StayAhead Training

the training specialist

bringing people *and* technology together

Oracle Database	Oracle Development	Oracle Fusion Middleware	Oracle Business Intelligence
Oracle e-Business	MySQL	MariaDB	MongoDB
Unit Testing Frameworks	Maven	Hibernate	Acceptance Testing Frameworks
Java Technology	Web Development HTML & CSS	JavaScript & jQuery	PHP, Python Perl & Ruby
OOAD	Spring	UNIX	Solaris, AIX & HP-UX
Red Hat Linux	SUSE Linux	LPI Linux	Business Analysis ITIL® & PRINCE2®

London | Birmingham | Leeds | Manchester | Sunderland | Bristol | Edinburgh

scheduled | closed | virtual training

www.stayahead.com +44 (0) 20 7600 6116 sales@stayahead.com

JavaScript 1

Update 1.0

JavaScript 1

3 days training

Home Enquiries

JavaScript 1 Course Overview

The JavaScript 1 course comprises sessions dealing with embedding a script in a web page, variables and expressions, conditions and loops, functions, objects, arrays, errors and debugging, the DOM, event handling, the BOM, and AJAX.

The delegate will learn how to add dynamic and interactive behaviour to their web pages.

Exercises and examples are used throughout the course to give practical hands-on experience with the techniques covered.

Skills Gained

The delegate will practice:

- Embedding or linking to JavaScript code in a web page
- Declaring and initialising variables
- Constructing expressions
- Constructing conditional and iterative statements
- Declaring and invoking functions
- Creating and manipulating objects
- Creating and manipulating arrays
- Handling errors and debugging
- Navigating and manipulating the DOM
- Writing and assigning event handlers
- Using the DOM to obtain and store useful information
- Obtaining data from the server asynchronously

Who will the course benefit?

The JavaScript 1 course is designed for web designers and developers who are familiar with HTML & CSS, and want to be able to add dynamic and interactive elements to their web sites.

Couse objectives

This course aims to provide the delegate with the knowledge to be able to add dynamism and interactivity to a web site by producing scripts that exploit all core elements of the JavaScript language including AJAX.

Requirements

Delegates should be able to build and style simple web pages using HTML & CSS. This knowledge can be obtained by attendance on the pre-requisite HTML & CSS course.

Pre-requisite courses

- HTML & CSS

Follow-on courses

- HTML5 & CSS3 with JavaScript
- JavaScript 2
- jQuery Web Development
- Bootstrap
- Angular Development
- Aurelia Framework

NOTE: Course technical content is subject to change without notice.

Table of Contents

Chapter 1: Introduction

Chapter 2: Structure

Chapter 3: Variables & Expressions

Chapter 3: Variables & Expressions (continued)

Chapter 3: Variables & Expressions (continued)

Chapter 4: Conditions & Loops

Chapter 5: Functions

Chapter 5: Functions (continued)

Chapter 6: Objects

Chapter 6: Objects (continued)

Chapter 7: Arrays

Chapter 7: Arrays (continued)

Chapter 8: Errors & Debugging

Chapter 9: DOM

Chapter 10: Event Handling

Chapter 11: The Window Object (continued)

Chapter 12: AJAX

CHAPTER 1

Introduction

What is JavaScript?

JavaScript is the programming language of the web. It enables live changes to the HTML page via user interactions which can affect structure, style, content, and behaviour. JavaScript-enabled web pages may fetch data from sources outside the page and display them in the page without reload.

All major browsers have inbuilt support for JavaScript but it may be disabled by the user.

There are many frameworks available that are built using JavaScript. To exploit these frameworks fully and wisely it pays to have a core understanding of the underlying language, and debate is always vigorous about whether or not any given thing should be done in pure "vanilla" JavaScript.

Dynamism and Interactivity

Think of a house, if you will, a new build. HTML is the work of the bricklayers and engineers who lay the foundation and raise the structure. CSS is the work of the plasterers and decorators, who provide the style, right down to the internal furnishings. JavaScript is the electrician who wires it all up and makes it work.

How do we make changes to our "house", our web page, in response to what the user does? JavaScript is the live wiring of the front end developer.

JavaScript code can be written to detect and respond to keyboard presses, mouse events, input into form fields, button clicks, device sizes, touch, tap and gesture events, scroll behaviours, and much more.

JavaScript and ECMAScript

JavaScript's initial codebase was developed by Brendan Eich for Netscape in 1995. At this time Eich was co-founder of Mozilla, which still maintains the developer API. He now has interests in WebAssembly language and, in 2017, he launched a new cryptocurrency and a new browser.

JavaScript wasn't called that at first and bears no intentional resemblance to Java except in name only. JavaScript version numbers are rarely quoted. It's just JavaScript.

JavaScript is an implementation of the ECMA-262 standard, adopted in 1997 by the European Computer Manufacturer's Association. Other languages implement the ECMA specification such as Jscript (Microsoft), and TypeScript. Version numbers come from the ECMA standard, and are one ahead of the release year. ECMAScript 7 (ES7 for short) is also ECMAScript 2016.

Versions

The table below lists JavaScript and ECMAScript versions. Note that ECMAScript 4 was never released.

Year	JavaScript	ECMAScript
1996	1.0	None
1997	1.2	1
1998	1.3	-
1999	-	2
2000	1.5	3
2005	1.6	-
2006	1.7	-
2008	1.8	-
2009	1.8.1	5
2015	-	6
2016	-	7
2017	-	8

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Editor setup

If necessary, download and install a text editor for writing JavaScript code. Your trainer is likely to be using one of the following:

- Sublime https://www.sublimetext.com/
- Brackets http://brackets.io/
- Visual Studio Code https://code.visualstudio.com/

If you'd prefer to use something else by all means do so. If you're not sure, we recommend that you use whichever editor your trainer is using. That way, he/she will be able to help you if you get stuck.

NOTE

Sublime may be downloaded and evaluated for free, however a licence must be purchased for continued use.

Browser setup

We recommend and support Google Chrome. It should be installed by default.

Dynamism and interactivity

The code provided for this exercise serves as a simple demonstration of what JavaScript enables you to do. You are not expected to understand it… yet.

1. Create a folder in which to save your work on the desktop.
2. Create an HTML page named demo.html.
3. Add a heading of your choosing.
4. Add a text field labelled Your name with an id of name.
5. Add a button labelled Submit with an id of submit.
6. Add a paragraph with an id of welcome.
7. Style the page as you see fit.
8. Copy and paste the code below just before the closing body tag:

```
<script>
   function getAndDisplayName() {
      var name = document.querySelector("#name").value;
      document.querySelector("#welcome").innerHTML =
         "Welcome to my site, " + name;
   }
   document.querySelector("#submit").addEventListener(
      "click", getAndDisplayName);
</script>
```

9. Save your changes.
10. Open the file in Chrome.
11. Enter your name into the text field and click the Submit button.

    If you've done everything correctly, a welcome message should be displayed. This demonstrates both interactivity (the user is able to interact with the page by clicking the Submit button), and dynamism (the content of the page is changed).

12. You might like to save a version of this page as a template for the exercises to come in the following chapters.

CHAPTER 2

Structure

The script and noscript Tags

<script>

Code to be interpreted as JavaScript may be placed between **<script>** tags. External code may be embedded by adding the **src** attribute to the script tag. HTML5 no longer requires the **type** attribute to be set as JavaScript is the default scripting language.

<noscript>

The noscript tag is used to include code in the HTML page ONLY IF a script type (in our case JavaScript) is unsupported or if scripting is currently turned off in the browser.

An example of good use might be a fall-back text menu:

```
<noscript>
   <ul>
      <li><a href="index.html">Home</a></li>
      <li><a href="about.html">About</a></li>
      <li><a href="contact.html">Contact</a></li>
   </ul
</noscript>
```

You can't have one set of **<noscript>** tags inside another.

Linked, Local, and Inline Scripts

Linked

A script file with a .js extension may be embedded in the HTML document:

```
<script src="my-script.js"></script>
```

Advantages:

- all your JavaScript code site-wide is in one place
- the script loads into browser memory cache on first load, and is available from the cached copy for the rest of the visit without separate requests

Disadvantages:

- the script can load before the HTML and cause referencing problems

Local

Code between script tags may be placed anywhere in the HTML document:

```
<script>
   // code here
</script>
```

Advantages:

- handy during development and testing

Disadvantages:

- code duplication site-wide if the same code is needed on different pages

Inline

Code can be embedded inline in HTML elements like so:

```
<article onclick="alert('Please subscribe to continue reading')">
```

The implications for code duplication should be obvious.

Case Sensitivity

JavaScript is case sensitive. Consistency in the use of keywords, variables, functions and identifiers is therefore needed.

HTML, however, is not strictly case-sensitive – this can cause confusion, as you will see, between JavaScript code within HTML tags and vice versa.

Valid HTML: ONCLICK, onClick, onclick

Whitespace

JavaScript tools are available to compress your JavaScript code. It therefore follows that interpreters ignore carriage returns, spaces and tabs between tokens.

However, developers expect your code to be easy to read, and so it is best practice to space and indent your code as you would in any third generation programming language.

Common compression or code minification tools include:

- JSMIN (early and offline)
- DOJO Compressor (tiered licensing)
- Packer (online copy and paste)
- Minifier.org (online or as a dependency – does CSS too)

Code may be spaced and indented as you choose, but it is advisable to follow some form of convention. While there is no absolute standard set of guidelines for JavaScript, there are some useful resources online at the Google and Mozilla developer networks that can give you some good pointers, or form the basis of your own standards.

Comments

Add a single line comment using `//`. For example:

```
// This is a single line comment
alert('Close me'); // A comment can be placed on the same line as code
```

You can add comments in your code with more versatility using `/*` at the beginning and `*/` at the end of the text you want the browser to ignore. In part of a line only:

```
alert(/* Can't see me */);
```

…or around multiple lines:

```
/*
Copyright 2018 StayAhead Training Ltd. All Rights Reserved.
@fileoverview Description of file, its uses and information
about its dependencies.
@author user@stayahead.com (Firstname Lastname)
*/
```

In general, using the C++/Java style for comments is a good idea, so:

- Copyright and authorship notice.
- A top-level (file-level) comment to show what's in the file (e.g. a one-paragraph summary of what the major pieces are, how they fit together, and with what they interact).
- Class, function, variable, and implementation comments as necessary.
- An indication of the browsers in which the code is expected to work (if applicable).
- Proper capitalization, punctuation, and spelling.
- Avoid sentence fragments.

Code on the assumption that someone else is going to read and maintain your code.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Linked, local, and inline scripts

1. Create a JavaScript file named external.js.
2. Copy and paste the following code:

```
function sayHi() {
   alert("Hi external!");
}
```

3. Save your changes.
4. Create an HTML page named linked-local-inline.html.
5. Add a heading of your choosing.
6. Add a button labelled Say hi as follows:

```
<button onclick="sayHi()">Say hi</button>
```

7. Embed the external.js script just before the closing body tag:

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

8. Save your changes.
9. Open the file in Chrome and click the button.
10. Copy and paste the following code just before the closing body tag:

```
<script>
   function sayHi() {
      alert("Hi inline!");
   }
</script>
```

11. Save your changes.
12. Refresh the file in Chrome and click the button.
13. Modify the button code as follows:

```
<button onclick="function sayHi() { alert('Hi inline!'); } sayHi()">
   Say hi</button>
```

14. Save your changes.
15. Refresh the file in Chrome and click the button.

    You should have seen that inline code takes precedence over local code which takes precedence over external code. Having said that, like CSS, external JavaScript code is preferred for the reasons described previously.

CHAPTER 3

Variables & Expressions

Variable Declaration and Assignment

Variables are names associated with values. They store or contain the value, either by reference or by value. They generally allow for storage and manipulation of data and are fundamental to programming.

Declaration

Before using a variable, it should be declared using the **var** keyword:

```
var width;
```

Variable declarations, wherever they occur, cause the JavaScript engine to reserve some memory, but you do not need to give it any value at that time.

If the **var** keyword is absent, JavaScript will still create the variable. This looseness is deceptive: variables declared this way are global and can lead to instability in your application. This issue, called scope, is explored in chapter five.

Assignment

The assignment operator **=** is used to associate a value with a variable.

```
width = 39.5;
```

Variables can be assigned either:

- a **value**, e.g. some data
- a **reference**, e.g. a pointer to some data elsewhere in memory
- an **expression**, which is some code that **evaluates** to a value or a reference

Initialisation

When a variable is assigned a value for the first time, it is said to be **initialised**. This may occur after declaration, as in the example above, or at the same time:

```
var width = 39.5;
```

It is possible, but not always readable, to declare multiple variables on the same line, mixing declaration with initialisation:

```
var foo; var bar = 10; // not good practice
```

An un-initialised variable remains **undefined** until a value is assigned to it.

Naming conventions

It is best practice to name variables as descriptively as possible, starting with a lower-case letter and using camel casing to distinguish separate words in ordinary language:

```
var screenWidth = 32.0;
var outerWidth = 39.5;
var packagingWidth = 64.0;
```

It is illegal to start a variable name with a digit but by all means use them elsewhere:

```
var screenWidthMonitorType2 = 24.0;
```

The rule:

Variable (and other) names may only contain alphanumeric characters (or $, or _).

When to declare variables

It is good practice to declare a variable if your logic demands it be reused at some stage. Consider the following, where two variables are declared:

```
var form = window.document.forms[0];
var checkBox = form.elements[0];
checkBox.checked = true;
```

Instead, the element can be accessed directly and assigned **true**:

```
window.document.forms[0].elements[0].checked = true;
```

However, as a general rule, a few milliseconds of performance detriment is better than dealing with untidy and difficult to interpret code. What we may want to end up with is:

```
var isChecked = window.document.forms[0].elements[0].checked;
isChecked = true;
```

NOTE

If you re-declare a variable it will not lose its value:

```
var screenHeight;    // undefined
screenHeight = 13.5; // 13.5
...
var screenHeight;    // 13.5
```

Only if you were to re-assign it using the = operator would the value change. The third line is confusing though, as it seems like the variable contains no value and that its first use should come later on.

