Basic Javascript Programming

This chapter covers the basic principles of Javascript programming: files and program structure, variables and values, intrinsic data types, functions, objects and Object Oriented Programming.

A well-organised Javascript Program

A newcomer to Javascript will often create a single document (file) that contains HTML, Javascript and sometimes even Style-Sheets. In many cases, Javascript is used to perform very simple functions within a website; for example, making sure that a date the user has entered is valid (no 31st of April) or that a number entered is actually a number (e.g. not L3).

Since the single-file format works for simple systems, it is often accepted as 'the way to do Javascript'. There are even some books that advocate this approach to Javascript programming. In this module, our aim is to create Javascript programs that are easy to write, easy to maintain, modular and well structured:

- Easy to Write: keeping things simple suggests that we try to create small code files that do relatively small tasks rather that pouring all of an application's code into a single big file. This does mean that we need to know how to make the code in the separate files interact, but that is a fairly simple task, compared with building massive complex systems with no separation of responsibilities
- Easy to maintain: by putting the code for a simple task into a single, small file, we can make sure that the task is well tested. If something goes wrong, it is usually clear which area of responsibility has been the problem and therefore which file the problem is in
- Modular: modularity in programming is simply the idea that different responsibilities are kept separate – one job per code file is a good starting point
- Well structured: code is well structured when it is easy to describe as a set of interacting components.

For these reasons, we will try to arrange things so that HTML **documents** are kept separate from Javascript code. Figure 3.1 shows a sensible structure for a small to medium sized Javascript program.

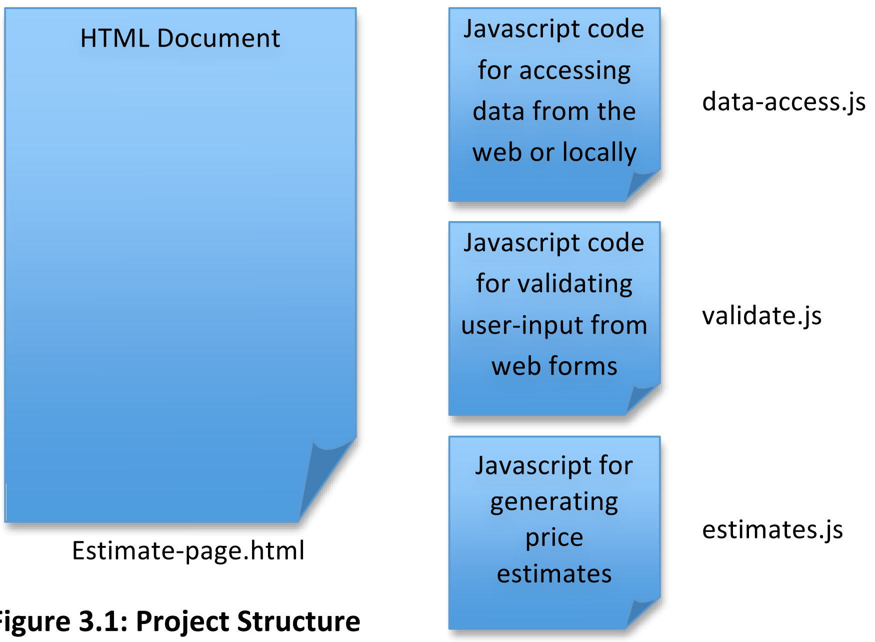

HTML Document
Estimate-page.html
Javascript code for accessing data from the web or locally
data-access.js
Javascript code for validating user-input from web forms
validate.js
Javascript for generating price estimates
estimates.js

Figure 3.1: Project Structure

Why arrange it like this? There are several reasons – all very good ones:

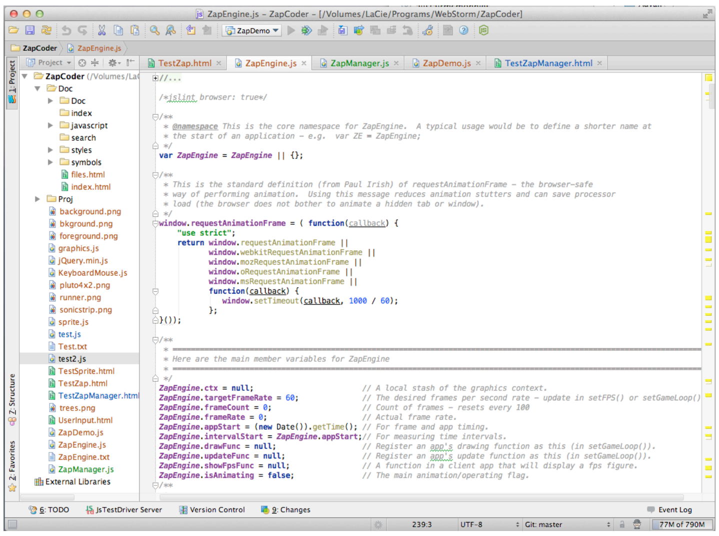
- I may want to access data from other HTML pages, so keeping the code for that in a separate file means I won't have to duplicate it or cut & paste the code into another file
- I may need to validate data entered into forms on other HTML pages – I can re-use that code
- By keeping the main program code in a separate file (in this case estimates.js is almost certainly the main code since it performs the main operations that the Estimate-page.html web-page needs) I can test the code in a much simpler scenario than one that needs to incorporate validation and data-access
- The code files are smaller, and all of their contents are thematically related, so there are no major problems in finding a bit of code that I need to check, modify or fix
- In future projects, I can apply a mix & match process – adding in Javascript modules that suit particular requirements of web pages

Javascript has a reputation for being an 'awkward' language: there are parts of its design that do not promote good programming principles (e.g. globally scoped variables, conflicting array syntax), and other parts that were just plain bad ideas (e.g. accepting undeclared variables, optional semicolons). You would not find many programmers prepared to argue against this perception.

