSES: STATIC

```
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
```

- 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

229



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
```

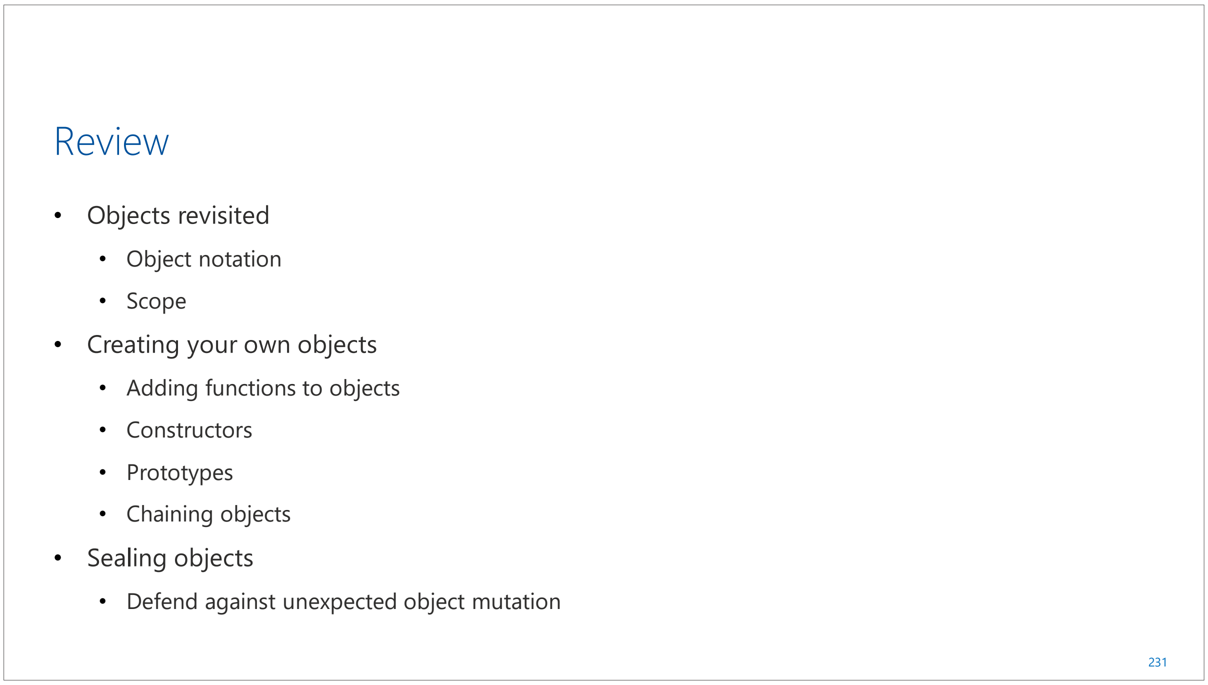
230

A new 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 )
```

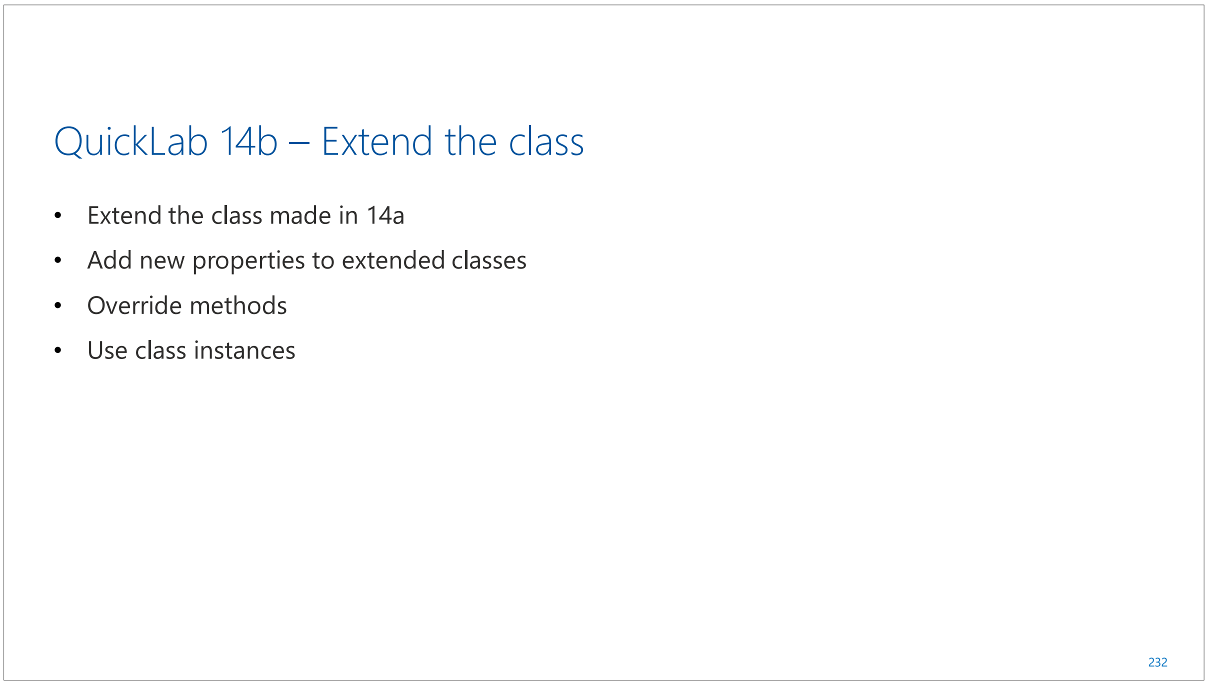
preventExtensions locks down an object and prevents any future property additions from occurring.



Review

- Objects revisited
  - Object notation
  - Scope
- Creating your own objects
  - Adding functions to objects
  - Constructors
  - Prototypes
  - Chaining objects
- Sealing objects
  - Defend against unexpected object mutation

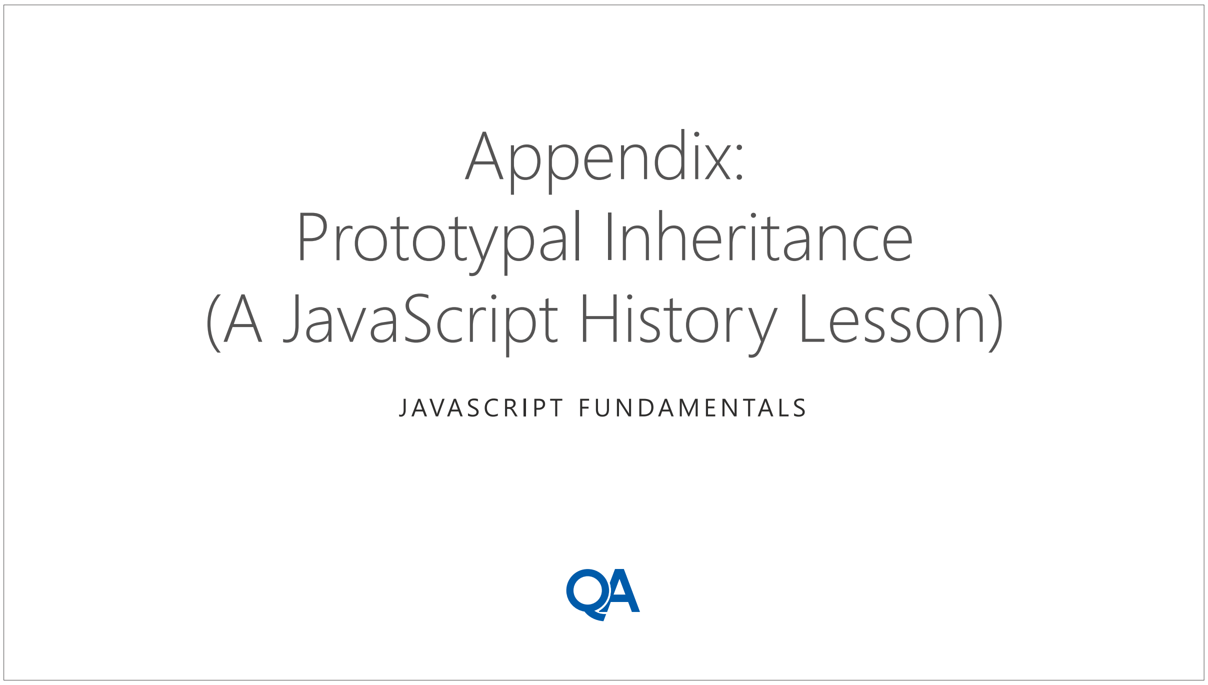
231



QuickLab 14b – Extend the class

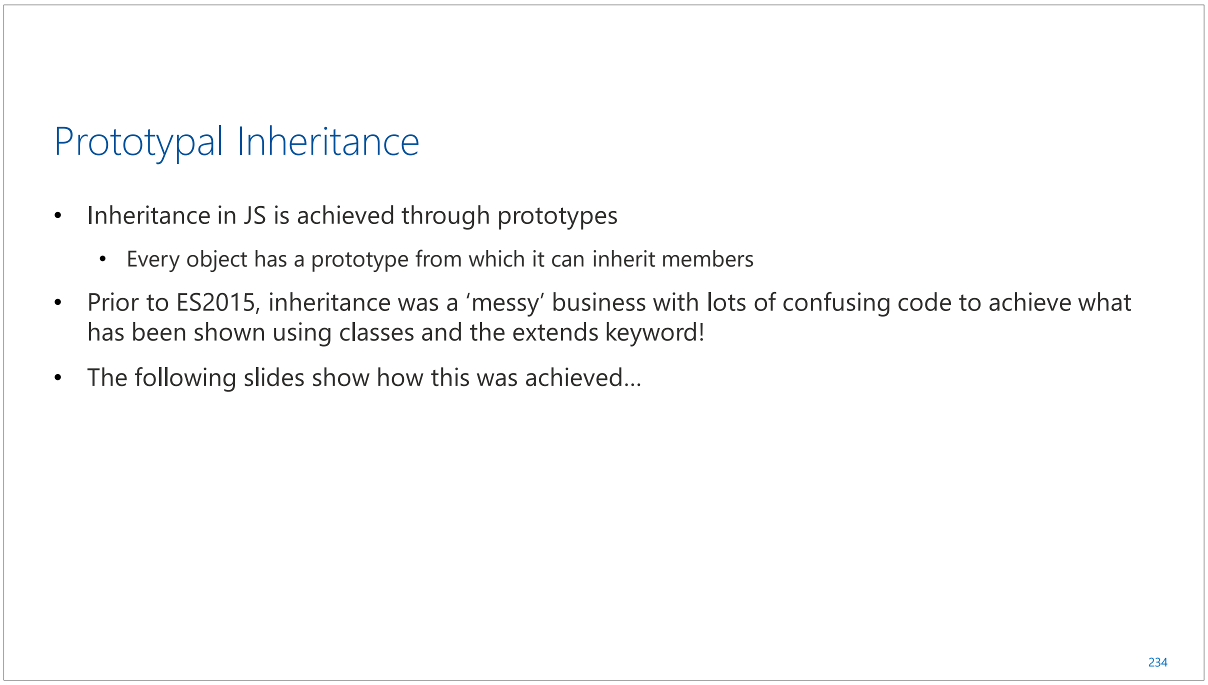
- Extend the class made in 14a
- Add new properties to extended classes
- Override methods
- Use class instances

232



Appendix: Prototypal Inheritance (A JavaScript History Lesson)

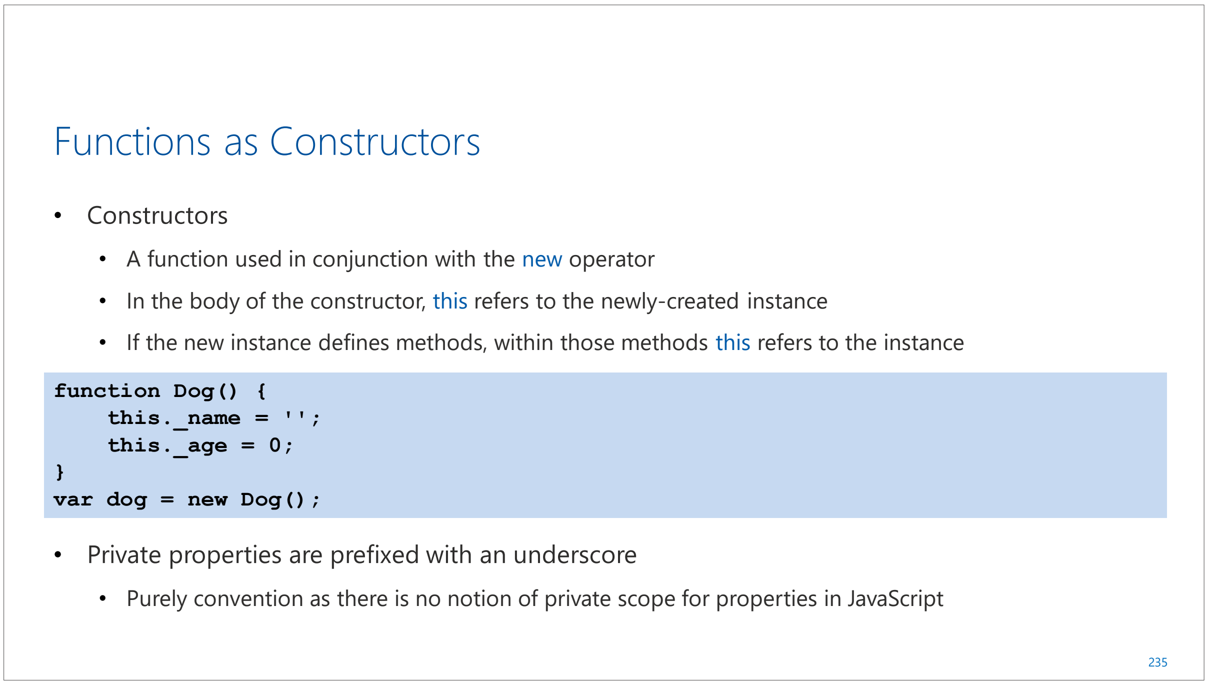
JAVASCRIPT FUNDAMENTALS



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…

234



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

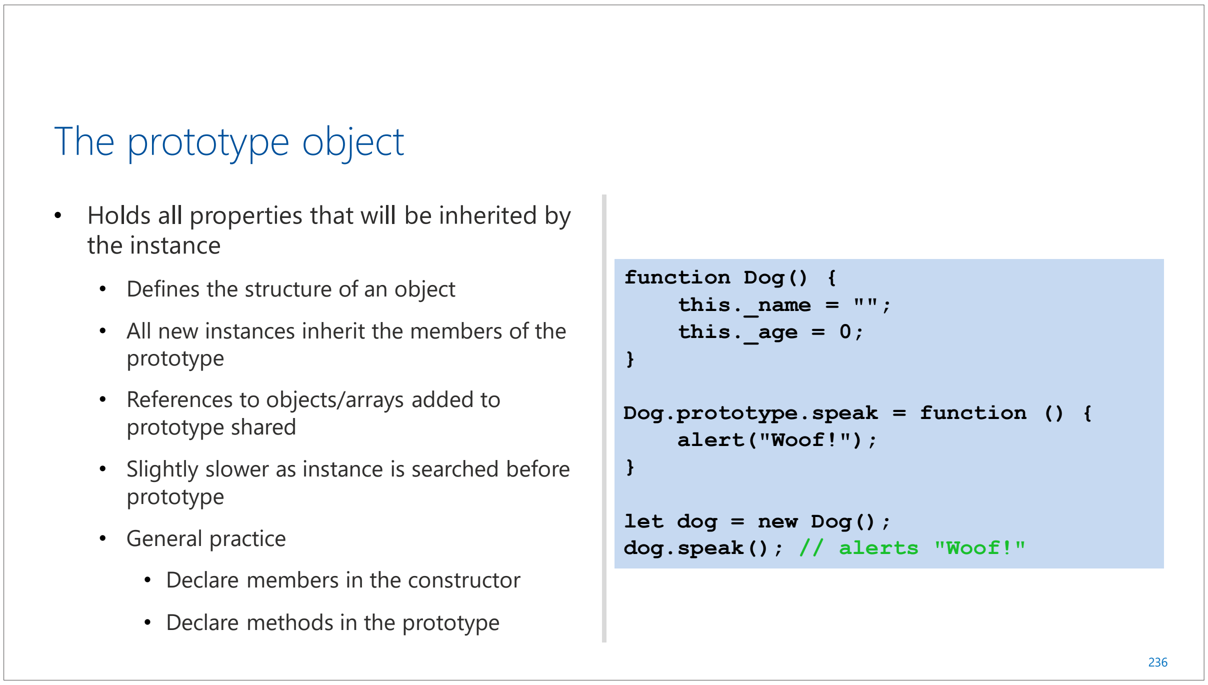
235

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

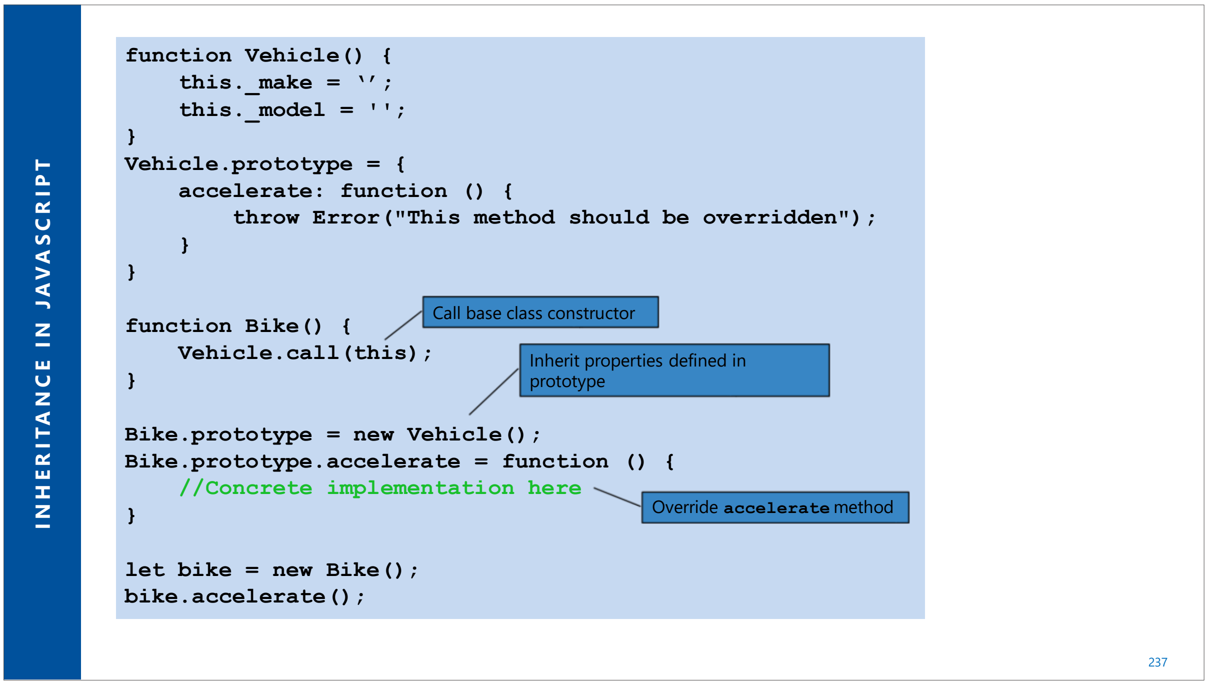
```
function Dog() {
    this._name = "";
    this._age = 0;
}

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

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

236

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