Data Types

JavaScript is a dynamically typed language. The data type of a variable is determined when a value is assigned, and so may be changed:

```
var x = 'Hello';  // data type of x is string
x = 123;          // data type of x is now number
x = false;        // data type of x is now boolean
```

There are six principle data types in JavaScript as follows:

Data Type	Description
string	a value surrounded by either single or double quotes
number	both integer and floating point numbers
boolean	true or false
object	a collection of key value pairs (properties)
null	a value that references/points to nothing
undefined	the value assigned to a variable that has been declared but not initialised

The typeof operator may be used to determine the data type of a value/variable.

```
typeof 1; // 'number'
var x = 'Hello';
typeof x; // 'string'
```

JavaScript may automatically convert types, for instance when attempting to append a string to a number. There is an element of simplicity afforded by this automatic conversion; however, it is important to be aware of the conversions that take place, in order to avoid pitfalls in longer and more complex programs.

Type Conversion

When the JavaScript engine encounters a string and a number and is asked to combine them using **+**, it will automatically convert the number to a string. Note that expressions, in the absence of parentheses, are evaluated from left to right. Consider the following:

```
10 + 10           // 20
'10' + '10'       // '1010'
10 + '10'         // '1010'
10 + '10' + 10    // '101010'
10 + 10 + '10'    // '2010'
```

As is shown here this can lead to unexpected output:

```
var foo = 1 + 2 + ' farmers';          // '3 farmers'
var bar = 'Num farmers: ' + 1 + 2;     // 'Number of farmers: 12'
```

This is because expressions are evaluated from left to right, unless parentheses change this order. The second line above may be changed to yield the correct result by adding parentheses such that 1 + 2 is evaluated first:

```
var bar = 'Num farmers: ' + (1 + 2);   // 'Number of farmers: 3'
```

Operators

An operator is a special symbol that instructs the computer to do something, e.g. subtract one value from another, determine if one value is greater than another etc.

Operators work on operands. Operands can be variables or data. Some operators require two operands; some only require one:

```
var x = 8;
var result = 10 > x; // > is the operator, 10 and x are operands
```

A compound expression is one that comprises many operators and operands.

Expressions

An expression is any valid unit of code that resolves to a value. Expressions usually occur to the right of the assignment operator:

```
var x = 5;
var result = x * 7; // x * 7 is the expression
```

The value that an expression evaluates to may be a primitive type, or a more complex type like a list or an object.

Arithmetic Operators

Addition +

Where each operand is a number, an expression using the addition operator yields the sum of the operands. Note that if one of the operands is a string, the operands are concatenated together.

```
var a = 3;
var b = 4;
var result = a + b; // 7

var c = 2;
var d = '5';
var result = c + d; // '25'
```

Subtraction –

Where each operand is a number, an expression using the subtract operator yields the result of subtracting the second operand from the first.

```
var a = 3;
var b = 4;
var result = a – b; // -1
```

Multiplication *

Where each operand is a number, an expression using the multiplication operator yields the product of the operands.

```
var a = 3;
var b = 4;
result = a * b; // 12
```

Exponential **

Where each operand is a number, an expression using the exponential operator yields the value of the first operand to the power of the second.

```
var a = 3;
var b = 4;
var result = a ** b; // 81
```

Division /

Where each operand is a number, an expression using the division operator yields the result of dividing the first operand by the second. All numbers are floating-point, so all divisions have floating-point results, e.g. 7/2 evaluates to 3.5, not 3.

```
var a = 3;
var b = 4;
var result = a / b; // 0.75
```

Modulo %

Where each operand is a number, an expression using the modulo operator yields the remainder of dividing the first operand by the second.

```
var a = 3;
var b = 4;
var result = a % b; // 3
```

Negation –

The negation operator requires only one operand and yields the negative of its operand. Note that the negation operator does not change the value of the operand.

```
var a = 3;
var result = -a; // -3 (a is still 3)
```

Increment ++

The increment operator requires only one operand and increments it by one. The behaviour depends on its position relative to the operand:

- Used before the operand, where it is known as the **pre-increment** operator, it increments the operand first, and then the expression is evaluated.
- Used after the operand, where it is known as the **post-increment** operator, the expression is evaluated first, then the value of the operand is incremented.

```
var a = 3;
var result = ++a; // result is 4, a is 4 (pre-increment)
var result = a++; // result is 3, a is 4 (post-increment)
```

Decrement --

The decrement operator requires only one operand and decrements it by one. The behaviour depends on its position relative to the operand:

- Used before the operand, where it is known as the **pre-decrement** operator, it decrements the operand first, and then the expression is evaluated.
- Used after the operand, where it is known as the **post-decrement** operator, the expression is evaluated first, then the value of the operand is decremented.

```
var a = 3;
var result = --a; // result is 2, a is 2 (pre-decrement)
var result = a++; // result is 3, a is 2 (post-decrement)
```

Assignment Operators

The assignment operator is used to assign a value to a variable. For example:

```
var foo = 0;
```

The assignment operator expects its left-hand operand to be a variable, an array element, or object property, and expects its right-hand operand to be a value or expression.

You can assign a variable to another variable. The value or reference of one is copied into the other.

```
var a = 3;
var b = a; // b is 3
```

In the example above the value of a (3) is copied into b.

The assignment operator has right-to-left associativity. This means that the expression on the right of the operator is evaluated first. Consider the following;

```
var a = 3;
var b = 4;
var result = a + b;
```

The expression a + b is evaluated first, and then the result is assigned to the variable named result.

There are a number of other assignment operators (tabled below) that provide a shortcut by combining assignment with some other operation.

Assignment Operator	Example	Long-hand Equivalent
+=	a += b	a = a + b
-=	a -= b	a = a – b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b

Comparison Operators

Comparison operators are used to compare values. An expression comprising a comparison operator will yield a boolean - true or false. Comparison operators are commonly used in conditional statements and loops to control program flow.

Loose equality ==

An expression using the loose equality operator yields true if the operands are loosely equal. Loose equality means that operands are compared by value/reference only.

Numbers, strings, and booleans are compared by value. This means that two operands are equal if they contain the same value regardless of the data type.

Objects and arrays are compared by reference. This means that two operands are equal only if they refer/point to the same object. Two separate arrays will never be equal, by the definition of the loose equality operator, even if they contain identical elements.

```
var a = 3;
var b = 4;
var result = a == b; // false

var c = '3';
var result = a == c; // true
```

NOT loose equality !=

An expression using the NOT loose equality operator yields true if the operands are NOT loosely equal.

```
var a = 3;
var b = 4;
var result = a != b; // true
```

Strict equality ===

An expression using the strict equality operator yields true if the operands are strictly equal. Strict equality means that operands are compared by value/reference and data type.

```
var a = 3;
var b = 4;
var result = a === b; // false

var c = '3';
var result = a === c; // false (a is a number, c is a string)
```


NOT strict equality !==

An expression using the NOT strict equality operator yields true if the operands are NOT strictly equal.

```
var a = 3;
var b = 4;
var result = a !== b; // true

var c = '3';
var result = a !== c; // true
```

Less than <

An expression using the less than operator yields true if the first operand is less than the second. Note that strings are compared lexicographically.

```
var a = 3;
var b = 4;
var result = a < b; // true
```

Greater than >

An expression using the greater than operator yields true if the first operand is greater than the second.

```
var a = 3;
var b = 4;
var result = a > b; // false
```

Less than or equal to <=

An expression using the less than or equal to operator yields true if the first operand is less than or equal to the second.

```
var a = 3;
var b = 4;
var result = a <= b; // true
```

Greater than or equal >=

An expression using the greater than or equal to operator yields true if the first operand is greater than or equal to the second.

```
var a = 3;
var b = 4;
var result = a >= b; // false
```

in

An expression using the in operator yields true if the first operand is a property of the second. The in operator should only be used when working with objects.

instanceof

An expression using the instanceof operator yields true if the first operand is an instance of the second. The instanceof operator should only be used when working with objects.

Logical Operators

Logical operators are used to construct complex expressions comprised of more than one comparison.

And &&

An expression using the and operator evaluates to true if each of its operands evaluates to true. If the first operand evaluates to false, then the result will be false regardless of the second operand. In these circumstances, the second operand will not be evaluated.

```
var a = 3;
var b = 4;
var result = a < b && b > a;     // true
var result = a > b && ++b == 5;  // false (and b is still 4)
```

Or ||

An expression using the or operator evaluates to true if either operand evaluates to true. If the first operand evaluates to true, then the result will be true regardless of the second operand. In these circumstances, the second operand will not be evaluated.

```
var a = 3;
var b = 4;
var result = a > b || b > a;     // true
var result = a < b || ++b == 5;  // false (and b is still 4)
```

Not !

The not operator requires only one boolean operand and yields its inverse value. Note that the not operator does not change the value of the operand.

```
var a = true;
var result = !a; // false (a is still true)
```

Ternary

Whilst often regarded as a logical or comparison operator, the ternary operator is technically a control structure as it affects the program flow, and is dealt with in chapter four.

Order of Precedence

Operators have a strict order of precedence. Where an expression is comprised of more than one operator, the order of precedence dictates the order in which expressions are evaluated. For example, multiplication and division have higher precedence than do addition and subtraction. Consider the following:

```
var result = 20 + 3 * 10;
```

The multiplication expression is evaluated first (3 * 10 = 30), followed by the addition expression (20 + 30 = 50).

The order of precedence may be circumvented using the grouping operator (parentheses). Consider the following:

```
var result = (20 + 3) * 10;
```

The expression in parentheses is evaluated first (20 + 3 = 23), followed by the multiplication expression (23 * 10 = 230).

Where the expression comprises more than operator of the same precedence, the expressions are evaluated from left to right. Consider the following:

```
var result = 21 / 7 * 4; // 12
```

You can see the full order of precedence table on the MDN website here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Employee data

1. Create an HTML page named employee.html.
2. Add a heading of your choosing.
3. Add a script element in the head.
4. Declare and initialise variables to store the following data. Be sure to choose descriptive variable names:

Employee name	Tim Smith
Started with the company	2017
Salary per month	£2409.50
Permanent	Yes

5. Change the value of the salary per month variable to £2510.90.
6. Declare and initialise a variable named tim and assign it a string comprised of his name, the year he started with the company, his salary, and his permanent status, e.g. Name: Tim Smith, Started with the company: 2017, …
7. Display the variable tim in an alert:

```
alert(tim);
```

8. Save your changes.
9. Open the file in Chrome.
10. Is the heading rendered before or after the popup is displayed?

Operators

Assuming:

```
var a = 7;
var b = 3;
var truth = false;
```

Complete the table below by writing the result of each expression.

NOTE

You should assume a, b, and truth are reset to 7, 3, and false respectively after each expression.

Arithmetic	**Result**
a + b	
a - b	
a * b	
a / b	
a % b	
Unary	**Result**
+a	
-b	
++a	
--b	
a++	
b--	
!truth	
Relational	**Result**
a == b	
a != b	
a > b	
b >= 4	
a < b	
a <= 7	

Logical	**Result**
a > b && b < a	
a == b && b < a	
a < b && b > a	
a > b || b < a	
a == b || b < a	
a < b || b > a	
a > b ^ b < a	
a > b ^ b > a	
Order of precedence	**Result**
a + b * 2	
(a + b) * 2	
15 / b * a	
14 % b + b	
--a * b++	
a++ * 2 + --b	

CHAPTER 4

Conditions & Loops

if else

The **if** statement enables the making of decisions, or the executing of statements conditionally. This statement has three forms. The first is:

```
if (boolean expression)
   statement
```

The expression is evaluated. If it is true, then the statement is executed. If the expression is false, then the statement is not executed. For example:

```
if (name == null)
   name = 'Larry';
```

A single statement may be replaced by a statement block:

```
if (surname == null || surname == '') {
   surname = 'undefined';
   alert('Please enter a surname');
}
```

The second form of the if statement introduces an **else** clause to be executed when the expression evaluates to false. Its syntax is:

```
if (boolean expression)
   statementA
else
   statementB
```

For example:

```
if (firstName != null)
   alert('Hello ' + firstname);
else {
   var firstName = prompt('What is your first name?');
   alert('Hello ' + firstname);
}
```

The third form of the if statement provides for more than two branches. Its syntax is:

```
if (boolean expression)
   statementA
else if (boolean expression)
   statementB
else
   statementC
```

There is no limit to the number of else if clauses.

For example:

```
if (team1Score > team2Score)
   alert('Team 1 wins!');
else if (team2Score > team1Score)
   alert('Team 2 wins!');
else
   alert('It\'s a tie!');
```

With nested if else statements, caution is required to ensure that the else clause goes with the appropriate if statement:

```
var i = 1;
var j = 1;
var k = 2;
if (i == j)
   if (j == k)
   console.log('i equals k');
else
   console.log('i doesn't equal j');   // WRONG!
```

In this example, the else clause appears to be matched to the outer if statement because the indenting is poor. It is, in fact, matched to the inner if statement.

The rule in JavaScript (as with most languages) is that an else clause is matched to the closest if statement. To make this example easier to read, understand, maintain, and debug, you might use braces:

```
if (i == j)  {
   if (j == k) {
      console.log ('i equals k');
   }
}
else {
   console.log ('i doesn't equal j');
}
```

Braces or no, indenting is key.

switch

The switch statement is used to execute one of several code blocks of code:

```
switch(expression) {
   case 1:
      execute block 1
      break;
   case 2:
      execute block 2
      break;
   case 3:
      execute block 3
      break;
   default:
      execute if value is none of 1, 2 or 3
}
```

An expression (usually a variable) is tested for strict equality against each case in turn. A match means the block of code associated with that case is executed. The **break** keyword prevents the flow from running into the following case:

```
var date = new Date();
var day = date.getDay();
switch (day) {
   case 5:
      alert('Friday!');
      break;
   case 6:
      alert('Saturday!');
      break;
   case 0:
      alert('Sunday!');
      break;
   default:
      alert('Roll on the weekend!');
}
```

The switch structure can be used without break statements if it fits business logic. When breaks are omitted the switch becomes additive. The block associated with the matching case is executed, as is each block that follows.

Ternary Operator

The ternary operator is used to construct a conditional expression, which is assigned to a variable, and that evaluates to ONLY ONE of two outcomes depending on the truth of the conditional test. Both outcomes cannot be true, neither should both be false, though the way the expression evaluates is usually down to business logic.