However, we have two compelling reasons for learning to use Javascript well:

1. Along with its bad features, Javascript has some really good features that simplify commonly complicated programming tasks: literal object notation, functions that are objects, easy to use and powerful data structures are among these
2. Javascript is the standard language of the browser, and so we are stuck with it! Beyond this reason, the main browser manufacturers (Mozilla, Apple, Microsoft, Opera, Google) are aware of the limitations had have done a lot of work to turn the pigs ear into a silk purse. All mainstream browsers now have a Just-In-Time compiler for Javascript, which means that the code executes very quickly (compared to Javascript of a decade ago, but also compared to compiled languages like Java and C#), and most browsers now include great tools for de-bugging and testing Javascript code.

Creating and testing simple Javascript programs can be done with no more than a browser and a copy of Windows Notepad. Generally it would be better to use a more capable development environment, and we have a site-licence for Jetbrains' WebStorm – one of the best (I think the best) Javascript editors and project managers available. Figure 3.2 shows the WebStorm IDE, and you will learn more about it in Lab1.

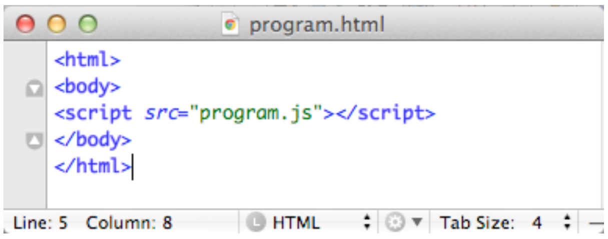


```
//...

/*jslint browser: true*/

/**
 * @namespace This is the core namespace for ZapEngine.  A typical usage would be to define a shorter name at
 * the start of an application - e.g.  var ZE = ZapEngine;
 */
var ZapEngine = ZapEngine || {};

/**
 * This is the standard definition (from Paul Irish) of requestAnimationFrame - the browser-safe
 * way of performing animation.  Using this message reduces animation stutters and can save processor
 * load (the browser does not bother to animate a hidden tab or window).
 */
window.requestAnimationFrame = ( function(callback) {
    "use strict";
    return window.requestAnimationFrame ||
           window.webkitRequestAnimationFrame ||
           window.mozRequestAnimationFrame ||
           window.oRequestAnimationFrame ||
           window.msRequestAnimationFrame ||
           function(callback) {
              window.setTimeout(callback, 1000 / 60);
           };
}());

/**
 * ==============================================================================
 * Here are the main member variables for ZapEngine
 * ==============================================================================
 */
ZapEngine.ctx = null;                           // A local stash of the graphics context.
ZapEngine.targetFrameRate = 60;                 // The desired frames per second rate - update in setFPS() or setGameLoop()
ZapEngine.frameCount = 0;                       // Count of frames - resets every 100
ZapEngine.frameRate = 0;                        // Actual frame rate.
ZapEngine.appStart = (new Date()).getTime();    // For frame and app timing.
ZapEngine.intervalStart = ZapEngine.appStart;// For measuring time intervals.
ZapEngine.drawFunc = null;                      // Register an app's drawing function as this (in setGameLoop()).
ZapEngine.updateFunc = null;                    // Register an app's update function as this (in setGameLoop()).
ZapEngine.showFpsFunc = null;                   // A function in a client app that will display a fps figure.
ZapEngine.isAnimating = false;                  // The main animation/operating flag.
/**
```

Figure 3.2: The WebStorm IDE

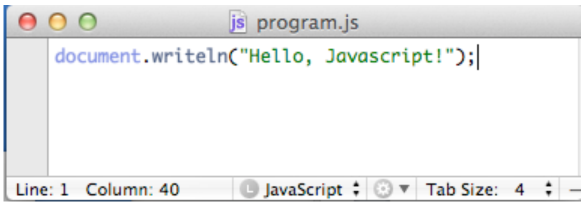
A Simple Test Program

To get a Javascript programming up and running, you need to provide a certain amount of framework. Using the principle described above that says that it is best to separate Javascript code from the HTML file that hosts it, a minimum JS (I'll use this from now on as shorthand for Javascript) program would occupy one HTML file and one JS file, as shown below:



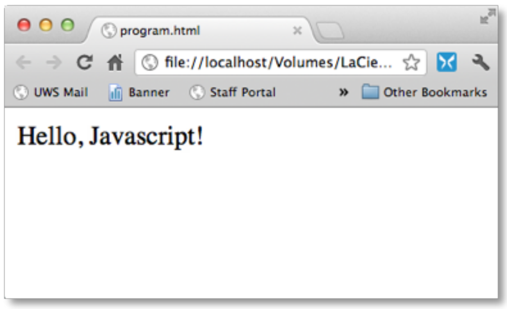
```
<html>
<body>
<script src="program.js"></script>
</body>
</html>
```

Figure 3.3: The HTML "host" file



```
document.writeln("Hello, Javascript!");
```

Figure 3.4: The Javascript code file



```
Hello, Javascript!
```

Figure 3.5: The resulting output, shown in a Google Chrome browser

Figure 3.3 shows a text editor (TextMate – a popular one for Apple Macs) with the HTML code required to define a "host" web page for the JS program – the page is empty but includes a <script/> tag which loads and executes the named JS file. The 'src' attribute HTML file names the JS code file and gives its location (in this case, the JS file is expected to be in the same disk folder as the HTML file). You can use Notepad on a Windows PC to write this file.