```
function Vehicle() {
    this._make = '';
    this._model = '';
}
Vehicle.prototype = {
    accelerate: function () {
        throw Error("This method should be overridden");
    }
}

function Bike() {
    Vehicle.call(this);
}

Bike.prototype = new Vehicle();
Bike.prototype.accelerate = function () {
    //Concrete implementation here
}

let bike = new Bike();
bike.accelerate();
```

Call base class constructor
Inherit properties defined in prototype
Override accelerate method
237

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.

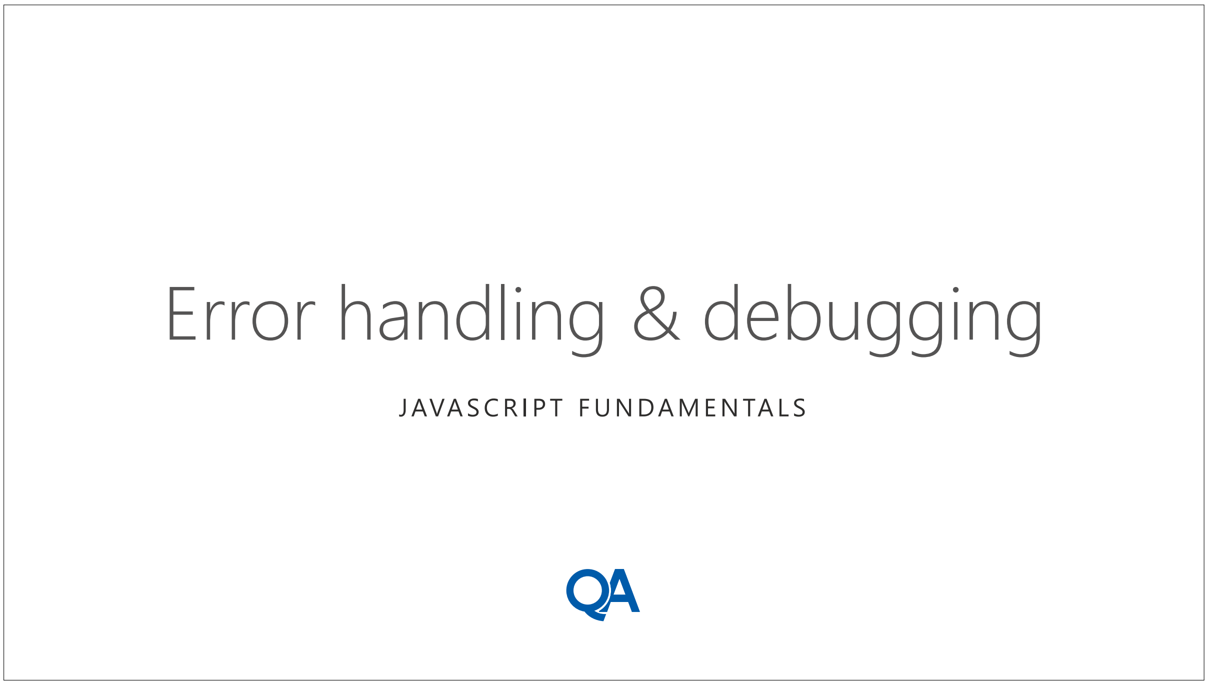
We use the call method, which allows us to change the context of the call so that **this** will refer to a different object (we pass in our new Bike – **this**). So, when the Vehicle constructor is called, the _make and _model will be added to the new Bike – we have inherited the field members.

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.

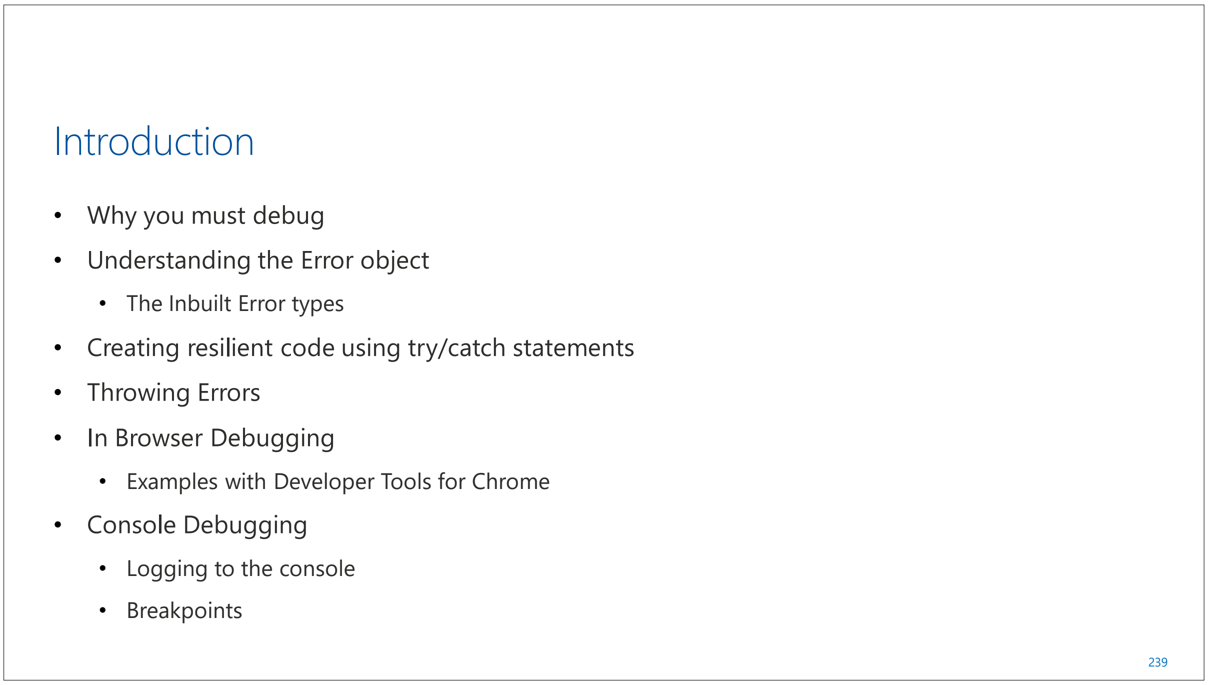
We can then use the Bike class and it will have all the capabilities of a Bike that inherits from Vehicle.

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.



Error handling & debugging

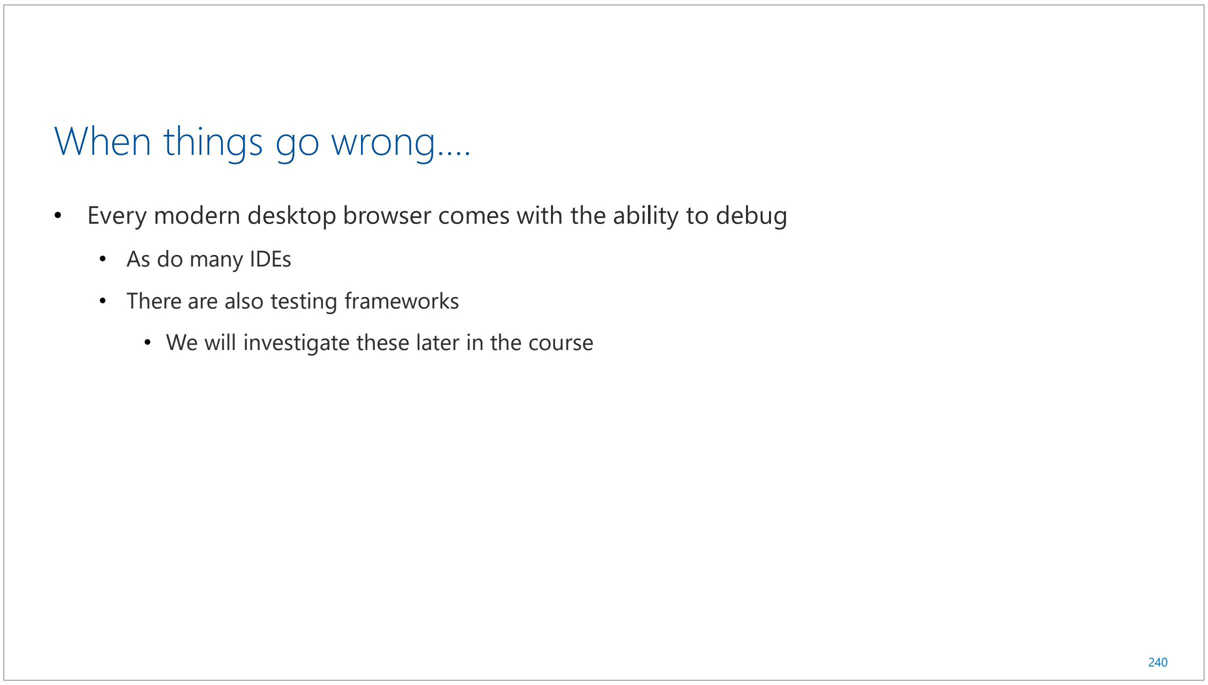
JAVASCRIPT FUNDAMENTALS



Introduction

- Why you must debug
- Understanding the Error object
  - The Inbuilt Error types
- Creating resilient code using try/catch statements
- Throwing Errors
- In Browser Debugging
  - Examples with Developer Tools for Chrome
- Console Debugging
  - Logging to the console
  - Breakpoints

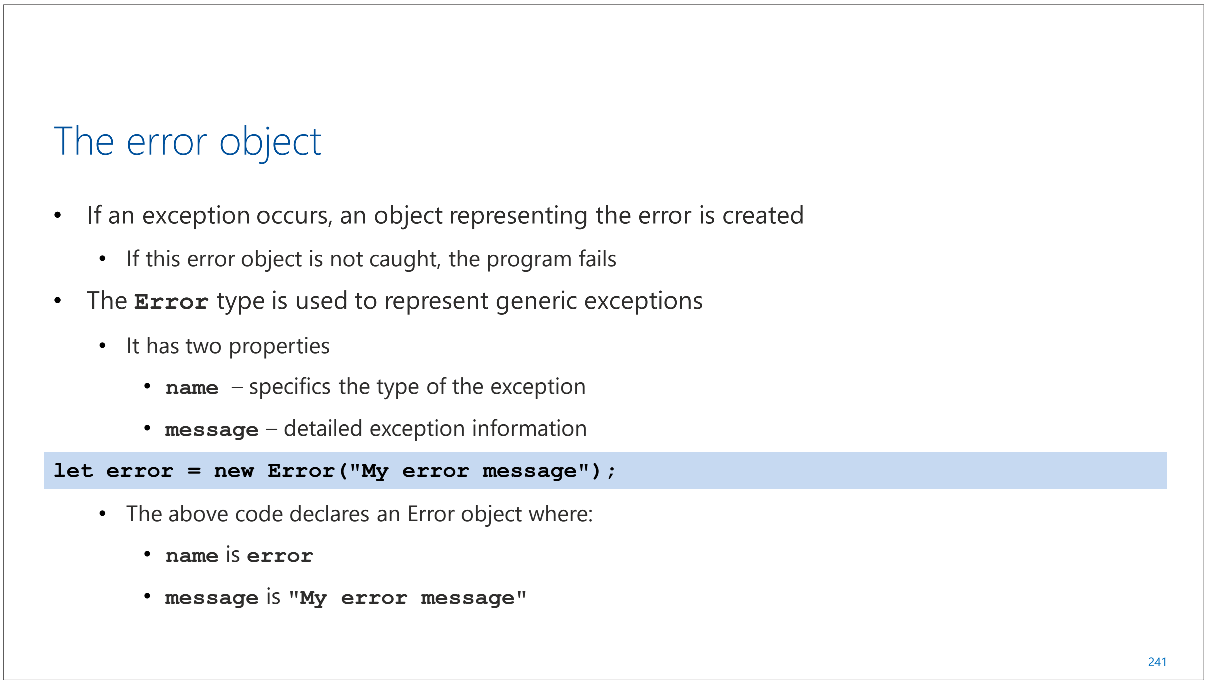
239



When things go wrong….

- Every modern desktop browser comes with the ability to debug
  - As do many IDEs
  - There are also testing frameworks
    - We will investigate these later in the course

240



The error object

- If an exception occurs, an object representing the error is created
  - If this error object is not caught, the program fails
- The **`Error`** type is used to represent generic exceptions
  - It has two properties
    - **`name`** – specifics the type of the exception
    - **`message`** – detailed exception information

```
let error = new Error("My error message");
```

  - The above code declares an Error object where:
    - **`name`** is **`error`**
    - **`message`** is **`"My error message"`**

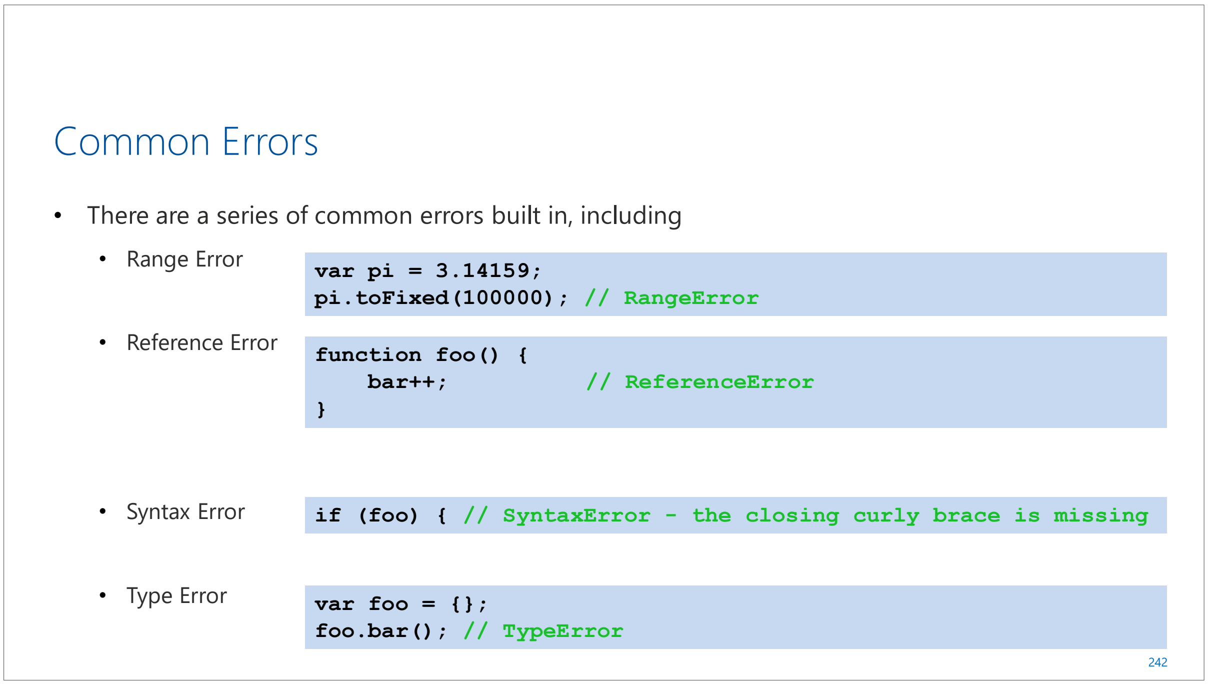
241

When a JavaScript statement generates an error, it is said to throw an exception. Instead of proceeding to the next statement, the JavaScript interpreter checks for exception handling code. If there is no exception handler, then the program returns from whatever function threw the exception. This is repeated for each function on the call stack until an exception handler is found or until the top level function is reached, causing the program to terminate.

The “Error” type is used to represent generic exceptions. This type of exception is most often used for implementing user defined exceptions. The topic of creating user-defined exceptions will be revisited later in this article. “Error” objects are instantiated by calling their constructor as shown in the following example:

```
let error = new Error("error message");
```

“Error” objects contain two properties: “name” and “message”. The “name” property specifies the type of exception (in this case “Error”). The “message” property provides a more detailed description of the exception. The “message” gets its value from the string passed to the exception’s constructor. The remaining exception types represent more specific types of errors, but they are all used in the same fashion as the generic “Error” type.



Common Errors

- There are a series of common errors built in, including
  - Range Error

```
var pi = 3.14159;
pi.toFixed(100000); // RangeError
```

  - Reference Error

```
function foo() {
    bar++;          // ReferenceError
}
```

  - Syntax Error

```
if (foo) { // SyntaxError - the closing curly brace is missing
```

  - Type Error