It is the programmatic equivalent of an exclusive OR (XOR) gate in electronics and its syntax is as follows:

```
var result = (boolean expression) ? Value if true : Value if false;
```

For example:

```
var result = (mark >= 50) ? 'PASS!' : 'FAIL!';
```

The ternary operator is so-called as it is the only JavaScript operator to take three operands (the boolean expression, the value if true, the value if false). In addition, the ternary operator may also take multiple conditions and evaluations – for a fuller specification see: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

for

The **for** loop enables the repetition of a statement or block of statements. It consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement or block of statements to be executed in the loop. Its syntax is:

```
for (control variable; boolean expression; change control variable)
   statement
```

For example:

```
var output = '';
for (var i = 0; i < 10; i++) {
   output += i;
}
alert(output); // '0123456789'
```

Note that the control variable is initialised once at the beginning. The boolean expression is evaluated at the beginning of each loop, and the control variable is changed at the end of each loop.

The control variable need not be incremented, as in the example above. It may be incremented by any number, e.g. i += 3, or decremented.

You may also specify any number of control variables. For example:

```
var output = '';
for (var i = 0, j = 0; i < 5; i++, j += 2) {
   output += i * j;
}
alert(output); // '0281832'
```

For loops are typically used for iterating over the elements of an array.

while

Like the for loop, the **while** loop enables the repetition of a statement or block of statements. It differs from the for loop because its control variable is declared outside the loop and is changed in the body of the loop. Its syntax is:

```
while (expression)
   statement
```

The statement or block of statements is executed repeatedly while the expression evaluates to true. Note that the statement of block of statements may never be executed.

For example:

```
var output = '';
var i = 0;
while (i < 10) {
   output += i;
   i++;
}
alert(output); // '0123456789'
```

do

The **do while** loop differs from the while loop in that its boolean expression is at the bottom of the statement body. Its syntax is:

```
do
   statement
while (expression);
```

The statement or block of statements will always execute at least once.

For example:

```
var output = '';
var i = 0;
do {
   output += i;
   i++;
} while (i < 10);
alert(output); // '0123456789'
```

break

We have seen **break** statements used in the switch structure. In iterative statements, the break keyword is used to terminate the currently executing loop.

The following example searches the elements of an array for a specific value, terminating the search if the value is located:

```
var array = [1, 2, 3, 4, 5];
var target = 3;
for(var i = 0; i < array.length; i++) {
   if (array[i] == target)
      break;
}
```


continue

In iterative statements, the **continue** keyword skips the current iteration of the loop.

The continue keyword is used in the following example to skip each empty array element:

```
var list = ['one','two','three','','four','','five'];
for(var i = 0; i < list.length; i++) {
   if (list[i] == '')
      continue;  // can't proceed when element is empty
   alert(list[i]);
}
```

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Exercise preparation

Previously, you used the **alert** function to display a message in a popup. The exercises that follow require you to obtain input from the user. The **prompt** function is similar to alert - it displays a popup with a message and a text box. If the user clicks OK, the value entered into the text box is returned by the function and may be assigned to a variable. If the user clicks the Cancel button, the function returns null:

```
var strAge = prompt('Please enter your age);
```

The data type of the returned value is string, regardless of what is entered. There are many ways to convert a string to a number. Consider the following:

```
var numAge = Number(strAge);
var numAge = parseInt(strAge);
```

In the event the string cannot be converted to a number, (e.g. it comprises non numeric characters), then both Number and parseInt functions return NaN (Not a Number).

Simple calculator

1. Create a HTML page named calculator.html.
2. Add a heading of your choosing.
3. Add a script element just before the closing body tag.
4. Prompt the user for the first number.
5. Prompt the user for the second number.
6. Prompt the user for an operator (+ - / *).
7. Convert each number value entered from a string to a number.
8. Declare a variable named result and initialise it to 0. You should assign the result of the arithmetic expression to this variable.
9. If the operator is +, add the two numbers.
10. If the operator is -, subtract the second number from the first.
11. If the operator is *, find the product of the two numbers.
12. If the operator is /, divide the first number by the second.
13. Display the result in an alert with an appropriate label.
14. Save your changes.
15. Open the file in Chrome and test the code thoroughly.
16. What could be done to make this code more robust?

Chrono quiz (version 1)

1. Create an HTML page name chrono-quiz-1.html (first version).
2. Add a heading of your choosing.
3. Add a script element just before the closing body tag.
4. Declare and initialise three variables to store three questions as follows:

   In what year did the Titanic sink on her maiden voyage?
   In what year did WW1 begin?
   In what year was the Qualification of Women act passed?

5. Declare and initialise three variables to store three answers as follows:

   1912
   1914
   1918

6. Construct a for loop with a control variable initialised to 1, a boolean expression that tests if the control variable is less than or equal to 3, and that increments the control variable by 1 at the end of each loop.
7. Add a switch statement inside the loop that tests the loop's control variable.
8. If the control variable is 1, prompt the user to answer question 1. If the user's answer equals the correct answer, display an alert with the message "Right!", else display an alert with the message "Wrong! It was <correct answer>".
9. Repeat step 8 twice more for questions 2 and 3.
10. Save your changes.
11. Open the file in Chrome and test the code thoroughly.
12. How might this code be improved?

13.

CHAPTER 5

Functions

Introduction

A **function** is a block of code that is defined once but may be invoked multiple times.

As a rule of thumb, one or more lines of code repeated more than two or three times is a candidate function. It is good practice to break large blocks of code up into functions. They're easier to read, test, and maintain. Functions may be assigned to variables, and passed around your code as data, even as arguments to other functions.

As we shall see, functions are central to the JavaScript language - promoting code re-use and controlling variable scope and data privacy.

Function Declaration

Functions are declared using the function keyword, followed by:

- An **identifier** as the name of the function
- An optional, comma-separated list of **parameters** in parentheses
- The **statements** comprising the body of the function, within braces
- An optional **return** statement

For example:

```
// a function with no input (parameters) nor output (return)
function greet()  {
   alert('Hello!');
}

// a function with input but no output
function greet(name) {
   alert('Hello ' + name);
}

// a function with both input and output
function greet(name)  {
   return 'Hello: ' + name;
}
```

Input (parameters)

A parameter is a value input into a function. In the example above, the parameter named name is a variable that will store the value input into the function when it is invoked. A function may be declared with none, one, or more parameters, depending on what data it needs to do its job. Where there is more than one parameter, they must be comma separated. Consider the following:

```
function getProduct(number1, number2) {
   return product1 * product2;
}
```

The getProduct function specifies that it expects to be passed two values to do its job.

Output (return statement)

A function may be coded to return a value to the caller. The value returned is effectively the function's output. Only one value may be returned, but that value may be an array or object composed of many parts.

Function expressions

A function expression is an expression that yields a function.

```
var greet = function(name) { return 'Hello: ' + name; }
```

Note that the expression on the right side of the assignment operator is a function declaration without an identifier. It is assigned to a variable named greet. The greet variable is now a function. This is identical to:

```
function greet(name)  {
   return 'Hello: ' + name;
}
```

Function Invocation

To invoke a function is to execute the statements in the body of the function declaration. To invoke a function, you need only know its name, and what data, if any, it expects to be passed as input. A value passed to a function is an argument.

No arguments

A function invocation must include parentheses even if there are no parameters specified, for example:

```
function greet() {
   alert('Hello');
}
greet(); // no arguments invocation
```

One or more arguments

Consider again the version of the greet function declaration that specifies some input is required:

```
function greet(name) {
   alert('Hello ' + name);
}
```

When we come to invoke this function, we must pass it an argument. That argument may be a value or a variable:

```
greet('David Smith');
var sara = 'Sara Jones';
greet(sara);
```

Note that, in the case of passing a variable, the name of the argument need not match the name of the parameter.

The wrong arguments

You cannot specify data types for function parameters. If type is important, use the **typeof** operator within the function to check it. JavaScript does not check for the correct number of arguments either. Any extra arguments will be ignored. Fewer arguments will result in undefined parameters.

It is possible to check the arguments passed to a function. You can use the arguments object (every function has one) to verify that a function is invoked correctly:


```
function addTwoNumbers(a, b) {
   if(arguments.length != 2) {
      throw new Error('Incorrect number of arguments');
   }
   ...
}
```

Return values

If the function you're invoking returns a value, you will most likely want to use it. Consider again the version of the greet function declaration that returns a value:

```
function greet(name)  {
   return 'Hello: ' + name;
}
```

To get the return value, we must assign the invocation to a variable:

```
var greeting = greet('David Smith');
```

A function invocation is an expression, that is, it yields a value.

Variable Scope and Hoisting

The scope of a variable is that part of your program where the variable was declared and assigned a value. If a variable is declared outside any function it is global (accessible anywhere). However, if the variable is declared within a function, it is local to that function (accessible only within the function).

Variables declared in a function but without the var keyword are made global when the function is invoked. This is dangerous as you may inadvertently overwrite data unintentionally. You should always declare variables using the var keyword unless you mean to affect a global variable.

```
var i = 'global';
function whatScope() {
   i = 'local';
   ...
}
whatScope();
alert(i); // 'local' (did you mean to change the global variable?)
```

A variable declared anywhere within the body of the function is accessible everywhere in the function: there is no block scope using var. Consider the following:

```
function noBlockScope() {
   if(true) {
      var inner = 'I was declared in a block';
   }
   alert(inner);
}
noBlockScope(); // 'I was declared in a block' (in an alert)
```

What is seldom understood when starting out is that JavaScript is actually a compiled language, and the browser's JavaScript engine makes several passes of the code before execution. The first pass always looks for declarations of variables or functions, and will "hoist" all these to the top of the order of execution. Consider the following – the function is invoked before it is declared:

```
whatGetsHoisted();
...
function whatGetsHoisted() {
   alert('Hello world');
}
```

The statements are executed in what appears to be the wrong order. The function declaration above is compiled into memory before it executes any code segment, allowing you to call the function before you declare it in your code. **Crucially, only declarations are hoisted, not initialisations.**

Functions as Data

Functions are data. They can be assigned to variables, stored in the properties of objects or the elements of arrays, passed to functions etc. Because JavaScript is interpreted, and because it treats functions as a distinct data type, the language allows functions to be defined dynamically.

It is important to understand what the **function** keyword actually does. The function keyword creates a function but also defines a variable. In this way, the function keyword is like the **var** keyword. Consider the following:

```
function double(x) { return x + x; }
```

This code:

- Defines a new variable: addition
- Creates a new value of type function
- Assigns the newly created function value to the newly defined variable

The name of a function then, is the name of a variable that stores the function. Therefore, the function could be assigned to another variable, and will work as before:

```
function double(x) { return x + x; }
var a = double(4);   // a contains the number 8
var b = double;      // b refers to the same function as double does
var c = b(5);        // c contains the number 10
```

Anonymous Functions

Anonymous functions are functions with no name. The function keyword returns a reference to the function which can then be assigned to a variable, parameterized, invoked, or returned from another function.

The following is a regular function declaration:

```
function fooBar() {
   alert('fooBar');
};
```

The following is the same function declared anonymously:

```
var fooBar = function() {
   alert('fooBar');
};
```

In either case the invocation is the same:

```
fooBar();
```

Passing logic

Anonymous functions can be used to pass code snippets from one function to another:

```
window.addEventListener('load', function() { alert('Complete'); });
```

The second argument passed to the addEventListener function invocation is a function declaration. This is the same as having a reference variable for the function separately defined. Consider the following:

```
function complete() {
   alert('Complete');
}
window.addEventListener('load', complete);
```


Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Mark grader

1. Create a HTML page named grader.html.
2. Add a heading of your choosing.
3. Add a script element just before the closing body tag.
4. Declare a function named grade that accepts a mark. The grade function should return a grade according to the following rules:

Mark	Grade
90+	A
80+	B
70+	C
60+	D
50+	E
<50	F

5. Declare a variable named more and assign it true.
6. Construct a loop that iterates while more is true. Inside the loop:
   a. Prompt the user to enter a mark.
   b. Grade the mark and display the result in an alert.
   c. Prompt the user to specify whether there are more marks to grade.
   d. If there are no more marks to grade, set more to false.

NOTE

In addition to alert and prompt, there exists a built-in function named confirm which displays a popup with OK and Cancel buttons. If the OK button is clicked, the function returns true. If the Cancel button is clicked, the function returns false. You may be able to use this function to improve your solution.

7. Save your changes.
8. Open the file in Chrome and test the code thoroughly.
9. Create a JavaScript file named grader.js.
10. Cut and paste the grade function from the HTML page to the JavaScript file.
11. Embed the grader.js script in the head of your HTML page.
12. Save your changes.
13. Open the file in Chrome and test the code thoroughly.
14. How many potential bugs can you identify in this code?

Chrono quiz (version 2)

1. Save chrono-quiz-1.html as chrono-quiz-2.html.
2. The code that is executed for each case of the switch statement is the same (save for the question and answer). Extract this code into a function and invoke the function in the switch statement.

   Hint: the function should accept a question and the right answer.
3. Save your changes.
4. Open the file in Chrome and test the code thoroughly.

CHAPTER 6

Objects

Object Literal

An **object** is a collection of named values, called properties. The property of an object can be of any data type – string, number, boolean, function, object, array. Objects are used to group related data and functionality together in one entity.

An object literal is an object without a template. More about templates later. An object literal is a comma-separated list of key value pairs within braces. Each property may be a value, variable, or expression:

```
// empty object
var person1 = {};

// each property is assigned a value
var person2 = {
   name: 'Fred',
   age: 25,
   address: {
      line1: '2 New Street',
      city: 'London'
   }
   greet: function() {
      alert('Hello');
   }
};

// each property is assigned a variable
var person3  = {
   name: name,
   age: age,
   address: address,
   greet: greet
};
```

Object Properties

The dot notation is used to access, change, and create object properties.

```
object.property
```

The thing on the left of the dot is a variable that references/points to the object. The thing on the right of the dot is the property name. Consider again the person object literal created previously:

```
var person2 = {
   name: 'Fred',
   age: 25,
   address: {
      line1: '2 New Street',
      city: 'London'
   }
   greet: function() {
      alert('Hello');
   }
};
```

The variable person2 contains a reference to the object. It is used to access the object's properties:

```
person2.name      // 'Fred'
person2.age       // 25
person2.address   // { line1: '2 New Street', city: 'London' }
person2.greet()   // 'Hello' (in an alert)
```

The values of existing properties may be changed using the dot notation:

```
person2.name = 'Fred Harris';
```

And new properties may be added:

```
person2.height = 1.73;
```


Constructor Function

An object literal is okay when you need only represent one person (or whatever it is you're trying to represent). But what if you need to create many people? Consider the following:

```
var person1 = {
   name: 'Karen King',
   age: 31
}
var person2 = {
   age: 35,
   firstName: 'Simon',
   lastName: 'Black'
}
var person3 = {
   first = 'Tabatha',
   last = 'Godwin'
}
```

They're all person objects, but they're not all the same. This may not be a problem unless you want to process all the person objects in, say, a loop. There's no way to write code to process these objects uniformly. What is more, we're having to specify the keys each time we create a new person object.