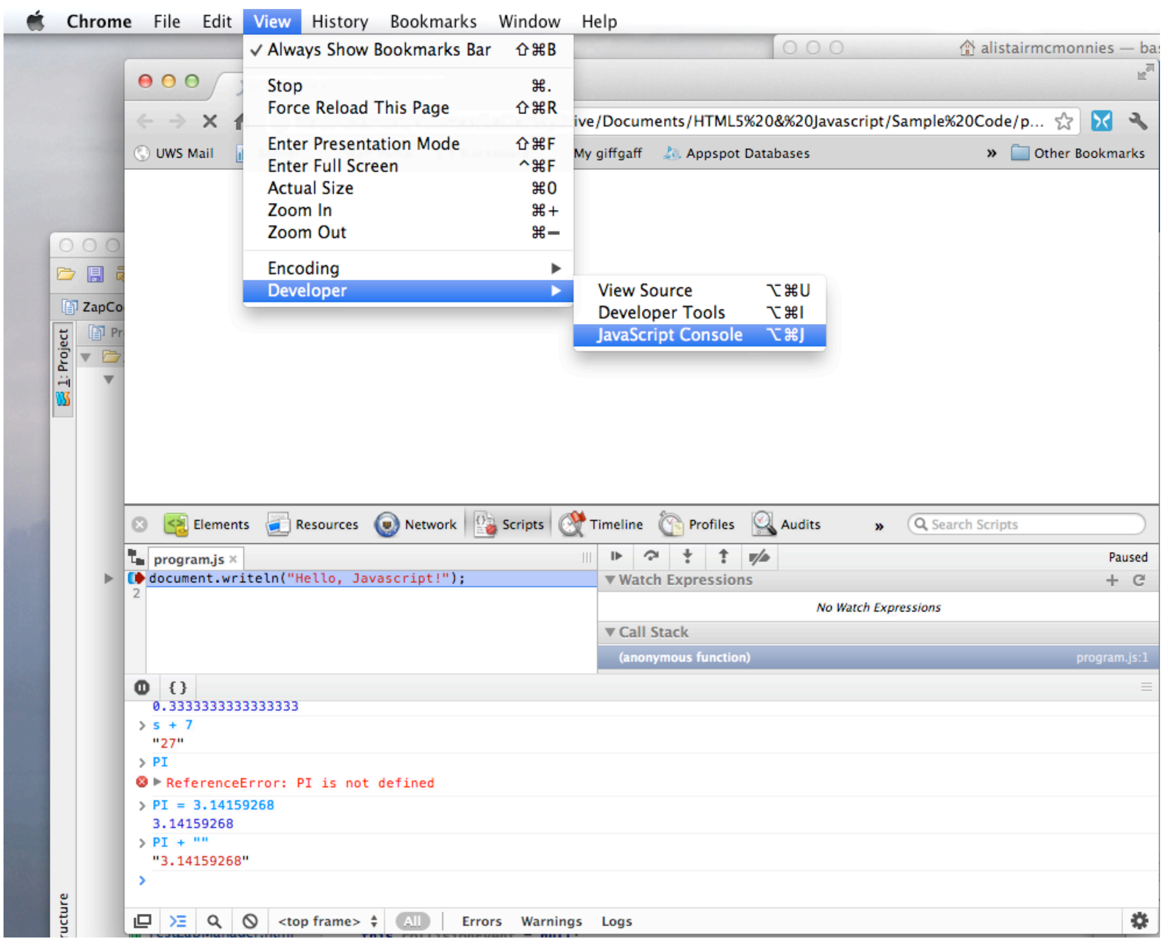
Figure 3.4 is the JS code file with a single statement, which adds a message to the current web page (an HTML document). Figure 3.5 shows the result in a browser – in this case Google Chrome, although any current browser would show the same result.

If you have a PC, Mac or Linux computer of almost any vintage from the past 10 years, you ought to be able to create these two files and execute the program in whatever your default browser is. In Windows and Mac, once you have the code files written, a simple double-click on the HTML file in an Explorer (Windows) or Finder (Mac) window should cause a browser to load and the program to execute in it. I suggest you try this at the first opportunity, for no other reason that to demonstrate to yourself that it is very easy to do. If for some reason it does not work, check the spelling and punctuation in your code (both files).

Now that we have that out of the way, we'll start with a more rigorous description of the most important features of JS.

Using a Console Window

You can test out a lot of Javascript code directly in a Console window. The Chrome and Safari and Internet Explorer browsers have these built-in, and Firefox can access one using the Firebug plug-in. Basically a console window lets you enter Javascript statements and expressions and will execute them immediately, often printing out what the result is. This is great for testing out small bits of code when you're not sure how they will work in a program.


Chrome File Edit View History Bookmarks Window Help
✓ Always Show Bookmarks Bar ⇧⌘B
Stop ⌘.
Force Reload This Page ⇧⌘R
Enter Presentation Mode ⇧⌘F
Enter Full Screen ^⌘F
Actual Size ⌘0
Zoom In ⌘+
Zoom Out ⌘−
Encoding
Developer
View Source ⌥⌘U
Developer Tools ⌥⌘I
JavaScript Console ⌥⌘J
alistairmcmonnies
ive/Documents/HTML5%20&%20Javascript/Sample%20Code/p...
UWS Mail
My giffgaff
Appspot Databases
Other Bookmarks
Elements Resources Network Scripts Timeline Profiles Audits
Search Scripts
program.js
Paused
Watch Expressions
No Watch Expressions
Call Stack
(anonymous function) program.js:1
<top frame> All Errors Warnings Logs

```
document.writeln("Hello, Javascript!");
```

```
0.3333333333333333
> s + 7
"27"
> PI
ReferenceError: PI is not defined
> PI = 3.14159268
3.14159268
> PI + ""
"3.14159268"
>
```

Figure 3.6: The Javascript Console in the Chrome browser

You can see how to access the console in Chrome in figure 3.6; also the actual console (at the bottom of the screen-shot) with some expressions entered. When you first open a console (in Chrome – others may differ), it is attached to the bottom of the browser window. You can detach it using the button that looks a little like overlapping rectangles at the bottom left.