```
var foo = {};
foo.bar(); // TypeError
```

242

RangeError

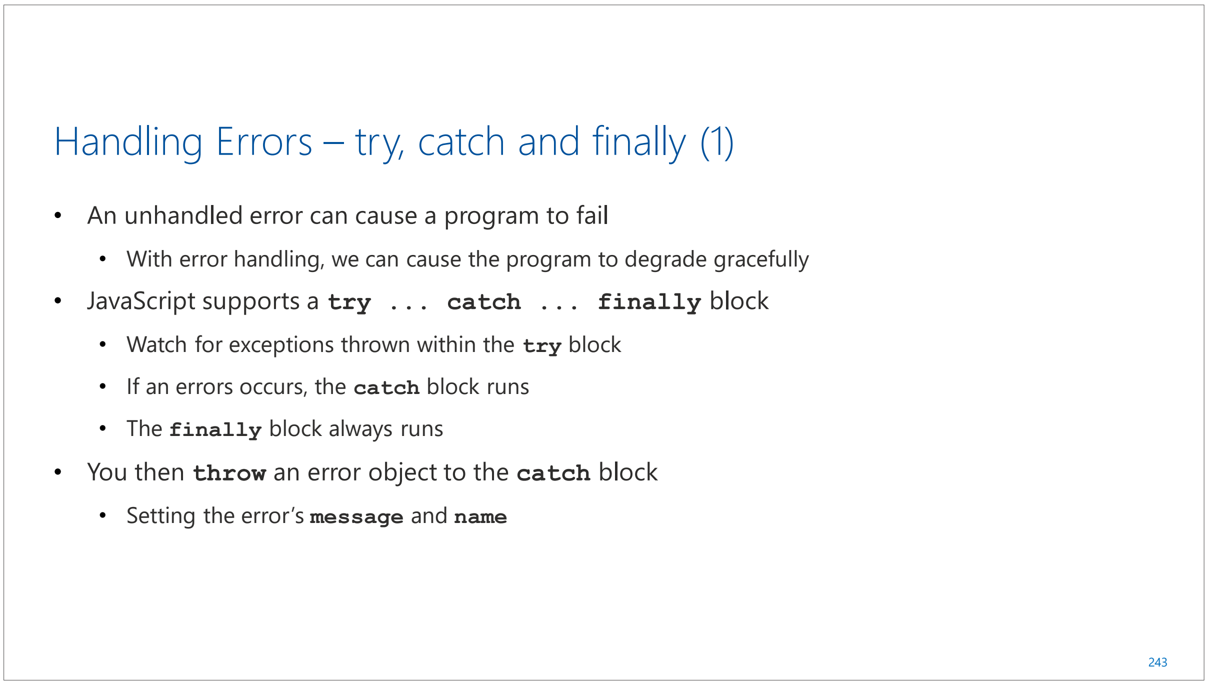
Are generated by numbers that fall outside of a specified range. In the code above, the argument is expected to be between 0 and 20 (although some browsers support a wider range). If the value of "digits" is outside of this range, then a "RangeError" is thrown.

ReferenceError

Is thrown when a non-existent variable is accessed. These exceptions commonly occur when an existing variable name is misspelled. In the example above, a "ReferenceError" occurs when "bar" is accessed. Note that this example assumes that "bar" does not exist in any active scope when the increment operation is attempted.

SyntaxError

Occurs when the rules of the JavaScript language are broken. Developers who are familiar with languages, such as C and Java are used to encountering syntax errors during the compilation process. However, because JavaScript is an interpreted language, syntax errors are not identified until the code is executed. Syntax errors are unique as they are the only type of exception that cannot be recovered from. The following example generates a syntax error because the "if" statement is missing a closing curly brace.



Handling Errors – try, catch and finally (1)

- An unhandled error can cause a program to fail
  - With error handling, we can cause the program to degrade gracefully
- JavaScript supports a **`try ... catch ... finally`** block
  - Watch for exceptions thrown within the **`try`** block
  - If an errors occurs, the **`catch`** block runs
  - The **`finally`** block always runs
- You then **`throw`** an error object to the **`catch`** block
  - Setting the error's **`message`** and **`name`**

243

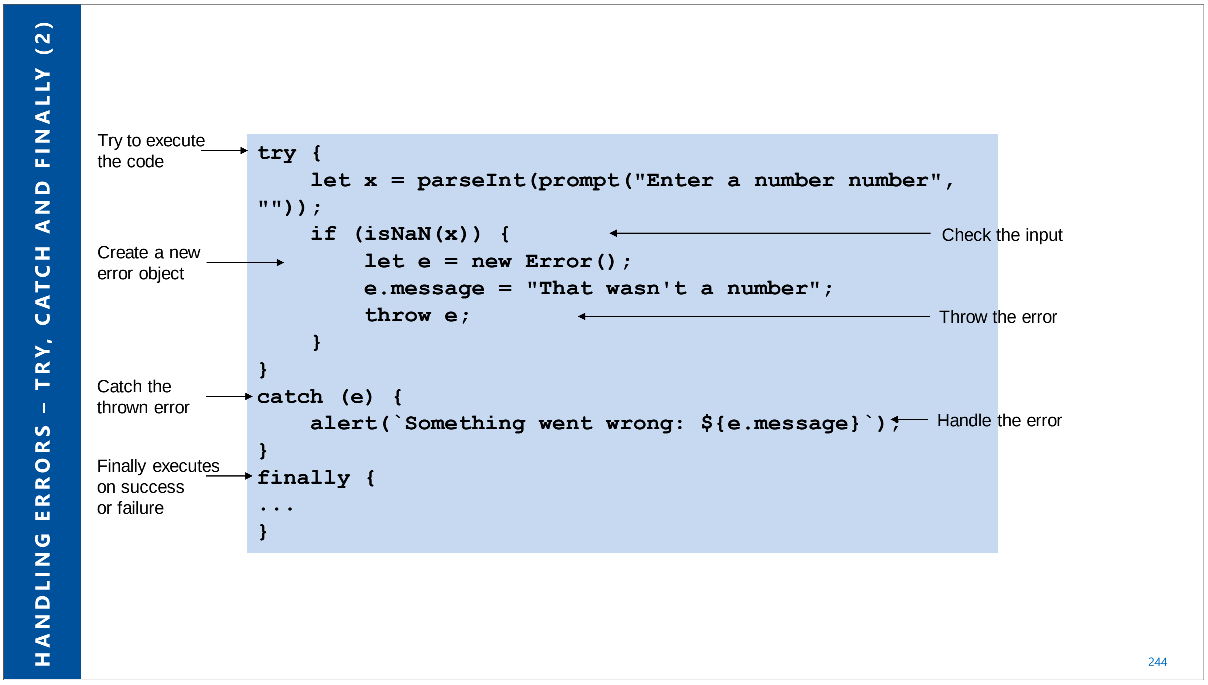
The try...catch...finally block is common to most modern languages. Processing proceeds as follows:

within the try block, if no exceptions are thrown, the finally block is run.

If an exception is thrown, the catch block is run followed by the finally block.

Please note that the finally block runs, even if the try or catch blocks return out of the current function.

Both the catch and the finally blocks are optional: you do not need to catch the exceptions, you do not need to have code that runs after the try ... catch. If the exception is not caught, it is thrown to calling function. If the exception is not caught at any level, then browser will log the error.



HANDLING ERRORS – TRY, CATCH AND FINALLY (2)

```
try {
    let x = parseInt(prompt("Enter a number number",
""));
    if (isNaN(x)) {
        let e = new Error();
        e.message = "That wasn't a number";
        throw e;
    }
}
catch (e) {
    alert(`Something went wrong: ${e.message}`);
}
finally {
...
}
```

Try to execute the code

Create a new error object

Catch the thrown error

Finally executes on success or failure

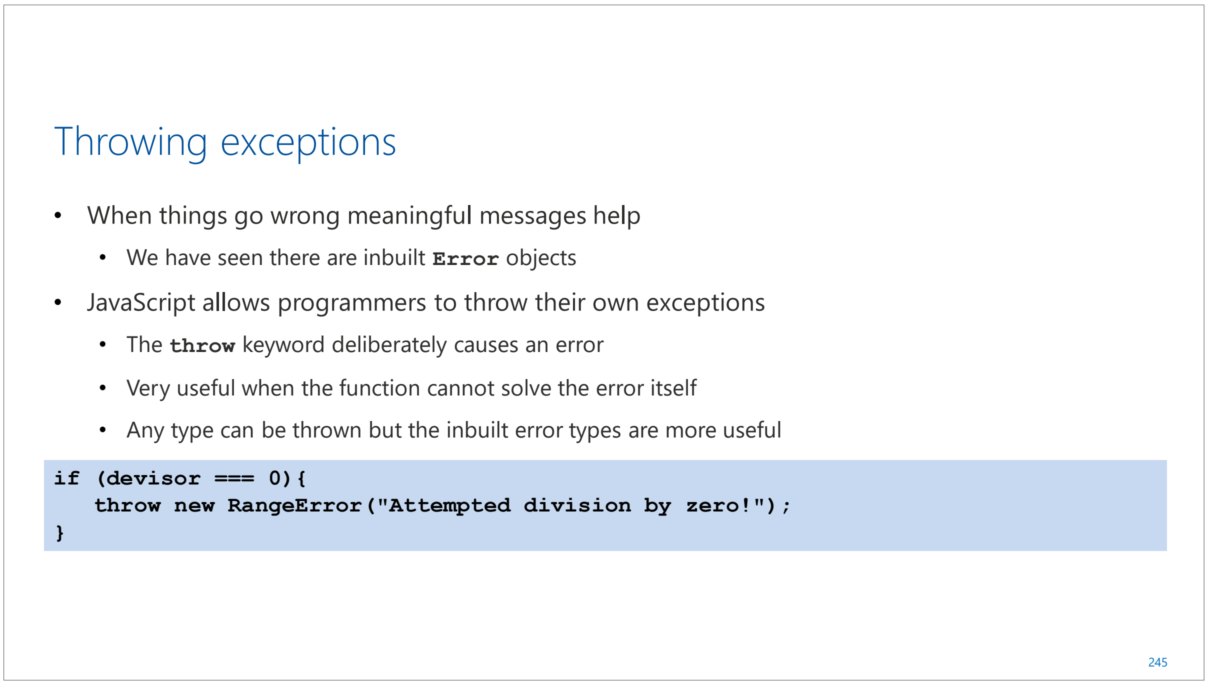
Check the input

Throw the error

Handle the error

244

The above code takes us through an error handling scenario using try catch and finally. It also introduces us to the JavaScript error object. The Error object represents a runtime error allowing a developer to wrap up useful information about what went wrong. As we can see above, it was possible to provide a message and even name the error.



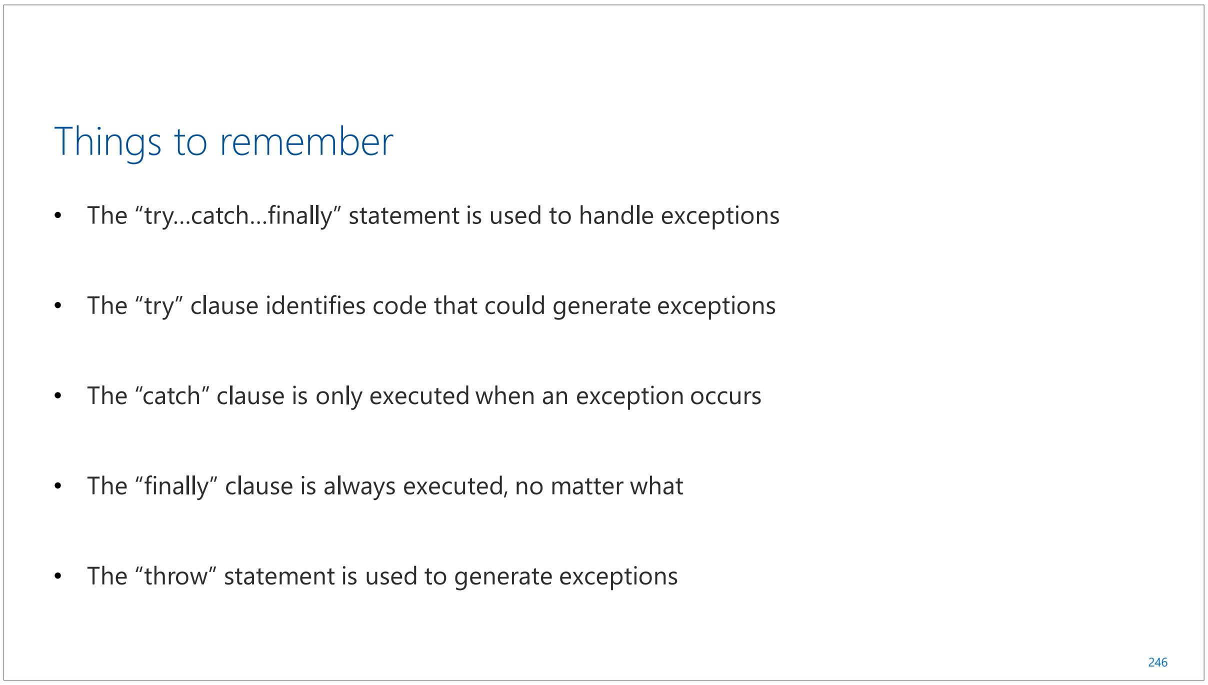
Throwing exceptions

- When things go wrong meaningful messages help
  - We have seen there are inbuilt **`Error`** objects
- JavaScript allows programmers to throw their own exceptions
  - The **`throw`** keyword deliberately causes an error
  - Very useful when the function cannot solve the error itself
  - Any type can be thrown but the inbuilt error types are more useful

```
if (devisor === 0){
   throw new RangeError("Attempted division by zero!");
}
```

245

JavaScript allows programmers to throw their own exceptions via the appropriately named “throw” statement. This concept can be somewhat confusing to inexperienced developers. After all, developers strive to write code that is free of errors, yet the “throw” statement intentionally introduces them. However, intentionally throwing exceptions can actually lead to code that is easier to debug and maintain. For example, by creating meaningful error messages, it becomes easier to identify and resolve problems.



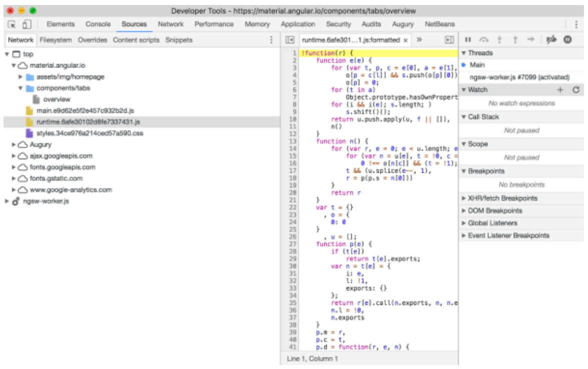
Things to remember

- The “try…catch…finally” statement is used to handle exceptions
- The “try” clause identifies code that could generate exceptions
- The “catch” clause is only executed when an exception occurs
- The “finally” clause is always executed, no matter what
- The “throw” statement is used to generate exceptions

246

Debugging Demonstration

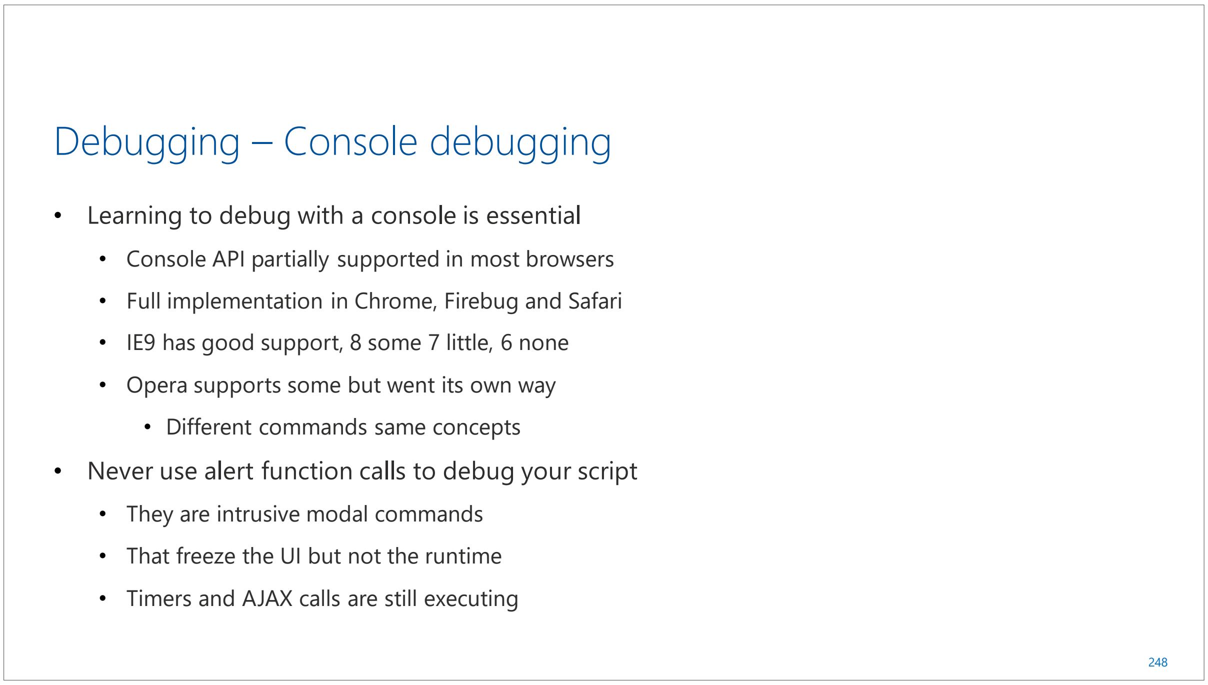
- Contains a JavaScript Console
  - DOM explorer
  - Console
  - CSS style browser
  - Can debug
    - Command line interface
    - Not trivial