A **constructor function** is a special function that is used to create objects. It is, effectively, a template, ensuring that each object created has the same set of properties. Consider the following:

```
function Person(firstName, lastName, age) {
   this.firstName = firstName;
   this.lastName = lastName;
   this.age = age;
}
```

Note that the function name begins with a capital letter. This has no effect on the function's behaviour, it is simply a convention. It indicates that this function is to be used to create Person type objects.

The **this** keyword is a reference to the object under construction. In this example we are adding firstName, lastName, and age properties to the object under construction and assigning each property the value of an argument passed to the function.

To create an object using a constructor function, you must invoke it with the **new** keyword as follows:

```
var person1 = new Person('Karen', 'King', 31);
```

Note that we didn't have to specify the property names. Every person object created using the Person constructor function will have firstName, lastName, and age properties.

Prototype Manipulation

You can add function type properties in the constructor function as follows:

```
function Person(firstName, lastName, age) {
   this.firstName = firstName;
   this.lastName = lastName;
   this.age = age;
   this.greet = function() {
      alert('Hello ' + this.firstName);
   }
}
```

Note that to access a property of the object within the function its name must be preceded by the keyword this.

Adding function type properties in this way is, however, bad practice. That's because every object created will, effectively, get a copy of the function code, and that's unnecessary.

The solution is to exploit the constructor function's prototype. Here's where it gets a little weird. Every function is an object. And every function has a property named prototype. The prototype property references an object to which properties may be added. Consider the following:

```
Person.prototype.greet = function() {
   alert('Hello ' + this.firstName);
}
```

So, we've added a function property to the constructor function's prototype. The crucial thing is that every object created using the constructor function has access to the properties of the prototype object. In this example, that means that the greet function code exists in memory only once, and all Person objects have access to it.

Let's put it all together:

```
function Person(firstName, lastName, age) {
   this.firstName = firstName;
   this.lastName = lastName;
   this.age = age;
}
Person.prototype.greet = function() {
   alert('Hello ' + this.firstName);
}
var person1 = new Person('Karen', 'King', 31);
person1.greet(); // 'Hello Karen' (in an alert)
```

If a given function should be a property of all objects of a given type, then it should be added to the constructor function's prototype.

String

A **string** is an array of characters. As such, strings have a length and each character may be accessed by specifying its index. More on arrays later.

string vs. String

A string type thing is not an object, but a String type thing is.

When you declare a variable and assign it a value surrounded in quotes, you create a primitive string (lower case s). It is not an object:

```
var s1 = 'Hello world';
typeof s1 // 'string'
```

There exists a built-in constructor function called String (upper case S) that may be used to create String objects:

```
var s2 = new String('Hello world');
typeof s2 // 'object'
```

String objects have lots of function type properties, e.g.:

```
s2.replace('world', 'JavaScript');
```

However, this is also valid:

```
s1.replace('world', 'JavaScript');
```

But s1 is a primitive string, it does not reference a String object. What's going on?
If you try to invoke a method on a primitive string it is automatically converted to a String object, but only for as long as it takes to execute the method.

Casting

Recall that JavaScript is dynamically typed. That means the data type of a variable may change depending on what is assigned to it. However, the data type of a value does not change. For example, the value 123 is always a number. It cannot be changed.

Casting is the process of creating a new value from an existing one, where the new value is of a different type. Consider the following:

```
var num = 123; // 123 is a number and cannot be changed
var str = String(num); // cast num as a string – str contains '123'
```

When used in this way, the String constructor function creates a string from the argument provided. Note that it is a primitive string that is created, not an object.

Methods

A **method** is a function property of an object. String objects have many methods. Tabled below are some of the more commonly used ones.

Method	Example (var s = 'Hello world';)	
charAt(index): string	s.charAt(6);	// 'w'
endsWith(string): boolean	s.endsWith('rld');	// true
indexOf(string): number	s.indexOf('e');	// 1
replace(string, string): string	s.replace('world', 'JavaScript');	// 'Hello JavaScript'
split(regex): array	s.split(' ');	// ['Hello', 'world']
startsWith(string): boolean	s.startsWith('Goodbye');	// false
substring(number): string	s.substring(6);	// world
toLowerCase(): string	s.toLowerCase();	// 'hello world'
toUpperCase(): string	s.toUpperCase();	// 'HELLO WORLD'

There are many more. For a full list see: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Escape characters

Sometimes you will want to include a character in a string value that has special meaning in JavaScript, e.g. a quotation mark. The escape character \ signals that the next character should be interpreted literally:

```
var quote = 'He said \'I like JavaScript\'';
```

In the example above the backslash signals that the single quotation mark that follows should be interpreted literally, that is, it should form part of the string value.

Escape characters are also used to insert non-printing characters such as the newline and tab characters:

```
var manyLines = 'This is line 1\nThis is line 2\nThis is line 3';
```

Finally, the escape character is used to insert Unicode characters:

```
var greekAlpha = '\u03B1';
```


Number

A **number** is an integer or floating point number.

number vs. Number

A number type thing is not an object, but a Number type thing is.

When you declare a variable and assign it a number, you create a primitive number (lower case s). It is not an object:

```
var n1 = 123.456;
typeof n1 // 'number'
```

There exists a built-in constructor function called Number (upper case N) that may be used to create Number objects:

```
var n2 = new Number(123.456);
typeof n2 // 'object'
```

Number objects have lots of function type properties, e.g.:

```
n2.toFixed(2);
```

However, this is also valid:

```
n1.toFixed(2);
```

But n1 is a primitive number, it does not reference a Number object. What's going on? If you try to invoke a method on a primitive number it is automatically converted to a Number object, but only for as long as it takes to execute the method.

Casting

Recall that JavaScript is dynamically typed. That means the data type of a variable may change depending on what is assigned to it. However, the data type of a value does not change. For example, the value '123' is always a string. It cannot be changed.

Casting is the process of creating a new value from an existing one, where the new value is of a different type. Consider the following:

```
var str = '123'; // '123' is a string and cannot be changed
var num = Number(str); // cast str as a number – num contains 123
```

When used in this way, the Number constructor function creates a number from the argument provided. Note that it is a primitive number that is created, not an object.
If a number cannot be created from the argument provided, the cast will yield the value NaN (Not a Number).

Methods

A **method** is a function property of an object. Number objects have many methods. Tabled below are some of the more commonly used ones.

Method	Example (var n = 123.456;)
toExponential(number): string	n.toExponential(3); // '1.235e+2'
toFixed(number): number	n.toFixed(2); // 123.46
toLocaleString(string): string	n.toLocaleString('fr'); // '123,456'
toPrecision(number): string	n.toPrecision(4); // '123.5'
toString(): string	n.toString(); // '123.456'

There are many more. For a full list see: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

Boolean

A **boolean** is a value that is either true or false.

boolean vs. Boolean

A boolean type thing is not an object, but a Boolean type thing is.

When you declare a variable and assign it either true or false (or an expression that evaluates to true or false), you create a primitive boolean (lower case b). It is not an object:

```
var b1 = true;
typeof b1 // 'boolean'
```

There exists a built-in constructor function called Boolean (upper case B) that may be used to create Boolean objects:

```
var b2 = new Boolean(true);
typeof b2 // 'object'
```

Boolean objects have a few function type properties, e.g.:

```
b2.toString();
```

However, this is also valid:

```
b1.toString();
```

But b1 is a primitive string, it does not reference a Boolean object. What's going on? If you try to invoke a method on a primitive boolean it is automatically converted to a Boolean object, but only for as long as it takes to execute the method.

Casting

Recall that JavaScript is dynamically typed. That means the data type of a variable may change depending on what is assigned to it. However, the data type of a value does not change. For example, the value 123 is always a number. It cannot be changed.

Casting is the process of creating a new value from an existing one, where the new value is of a different type. Consider the following:

```
var num = 123; // 123 is a number and cannot be changed
var boo = Boolean(num); // cast num as a boolean – boo contains true
```

When used in this way, the Boolean constructor function creates a boolean from the argument provided. Note that it is a primitive boolean that is created, not an object.

The rules that determine what values are true and false are as follows:

- Non-empty strings are true
- Empty strings are false
- Non-zero numbers are true
- Zero numbers are false
- Objects are true (even if they're empty)
- null is false
- undefined is false
- NaN is false

Date

The built-in Date constructor function is used to create Date objects:

```
var today = new Date();
```

The variable today contains a reference to a Date object representing today's date.

Whilst there are many ways of creating a Date object that represents a date in the past/future, probably the simplest way is to pass the constructor function arguments representing the year, month, day, and (optionally), hours, minutes, and seconds:

```
var magnaCarta = new Date(1215, 5, 15);
```

But King John affixed his seal to the great charter in June, not May! You're right, of course, but for JavaScript Date objects, January is month 0, not 1.

Methods

A **method** is a function property of an object. Date objects have many methods. Tabled below are some of the more commonly used ones.

Method	Example (var d = new Date(1215, 5, 15);)
getDate(): number	d.getDate(); // 15 (day of month)
getDay(): number	d.getDay(); // 1 (day of week; Sun = 0)
getMonth(): number	d.getMonth(); 5
getYear(): number	d.getYear(); -685 (DEPRECATED - DO NOT USE!)
setDate(number)	d.setDate(1); (changes the day of month to 1)
setMonth(number)	d.setDate(9); (changes the month to October)
setYear(number)	d.setYear(2015); (DEPRECATED - DO NOT USE!)

There are many more. For a full list see: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

Adding and subtracting time from a date is relatively easy. Let's assume we want to know what the date will be 10 days from now:

```
var todayPlus10Days = new Date();
var dayOfMonth = todayPlus10Days.getDate();
todayPlus10Days.setDate(dayOfMonth + 10);
```

The same approach can be applied for adding and subtracting months and years.

There is no easy way to natively calculate the period between two dates. The getTime method returns a number representing the number of milliseconds that have elapsed between 1 January 1970 00:00:00 UTC and the given date. As this is a number, you can subtract one from the other and convert the resultant value back to days/months/years.

Math

The built-in Math object comprises many properties and methods that may be used for performing mathematical calculations. Unlike, String, Number, Boolean, and Date, Math is not a constructor function. It cannot be used to create Math objects.

Data properties

The Math object has many data properties. Tabled below are just a few of them.

Data Property	Value
Math.E	2.718281828459045 (Euler's constant)
Math.PI	3.141592653589793

Methods

A **method** is a function property of an object. The Math object has many methods. Tabled below are just some of them.

Method	Example
Math.abs(any): number	Math.abs(-1); // 1
Math.floor(number): number	Math.floor(3.7); // 3
Math.max(numbers): number	Math.max(3, 1, 6, 2, 4); // 6
Math.min(numbers): number	Math.min(3, 1, 6, 2, 4); // 1
Math.pow(number, number): number	Math.pow(2, 4); // 16
Math.random(): number	Math.random(); // (number between 0 and 1)
Math.round(number): number	Math.round(123.456); // 123

There are many more. For a full list see: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Regular Expressions

A regular expression is a powerful tool for performing pattern matching, especially for the validation of user input. JavaScript's regular expressions follow PERL style syntax.

It is possible to build a regular expression using literal syntax or the RegExp constructor function. For example:

```
var pattern = /^\d{4}$/; // or...
var pattern = new RegExp('^\\d{4}$');
```

Note that the literal expression is surrounded by forward slashes.

As RegExp object may be used to validate data as follows:

```
var input = '1234';
input.match(pattern); // test the string is a match for the pattern
```

More on how to use RegExp objects to validate data later. The following tables represent some of the more common special characters used to construct regular expressions.

Character classes

.	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

Character sets

[abc]	Find any one of a, b, or c
[^abc]	Find any one except a, b, or, c
[0-9]	Find any one digit from 0 to 9
[A-Z]	Find any one uppercase character from A to Z
[a-z]	Find any one lowercase character from a to z
[A-z]	Find any one character from A to z

Quantifiers

n+	At least one n
n*	Zero or more occurrences of n
n?	Zero or one occurrences of n
n{X}	X number of n's
n{X,Y}	At least X and at most Y number of n's
n{X,}	At least X number 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
?=n	Matches any string that is followed by n
?!n	Matches any string that is not followed by n

Modifiers

Placed in a unique position AFTER the trailing forward-slash, modifiers define how the whole expression is applied.

i	Perform case-insensitive matching
g	Perform a global match (finding all matches)
m	Perform multiline matching

Regular expressions in common use

Methods for applying regular expressions fall broadly into two categories.

- Those that are called on a RegExp object and take a string as an argument
- Those that are called on a string and take a RegExp object as an argument

Here are some of the more commonly used ones:

test

The RegExp method, test, tests a regular expression for a match in a given string. It returns true if the string matches the pattern:

```
var str = 'Hello crazy world!';
var pattern = /crazy/g;
var result = pattern.test(str); // true
```

match

The String method, match, looks at the string for matches of a given pattern. With the global modifier it returns an array of matches. Without the global modifier it returns only the first match (or null if there are no matches).

```
var str = 'I can\'t see the sea.';
var pattern = /s?ea?/g;
var result = str.match(pattern); // ['se','e','e','sea']
```

search

The String method, search, looks at the string for matches of a given pattern. It returns the index of the beginning of the first match only (or -1 if there are no matches).

```
var str = 'I can\'t see the sea.';
var pattern = /s?ea?/g;
var result = str.search(pattern); // 8
```

replace

The String method, replace, looks at the string for matches of a given pattern. It returns a new string with each match replaced by the string given as the second argument.

```
var str = 'She sells sea shells';
var pattern = /sea/g;
var result = str.replace(pattern, 'pea'); // 'She sells pea shells';
```

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Bank account (object literal)

1. Create a HTML page named bank-account-literal.html.
2. Add a heading of your choosing.
3. Add a script element just before the closing body tag.
4. Declare a variable named account and assign it an object literal as follows:

Name	Value
name	Davidson
number	6657
balance	198.77

5. Add two function properties to the object, one named deposit, the other named withdraw. Each function should accept an amount and adjust the balance.
6. Add a function property to the object named display. The function should display the account name, number, and balance in an alert with appropriate labels.
7. Deposit £100 into the Davidson account.
8. Withdraw £50 from the Davidson account.
9. Invoke the account's display method.
10. Save your changes.
11. Open the file in Chrome and test the code thoroughly.