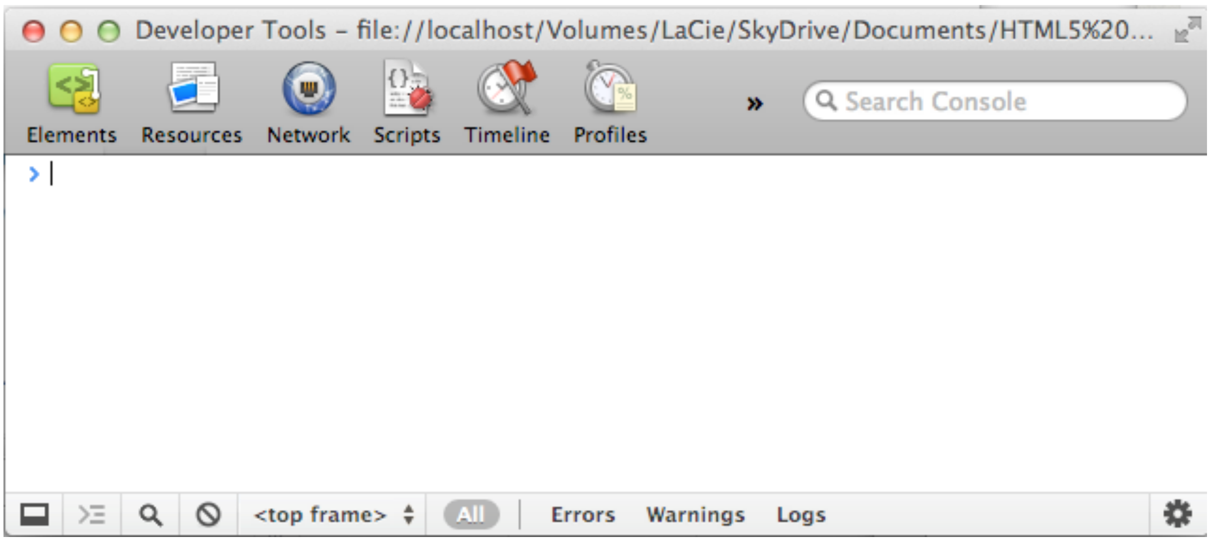


Figure 3.7: The Chrome console – detached.

From here you can enter a variety of different types of code fragments. The console will try to respond with the most useful output. For example:

- Enter a valid variable value, like 11, "fred" or true, and it will simply echo the value back to you
- Enter an expression and it will try to evaluate it and print the result. e.g
  - 2 * 4 // prints 8
  - "Hello " + "Javascript" // prints "Hello Javascript"
  - 12 + " days of Xmas" // prints "12 days of Xmas"
  - 1/0 // prints Infinity
  - Infinity – Infinity // prints NaN (not a number)
- Enter a statement to assign a value to a variable and it will print out the value assigned. e.g.
  - PI = 3.14 // prints 3.14
  - Radius = 2 // prints 2
  - Circumference = 2 * PI * Radius // prints 12.56
- Enter a function call or an object's method call, and it will print the result. e.g.
  - Math.sqrt(3) // prints 1.7320508075688772
  - parseInt("12 feet) // prints 12
  - confirm("Are you ready?") // prints **true** or **false** depending on the user's click

In particular, the last option very useful since to produce the displayed output, the console had to run the function that was called. The **Math.sqrt()** example evaluates the square root of 3, the **parseInt()** example tries to extract an integer value from the string passed to it and the **confirm()** example would put a box on the screen with OK and Cancel buttons, and evaluate the result. ***This makes the console useful for testing your own code***. If you have defined a function somewhere in the current web page (the one that is in the main browser window), you can test it by calling it from the console. Since this is a controlled situation, you get a better chance to figure out what your function really does than if you simply rely on it being called as part of the web page.

The Javascript Language

Variables & Values

Javascript is based on a very simple philosophy – everything is an Object. What this means is that every value, variable and function in a JS program has some properties and behaviours associated with it. For example, the number 3.14159268, an approximation of PI, is a value that you could assign to a variable – e.g. the first statement in a JS file could be:

```
var PI = 3.14159268;
```

From this point on, PI is a valid identifier that refers to an Object of the Number type. If you wanted to get rid of some of the decimal places so that this value printed out more neatly, you could use:

```
document.write( PI.toFixed(2) );
```

Using the .toFIxed() method of the Number type will return a string that contains the number expressed to the given number of decimal places (2 here). What is easy to miss, though, is that the number itself is treated as an Object in JS code, so we could instead use:

```
document.write( 3.14159268.toFixed(2) );
```

and get the same result. The number is converted to a Number object directly, and so all of the functions available for Number variables are available to the literal value.

This facility (using values directly as objects) is not one you'll use very much in JS programming, but it is worth being aware of the principle, since the same principle (a value is an object) has got some very important and useful consequences.

JS gives use three basic types – Number, String and Boolean – that work in this way, usually referred to as "Primitive" types. Since these are all of the type of values available to you (compare this with Java, which has 6 numeric types alone), you have to be very careful how you use them. Using floating-point arithmetic, for example, you can get odd rounding effects:

```
document.write( 9.2 * 100.0 );          // prints 919.9999999999999
```

This is normal in all languages which provide floating-point calculations, but without an integer type you need to be careful that this type of error does not upset the counting in a 'for' loop, for example.

It is useful to know that if you want to find out what type a variable is, you can use the ***typeof*** operator: e.g.

```
typeof PI;  // returns "number"
```

Numbers

Assign any integer or decimal value, or the result of a numeric expression to a variable, and it will assume the "number" type. You can also assign hexadecimal values by prefixing with 0x, and octal values by prefixing with a 0. While the hex assignment is fairly clear ('x' doesn't appear in normal numbers), you can get caught out using a 0 prefix. For example:

```
0344 // 344 octal is 228 decimal
0393 // 393 can not be an octal number (9 isn't a possible
digit), so 393 decimal
0x234  // 234 hex is 564 decimal
```

```
    0xff45 // 65349
```

Octal and hex values are mostly used as a quick notation for specific binary operations – e.g. assigning colour values, so for now you can ignore them in your own code (but be aware of their possible use in other peoples' code).