```
!function(r) {
    function e(e) {
        for (var t, p, c = e[0], a = e[1],
            o[p = c[l]] && s.push(o[p][0])
            o[p] = 0;
        for (t in a)
            Object.prototype.hasOwnPropert
        for (i && i(e); s.length; )
            s.shift()();
        return u.push.apply(u, f || []),
        n()
    }
    function n() {
        for (var r, e = 0; e < u.length; e
            for (var n = u[e], t = !0, c =
                0 !== o[n[c]] && (t = !1);
            t && (u.splice(e--, 1),
            r = p(p.s = n[0]))
        }
        return r
    }
    var t = {}
      , o = {
        0: 0
    }
      , u = [];
    function p(e) {
        if (t[e])
            return t[e].exports;
        var n = t[e] = {
            i: e,
            l: !1,
            exports: {}
        };
        return r[e].call(n.exports, n, n.e
        n.l = !0,
        n.exports
    }
    p.m = r,
    p.c = t,
    p.d = function(r, e, n) {
```

247

Your instructor may choose to give a demonstration of using a browser's debug tools.



Debugging – Console debugging

- Learning to debug with a console is essential
  - Console API partially supported in most browsers
  - Full implementation in Chrome, Firebug and Safari
  - IE9 has good support, 8 some 7 little, 6 none
  - Opera supports some but went its own way
    - Different commands same concepts
- Never use alert function calls to debug your script
  - They are intrusive modal commands
  - That freeze the UI but not the runtime
  - Timers and AJAX calls are still executing

248

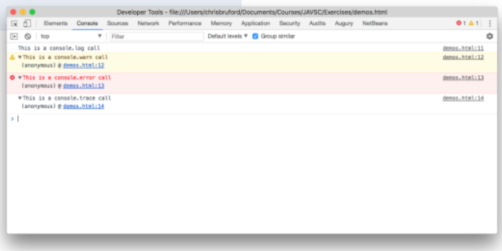
As a client side developer, you need to learn to debug and test. As we have discovered so far, JavaScript is an amazingly powerful programming language, but things can easily go wrong and you will need to identify how and why.

Some developers use alert calls to test a function is being called. One simple piece of advice: DON'T! Debugging with alerts holds the UI in a state of limbo, but does not hold any asynchronous operations, such as timers or AJAX feeds.

Debugging – Console logging

- The console API has a number of useful functions

Function	Description
`console.log()`	Writes a message to the console.
`console.warn()`	As above with visual "warning" icon
`console.error()`	As above with visual "error" icon
`console.trace()`	As above with a call trace



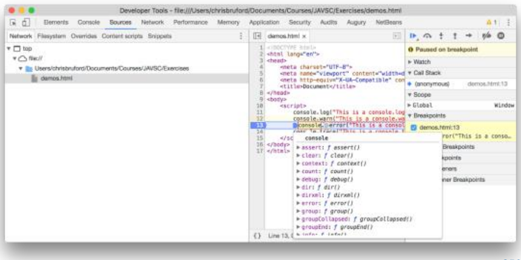
```
This is a console.log call                                   demos.html:11
▼This is a console.warn call                                 demos.html:12
  (anonymous) @ demos.html:12
▼This is a console.error call                                demos.html:13
  (anonymous) @ demos.html:13
▼This is a console.trace call                                demos.html:14
  (anonymous) @ demos.html:14
>
```

249

%s	String
%d, %i	Integer (numeric formatting is not yet supported)
%f	Floating point number (numeric formatting is not yet supported)
%o	Object hyperlink
%c	Style formatting

Debugging – breakpoints

- Breakpoints pauses the code allowing examination and debugging
  - From the developer tools select the sources panel and select from a JavaScript source file
  - Set breakpoint by clicking in the gutter of the line you want to pause execution at
  - Hover over the source code to inspect variables and functions
  - Delete the breakpoint by clicking the blue tag breakpoint indicator

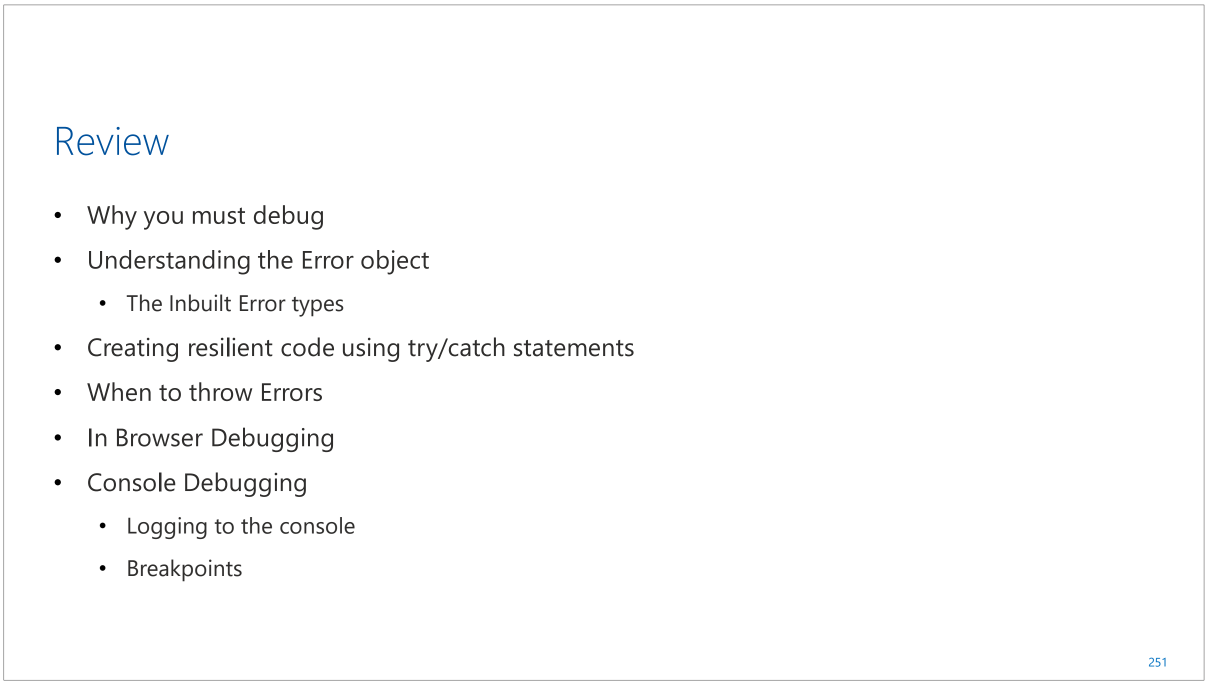


```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=d
    <meta http-equiv="X-UA-Compatible" con
    <title>Document</title>
</head>
<body>
    <script>
        console.log("This is a console.log
        console.warn("This is a console.wa
        console.error("This is a consol
    </sc
</body>
</html>
```