Bank account (constructor function)

1. Create a HTML page named bank-account-constructor-function.html.
2. Add a heading of your choosing.
3. Add a script element just before the closing body tag.
4. Declare a constructor function named Account. It should accept a name, number, and balance, and assign each one to a property of the object under construction.
5. Add two function properties to the constructor function's prototype, one named deposit, the other named withdraw. Each function should accept an amount and adjust the balance.
6. Use the Account constructor function to create two Account objects (referenced by variables named a1, and a2), passing arguments as you see fit.
7. Deposit £100 into each account.
8. Add a function property named display to the object referenced by a1. The function should display the account name, number, and balance in an alert with appropriate labels.

9. Is the display method available to both Account objects? Write some code to test your theory.
10. If you now add the display function to the Account constructor function's prototype, will it be available to each of the Account objects created previously? Write some code to test your theory.
11. Save your changes.
12. Open the file in Chrome and test the code thoroughly.

Email validator (version 1)

1. Create a HTML page named email-validator.html-1(first version).
2. Add a heading of your choosing.
3. Add a script element just before the closing body tag.
4. Prompt the user to enter his/her email address.
5. If the email address entered is valid - has the symbols @ and . then…
   a. Assign the index of the @ symbol to a variable named indexOfAt.
   b. Get the domain part of the email address using the substring method and assign it to a variable named domain.
   c. Display an all caps version of the domain in an alert.
6. If the email address entered is invalid, then display an error message in an alert.
7. Save your changes.
8. Open the file in Chrome and test the code thoroughly.

Email validator (version 2)

1. Save email-validator-1.html as email-validator-2.html.
2. Simply testing for the existence of the symbols @ and . is not a satisfactory method of validating an email address. Refactor the code to use a regular expression, instead.
3. Save your changes.
4. Open the file in Chrome and test the code thoroughly.

Transaction date

1. Make the following additions to the code in the file bank-account-constructor-function.html.
2. In the constructor function, add a property named lastTransactionDate and set it to null. You need not change the parameter list.
3. In each of the deposit and withdraw methods, and after the balance has been adjusted, set the lastTransactionDate property to today's date.
4. Update the display method such that the lastTransactionDate is included.
5. If you're feeling lucky, add a method to the constructor function's prototype named getDaysSinceLastTransaction that calculates and returns the number of days since the last transaction.
6. Save your changes.
7. Open the file in Chrome and test the code thoroughly.

CHAPTER 7

Arrays

Array Instantiation

An **array** is a zero-indexed list of values. An array can store various types of things, including other arrays, but it is generally bad practice to store various types of things in any given array – it makes the array very hard to process. An array can grow and shrink.

Array literal

An array literal is a collection of values between square brackets, each value separated by a comma. For example:

```
var oddNumbers = [1, 3, 5, 7, 9];
```

Array constructor function

An array may also be created using the built-in Array constructor function:

```
var oddNumbers = new Array(1, 3, 5, 7, 9);
```

Passing the constructor function a single argument sets the initial length of the array, with each elements undefined:

```
var oddNumbers = new Array(5);
```

Array Elements

An element within an array is accessed by specifying the array name and the index of the element in question between square brackets. This syntax can be used to both read from and write to the array. Consider the following:

```
var names = ['James', 'George', 'Olivia', 'Tom'];
```

To get an element form the array:

```
var secondName = names[1]; // 'George'
```

And to set an element in the array:

```
names[3] = 'Thomas';
```

You can use a variable or an expression to refer to an array element:

```
var position = 2;
var name3 = names[position]; // 'Olivia'
var name4 = names[position + 1]; // 'Thomas'
```

NOTE

Strings are arrays too. Consider the following:

```
var name = 'James';
typeof name; // 'string'
var firstChar = name[0]; // 'J'
```

Array Length

Note that arrays are objects (albeit with special properties). The Array constructor function initializes a length property. The length property is automatically updated whenever necessary and is always one greater than the largest index:

```
var list = new Array(); // list.length = 0

var list = new Array(11); // list.length = 11 (all elements undefined)

var list = new Array(10,20,30); // list.length = 3
```

The length property is writable. Setting the length to a value less than than its current value will truncate the array. Setting the length to a value greater than its current value will make the array larger and new, undefined elements will be added at the end.

JavaScript arrays are sparse; elements do not have to be added in contiguous blocks:

```
var list = new Array();
list[5] = -1; // elements at indexes 0-4 are undefined
```

Array Traversal

There are a number of ways of traversing the elements of an array – traditional for loop, the for of construct (ES6), and the Array prototype method forEach. There also exists a for in construct, but this should not be used for traversing an array. It is intended to traverse the properties of an object, and can result in unexpected behaviour if used to traverse the elements an array.

Traditional for loop

Note that any one of the three loop types – for, while, do while – may be used to traverse the elements of an array, but the for loop is most commonly used. Consider the following (assuming an array named list):

```
for(var i = 0; i < list.length; i++){
   var element = list[i];
   ...
}
```

Note that undefined elements will be included using this approach.

for of (ES6)

The for of construct may be used to traverse any Iterable type thing, of which Array is one. It is similar to for each constructs in other programming languages:

```
for(var element of list) {
   // do something with element
}
```

Note that we need not concern ourselves with the index. This is simpler than the traditional for loop but is only suitable where the index is not needed.

forEach

The forEach method executes the function argument once for each element in the array. It is not invoked for elements that are undefined.

The function that you pass to the forEach method is invoked with three arguments:

- the element **value**
- the element **index**
- the **array** being traversed

Consider the following:

```
function processElement(element, index, array) {
   array[index] = element + 2;
}
var numbers = [1, 2, 3, 4, 5];
numbers.forEach(processElement);
```

The forEach method is passed the processElement function, which is invoked for each element in the array. The processElement function increments each element by 2. The numbers array now contains the numbers 3, 4, 5, 6, and 7.

More often than not, the forEach method is passed an anonymous function. Consider the following version of the code above.

```
var numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(element, index, array) {
   array[index] = element + 2;
});
```

Array Methods

Each array object has many methods. What follows is a description of some of the more commonly used ones. Consider the following:

```
var monarchs = ['William I', 'Rufus', 'Henry I', 'Stephen'];
```

push

The push method adds one or more elements to the end of an array and returns the updated length.

```
monarchs.push('Matilda'); // updated length = 5
```

pop

The pop method removes the last element from the array and returns it.

```
var lastIn = monarchs.pop(); // 'Matilda'
```

find

The find method returns the value of the first element in the array that satisfies the condition specified in the function argument. If there is no match, null is returned.

```
var rufus = monarchs.find(function(element) {
   return element.startsWith('R');
}); // 'Rufus'
```

includes

The includes method determines whether an array contains a certain element, returning true or false as appropriate.

```
var exists = monarchs.includes('Matilda'); // false
```

join

The join method creates and returns a string by concatenating all the elements of the array, separating them with an optionally specified delimiter. With no delimiter specified, a comma is used.

```
var all = monarchs.join(); // 'William I,Rufus,Henry I,Stephen'
```

The join method is effectively the opposite of the String method, split.

reverse

The reverse method reverses the order of the array elements. It does not create a new array with the elements rearranged, rather it rearranges the elements in place.

```
monarchs.reverse(); // ['Stephen', 'Henry I', 'Rufus', 'William I']
```

sort

The sort method sorts the array elements. It does not create a new array with the elements rearranged, rather it rearranges the elements in place. With no arguments, the elements are sorted lexicographically.

```
monarchs.sort(); // ['Henry I', 'Rufus', 'Stephen', 'William I']
```

You must the sort method a function argument to sort the array elements in alternative order. This function will be passed two elements that it should compare. If the first element should appear before the second, then the function should return a number less than zero. If the first element should appear after the second, then the function should return a number greater than zero. And if the two values are equivalent (order is irrelevant), then the function should return zero.

Let's sort the monarchs array by length descending:

```
monarchs.sort(function(element1, element2) {
   if(element1.length > element2.length)
      return -1;
   else if(element1.length < element2.length)
      return 1;
   else
      return 0;
}); // ['William I', 'Henry I', 'Stephen', 'Rufus']
```

slice

The slice method takes a slice out of the array (like cutting a cake) and returns the slice as a new array. The original array is unaffected.

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var slice = numbers.slice(3, 7); // [4, 5, 6, 7]
```

Note that the first argument is the index of the start of the slice, and the second argument is the index + 1 of the end of the slice.

splice

The splice method is used to remove elements from an array. It accepts a start index and a number of elements to remove. It returns the removed element(s).

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var removed = numbers.splice(3, 2); // [4, 5]
// numbers is now [1, 2, 3, 6, 7, 8, 9]
```

Others

There are many, many more Array methods. For a full list see:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Training course

1. Create a HTML page named training-course.html.
2. Add a heading of your choosing.
3. Add a script element just before the closing body tag.
4. Declare a variable named trainees and assign it an array of names (one for each of your co-trainees).
5. Declare a variable named allTrainees and assign it an empty string.
6. Traverse the trainees array using either a traditional for loop, the for of construct, or the forEach method, and append each name to allTrainees along with a newline character \n.
7. Display allTrainees in an alert.
8. Add the trainer's name to the end of the trainees array.
9. Sort the trainees array alphabetically.
10. Remove your name from the trainees array.
11. Assign an empty string to the allTrainees variable.
12. Traverse the trainees array again (using a different approach from that you chose earlier), and append each name to allTrainees along with a newline character \n.
13. Display allTrainees in an alert.
14. Save your changes.
15. Open the file in Chrome and test the code thoroughly.

Chrono quiz (version 3)

1. Save chrono-quiz-2.html as chrono-quiz-3.html.
2. As there is more than one question, they should be elements in an array. Declare and initialise two arrays – one for questions and one for answers.
3. Refactor the for loop (the switch statement is no longer needed).
4. Save your changes.
5. Open the file in Chrome and test the code thoroughly.
6. The questions and answers are stored in separate variables and yet the data is related. How might we improve the code such that questions and answers are stored together?

7.

CHAPTER 8

Errors & Debugging

In-browser Developer Tools

Chrome has an incredibly powerful backstage area – Developer (Dev) Tools – for inspecting, debugging and testing front-end code of all types including HTML, CSS and JavaScript. All the major browsers have this now: here we stick to Chrome as it is in use on about 70% of all platforms worldwide.

Chrome is updated regularly and experimental technology soon becomes available as, for example, the CSS Grid Inspector did in 2017. So it's a good idea to accept all the updates if only for handy new Dev Tools features. You should, of course, test your code on older browser versions if the business case requires backwards-compatibility.

Here is a list of just some of the things you can see in Chrome's Dev Tools:

- Live source code
- Linked files e.g. .js files
- Animations as style changes
- Variables input from the user
- JavaScript functions working
- Mobile simulations
- Cookies and web storage
- Memory usage
- Security certificates
- Console messages

The code you see in Dev Tools underlying the visual render of the page is not necessarily the same as the HTML. It is a product of everything that has an influence on the page: the HTML, the CSS, the JavaScript, the device, etc. and changes with user input.

Access

To access Chrome's Dev Tools:

- Select View | Developer | Developer Tools or…
- Click the More Options button on the Address bar (three dots), then select More Tools | Developer Tools or…
- Press Ctrl + Shift + I (Windows) or Cmd + Alt + I (Mac)

Panels

Chrome's Dev Tools has many panels. What follows is a brief description of each one. Note that the Console and Sources panels are of particular interest here.

Panel	Description
Console	The Console (active by default) is the JavaScript engine's terminal. You can send output to the Console from within your script using the console object (see next section), and you can write code in it directly for the purpose of testing small snippets.
Elements	The Elements panel provides for the inspection of HTML elements. As you mouse over elements in the code they are highlighted in the browser window. This is particularly useful for debugging your page layout.
Sources	The Sources panel is probably the most useful for you at this stage. It provides for the debugging of JavaScript code. Debugging is covered later in this chapter. You can also edit your JavaScript code from here directly.
Network	The Network panel provides information about page load – what files are loaded by the browser and how long it took etc.
Performance	The Performance panel provides for the recording of page load, and the inspecting of the various events that happen during the page's lifecycle.
Memory	The Memory panel provides for the profiling of memory use and is helpful in identifying and correcting memory leaks.
Application	The Application panel provides for the inspection of all resources that are loaded, including images, fonts, stylesheets, cookies, local and session storage, and indexed databases.
Security	The Security panel provides for the debugging of mixed content and certificate problems.

For more information about each the panels, see:
https://developers.google.com/web/tools/chrome-devtools/

Console Object

The **console** object is built-in. It has many methods but the most commonly used one is log. When you invoke console.log in your script the argument passed to it is written to the browser's console/terminal.

```
console.log('This will be written to the console');
```

A web user will not see that which is written to the console and it is worth noting that some error messages, e.g. http status 404, are commonly logged to the console too.

Live pages

You can write JavaScript code directly in the console and execute it on a live page. For example, you can inspect a variable or invoke a function just by typing its name into the console. Consider the following script embedded in a HTML page:

```
<script>
   var person = {
      name: 'Francis',
      age: 21,
      sayHello: function() {
         alert('Hello, I\'m ' + this.name);
      }
   }
   ...
</script>
```

With this page loaded in Chrome, if you were to type person.age into the Console panel, it would yield the number 21. If you were to type person.sayHello() into the Console panel, an alert would appear displaying Hello, I'm Francis.