The numeric range can go from very small (0.000000000001) to very large (999999999999999), so sometimes you'll see values expressed with exponential notation – particularly values that are fundamental physical constants like the charge on an electron or the number of molecules in a mole of a substance (Avogadro's number):

```
    1.6e-19  // 1.6 divided by a 1 with 19 zeroes after it (charge of
  an electron)
    6.02e+23 // 6.02 multiplied by 10, 23 times (Avogadro's constant)
```

Using this notation, there is less chance of losing count of the number of digits. The part before the 'e' is the number that is modified by either dividing (for a negative exponent) or multiplying by (for a positive exponent) 10 the number of times indicated by the value after the 'e'.

Of course there are maximum and minimum values that can be represented by a number in JS – Number.MAX_VALUE returns the biggest possible number (1.7976931348623157e+308), and Number.MIN_VALUE returns the smallest (5e-324). Given these extremes, it is unlikely you'll find the max and min values a restriction.

If you try to exceed the maximum value in an assignment, the variable will be set to the special numeric value "Infinity". Of course this is not infinity in the mathematical sense (whatever that is); just that JS can't cope with the value:

```
    Number.MAX_VALUE + 1;                    // 1.7976931348623157e+308
    Number.MAX_VALUE * 1.0000000000000001    // 1.7976931348623157e+308
    Number.MAX_VALUE * 1.000000000000001     // Infinity
```

Note that adding 1 to MAX_VALUE has no effect (it gets swallowed up in the rounding errors), but somewhere between multiplying by 1+1e-16 and 1+1e-15, the rounding errors become less significant and the value exceeds the maximum possible.

One other special number value is NaN (Not a Number). This is the result if you try to perform some operation that JS can't cope with where the result is not Infinity. For example:

```
        10 * 'g';               // NaN
        11 - 'x';               // NaN
        1 + NaN;                // NaN
        Infinity - Infinity;    // NaN
```

The last one is interesting – you might think that subtracting Infinity from itself would result in zero, but then an expression like "Infinity + 1 – Infinity" would be 1 and so on, which would be a bit like promoting Infinity to being a value that meant something. All that Infinity is (in JS) is an indication of "too-big to handle" – not a specific value.

One very weird thing that JS does is to report both Infinity and NaN as "number" type:

```
    typeof Infinity;    // “number”
    typeof NaN;   // “number” – deeply weird: “Not a Number” is a
  “number”
```

Strings

A string is a sequence of characters, or what we might call Text. However, it is a sequence of **any** characters available, including punctuation, digits, odd typographical symbols (like ~ and |) or anything else that could be printed as a character, so Text is too restrictive a name.

```
s = “character data”;
typeof s; // “string”
s2 = “!@£$^&(*((*&&&~|}`?”;
typeof s2;  // “string”
```

Note that we need to use quote marks (“ or ‘) to assign a literal string value to a variable (think about the alternative). The quotes used need to match on either side – e.g. “ and “, or ‘ and ‘. This lets us use one or other type of quite mark inside a string – “Fred’s keys”. Strings can be added together (this is called concatenation) to make bigger strings:

```
S1 = “Hello”;
S2 = “Fred”;
S1 + S2;    // “Hello Fred”
```

There is no equivalent of subtraction, multiplication or division. However, strings are pretty powerful because number-like strings are automatically converted to numbers if the sense of the statement makes this useful, and other types can automatically be converted to strings if that would be useful. For example, we *can* multiply a numeric string, like:

```
S = ‘2’;
V = 4 * s;  // 8
```

You need to be careful to not assume too much about this – multiplication, subtraction and division will all work appropriately:

```
S – 4;    // -2
S / 6     // 0.3333333333333333
```

However try to use addition the same way, and you’ll get a concatenation:

```
S + 7;    // “27” – not 9.
```

You can add anything to a string and the result will be a string because of how the + operator is defined for strings. This makes it easy in programs to generate meaningful results. For example:

```
pricePlusVAT = price * (1 + VATRate/100);
displayPrice = “£” + pricePlusVAT.toFixed(2);
```

If you need to simply convert a number to a string, you can add an empty string to it:

```
document.writeln( PI + “’);   // “3.14159268” – a string value
```

JS will pragmatically convert to and from string values as necessary, and most of the time it will make the correct assumptions. However, it is safer not to assume this – always at least test out any conversions you intend to make in a browser console window to see how they will be handled.

Some string values have special purposes. For example:

```
‘\n’ new-line
‘\t’ tab
‘\b’ backspace
‘\”’ a double quote
‘\’’ a single quote
```

The \ character is known as the ‘escape’ character – it has a special purpose is allowing otherwise non-usable characters or non-keyboard characters to be put into strings. If you want to use an escape character in a string, simply use \\.