```
console
▸ assert: ƒ assert()
▸ clear: ƒ clear()
▸ context: ƒ context()
▸ count: ƒ count()
▸ debug: ƒ debug()
▸ dir: ƒ dir()
▸ dirxml: ƒ dirxml()
▸ error: ƒ error()
▸ group: ƒ group()
▸ groupCollapsed: ƒ groupCollapsed()
▸ groupEnd: ƒ groupEnd()
```

250

	Windows/Linux	Mac
Select Next Call Frame	Ctrl-.	^.
Select Previous Call Frame	Ctrl-,	^,
Continue	F8 or Ctrl-/	F8 or ⌘/
Step Over	F10 or Ctrl-'	F10 or ⌘'
Step Into	F11 or Ctrl-;	F11 or ⌘;
Step Out	Shift-F11 or Ctrl-Shift-;	⇧F11 or ⇧⌘;
Evaluate Selection	Ctrl-Shift-E	⇧⌘E
Toggle Breakpoint Condition	Click on line number	Click on line number
Edit Breakpoint Condition	Right-Click on line number	Right-Click on line number



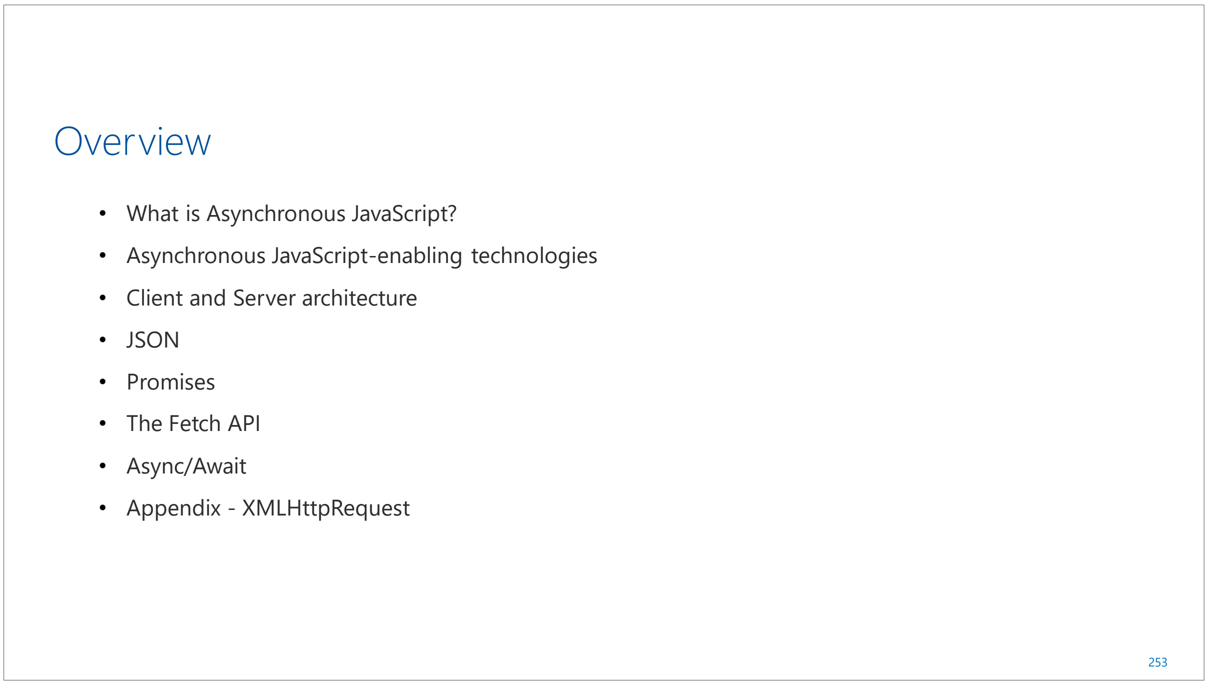
Review

- Why you must debug
- Understanding the Error object
    - The Inbuilt Error types
- Creating resilient code using try/catch statements
- When to throw Errors
- In Browser Debugging
- Console Debugging
    - Logging to the console
    - Breakpoints

251

Asynchronous JavaScript

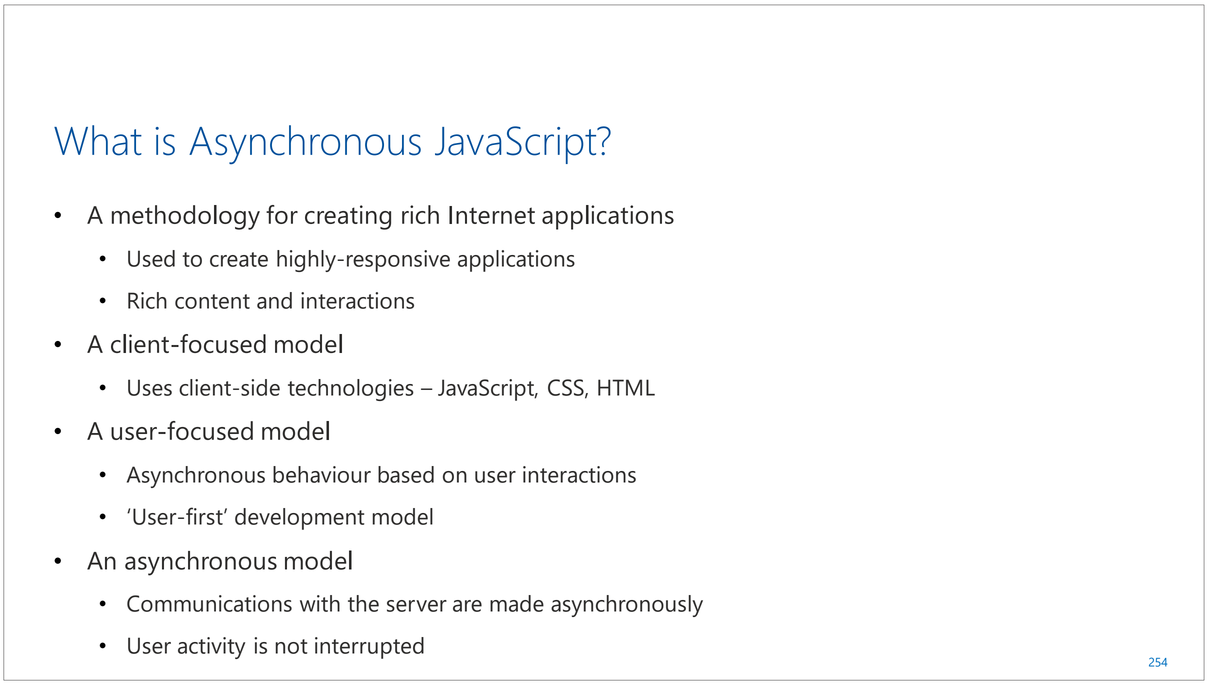
JAVASCRIPT FUNDAMENTALS



Overview

- What is Asynchronous JavaScript?
- Asynchronous JavaScript-enabling technologies
- Client and Server architecture
- JSON
- Promises
- The Fetch API
- Async/Await
- Appendix - XMLHttpRequest

253



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

254

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.

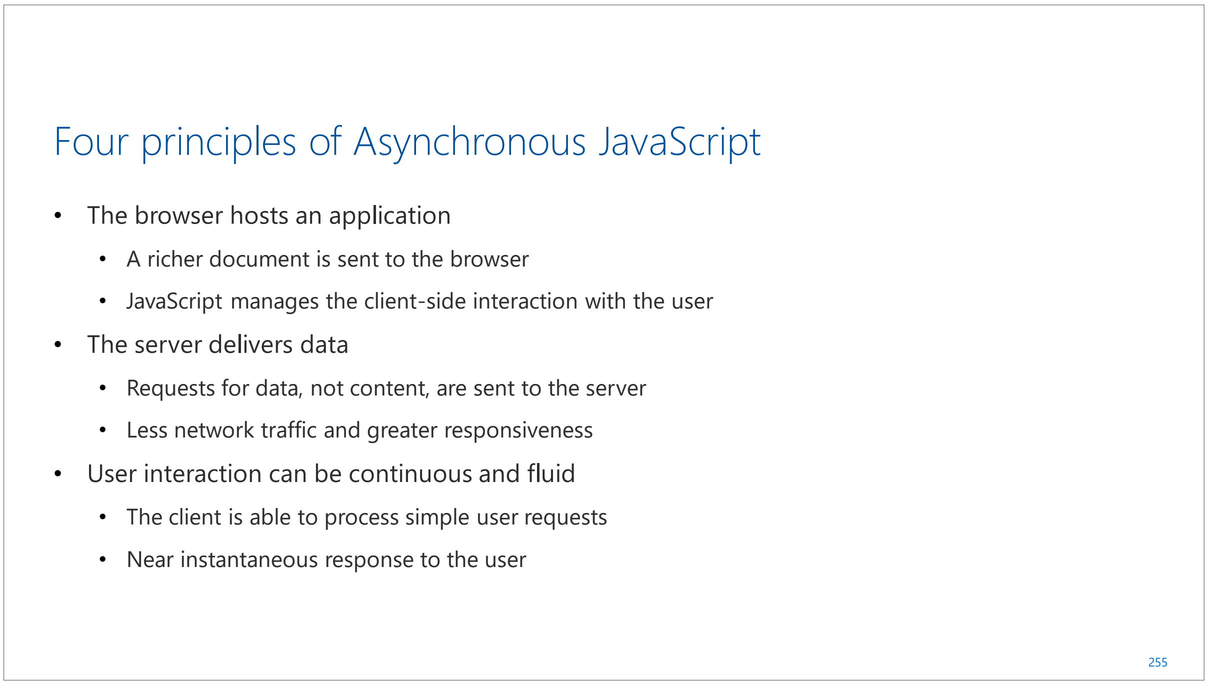
The typical server-bound web application cannot easily provide this because it has to defer to the server for any significant updates to the user interface.

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

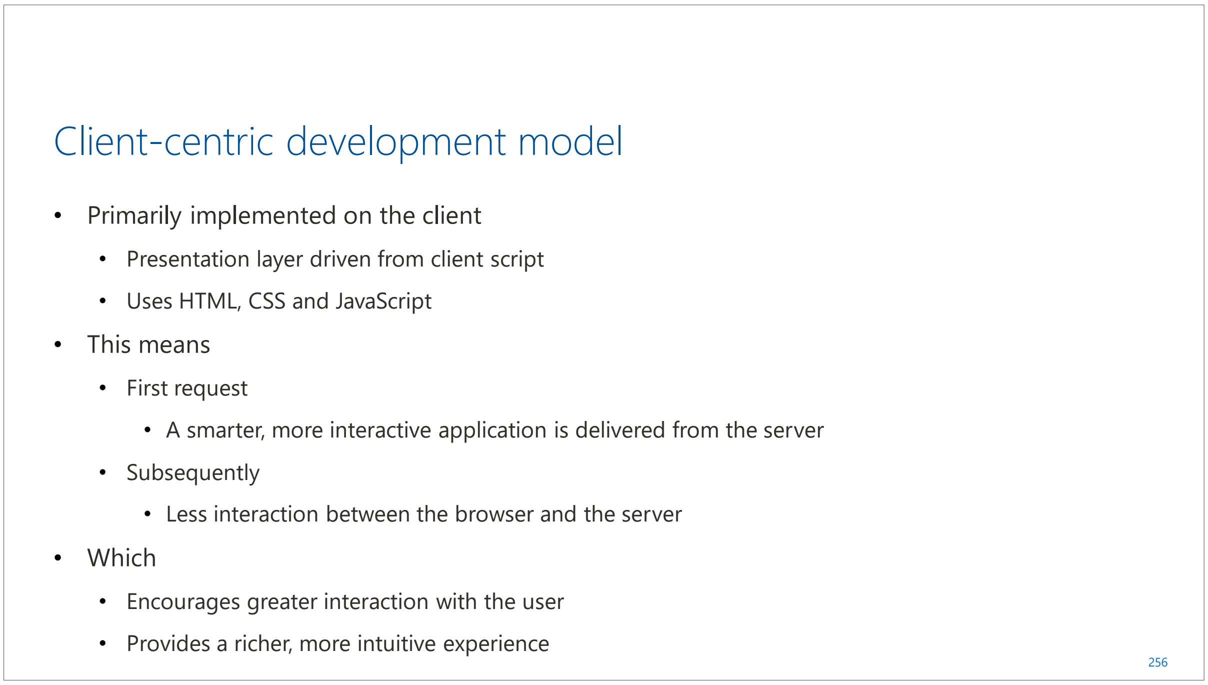
- The browser hosts an application
  - A richer document is sent to the browser
  - JavaScript manages the client-side interaction with the user
- The server delivers data
  - Requests for data, not content, are sent to the server
  - Less network traffic and greater responsiveness
- User interaction can be continuous and fluid
  - The client is able to process simple user requests
  - Near instantaneous response to the user

255

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

256

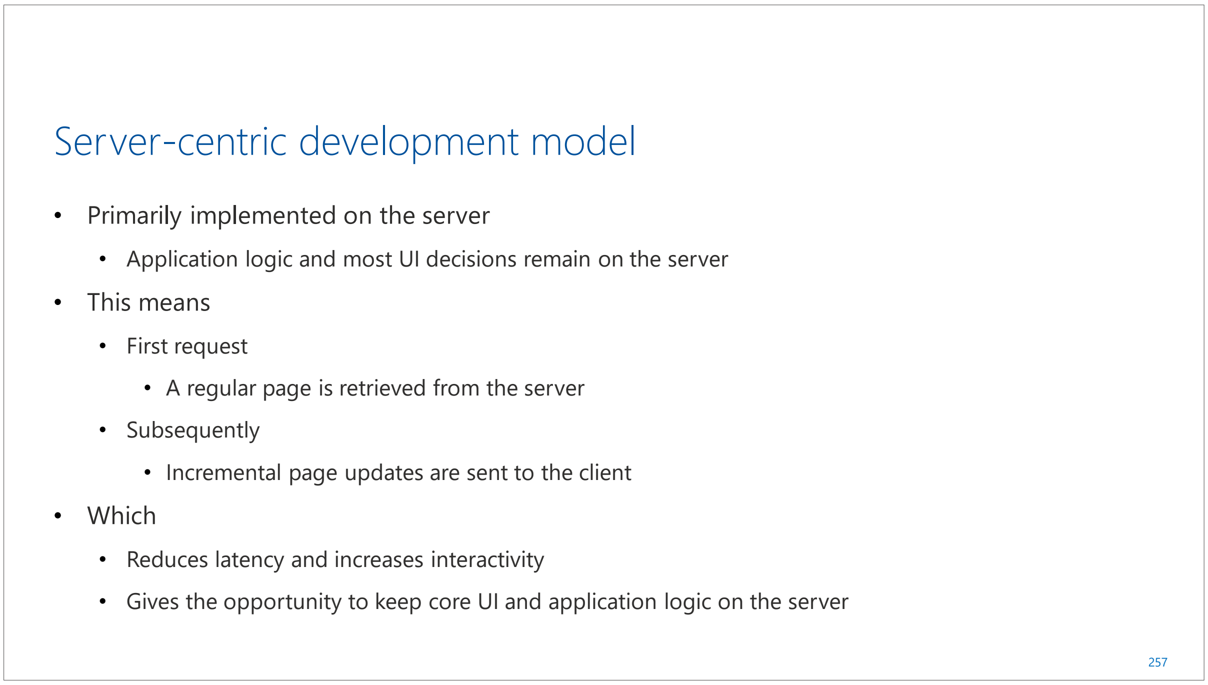
As mentioned, there are two main development models that we can consider within Asynchronous applications.

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

257

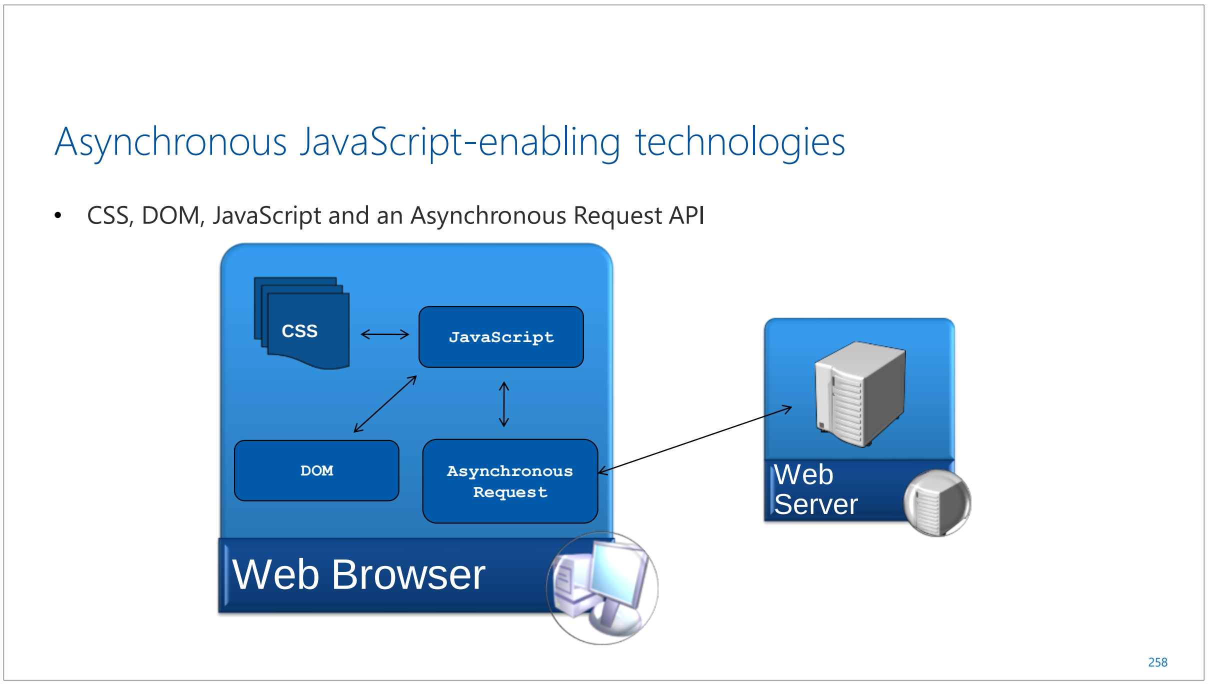
The second model is the server-centric development model.

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.

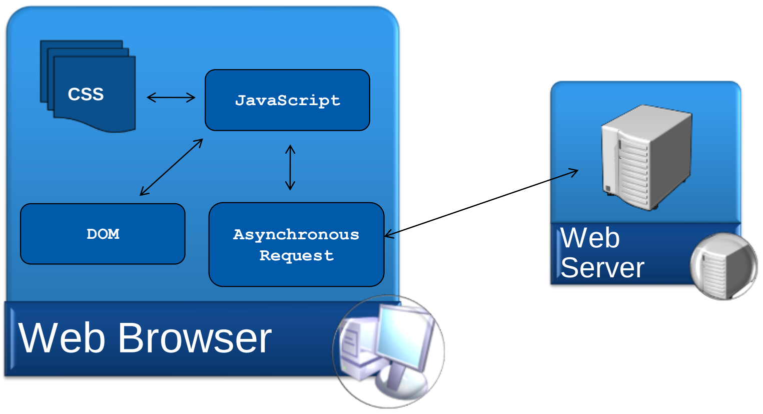
This happens incrementally, so reducing latency and increasing the level of interactivity possible without a full-page refresh. It also gives the opportunity to keep the core login of the Application and UI on the server which may be a required decision for some sensitive applications.

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



258

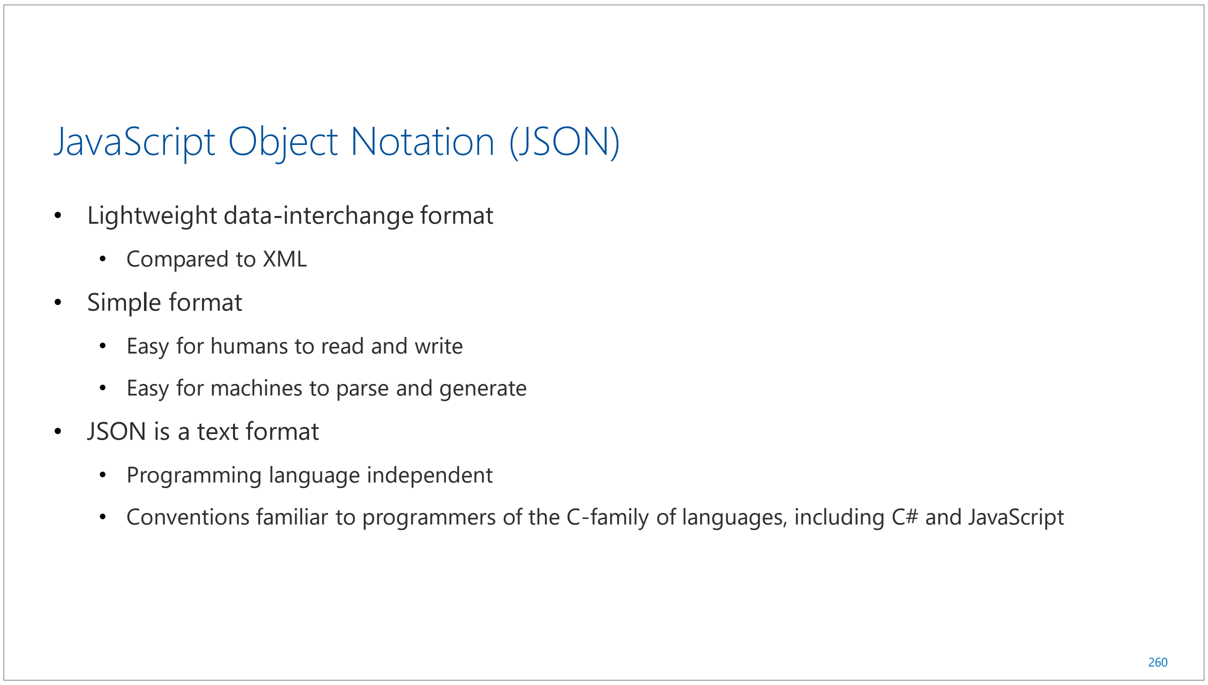
The diagram above shows the main enabling technologies of Asynchronous JavaScript.

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 FUNDAMENTALS



JavaScript Object Notation (JSON)

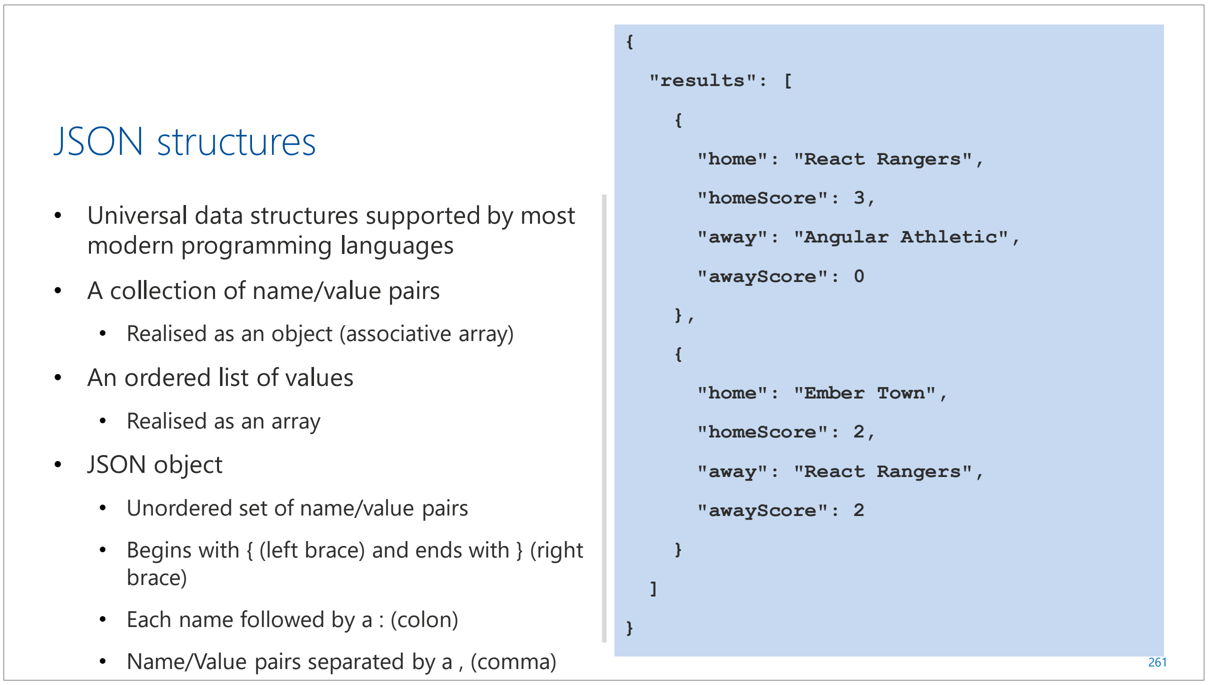
- Lightweight data-interchange format
  - Compared to XML
- Simple format
  - Easy for humans to read and write
  - Easy for machines to parse and generate
- JSON is a text format
  - Programming language independent
  - Conventions familiar to programmers of the C-family of languages, including C# and JavaScript

260

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

- 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)

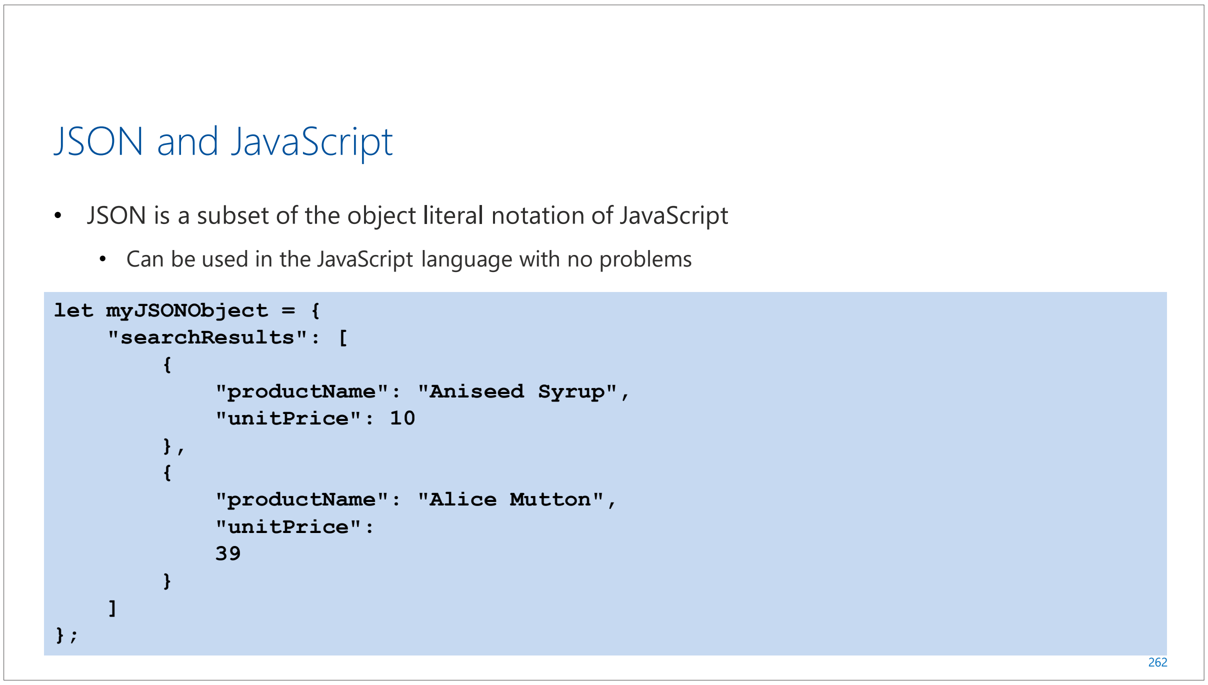
```
{
  "results": [
    {
      "home": "React Rangers",
      "homeScore": 3,
      "away": "Angular Athletic",
      "awayScore": 0
    },
    {
      "home": "Ember Town",
      "homeScore": 2,
      "away": "React Rangers",
      "awayScore": 2
    }
  ]
}
```

261

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

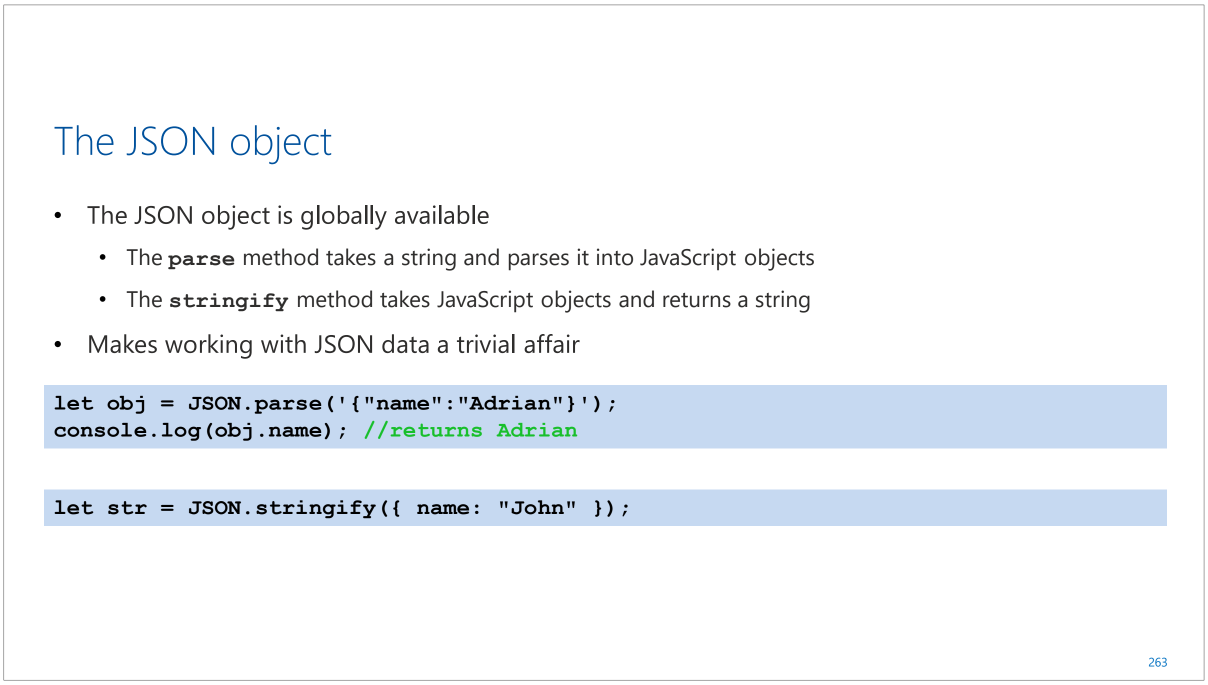
- JSON is a subset of the object literal notation of JavaScript
  - Can be used in the JavaScript language with no problems

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

262

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); //returns Adrian
```