Offline pages

Placing console.log statements in our own code can help to reassure you that some code does what you expect it to do. For example...

```
<button type="submit" onclick="changeColour()">SIGN UP</button>
```

…will NOT trigger a function named changeColor (note the spelling), so if we were to add a console.log statement as the first line of the function:

```
function changeColor() {
   console.log('changeColor reached');
   ...
}
```

…and we don't see the line 'changeColor reached' in the Console panel we know that the function was never invoked, and that the problem lies in the invoking of the function.

Debugging

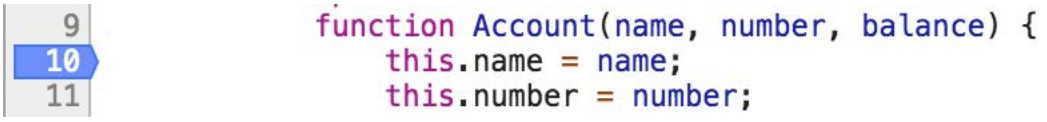
When working on a large codebase, it may be impractical to insert console.log statements. **Debugging** is the process of locating and fixing bugs. Dev Tools includes a Debugger – a tool that helps you to locate bugs. The Debugger enables the execution of code one statement at a time and, crucially, you control when the JavaScript engine moves onto the next statement. This allows you to see what is happening line-by-line (at your own pace).

Setting a breakpoint

The first step is to step a breakpoint. This tells the Debugger where you want to start debugging. It is from this point that each statement will be executed on your say so.

To set a breakpoint:

1. Navigate to the Sources panel.
2. If necessary, open the file you want to debug (Ctrl+P or Cmd+P).
3. Click in the margin (where the line numbers are) adjacent to the line at which you want to start debugging. A blue arrow will appear.



4. When you're done debugging, click the blue arrow again to remove it.

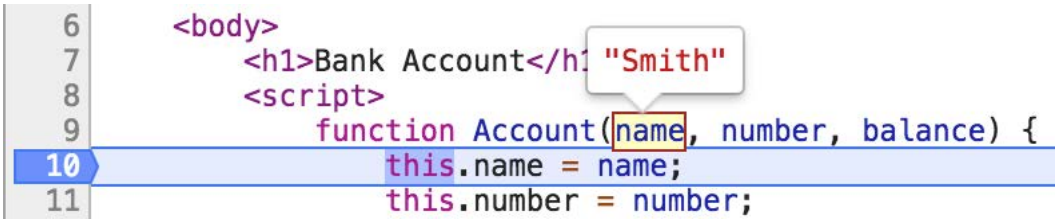
You can set any number of breakpoints as is necessary.

Executing the code

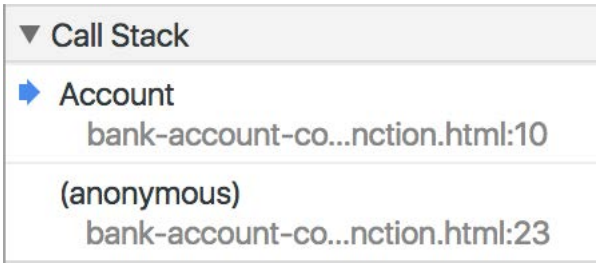
The second step is to execute the code to be debugged. This may mean refreshing the page or clicking a button etc. Do whatever you need to do to have the browser execute the code in question. The Debugger will pause execution when it reaches the breakpoint.

At this point you can inspect the current state of your application. This can be done in a number of ways as follows:

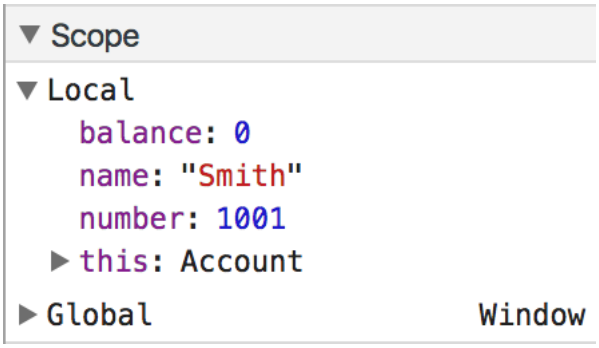
- Mouse over a variable to see its value



- Explore the Call Stack to see which functions are currently in execution

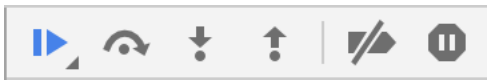


- Explore the Scope to see which variables are currently in Local and or Global scope and what their values are



Stepping over or into

When you're ready to move on, you can step over or step into. Each option is available in the top right corner. Step over is the second button from the left, step into is the third:



Both step over and step into execute the current statement. However, if the current statement involves the invoking of a function, step over will result in the execution of all statements within the function and pause again at the next statement. Step into, conversely, will step into the function and pause at the first statement within.

Whether you step over or step into will depend on the circumstances. If the function in question is a built-in JavaScript function, you will probably step over. It's unlikely that function is the source of the problem. If, however, the function is one of yours, then you may want to step into it, unless you know it to be bug-free.

Resuming execution

When you've located the bug or are otherwise done with debugging, you will want to resume execution. This effectively stops the debugger until the next breakpoint is reached. Clicking the first button in the top right corner resumes execution.

Throw, Try, and Catch

When something goes wrong, the JavaScript engine generates an error object which contains information about the problem. If an error is generated and nothing is done about it, it will effectively crash your script. That is, execution will stop. Consider the following code:

```
var x = 1;
var result = x * y; // ReferenceError
alert(result); // this line (and all that follow) are not executed
```

Assuming y has not been declared, the second line above will generate a ReferenceError and execution of the script will stop.

JavaScript, like most modern languages, provides a mechanism for handling errors so that, in the event of an error, you can do something about it and execution of your script may continue.

NOTE

Error handling should only be used for errors that are beyond your control. The example above does not require error handling, it requires that we fix the code – declare the variable y properly.

try catch

The try and catch blocks allow for the catching and handling of an error. The try block contains the code to be executed, and the catch block is executed if an error is generated:

```
try  {
   var result = x * y;
   ...
} catch(error)  {
   alert('An error occurred: ' + error);
}
```

If any statement in the try block results in the generation of an error, execution moves immediately to the catch block, skipping subsequent lines in the try block.

If no errors are generated in the try block, the catch block is skipped.

The catch block is passed the error object. The name error, in the example above, is the name given to the variable that references the error object. It need not be named error.

Crucially, if an error is generated in the try block, the code in the catch block is executed, and then execution continues with the statements that follow.

finally

The optional finally block is executed whether or not an error is generated. Consider the following:

```
try  {
   var result = x * y;
   ...
} catch(error)  {
   alert('An error occurred: ' + error);
} finally {
   // code here is executed no matter what happens above
}
```

throw

Sometimes, you may want to generate your own, business specific errors. This is necessary when your code may encounter a problem that must be dealt with elsewhere. Consider the following:

```
function processInput(input) {
   if(typeof input != 'number') {
      // I can't do my work if the input is not a number
   } else {
      // Do my work
   }
}
```

In this case, the solution to the problem lies elsewhere – whereever the data was entered. The throw keyword allows you to generate an error. Consider the following:

```
function processInput(input) {
   if(typeof input != 'number') {
      throw new Error('Invalid input: not a number');
   } else {
      // Do my work
   }
}
```

If the input is not a number, we create a new Error object and throw it to the caller. The caller can do one of two things:

1. Throw the error on
2. Handle it (catch it and deal with it)

Which of these it does depends on its ability to deal with the error. If the error is not caught it will end up being thrown to the JavaScript engine and execution of your script will stop.

CHAPTER 9

DOM

HTML and JavaScript

The elements of a HTML page are related to one another. Consider the following:

```
<html>
   <head>
      <title>My HTML Page</title>
   </head>
   <body>
      <h1>My HTML Page</h1>
      <p>Welcome to my page...</p>
   </body>
</html>
```

The head and body elements are children of the html element. They are siblings. The heading element is a child of the body element and a grandchild of the html element. We indent our HTML code (or at least we should) to reflect these relationships.

In the same way, the properties of a JavaScript object are related to one another. Consider the following:

```
var person = {
   name: 'Irene',
   age: 41,
   address: {
      street: 'Main street',
      city: 'Birmingham'
   }
}
```

The name, age, and address properties are children of the person object. They are siblings. The street and city properties are children of the address object and grandchildren of the person object. They, too, are siblings.

Each HTML element is, effectively, a JavaScript object and can be manipulated like any other. Look again at the HTML example at the top of this page. The head and body elements are properties of an object named document. The heading and paragraph elements are properties of the body object. Consider the following:

```
var document = {
   head: {
      title: { ... }
   },
   body: {
      h1: { ... },
      p: { ... }
   }
}
```

Note that in reality the tag name is (in most cases) not the property name.

The **DOM** (Document Object Model) is a built-in JavaScript object that represents all elements in a given HTML page. Each element object has properties that can be manipulated including the content of the element and its attributes (including style). You can also add elements to and remove elements from the DOM. In fact, if you were so inclined, you could construct an entire HTML page in JavaScript. This is not as silly as it may sound. Many modern JavaScript frameworks do just that.

Element Referencing

To manipulate the DOM (and consequently the HTML page) we must obtain a reference to the HTML element(s) we want to change. There are a number of ways of doing this. The document object is key. It has methods that we can use to obtain said references without having an intimate knowledge of the HTML structure.

By tag name

The getElementsByTagName method of the document object accepts a string tag name, and returns an array of element objects:

```
var paragraphs = document.getElementsByTagName('p');
```

The paragraphs variable references an array of paragraph objects. If there are no paragraph elements in the page the method will return an empty array.

By class

The getElementsByClassName method of the document object accepts a string class name, and returns an array of element objects:

```
var quotes = document.getElementByClassName('quote');
```

The quotes variable references an array of elements. Each element in the array is a member of the quotes class. If there are no elements in the page assigned to the quotes class, method will return an empty array.

By ID

The getElementById method of the document object accepts a string id, and returns only one element object. IDs should be unique in the page, but if there exists more than one element with the same ID, then this method will return a reference to the first one.

```
var basket = document.getElementById('basket');
```

The basket variable references an element object or null if the ID does not exist.

By selector

The querySelector method of the document object accepts a string CSS selector, and returns an array or element object.

```
var basket = document.querySelector('#basket');
var quotes = document.querySelector('.quote');
```

The querySelector method is more powerful than those described above, and is probably the most commonly used method for element referencing today.

Common Element Properties

Element objects have lots of data properties and methods. Grab a reference to any HTML element in the Console to see for yourself. There are, however, a small number of common element object properties you should be familiar with. The examples that follow assume the following:

```
<style>
   #basket {
      background-color: blue;
      color: white;
   }
   .blackOnWhite {
      background-color: white;
      color: black;
   }
</style>
...
<div id="basket">Your basket</div>
...
<script>
   var basket = document.querySelector('#basket);
   ...
</script>
```

innerHTML

The innerHTML data property references the element content:

```
var inner = basket.innerHTML; // 'Your basket'
basket.innerHTML = 'My basket';
```

outerHTML

The outerHTML data property references the entire element:

```
var outer = basket.outerHTML; // '<div id="basket">My basket</div>'
```

Whilst it's possible to add/remove/change completely an element by writing to the outerHTML property, it is messy, and there are better ways. See the section on DOM Manipulation.

hasAttribute

The hasAttribute method is used to test for the presence of a given attribute. It accepts an attribute name:

```
var hasName = basket.hasAttribute('name'); // false
```


setAttribute

The setAttribute method is used to set an attribute on an element. It accepts two arguments – the name and value of the attribute to be set:

```
basket.setAttribute('name', 'basket');
```

getAttribute

The getAttribute method is used to get the value of a given attribute. It accepts an attribute name:

```
var name = basket.getAttribute('name'); // 'basket'
```

removeAttribtue

The removeAttribute method is used to remove a given attribute. It accepts an attribute name:

```
basket.removeAttribute('name');
```

style

The style data property references the element's style object. It has many properties, and not only those styles you have explicitly set, but those that are inherited too.

The syntax for referencing style properties in JavaScript differs from CSS:

- Names are camel-case, not hyphenated
- Values are surrounded by quotes
- The equals symbol is used to assign a value to a name, not the colon

For example:

CSS	JavaScript
font-size: 2em;	fontSize = '2em';
background-color: #CCC;	backgroundColor = '#CCC';

Consider the following:

```
var backgroundColor = basket.style.backgroundColor; // 'blue'
basket.style.backgroundColor = 'white';
```

classList

The classList data property references an array of classes assigned to the element. What is more, the array object has a suite of methods for manipulating the array:

Method	Description
add(class…)	Adds one or more classes to the element
remove(class…)	Removes one or more classes from the element
toggle(class)	If the class is set then remove it, else add it
contains(class)	Returns true if the class is assigned, else it returns false
replace(class1, class2)	Replace class1 with class2

Consider the following:

```
var isBlackOnWhite = basket.contains('blackOnWhite'); // false
basket.classList.add('blackOnWhite');
```

Others

There are many more element properties available to you. For a full list see: https://developer.mozilla.org/en-US/docs/Web/API/Element and https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement

DOM Navigation

In addition to those properties described in the previous section, each element object has properties that enable you to navigate the DOM with little to no prior knowledge of the structure of the page. The examples that follow assume the following:

```
...
<body>
   <h1 id='heading'>My Web Page</h1>
   <p id='welcome'>Welcome to my web page...</p>
   <ul id='menu'>
      <li>Home</li>
      <li>About</li>
      <li>Contact</li>
   </ul>
</body>
```

children

The children data property references an array of child elements:

```
var menu = document.querySelector('#menu');
var menuItems = menu.children;
console.log(menuItems.length); // 3
for(var menuItem of menuItems) {
   console.log(menuItem.innerHTML);
}
```

Note that the for loop above yields the following output to the console:

```
> Home
> About
> Contact
```

NOTE

Each element object has a similar property named childNodes. It, however, includes references to text (whitespace) and comment nodes which you're probably not interested in.