Booleans

A Boolean value can be only true or false. These are not strings “true”/”false”, but instead the unquoted words, which are properly interpreted by Javascript as Booleans. Booleans have a special purpose in programming, in that they can be used to store the result of conditions – for example the expression x == 2 has a Boolean result (true if x has the value 2, false otherwise). Because of this, they are most often used in situations where an if… is appropriate. Usually, you don’t use a Boolean variable for this, but use an expression that evaluates to a Boolean. For example:

```
  if( x > 0 ) y = Math.sqrt(x);    // Prevents taking the square root
of a negative number
```

However, often a Boolean variable can be used to store a true or false result for later in a program. For example:

```
  validRoot = (x > 0);// We’ve stored the result of a test, later
in the program…
  if( validRoot ) y = Math.sqrt(x);    // Uses the result of the
test done earlier
```

Booleans also come with some special purpose operators, used for combining an/or manipulating Boolean values. e.g.

```
  iAmOver16 = confirm(“Are you 17 or older?”);
  iHavePassedADrivingTest = confirm(“Have you sat and passed a
driving test?”);
  iHaveConvictions = confirm(“Have you had any convictions for
drink driving?”);
  iHaveACurrentLicence = confirm(“Do you have a current driving
licence?”);
```

The answers to these four questions can, in combination, evaluate whether it is legal for you to drive a car in the UK. We can combine them like:

```
if(  iAmOver16 && iHavePassedADrivingTest &&
     !iHaveConvictions && iHaveACurrentLicence)
   document.write(“Vroom vroom!”);
```

The && operators are Boolean ***and*** operators. For an expression with a number of conditions 'anded' together to be true, all the parts have to be true. Note the ! (Boolean ***not***) operator, which turns the iHaveConvictions value into an iHaveNoConvictions value.

The set of Boolean operators are:

&&	Boolean AND	(true only if all are true)
||	Boolean OR	(true if any one is true)
!	Boolean NOT	(true if 'notted' expression is false & vice versa)

True and False Values

if(), while() and for() control statements in JS all use a Boolean to decide how to proceed. Usually, these are proper conditions, like if(x>0) etc. However, JS can use any value as a form of condition. Generally this is considered bad practice (a Java programmer would certainly throw their hands up in horror, while a Pascal programmer might just have a seizure at the idea), but that's never stopped a JS programmer from doing this as a short-cut:

```
change = amountPaid – amountCharged;
if(change) {
  returnChangeToCustomer();
}
```

The workings of this are that if the value of change is 0, it will be considered to be *false* and so the returnChangeToCustomer() function will not be called. There are five different values in JS that are taken as the equivalent of *false*:

```
undefined i.e. a variable that nothing has been assigned to yet is
undefined
NaN
null   i.e. no specific value
0
“”   i.e. the empty string
```

Every other value is considered to be *true*. In JS parlance, values can be "truthy" or "falsey". The significance of the 'falsey' values is that they are potentially default values in certain circumstances. For example, a variable that has been declared but has never had a value assigned to it will always have the value *undefined*. The value *null* can be assigned to an object variable specifically to mean "no object". NaN is, as we've seen, the possible result of an arithmetic operation that has failed in some way. An empty string ("") is "no text" so should be considered to be false.

Zero is more difficult to justify, since zero is a definite numerical value. JS takes its cue from the C language (which much of the JS syntax is based on) for this. In C, the numeric value 0 was the definition of Boolean false, and any non-zero value was therefore true. This follows in Javascript.

Declarations

Javascript was written to be a quick and easy language to assimilate. Because of this, a number of the constraints other programmers take for granted were "relaxed" – the result is useful if all you wish to do is write a short bit of validation code for a web form, but can promote chaos in a bigger project. When Javascript was first developed, no one imagined that a word processor or spreadsheet could ever be written to work inside a browser (Google Docs, Microsoft Online and Zoho all demonstrate that where there's a will, there's a way).

In JS code, simply assign a value to a new identifier and you have a new variable:

```
X = 10;
```

There are at least two problems with this:

1) X now belongs to the current context variable – usually the window object of the browser. Essentially, this makes it a Global Variable (one that is available anywhere in the program), and for years, programming languages have been designed to minimise the use of global variables because when there are many variables in a program, it is difficult to prevent a programmer creating another global with the same name, thereby destroying or overwriting the first

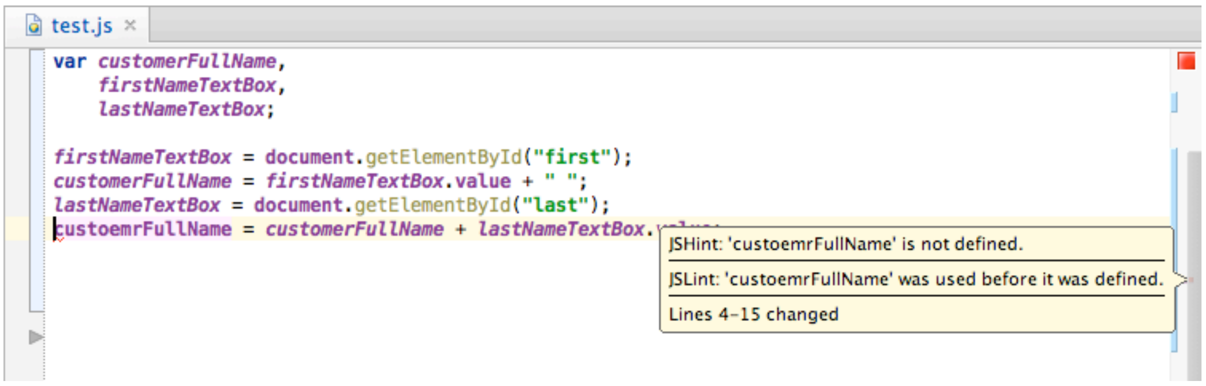
2) Although this is unlikely for a variable with a single letter name, longer variable names (e.g. currentWindowContext) are prone to being mistyped. When an assignment can just as easily create a new variable, this can cause serious (and very difficult to track down) problems in program code. For example:

   ```
   customerFullName = “Fred ”
   custoemrFullName = customerFullName + “Smith”;
   ```

   The simple typo above means that we now have two variables, one (with the wrong name) storing the customer's full name and the other (the one we expect) having only the first name.

Of course, Javascript has this bad behaviour built-in, and so errors in code that is run will not be detected until something bad happens. However, a good Javascript IDE (like WebStorm or Aptana for example) will point out the error. Both of these incorporate a program called jsLint (lint is fluff, and jsLint is a program that “picks the fluff” out of Javascript code), written by Douglas Crockford. This is also freely available as a separate program (it is written in Javascript) and can be used online at www.jslint.com.