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

263

There are a series of overloaded methods for the type:

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

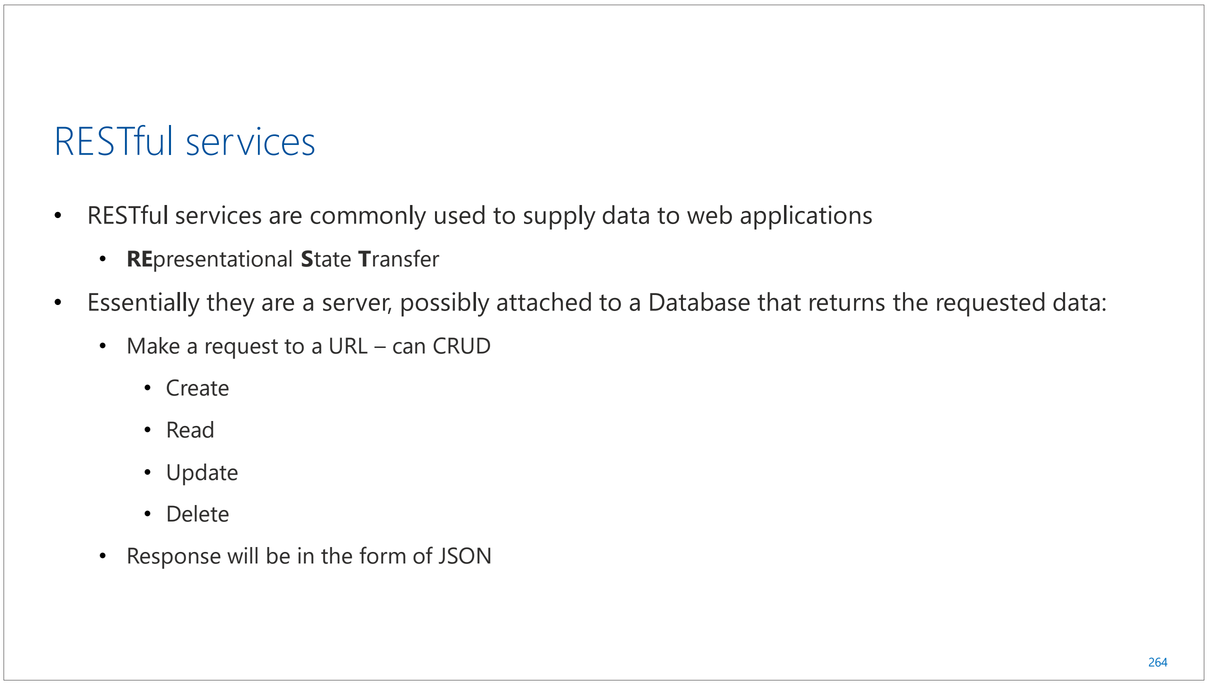
JSON.parse(text, translate) – Uses a translation function to convert values or remove them entirely.

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.

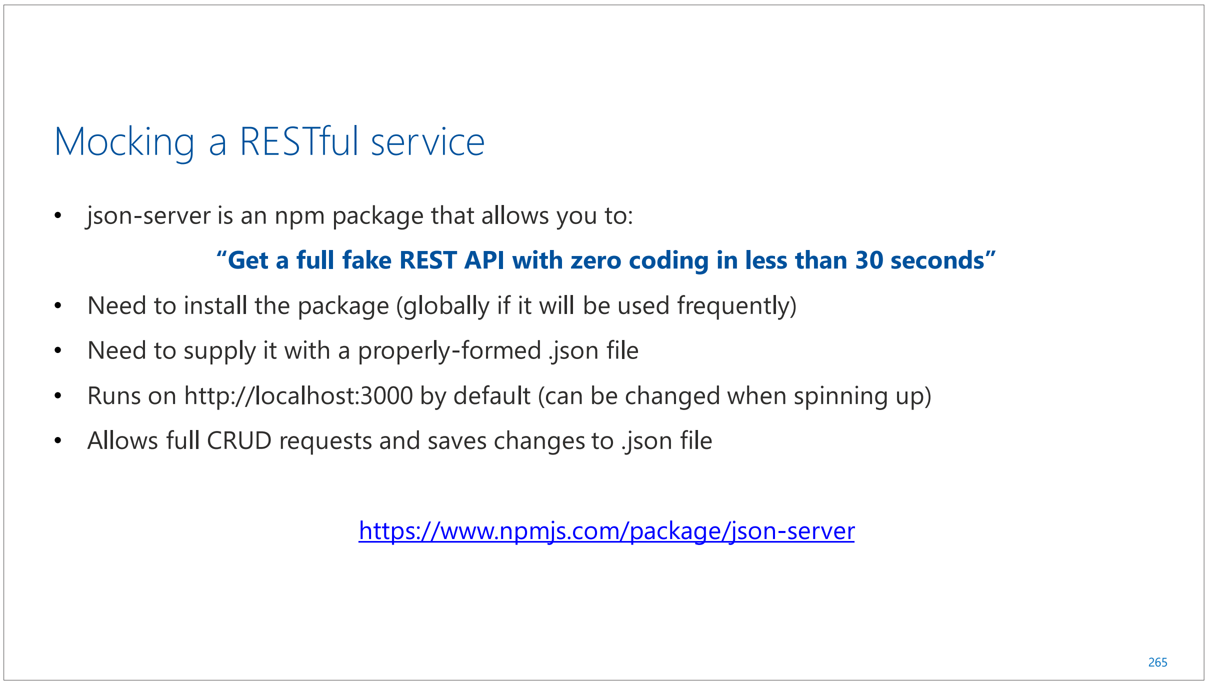
JSON.stringify(obj, null, 2) – Adds the specified number of spaces to the output, printing it evenly.



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

264



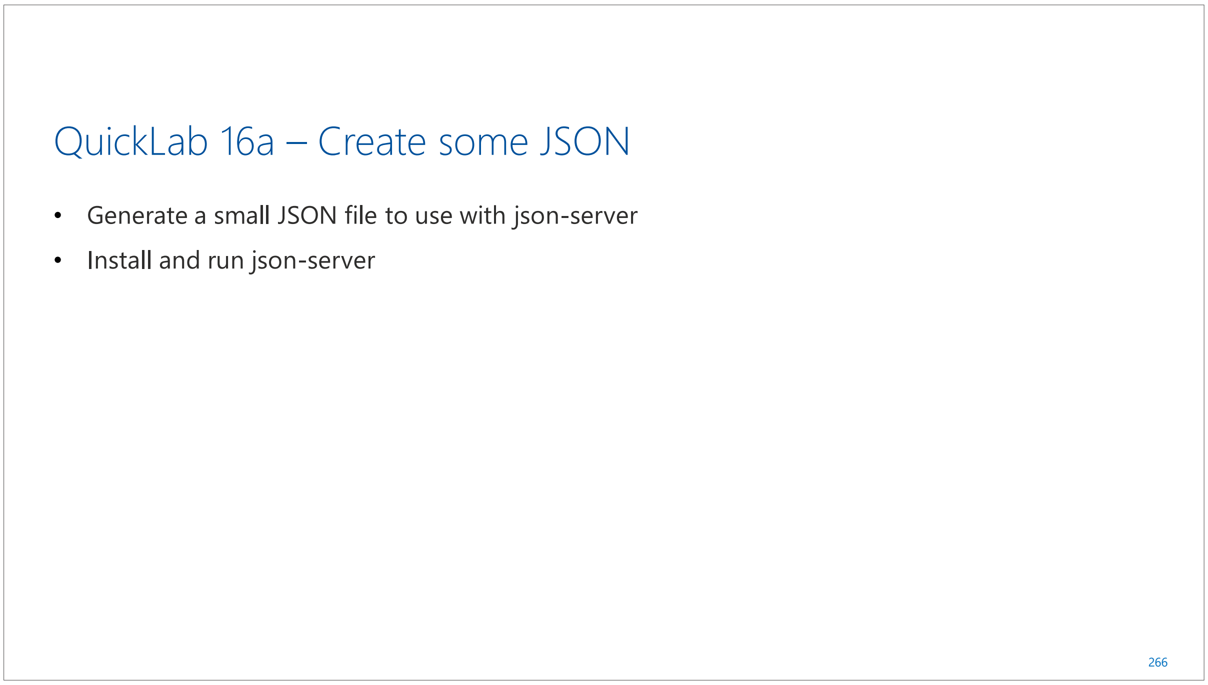
Mocking a RESTful service

- 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

265



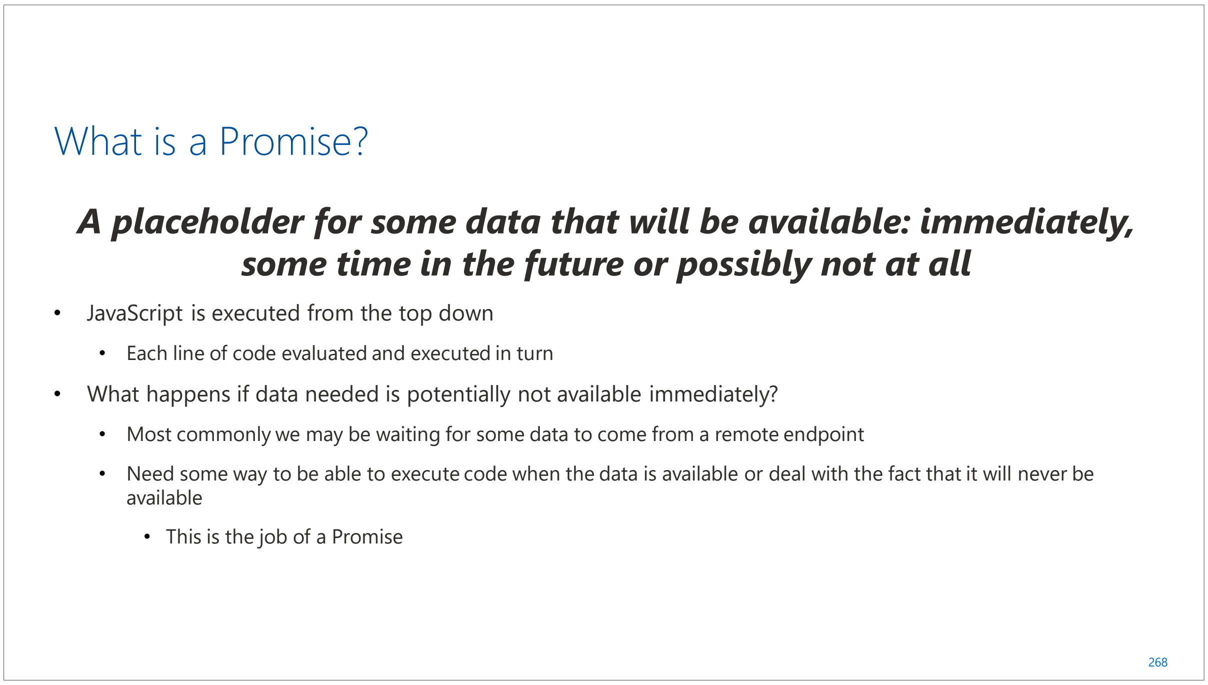
QuickLab 16a – Create some JSON

- Generate a small JSON file to use with json-server
- Install and run json-server

266

Promises

JAVASCRIPT FUNDAMENTALS

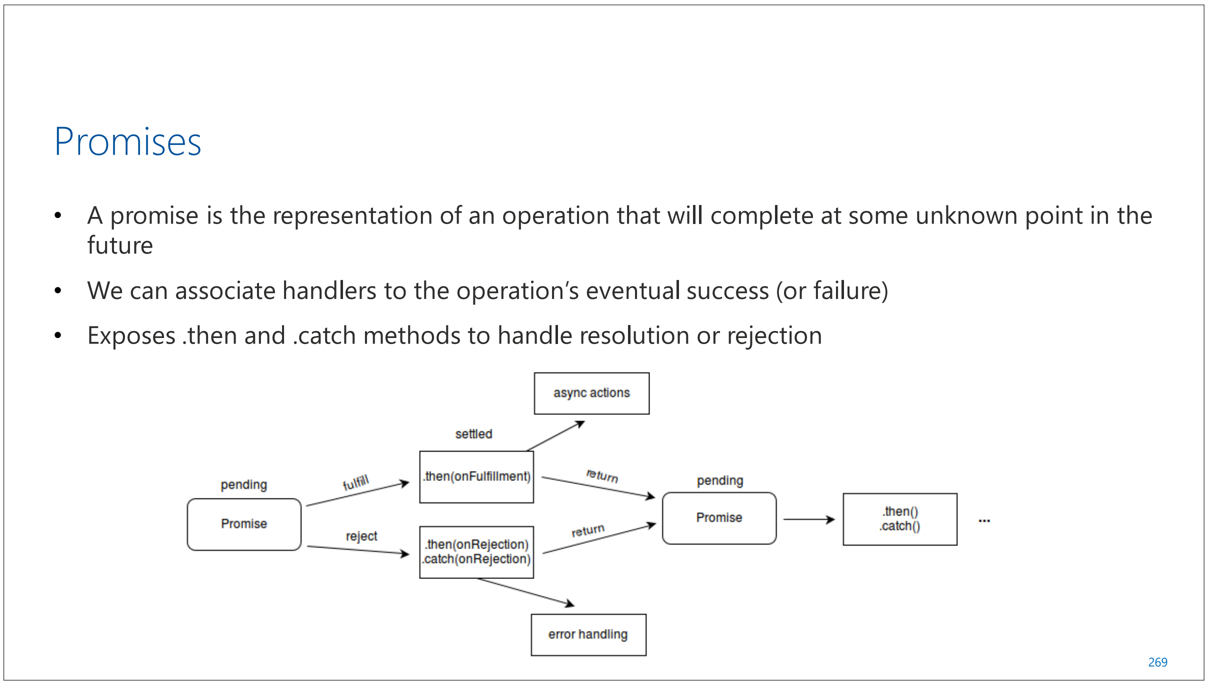


What is a Promise?

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

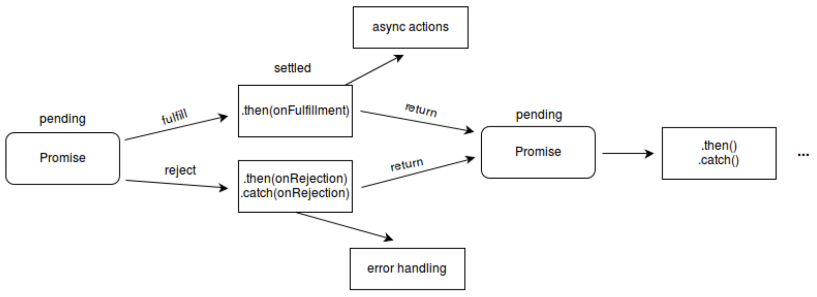
- JavaScript is executed from the top down
  - Each line of code evaluated and executed in turn
- What happens if data needed is potentially not available immediately?
  - Most commonly we may be waiting for some data to come from a remote endpoint
  - Need some way to be able to execute code when the data is available or deal with the fact that it will never be available
    - This is the job of a Promise

268



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



269

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

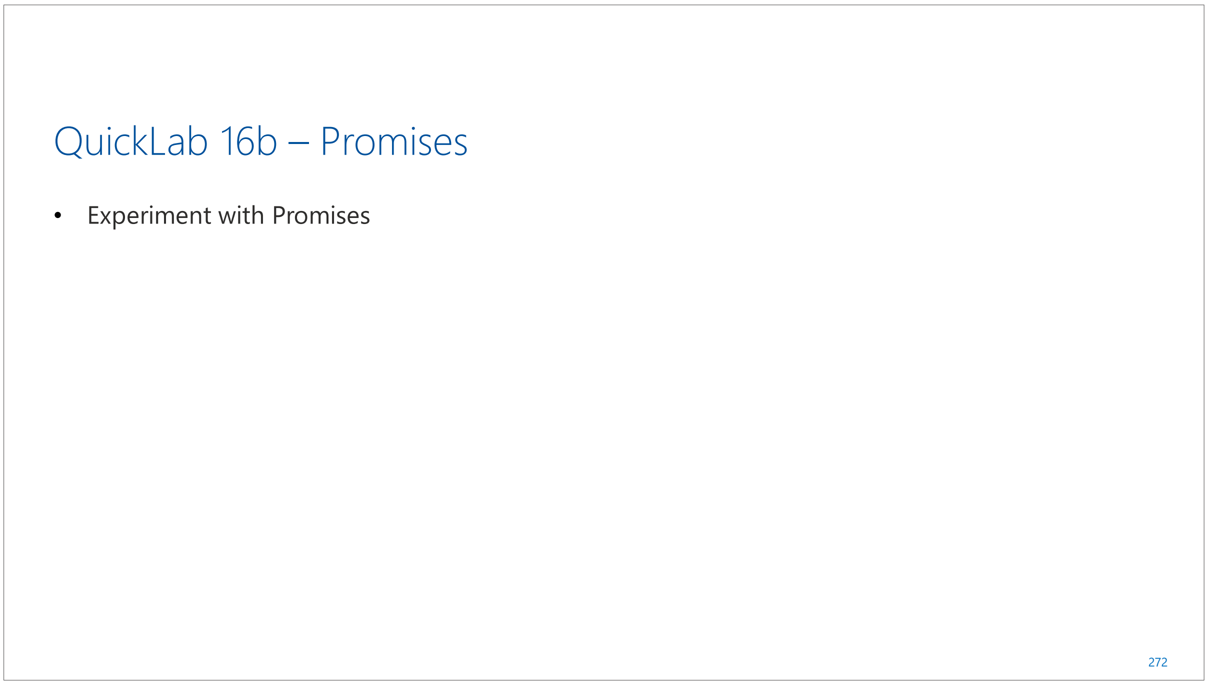
```
let newPromise = new Promise((resolve, reject) => { });
```