firstElementChild

The firstElementChild data property references the first child element:

```
var first = document.body.firstElementChild;
console.log(first.tagName); // 'H1' (each element has a tagName prop)
```

lastElementChild

The lastElementChild data property reference the last child element:

```
var last = document.body.lastElementChild;
console.log(last.tagName); // 'UL'
```

nextElementSibling

The nextElementSibling data property references the next sibling element:

```
var next = document.querySelector('#heading').nextElementSibling;
console.log(next.tagName); // 'P'
```

previousElementSibling

The previousElementSibling data property references the previous sibling element:

```
var prev = document.querySelector('#welcome').previousElementSibling;
console.log(prev.tagName); // 'H1'
```

parentElement

The parentElement data property reference the parent element:

```
var listParent = document.querySelector('#menu').parentElement;
console.log(listParent.tagName); // 'BODY'
```

DOM Manipulation

So far we have described how to reference, modify, and navigate existing HTML elements on the page. But you can create, append, insert, and remove elements too. The examples that follow assume the following:

```
...
<body>
   <h1 id='heading'>My Web Page</h1>
   <p id='welcome'>Welcome to my web page...</p>
   <ul id='menu'>
      <li>Home</li>
      <li>About</li>
      <li>Contact</li>
   </ul>
</body>
```

createElement

The createElement method of the document object is used to create an element. It accepts a string tag name and returns a reference to the newly created element object:

```
var div = document.createElement('div');
```

Note that creating an element in this way does not automatically add it to the DOM. The element exists only in memory until you explicitly append it to, or insert it into the DOM.

appendChild

Each element object has an appendChild method that may be used to append an element to its set of children, that is, the appended element becomes the last child. It accepts an element object:

```
document.body.appendChild(div);
document.querySelector('#menu').nextElementSibling.tagName; // 'DIV'
```

insertBefore

Each element object has an insertBefore method that may be used to insert an element in amongst its children. It accepts two arguments – the element object to be inserted, and a reference to the child element it is to be inserted before:

```
var productMenuItem = document.createElement('li');
productMenuItem.innerHTML = 'Products';
var menu = document.querySelector('#menu');
menu.insertBefore(productMenuItem, menu.children[1]);
```

The Product list item is inserted before the About list item.

removeChild

Each element object has a removeChild method that may be used to remove an element from its set of children. It accepts a reference to the child element to be removed:

```
var menu = document.querySelector('#menu');
menu.removeChild(menu.children[3]);
```

The Contact list item is removed from the list.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

DOM

1. Create a HTML page named dom.html.
2. Copy and paste the following code:

```
<!DOCTYPE html>
<html>
   <head>
      <title>DOM</title>
      <style>
         .highlight {
            background-color: yellow;
            font-weight: bold;
         }
      </style>
   </head>
   <body>
      <h1>DOM</h1>
      <p>All HTML elements are JavaScript objects.</p>
      <p>The <span>document</span> object is the root of the DOM.</p>
      <p>document methods for element referencing:</p>
      <ul id="documentMethodList">
         <li>getElementsByTagName</li>
         <li>getElementsByClassName</li>
         <li>getElementById</li>
         <li>querySelector</li>
      </ul>
      <table id="elementPropertyList">
         <tr>
            <td>PROPERTY</td>
            <td>DESCRIPTION</td>
         </tr>
         <tr>
            <td>innerHTML</td>
            <td>Element content</td>
         </tr>
         <tr>
            <td>style</td>
            <td>The element's style object</td>
         </tr>
      </table>
   </body>
</html>
```

3. Add a script element just before the closing body tag.
4. Obtain a reference to the heading element and set its innerHTML to *Document Object Model (DOM)*.
5. Obtain a reference to the body element and change its font to *Helvetica*.
6. Obtain a reference to the span element in the second paragraph and assign it the *highlight* class.
7. Obtain a reference to the unordered list and:
   a. set its *type* attribute with a value of *square*, and…
   b. display the innerHTML of its third child in an alert.
8. Declare a variable named tableRow and assign it a new table row element.
9. Declare a variable named propertyCell and assign it a new table cell element.
10. Declare a variable named descriptionCell and assign it a new table cell element.
11. Set the innerHTML of the propertyCell to *classList*.
12. Set the innerHTML of the descriptionCell to *Array of classes assigned to the element*.
13. Append the propertyCell and descriptionCell to the tableRow.
14. Obtain a reference to the table and append to it the tableRow.
15. Obtain a reference to the array of table cells and set each one with 10px of right padding.
16. Save your changes.
17. Open the file in Chrome and test the code thoroughly.

CHAPTER 10

Event Handling

Events

As the user navigates around your site and interacts with it, the browser generates events. For example, when the user moves the mouse from one part of the page to another, a mouse event is generated. When the user clicks a button, a click event is generated. When the user taps a touch screen, a touch event is generated, and so on. For a full list of event types see: https://developer.mozilla.org/en-US/docs/Web/Events

An event is a JavaScript object that contains information about the thing that just happened. If we don't write code to handle it, the event is lost. An event is associated with an HTML element – the button that was clicked, for example. Events can be set to either propagate upwards through the DOM (bubble up), or not.

Crucially, you can write JavaScript code that is executed in response to the generation of an event – an event handler. An event handler is a JavaScript function that must be associated with the HTML element from which you expect events to be generated. The handler can be assigned to the element using an HTML attribute, e.g.:

```
<button onclick="myHandler(event)">...
```

But this means mixing your HTML and JavaScript code, and that can make code hard to read and maintain. It is generally considered best practice to separate your HTML and JavaScript code as much as possible. The alternative to the example above is to obtain a reference to the element and assign the handler in your script.

Event Object

Event objects have many properties. Tabled below are some of those that may be of use:

Property	Description
bubbles	true if the event propagates upwards through the DOM, else false
target	a reference to the element that generated the event
type	string event type e.g. 'click'
x	x coordinate of the event
y	y coordinate of the event
preventDefault()	Instructs the browser not to respond in the normal way
stopPropagation()	Prevents the event propagating upwards through the DOM

There are many more. For a full list see: https://developer.mozilla.org/en-US/docs/Web/API/Event

Event Handler

An event handler is a function that will be invoked when a given event is generated. Provided it's assigned correctly, the browser will invoke the handler and pass it the event object. Consider the following example handler:

```
function eventHandler(event) {
   alert('A ' + event.type + ' event was generated by '
      + event.target);
}
```

Note that the parameter named event could be named anything. It is simply a reference to the event object that was generated. Indeed, if you don't need the event object, you need not specify it as a parameter at all.

Once we've created the handler, we must assign it to the appropriate element.

Event Handler Assignment

An event handler may be assigned to an element using an HTML attribute, or by obtaining a reference to the element and assigning the handler in your script. The examples that follow assume the following:

```
function eventHandler(event) {
   alert('A ' + event.type + ' event was generated by '
      + event.target);
}
```

HTML attribute

There exists a large number of HTML attributes whose names begin with the word on, one for each event type, in fact. And these may be used to assign event handling code to the element. Consider the following:

```
<div onmouseover="eventHandler(event)">...
```

We are specifying the JavaScript code that is to be executed when the mouseover event is generated. Note that when assigning a handler in this way we have to invoke the function within the quotation marks.

Technically, you don't have to invoke a function. You can insert any valid JavaScript code between the quotation marks:

```
<div onmouseover="for(var i = 0; i < 10; i++) console.log(i);">...
```

But, as you can imagine, this can get very messy indeed.

addEventListener

Each element object has an addEventListener method that may be used to assign a handler for a given event type. It accepts three arguments:

- The **type** of event (a string)
- A reference to the **handler** function
- An optional object of **options**

You should set the options argument to **false** as a fall-back for older browsers that only accept a boolean, while it may be set to an object of options for newer browsers. More information is available on the MDN. Consider the following:

```
<div id="main">...</div>
...
<script>
   var mainDiv = document.querySelector('#main');
   mainDiv.addEventListener('mouseover', eventHandler);
</script>
```


Here again we are specifying the JavaScript code that is to be executed when the mouseover event is generated, but the HTML is separated from the JavaScript code.

NOTE

The second parameter is a reference to the eventHandler function and must not include parentheses. Doing so will cause it to be executed on assignment.

The addEventListener method may be passed an anonymous handler function. Consider the following:

```
<div id="main">...</div>
...
<script>
   var mainDiv = document.querySelector('#main');
   mainDiv.addEventListener('mouseover', function(event) {
      alert('A ' + event.type + ' event was generated by '
         + event.target);
   });
</script>
```

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Accordion

1. Create a HTML page named accordion.html.
2. Add a heading of your choosing.
3. Add two buttons – one labelled Expand and with a matching id, the other labelled Collapse and with a matching id.
4. Style the Expand button to be hidden.
5. Add a paragraph of Lorem Ipsum text (https://www.lipsum.com/).
6. Add a script element just before the closing body tag.
7. Declare a variable named expandButton and assign it a reference to the Expand button element.
8. Declare a variable named collapseButton and assign it a reference to the Collapse button element.
9. Declare a variable named paragraph and assign is a reference to the paragraph element.
10. Assign a handler to expandButton to respond to click events. It should:
    a. set the paragraph's display property to block;
    b. set the Collapse button's display property to inline, and;
    c. set its own display property to none.
11. Assign a handler to collapseButton to respond to click events. It should:
    a. set the paragraph's display property to none;
    b. set the Expand button's display property to inline, and;
    c. set its own display property to none.
12. Save your changes.
13. Open the file in Chrome and test the code thoroughly.

Chrono quiz (version 4)

1. Save chrono-quiz-3.html as chrono-quiz-4.html.
2. Make changes to the app as follows:
    a. The questions should appear in the page (not in a prompt). Whether you choose to show all questions at once, or one after the other is up to you, but the latter is preferred.
    b. The user should type his/her answer to each question in a text box.
    c. The user should be able to click a button to check his/her answer.
    d. The feedback should appear in the page (not in an alert).
    e. Try to avoid repeating code. Use functions instead.
    f. Hint: it might be helpful to declare a function that renders a question on the page, another that checks the answer and renders feedback, and another that cleans up ready for the next question.

CHAPTER 11

The Window Object

Window Object

The document object with which we have been working is a child of the window object. To extend our house build simile, the HTML, CSS and JavaScript are all vital parts of the house, interacting with each other. To inspect the window object is like stepping back and looking at where the house is in relation to the street, with the ability to zoom back in to inspect any level of detail we want.

The window object is the parent object of everything else. That is, everything else in the DOM is a property of the window object or of one of its descendants.

Every time you declare a variable or a function, you are effectively adding a property to the window object. Consider the following:

```
var name = 'Tom';
function sayHello() {
   alert('Hello , I\'m ' + name);
}
```

The variable named name and the function named sayHello are properties of the window object, even though you did not specify them as such explicitly. Consider the following:

```
console.log(window.name); // 'Tom'
window.sayHello(); // 'Hello I'm Tom' (in an alert)
```

The window object is often referred to as the global object. This is because when we say that a variable or function is global or globally scoped, that means that it is a property of the window object. Note too that JavaScript code can now be executed outside of the browser where there is no window to speak of.

Window properties

The window object has hundreds of data properties and methods. Listed below is a small sample, some of which you've encountered already:

- `alert()`
- `prompt()`
- `console`
- `document`
- `cookie`
- `history`
- `location`
- `navigator`
- `screen`
- `setInterval()`
- `setTimeout()`
- `XMLHttpRequest()`

For a full list, enter `window` into the Console and expand the output - you may want to be sitting down. There are a lot!

Screen, History, and Navigator

Screen

The screen property of the window object references the screen object. It contains information about the screen on which the window is rendered.

Screen object property	Description
width	Width of the screen
height	Height of the screen
availWidth	Width of the screen excl. e.g. sidebars
availHeight	Height of the screen excl. e.g. taskbar
colorDepth	Depth of the color palette
pixelDepth	Color resolution in bits per pixel
orientation	Object containing orientation information

The screen object is key to responsive web design.

History

Moving backward and forward through the user's browsing history is done using methods of the history object, referenced by the history property of the window object:

```
history.back();      // same as pressing the back button
history.forward();   // same as pressing the forward button
```

The go method loads a specific page from session history, identified by its relative position to the current page (the current page is index 0):

```
history.go(-2);      // two pages back
history.go(1);       // same as pressing the forward button
```

The length property contains the length of the history list:

```
history.length;
```

The methods pushState and replaceState allow you to add and modify history entries. These methods work in conjunction with the onpopstate event. Their implementation is beyond the scope of this course but for dynamically-generated pages it can be useful to modify the browser history.

Navigator

The navigator property of the window object references the navigator object. It contains information about the state and identity of the user agent (the browser, computer, phone, tablet, screen reader etc.).

Navigator object property	Description
cookieEnabled	true if cookies are enabled, else false
geolocation	Object with methods for obtaining position information
language	The language of the browser
onLine	true if the browser is online, else false
platform	The platform on which the browser is running
product	The JavaScript engine
userAgent	The user agent header sent by the browser to the server

NOTE

The navigator object has an appName property too but it should not be used to perform browser detection. Use the userAgent property instead.

Alerts and Prompts

The alert, confirm, and prompt functions are properties of the window object. Crucially, they are modal – they remain active and focused until dismissed.

alert

The alert method displays a modal dialog with an OK button. It accepts a string message:

```
alert('Hello World!');
```

confirm

The confirm method displays a modal dialog with OK and Cancel buttons. It accepts a string message:

```
var result = confirm('Do you want to continue?');
```

If the user clicks OK, the confirm method returns true; if the user clicks Cancel, the method returns false.

prompt

The prompt method displays a modal dialog with a text field and OK and Cancel buttons. It accepts two arguments:

- a string message prompting the user to enter some value
- an optional string default value for display in the text field

If the user clicks OK, the prompt method returns whatever the user entered into the text field as a string; if the user clicks Cancel, the method returns null.

Timeouts and Intervals

setTimeout

The window object's setTimeout method allows for the execution of some code after a period of time. It accepts two arguments:

- a reference to the function to be executed
- the time to wait (in milliseconds) before executing the function argument

It returns an integer ID which may be passed to the clearTimeout method to cancel the function before it is invoked.

```
function sayHello() {
   alert('Hello!');
}
var id = setTimeout(sayHello, 5000);
```

The sayHello function will be invoked after AT LEAST five seconds. It may be longer than five seconds but never less. The invocation of the sayHello function might be cancelled if the following statement is executed before the time to wait has elapsed:

```
clearTimeout(id);
```

NOTE

Invoking setTimeout does not block your script. The statements that follow will be executed immediately after the call to setTimeout.

setInterval

The window object's setInterval method allows for the execution of some code repeatedly at a specified interval. It accepts two arguments:

- a reference to the function to be executed
- the interval (in milliseconds) at which the function is to be executed

It returns an integer ID which may be passed to the clearInterval method to cancel the function before it is next invoked.

```
function changeBanner() {
   // TODO
}
var id = setInterval(changeBanner, 5000);
```

The changeBanner function will be invoked approximately every five seconds. The invocation of the changeBanner function might be cancelled if the following statement is executed:

```
clearInterval(id);
```

NOTE

Invoking setInterval does not block your script. The statements that follow will be executed immediately after the call to setInterval.