The jsLint recommendation is that ALL variables in a program should be declared before they are used. It can then easily highlight variables that were not declared, including all the custoemrFullName variants, making it easy to fix errors and typos before they become a problem. Here is jsLint's response to the faulty program code:



```
var customerFullName,
    firstNameTextBox,
    lastNameTextBox;

firstNameTextBox = document.getElementById("first");
customerFullName = firstNameTextBox.value + " ";
lastNameTextBox = document.getElementById("last");
custoemrFullName = customerFullName + lastNameTextBox.value;
```

```
JSHint: 'custoemrFullName' is not defined.
JSLint: 'custoemrFullName' was used before it was defined.
Lines 4-15 changed
```

Figure 3.8: A WebStorm window, showing the jsLint hint from an error marker.

The screen shot in figure 3.8 shows up the error in four ways:

1. The big red square at the top of the scroll bar on the right hand side effectively says "there is at least one error in this file"

2. A small red rectangle (currently covered up by the hint window) shows the relative position of the error in the file

3. The first character in the line that contains the error has a red wavy underline

4. Hover the mouse pointer over the red rectangle for that error (see 2) and a hint pops up, explaining what the problem is

Armed with that information, even the most stupid or laziest programmer will have no problem fixing the problem during production.

The code above shows the standard recommended method for declaring variables:

1. There should be a single ***var*** statement at the top of a scope unit. In this case, the scope unit is global, although putting the same declarations in the first line of a function would make the variables local to that function

2. The full list of variables should appear in a single, comma separated list which is ended with a semicolon

3. Variables can optionally have values assigned to them within the declaration. jsLint flags this as a possible error (a warning) by using a small yellow rectangle, since it is slightly more efficient to separate the declaration and the first use of a variable. However, this is one of the settings in jsLint that can be switched off – many programmers do this

Variable Scope

A variable's ***scope*** is the range of code that can access it. What this means is that once a variable is declared, these is some code where its value can be read and altered, and other code where it can't. In the second lot, the code simply does not 'see' the variable at all. Generally it is a good idea to give variables the smallest scope possible, since this reduces the possibility of code inadvertently changing a variable and breaking a program.

JS provides three levels of scope:

1. Global scope: variables belong to the top syntactic unit in a program. Usually, this is the browser window, so a variable declared as: var x = 2; becomes a component variable of the window object. You can check this in a console window: enter x = 2 in one line, and then window.x in the next. The value 2 will be printed out

   All variables declared outside of any functions in a file that is part of a project will have global scope. All variables declared outside of any functions in another file that is part of the same project will also have global scope. Because of this, you need to be careful not to declare variables with the same name even in separate files in a project. This is difficult to do in a large project, which might have several programmers working on it

2. All variables declared inside a function are ***local*** to that function. This means that they don't even exist while the function is not executing. This is inherently safe, since values can't be left over from previous function calls – on every call, the local variables are re-created from scratch

3. All variables declared with a prefix of this. (e.g. this.x = 4;) are object members which belong to the current object. We'll see a lot more of this later, but effectively, object variables can be used to mimic different scopes – if I declare and use a variable X inside an object scope called MYAPP, it will not clash with other variables called X. We'll use this approach later in the module to create safe program contexts.

For now, don't get hung up about variable scope. Later, when it becomes important, the reasons for using scope management tricks in code will be a lot clearer.

Statements and Control Structures

A ***statement*** in JS is code that does some work. This distinguishes it from an expression, which gets evaluated but does not need to do something as a result. A short example explains this:

```
  X + 1;      // This is an expression – it has a value that the JS
interpreter evaluates
  X = X + 1;  // This is a statement – the value of X is changed
because of it
```

In much the same way, some function calls may take up processor time but have no overall result that is useful to the program. A function definition may return a value, but unless this value is used in some way, the call could just be wasting the processor's time.

Note that when you use Java interactively in the console, expressions always have some effect, because in the absence of it doing anything else, the console will print out the value of the expression. However, this is a convenience that the console provides – it does not happen in a program because in a program you are expected to say what gets printed out, what assignments are made and what is done with the result of an expression.

I often distinguish statements by calling them "executable statements". In fact this is a tautology (I'm saying the same thing twice), since in all programming languages, statements are required to make sure that code has a result. The easiest way to distinguish statements in "properly written" JS code is that a ***semicolon*** terminates them. Unfortunately this is not a law in JS, so I've had to say "properly written" – see below (Semicolons and Blocks) for more info on this. You should get used to typing a semicolon after each statement in JS because:

- It will prevent JS from having to perform automated semicolon insertion, which is can in some circumstances get wrong, destroying the logic of a block of code
- It will reinforce your recognition of the distinction between statements and control structures (see below)

JS recognises three types of program statement:

1. An assignment – e.g. ***x = x + 12;***

   This type of statement is the most obvious since some work is clearly being done. Assignments can come from simple operations (as above) or from the result of a function call, for example – ***x = Math.sqrt(y);***, in which the value assigned is one returned from a function
2. A function call with no assignment – e.g. ***myArray.sort();***. The sort() method of an array will reorganize its elements into some order (alphabetical or numeric). There is no return from this function, but instead it has a *side-effect* in that it alters the array contents
3. A declaration, such as: ***var x;***. In this case, the work that is being done is a bit more "under-the-hood" since it has no real effect on the results of the program. However, this type of statement is as important as the 'executable' ones because is gives the declared variable a scope (i.e. the parts of the program that can access it) which can have a major effect on the way the rest of the program behaves. Also, **jsLint**, which is built in to WebStorm and some other IDEs, will mark variables that have not been properly declared as errors in your code. Unfortunately, JS does not enforce the use of declaration statements in programs, ***but I do in assessments***.