- 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

270

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);
        }
    );
```



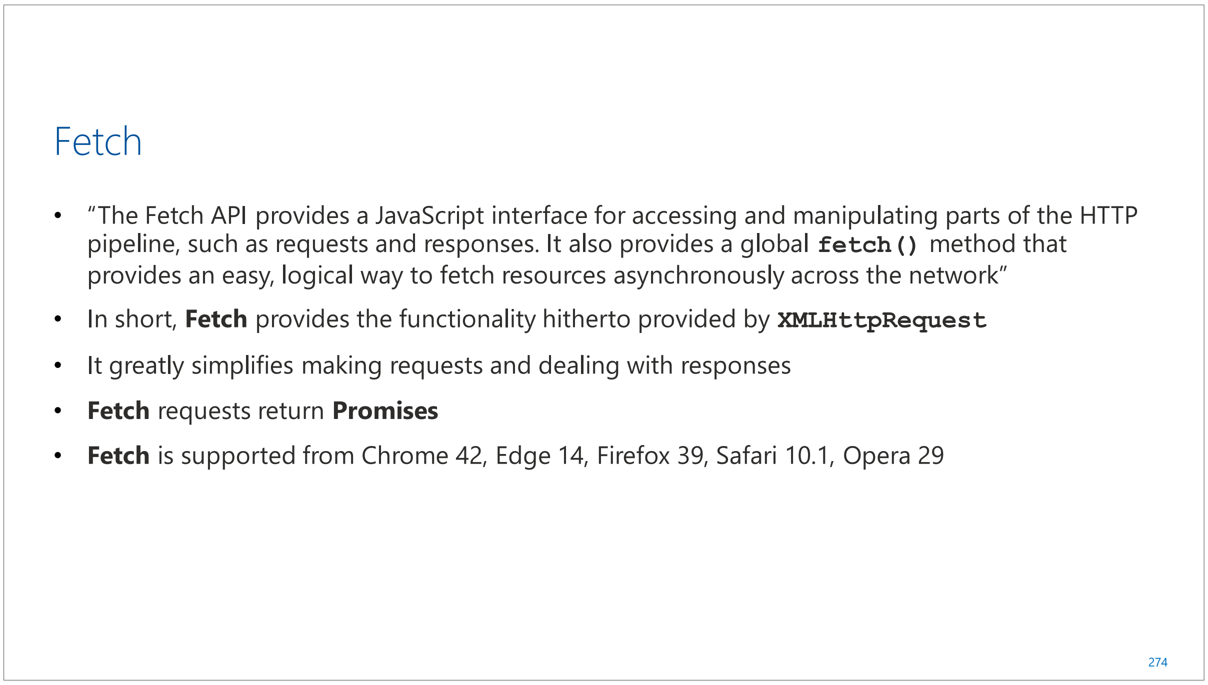
QuickLab 16b – Promises

- Experiment with Promises

272

Fetch

JAVASCRIPT FUNDAMENTALS



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

274

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`**

275

FETCH – FULL EXAMPLE

```
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));
```

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

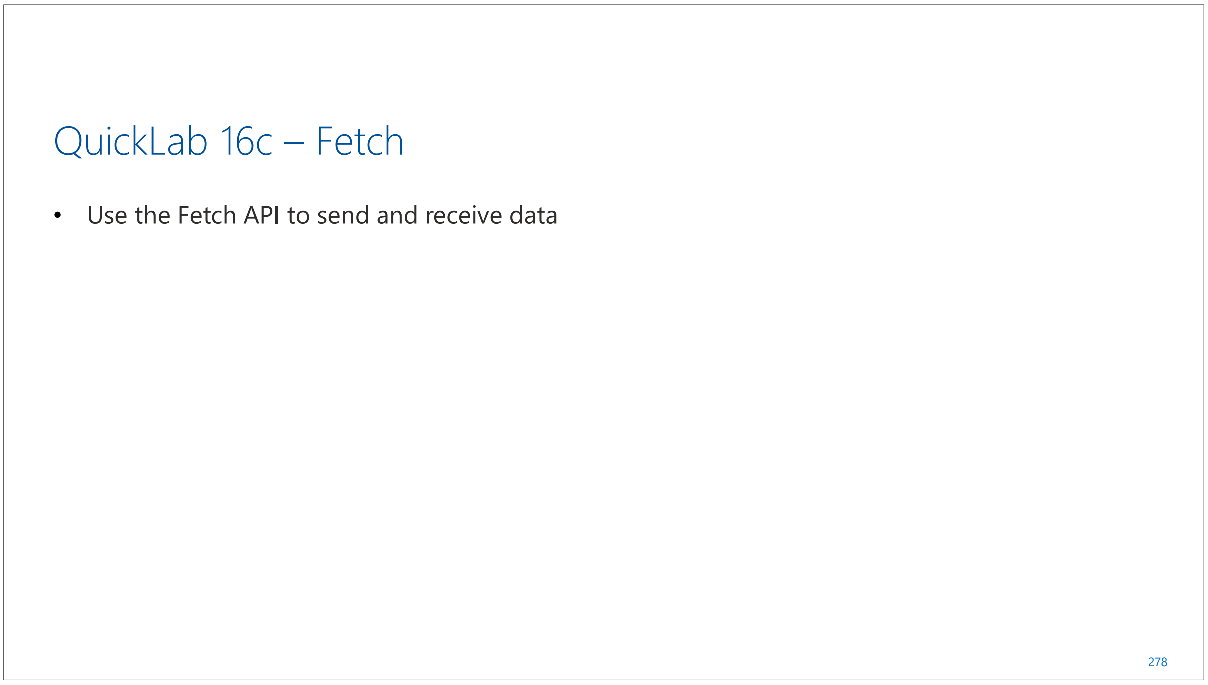
276

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
        }
    });
```

277



QuickLab 16c – Fetch

- Use the Fetch API to send and receive data

278

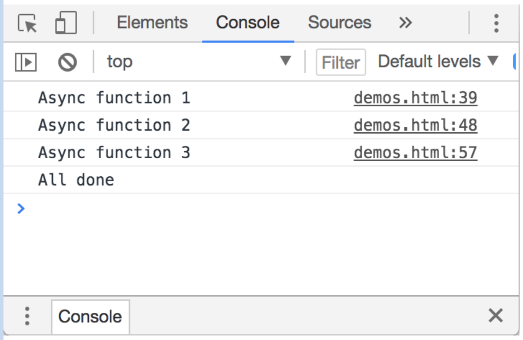
Async Functions

- 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 FUNCTIONS

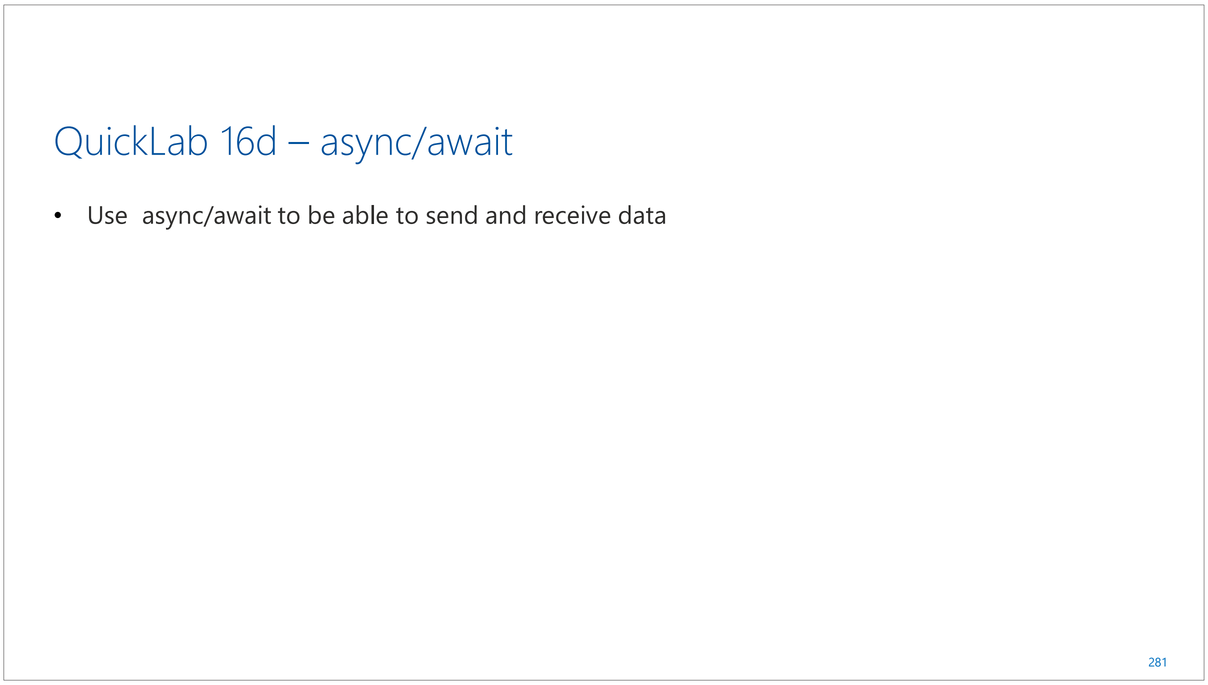
```
async function asyncFunc1() {
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log('Async function 3');
            resolve();
        },3000);
    });
}

async function asyncFunc2() {
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log('Async function 3');
            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                demos.html:39
Async function 2                demos.html:48
Async function 3                demos.html:57
All done
```

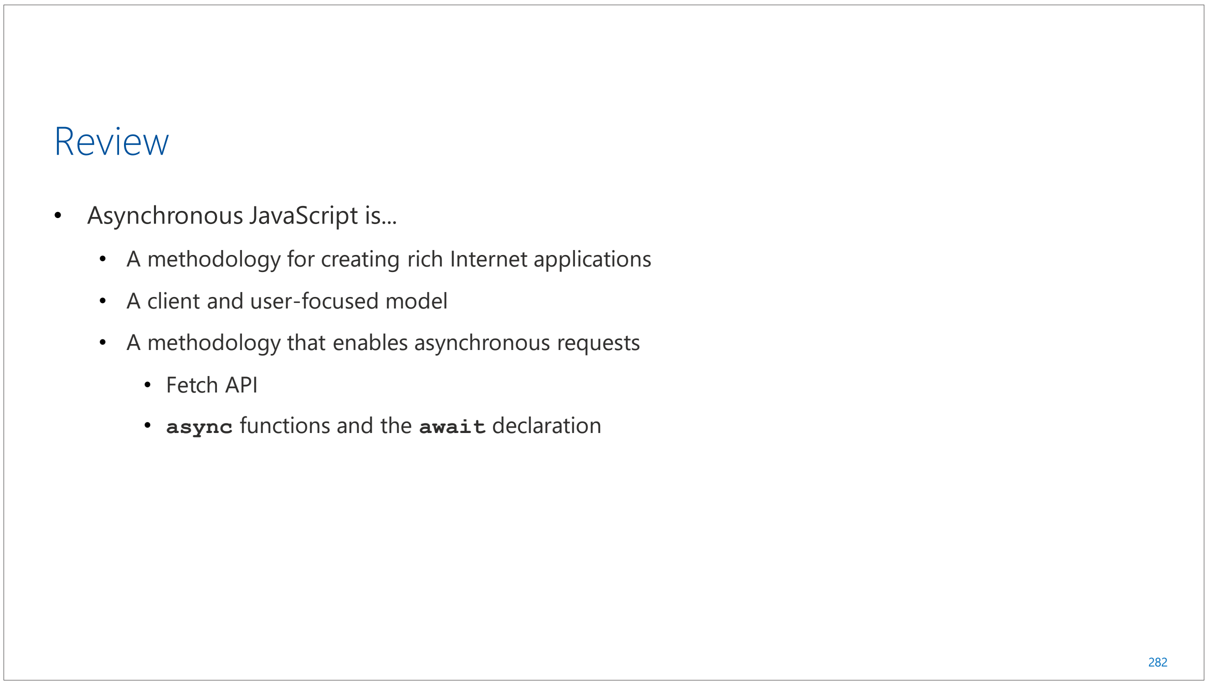
280



QuickLab 16d – async/await

- Use async/await to be able to send and receive data

281



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

282

Hackathon Part 2

- In this part Hackathon, you will build on a partially developed solution (whether that be your previous iteration or the provided starting point) for QA Cinemas' website by allowing submission of the user data from the form to a remote backend. This should be simulated by using json-server. All the necessary tools, knowledge and techniques have been covered in the course so far.
- This part of the Hackathon is intended to help you develop your skills and knowledge to be able to use JavaScript to submit data from a 'Sign-Up' form for users of the QA Cinemas website.

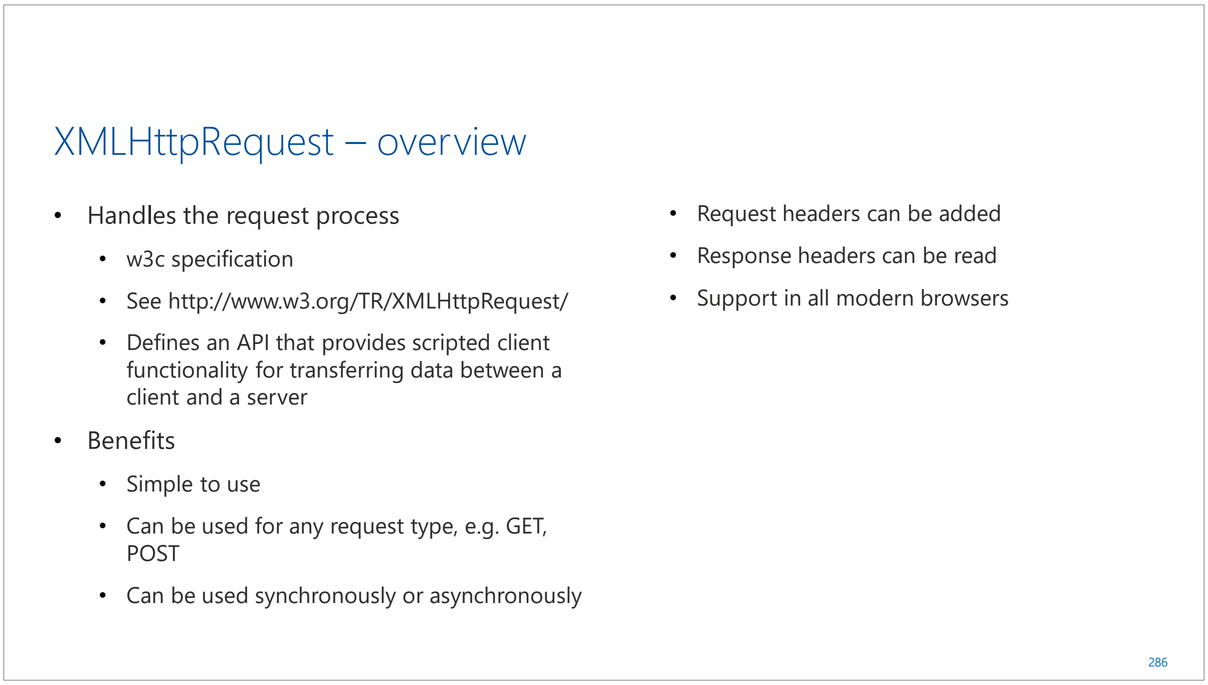
Appendix AJAX and XMLHttpRequest

JAVASCRIPT FUNDAMENTALS

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

285



XMLHttpRequest – overview

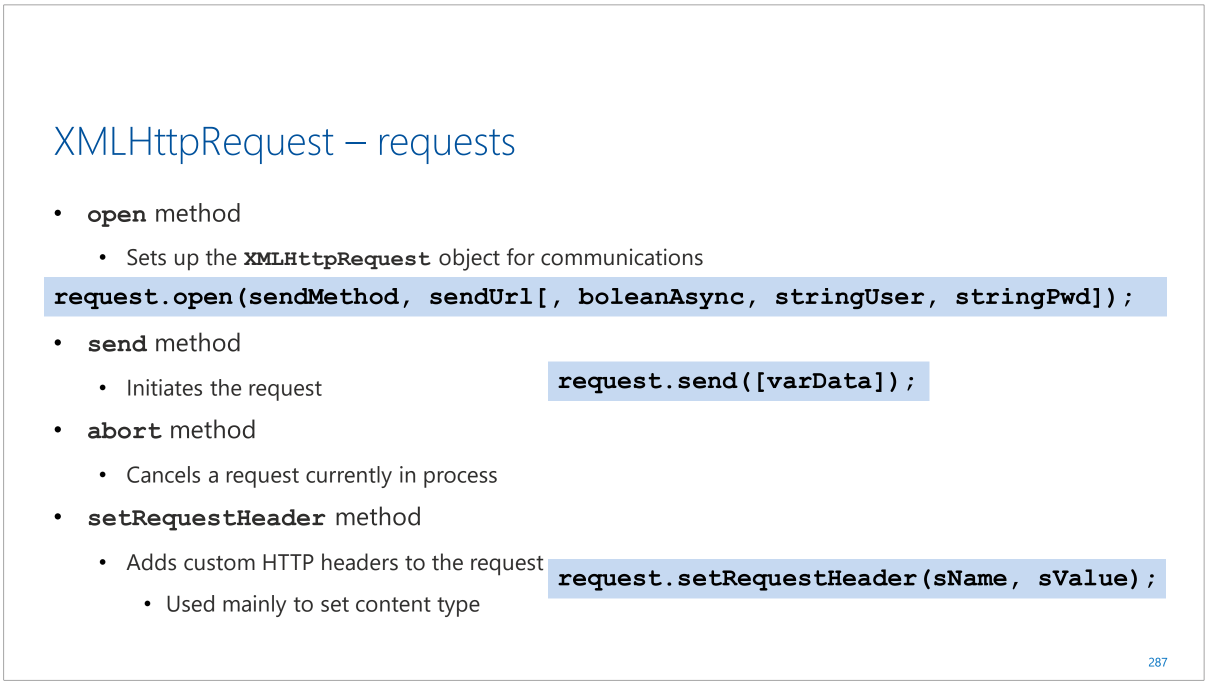
- Handles the request process
  - w3c specification
  - See http://www.w3.org/TR/XMLHttpRequest/
  - Defines an API that provides scripted client functionality for transferring data between a client and a server
- Benefits
  - Simple to use
  - Can be used for any request type, e.g. GET, POST
  - Can be used synchronously or asynchronously
- Request headers can be added
- Response headers can be read
- Support in all modern browsers

286

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.