Cookies

A cookie is a key value pair stored locally by the browser in association with a particular page/site. They may only be accessed, changed, and removed by the page/site that created them.

In JavaScript code, cookies are written to and read from the cookie property of the document object. Support for cookies has existed since the early web, but setting and getting them is a tedious, fiddly, low level affair.

Uses for cookies include authentication, the saving of preferences, and to maintain state between product selection, shopping cart and checkout pages in e-commerce.

Note that the user can choose to allow, disallow, optionally store from certain domains, as well as delete cookies on demand.

There are various types of cookies:

- Session cookies are those that exist for the duration of a user's visit to a site; they are deleted when the session expires.
- Persistent (tracking) cookies are those with a Max-Age attribute that persist between sessions. They are written to disk by the browser.
- Secure cookies are those subject to encryption.

No more than 20 cookies may be stored per site with a total size of approximately 4KB.

Consent

In 2011, EU law changed from requiring notification and opt-out, to notification and consent for cookies. This led to the General Data Protection Law or GDPR, legislating over a wider audience and platform combination.

In short, when cookies can identify an individual person through their device, they are considered to be personal data and the following measures must be in place.

- Consent must be of the opt-in sort through affirmative action by the user
- Opt-out must also be offered: it should be as easy to opt-out as it is to opt-in

Cookie policy should be taken seriously on a production site as breach of compliancy can result in a fine of up to €10 million or 2% of annual turnover, whichever the higher: https://www.theregister.co.uk/2017/03/01/planned_cookie_law_update_expert/ However the legality of cookie consent is hotly debated at government level and the lobby now wants to minimise user interference and distraction. Future changes in legislation may be specific to UK site: https://www.itgovernance.eu/blog/en/how-the-gdpr-affects-cookie-policies

Writing cookies

Writing cookies is straightforward. Consider the following:

```
document.cookie = 'name=George Johnson';
```

Neither the name nor the value should include white space, commas or semi-colons. There are two built-in (window) methods you can use to overcome this: escape and unescape. The escape method accepts a string and converts it into valid ASCII format. The unescape method does the reverse.

```
document.cookie = 'name=' + escape('George Johnson');
```

The value of document.cookie is now:

```
> 'name=George%20Johnson'
```

The data type of document.cookie is string. Writing another cookie has the effect of appending it to the cookie string. Consider the following:

```
document.cookie = 'age=52';
```

The value of document.cookie is now:

```
> 'name=George%20Johnson; age=52'
```

This makes reading a specific cookie tricky.

Reading cookies

We now know that document.cookie is a string, with each key value pair separated by a semicolon. Consider the following code for extracting and printing each cookie:

```
var pairs = document.cookie.split(/;\s/);
for(var pair of pairs) {
   var parts = pair.split(/=/);
   var key = unescape(parts[0]);
   var value = unescape(parts[1]);
   console.log(key + '=' + value);
}
```

We initially split document.cookie using a RegExp object representing a semicolon and a space. We then traverse the pairs array and split each pair using a RegExp object representing the equals symbol. Note that we also use unescape just in case the cookie string was escaped when it was written.

Setting an expiry date

The cookies we've created so far have been session type. They exist in memory only for as long as the browser is running.

To create a persistent cookie we must write it with an expiry date. The date you specify must be in GMT format, e.g.: Thu, 20-Mar-1975 22:30:00 GMT. Fortunately, Date objects have a toGMTString method.

So, to write a cookie with an expiry date of today + 7 days:

```
var today = new Date();
var expiryDate = new Date(today.setDate(today.getDate() + 7));
var expiryDateString = expiryDate.toGMTString();
document.cookie = 'email=george@me.com;expires=' + expiryDateString;
```

The cookie will be deleted from the local filesystem after seven days (if the user doesn't delete it manually beforehand).

NOTE

The expiry date string MUST NOT be escaped.

Setting a path and domain

By default, a cookie is associated with the web page that created it and any other pages in the same directory. For example, a cookie set by www.test.com/customers/details.html, is not accessible by www.test.com/index.html. To achieve this, we have to set the path of the cookie:

```
document.cookie='theme=black;path=/';
```

This has set the effect of setting the cookie's path to the app's root directory, so that any of the site's pages will be able to access it.

Some websites have lots of sub-domains. If a web page on sport.test.com writes a cookie, only pages on sport.test.com will be able to access it. If we want all sub-domains of test.com to be able to access the cookie, we have to set the domain of the cookie:

```
document.cookie='theme=black;path=/;domain=test.com';
```


CHAPTER 12

AJAX

The Request and Response Model

HTTP (HyperText Transfer Protocol) is the set of rules that govern communication between the browser (client) and the host (server). When you enter a URL into the address bar of the browser it sends a HTTP request to the server. The server then sends a HTTP response back to the client.

HTTP request and response

A HTTP request consists of:

- a request line e.g. GET /app/main.html
- headers e.g. User-Agent: Mozilla…
- an empty line
- an optional message body

A HTTP response consists of:

- a status code e.g. 200 OK
- headers e.g. Content-Type: text/html
- an empty line
- an optional message body e.g. the content of a HTML file

HTTP methods

HTTP specifies a set of methods (keywords) that indicate to the server what action is to be carried out for the given resource. These methods are tabled below.

HTTP Method	Description
GET	Get the specified resource
POST	Submit data to be processed by the specified resource
PUT	Replace the specified resource with the content of the message body
DELETE	Delete the specified resource

There are other HTTP methods, but these are the principle ones. Note that most publicly accessible web servers are configured to block PUT and DELETE requests.

You will have encountered GET and PUT as the values you can set on a form's method attribute. You may also know that it is possible to submit data to a resource by appending data to a GET request header, although it is better to do so using POST. Why? See the table overleaf for an answer…

GET Request	POST Request
Can be cached	Cannot be cached
Remains in browser history	Does not remain in browser history
Can be bookmarked	Cannot be bookmarked
Has length restrictions	Does not have length restrictions
ASCII characters only	Binary data allowed
Should not be used to send sensitive data	
Should only be used to retrieve data	

HTTP status codes

HTTP specifies a set status codes, one of which is issued by the server in response to the browser request. The status code groups are tabled below.

Status Code	Type	Description
1XX	Informational	the request was received and understood
2XX	Success	the request was accepted
3XX	Redirection	the client must take action to complete the request
4XX	Client error	the error is the result of a client error
5XX	Server error	the error is the result of a server error

MIME types

MIME (Multipurpose Internet Mail Extensions) type is a standard way to indicate the type of a document. Most browsers use the MIME type (Content-Type) specified in the header to determine how it will process it. MIME type structure follows the format:

```
type/subtype
```

Tabled below are the MIME types that you are most likely to encounter.

MIME Type	Description
text/plain	Plain text files
text/html	HTML content
text/css	CSS file (must be served with this type or may be ignored)
image/...	gif, jpeg, png, svg+xml are considered web safe
audio/… & video/…	Most common ones: wave, webm, ogg
application/octet-stream	Binary data (treated as an attachment)
multipart/form-data	Content of a completed form

AJAX

What does all this have to do with AJAX? And what is AJAX? First things first. When the browser receives a response comprising a html page from the server, the entire page must be rendered regardless of the differences between the current page and the previous one, and at the very least, the user will notice a flicker.

AJAX (Asynchronous JavaScript and XML) is the mechanism by which we can make an asynchronous request to the server in code. This means that we can discretely request small amounts of new content in the background and use it to update the DOM without having to render an entire page.

AJAX is not a thing so much as it is a way of working. The XMLHttpRequest constructor function (a property of the window object) provides for the creation of request objects, and each request object has methods that enable the sending of a request, and the facilities to process the response.

Making the Request

To make an HTTP request in code, we must:

1. Create an XMLHttpRequest (XHR) object
2. Initialise the request
3. Send the request

When the response is received, the XHR object will contain the requested data (assuming the request was successful).

Creating the XHR object

Creating the XHR object is simple:

```
var request = new XMLHttpRequest();
```

Note that the constructor function does not accept any arguments.

Initialising the request

To initialise the request is to specify:

- The HTTP method
- The URL of the resource
- Whether the request should be made asynchronously or not (optional)
- Username (optional)
- Password (optional)

The XHR object's open method is used to initialise the request:

```
request.open('GET', '/app/data/products1.csv');
```

The third argument indicates whether the request should by asynchronous (true) or not (false) and is true by default.

NOTE

All AJAX requests should be asynchronous. Indeed, support for synchronous requests is being phased out by all browser vendors. It's not called ASYNCHRONOUS JavaScript and XML for nothing.

Sending the request

The XHR object's send method is used to send the request. It accepts an optional message body:

```
request.send();
```


Obtaining the Response

Having made the request, we must wait for a response. We know that the response will ultimately form part of the XHR object, but we cannot know WHEN that will happen. Indeed, it may never happen if the request is unsuccessful. This uncertainty is managed by the generating of events.

The XHR object will generate an event if and when the response is loaded. We can therefore assign a handler to the XHR object to be invoked on load:

```
// DO THIS BEFORE INITIALISING AND SENDING THE REQUEST
request.addEventListener('load', function(event) {
   console.log(request.response);
});
```

Note that the XHR object has a property named response. This is a reference to the response body, the type of which will depend on the type of data requested.

Load is only one of several events the XHR object will generate and that you can assign handlers to. The others are tabled below.

Event	Description
loadstart	Generated when loading of the resource has begun
progress	Generated periodically
abort	Generated if and when loading of the resource is aborted
error	Generated if and when an error occurs
timeout	Generated if and when the preset time for loading expires
loadend	Generated after load, abort, or error
readystatechange	Generated when the request's readyState changes

The XHR object has many properties beyond the open and send methods, and the response itself. Some of the others are tabled below.

Property	Description
onreadystatechange	Can be used to assign one handler to manage all events and is supported in all browsers
readyState	0: UNSENT 1: OPENED 2: HEADERS_RECEIVED 3: LOADING 4: DONE
status	HTTP status code
timeout	Number of milliseconds a request may take before timing out

Submitting Data

To submit data using AJAX you must pass the data to the XHR object's send method. Consider the following:

```
var request = new XMLHttpRequest();
// assign handler(s)
request.open('POST', '/app/process-data.php');
request.send('Some data to be processed');
```

Note that the HTTP method is POST, not GET.

If you want to submit form data, the simplest way is to create and send a FormData object. It allows for the construction of form data entirely in code:

```
var formData = new FormData();
formData.append('name', 'Jeff Thomas');
formData.append('email', 'jefft@mail.com');
...
request.send(formData);
```

Alternatively, the FormData constructor function may be passed a reference to an HTML form element:

```
<form id="myForm">
   <input type="text" name="name">
   <input type="email" name="email">
</form>
...
<script>
   var myForm = document.querySelector('#myForm');
   var formData = new FormData(myForm);
   ...
   request.send(formData);
</script>
```

The form data will be submitted to the server asynchronously.

XML and JSON

Obtaining plain text via AJAX is easy, but parsing it is not. Depending on your requirements, obtaining HTML, XML, or JSON data is likely to be better.

eXtensible Markup Language (XML)

XML is very similar to HTML but allows for custom tags, and can therefore support a wide variety of document types. It can be used outside of browser-based applications to transport and store data, and is mainly concerned with the underlying structure of data.

An XML file always contains one root or parent element which contains all others. Consider the following example (tracks.xml):

```
<tracks>
   <track id="t1">
      <artist>Bruce Springsteen</artist>
      <title>American Skin (41 Shots)</title>
      <album>High Hopes</album>
   </track>
   <track id="t2">
      <artist>Rag 'n' Bone Man</artist>
      <title>Grace (We All Try)</title>
      <album>Human</album>
   </track>
</tracks>
```

When you request a HTML or XML resource using AJAX, you should set the responseType property of the XHR object to 'document'. By so doing the response will be a Document type object. This means that you can process it like you would other HTML element objects. Consider the following:

```
var request = new XMLHttpRequest();
// assign handler(s)
request.responseText = 'document';
request.open('POST', '/app/data/tracks.xml');
request.send();
```

The handler might look something like this (assuming the tracks.xml resource):

```
request.addEventListener('load', function(event) {
   var doc = request.response;
   var titles = doc.getElementsByTagName('title');
   var firstTitle = titles[0].innerHTML; // American Skin (41 Shots)
   ...
});
```

You can see that extracting information from an XML document can be tedious.

JavaScript Object Notation (JSON)

JSON format is effectively identical to JavaScript object notation (hence the name). Note however that JSON keys must be surrounded by double quotes. Consider the following example (tracks.json):

```
[
   {
      "id": "t1",
      "artist": "Bruce Springsteen",
      "title": "American Skin (41 Shots)",
      "album": "High Hopes"
   },
   {
      "id": "t2",
      "artist": "Rag 'n' Bone Man",
      "title": "Grace (We All Try)",
      "album": "Human"
   }
]
```

This example is an array of two objects. It is both valid JSON and JavaScript.

When you request a JSON resource using AJAX, you should set the responseType property of the XHR object to 'json'. By so doing the response will be a standard JavaScript object. This means that you can process it like you would any other. Consider the following:

```
var request = new XMLHttpRequest();
// assign handler(s)
request.responseText = 'json';
request.open('POST', '/app/data/tracks.json');
request.send();
```

The handler might look something like this (assuming the tracks.json resource):

```
request.addEventListener('load', function(event) {
   var tracks = request.response;
   var firstTitle = tracks[0].title; // American Skin (41 Shots)
   ...
});
```

Simpler than XML? We think so.

StayAhead Training Limited
6 Long Lane
London
EC1A 9HF

020 7600 6116

www.stayahead.com

Terms & Conditions