XMLHttpRequest – requests

- **`open`** method
  - Sets up the **`XMLHttpRequest`** object for communications

```
request.open(sendMethod, sendUrl[, boleanAsync, stringUser, stringPwd]);
```

- **`send`** method
  - Initiates the request

```
request.send([varData]);
```

- **`abort`** method
  - Cancels a request currently in process
- **`setRequestHeader`** method
  - Adds custom HTTP headers to the request
    - Used mainly to set content type

```
request.setRequestHeader(sName, sValue);
```

287

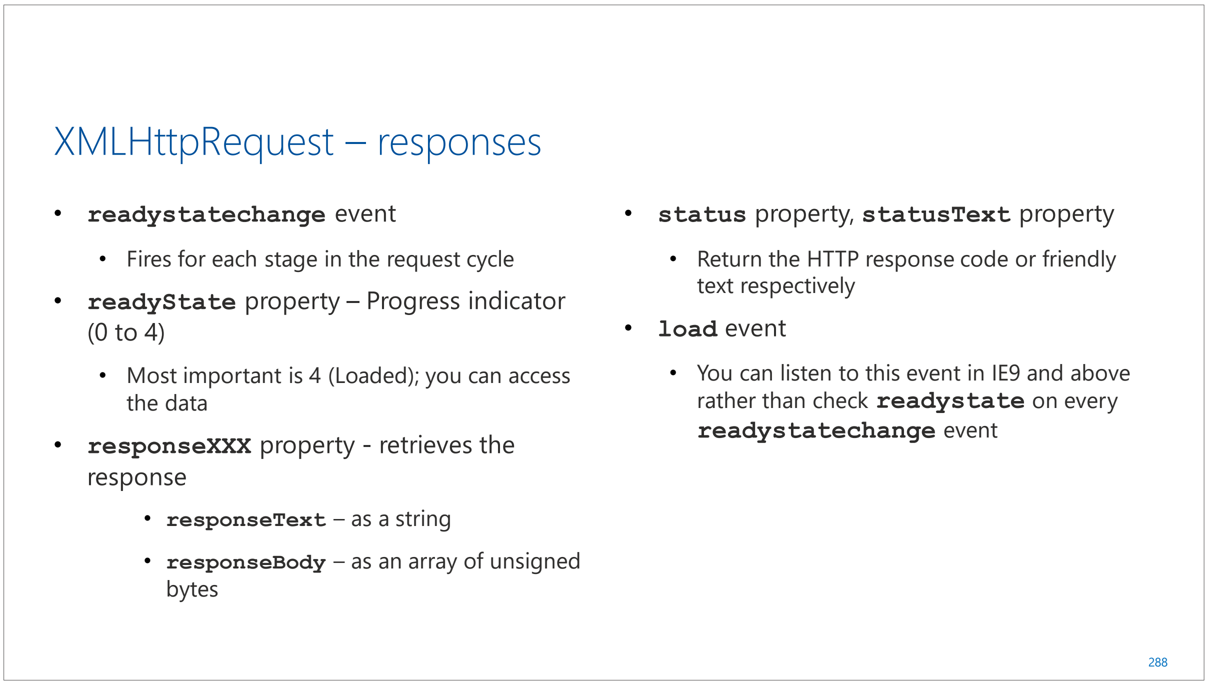
There are four named request methods; open, send, abort and setRequestHeader.

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

- **`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
- **`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

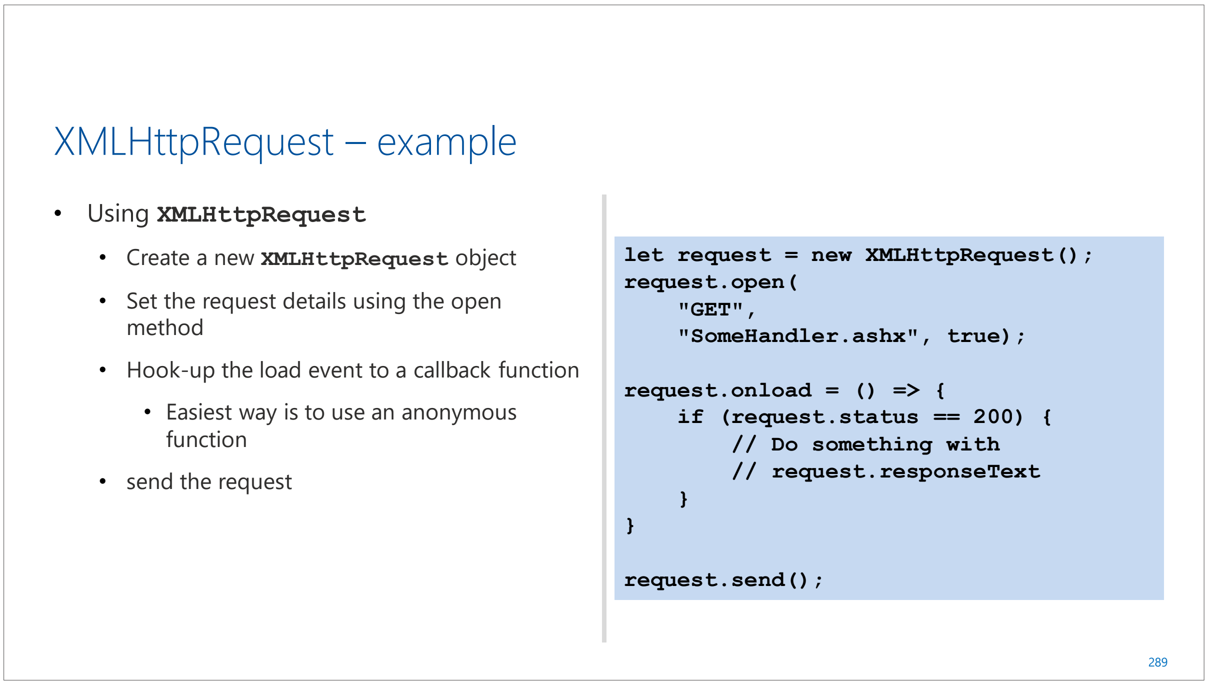
288

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

This event fires for each stage in the request lifecycle and can be used to retrieve the content of the response. The readyState property indicates what stage the request/response is at.

You can che ption from
the statusTe . You can
interrogate t

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();
```

289

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.

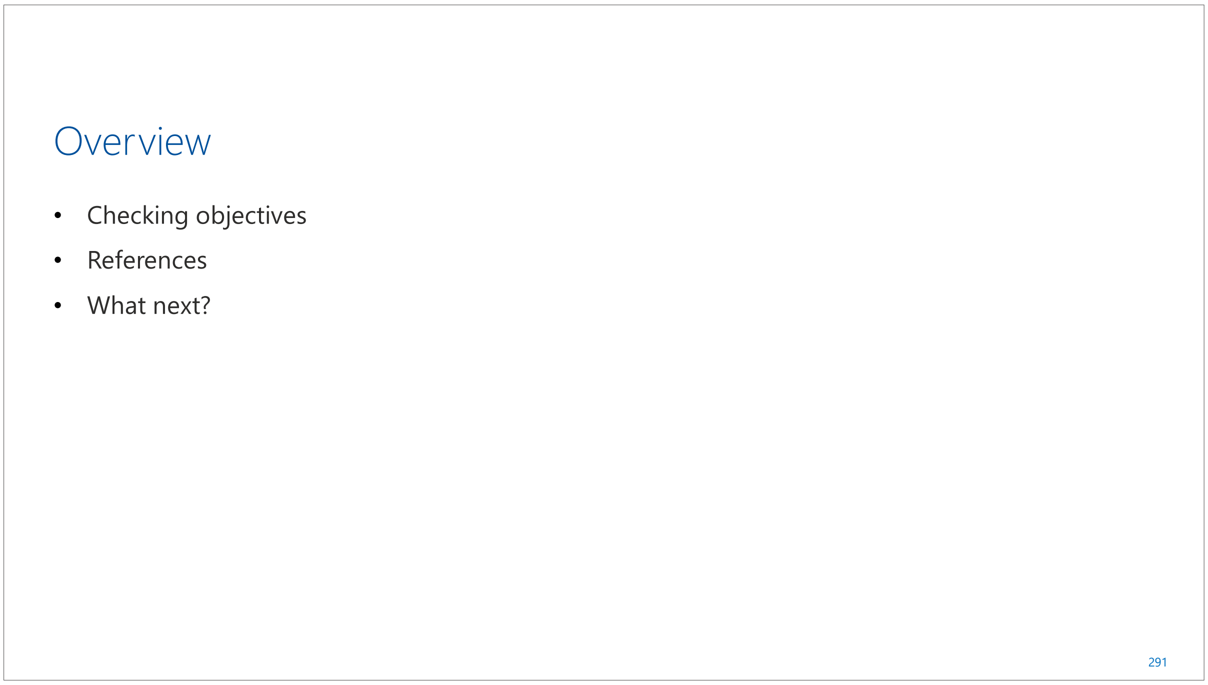
We then check to see that the request completed successfully (HTTP response code 200 – OK).

We can then do something with the responseText/responseXML.

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

Conclusion

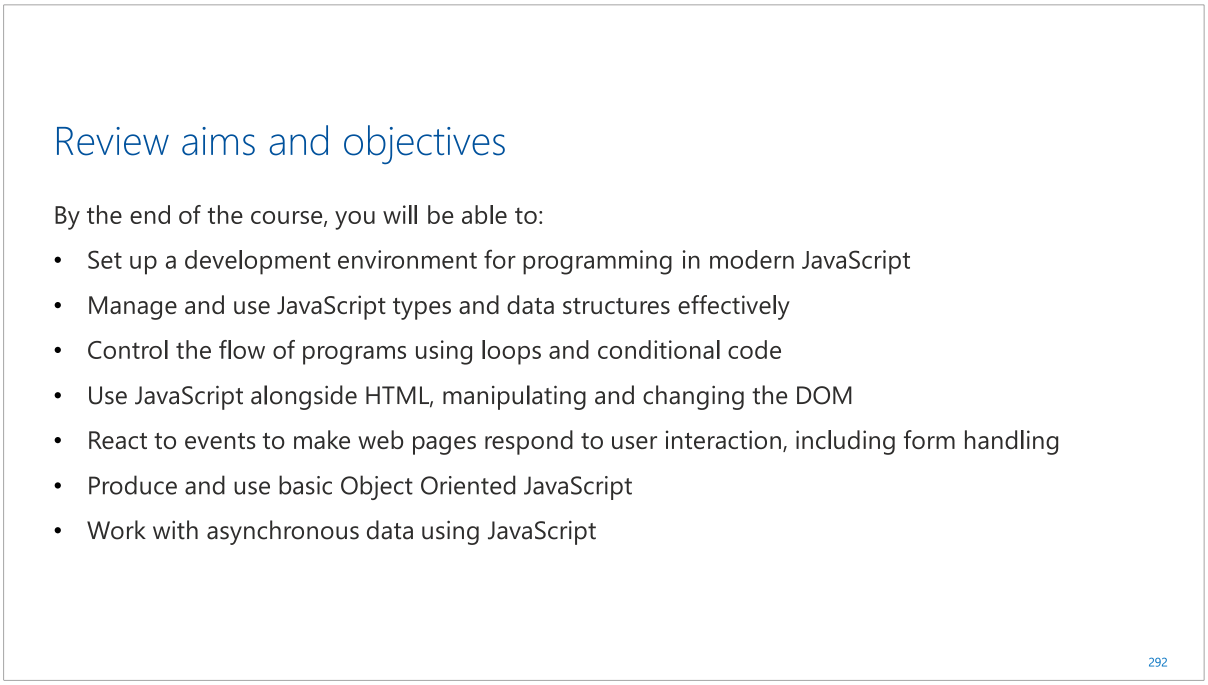
WEB DEVELOPMENT FUNDAMENTALS - JAVASCRIPT



Overview

- Checking objectives
- References
- What next?

291



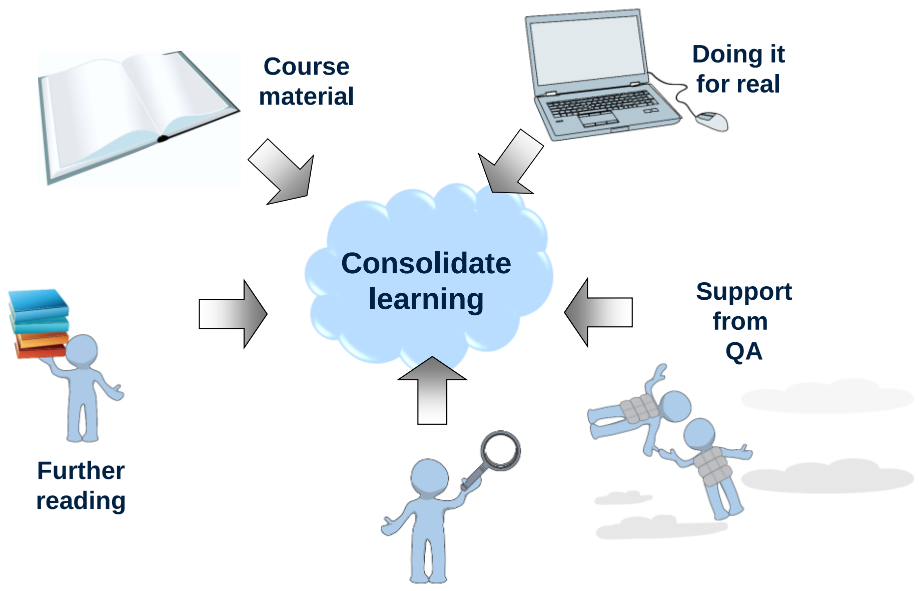
Review aims and objectives

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

- Set up a development environment for programming in modern JavaScript
- Manage and use JavaScript types and data structures effectively
- Control the flow of programs using loops and conditional code
- Use JavaScript alongside HTML, manipulating and changing the DOM
- React to events to make web pages respond to user interaction, including form handling
- Produce and use basic Object Oriented JavaScript
- Work with asynchronous data using JavaScript

292

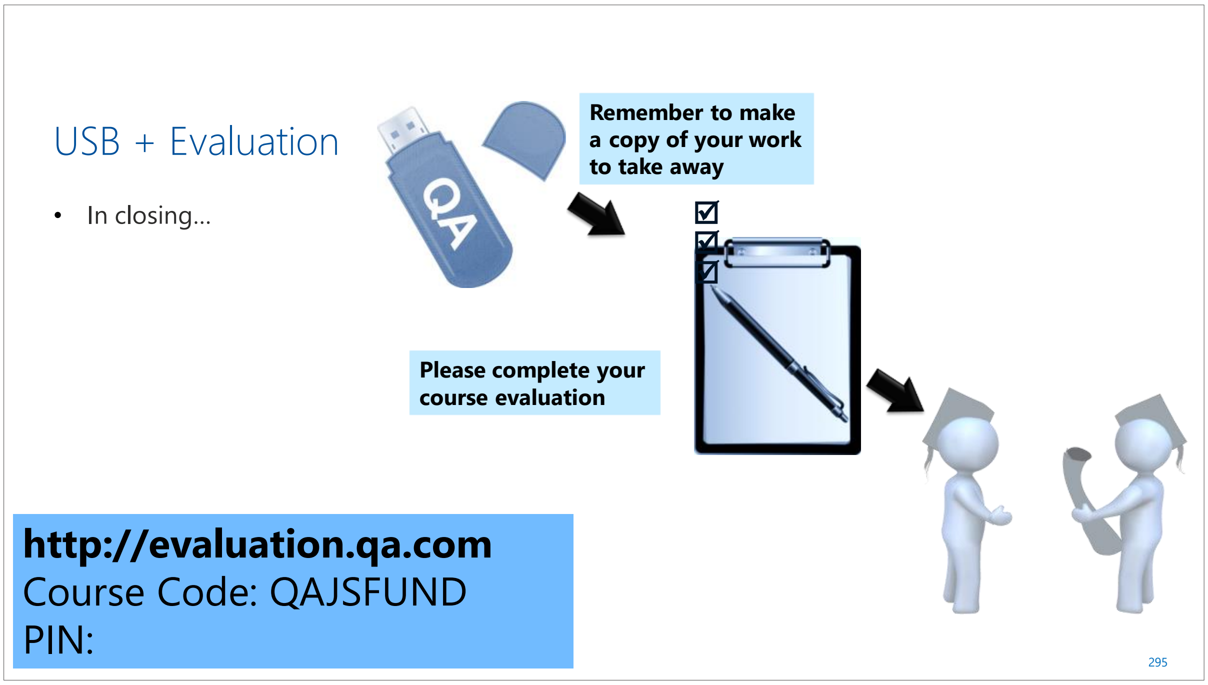
WHAT NEXT?


Course material
Doing it for real
Consolidate learning
Further reading
Support from QA

Help from QA

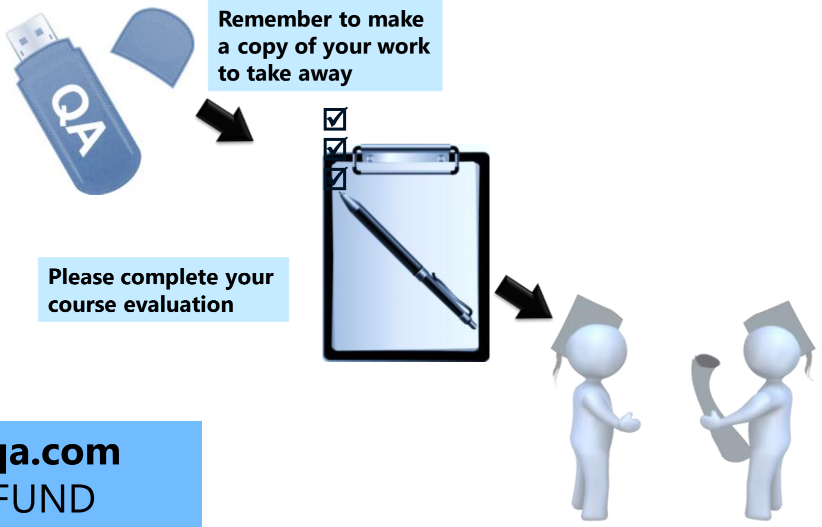
- Ever increasing range of courses
    - Vendor neutral skills
        - JavaScript
        - PHP
        - Agile
    - Application development
        - C#
        - Python
        - Java
    - DevOps
- Web site
    - www.qa.com
    - Social networking
    - Blogs http://www.qa.com/blogs

294



USB + Evaluation

- In closing…



Remember to make a copy of your work to take away

Please complete your course evaluation

http://evaluation.qa.com
Course Code: QAJSFUND
PIN:

295