JS, like almost all programming languages, provides structures that let you modify how code in normal statements runs. An **if()** structure lets you decide whether or not to run a chunk of code, an **if()…else** structure lets you decide which of two chunks of code to run, and **for()** and **while()** structures let you repeat a chunk of code either a set number of times or while a particular condition is true or false. These are the structural building blocks of programming, and as such are the elements that have given programming its power and flexibility.

Control Structures

As most of these structures operate around a Boolean condition (e.g. if(x>0), while(x<10) etc.), you need to know how to make efficient use of values in your programs as Boolean criteria. For example, you could write code to do something 10 times as follows:

```
x = 0;
while( x < 10 ) {
     // do something, then…
     x = x + 1;
}
```

This code will work fine, but it is less efficient and causes you to have to type more than:

```
x = 9;
while( x  ) {
     // do something, then…
     x = x – 1;
}
```

JS does not have to evaluate the Boolean condition (x < 10) since the value of x being zero or non-zero will act as a condition. Some programmers will say that this makes your code less easy to follow (most of the time I'd agree), but if you understand the Boolean equivalence of numeric variables this is not too strong an argument. I'd suggest using features like this only if you're happy with what is going on; they won't crop up too often in your programs anyway.

Comparison Operators

Every programming language needs the ability to compare things - compare values with the contents of variables, compare objects with other objects, check whether a value is within a given range etc. Generally languages define operators for comparison: comparison expressions that use these operators are placed within an if(), while() or for() statement so that comparison can be acted on.

Javascript has what appears to be a surplus of comparison operators:

Operator	*Name*	*Purpose*
==	Is equal	Test is two objects have the same value
===	Is the same	Test if two objects are the same value and type
!=	Is not equal	Test if two object have different values
!==	Is different	Test if two object have different value and type
>	Is greater	Test whether the first object has a bigger value
<	Is less	Test whether the first object has a smaller value
>=	Is greater or equal	Test whether the first object is ***not*** smaller
<=	Is less or equal	Test whether the first object is ***not*** bigger

Table 3.1: Javascript's comparison operators

You'll probably be familiar with most of the operators shown in table 3.1, but the first four (or, more specifically, the distinctions between them) confuse everyone initially. Let's see how they work with reference to a specific object. Assume we have a variable, v:

```
  v = 3;   // Assign the Number value, 3
  v == 4;  // false – the values are different
  v == 3   // true – the values are equivalent
  v == “3” // true – the values are equivalent (JS works it
out)
  v === “3”// false – the values are equivalent, but types are
different
  v === 3;       // true – value and type are equivalent
  v != 4;  // true – values are different
  v !== “3”;           // true – values are similar, types are
different
  v !== 3;       // false – values and types are the same
  v > 4;   // false – v is smaller
  v < 4;   // true – v is smaller
  v >= 4;  // false – v is less than (the test is NOT less
than)
  v <= 4;  // true – v is less
```

The two types of equality test are important because Javascript was designed to be able to do user-input comparisons as easily as possible (in my opinion, this didn't work out too well for them, since the solution demands that programmers have a good understanding of the underlying mechanisms). The best advice to programmers is that you should always use the === operator for equality and the !== one for inequality. These operators do what most programmers would accept to be an equality test, where you are aware you are comparing values of similar types.

In Java, for example, a comparison between a number and a string would simply not compile so the issue would never arise. Javascript attempts to make this possible, and the consequences are, at best, entertaining. For example, in Javascript:

```
”” == “0”             // false
”0” == “”             // true
“0” == 0              // true
false == “false”      // false
false == “0”                // true
false == undefined    // false
false == null               // false
null == undefined     // true
```

(You should try these in a console window to see the results for yourself).

Douglas Crockford, who is probably the best known Javascript advocate today, suggests that you should never use what he calls the evil twins (== and !=) because they do not follow any easy to define rules. Always use the longer === and !==, and it is guaranteed that two variables or values which have the same value and the same type will be properly evaluated as equal/unequal in tests.

Blocks and Statements

In JS, you de-lineate the code that is influenced by a control structure using braces (curly brackets). In a *for* loop, for example, braces are used to indicate the start and end of the *block of code* to be repeated. If there was only a single statement to repeat we can do it without braces:

```
for ( x = 0; x<10; x+=1 )
  // do something
```

This gives you a tiny bit less typing to do, but should be avoided for the simple reason that without them it can be more difficult to read a lot of code and figure 3. out the structure. Braces are visually more obvious than a little bit of indentation. Adding braces does not slow the code down and immediately makes what is to be repeated much more obvious:

```
for ( x = 0; x<10; x+=1 ) {
  // do something
}
```

From a “plain programming” perspective, you should always use braces. Your code will be clearer and easier to read. If every statement in a sequence of statements is to be part of the same operation, putting {} around them makes them into an indivisible (and very visible) block. If it is a single statement, braces make it clearer to the person reading the code that this one statement is a block.

Semicolons and Automatic Semicolon Insertion

In Javascript, every statement that is not a control statement should be terminated with a semicolon. Control statements are statements that modify the execution of others, such as if(), for() and while(). In an attempt to be more 'friendly' to programmers, JS will insert a semicolon wherever one is missing and that it detects that one was needed, so there is (seemingly) no penalty for missing them out. However, I'd consider omitting a semicolon where it was needed as bad practice. Two reasons for this: firstly, it indicates a lack of clarity in the code – you're leaving it up to the JS interpreter to add structure to your code and you should be ashamed of this. Secondly, there is at least one situation where the result will be code that does not do what you intended it to do.

The one situation in JS where a semi-colon can be automatically inserted into code that could change the way it works (compared to what you think your code will do), is illustrated in this example:

```
function sayHello(name){
  return
    "Hello " + name;
}
```

In this case, C, C# and C++ would all return the string "Hello Joe" for the call sayHello("Joe"), because these languages treat white-space (spaces, new-lines etc.) as non-existent. JS will return the value 'undefined', because the interpreter will insert a semicolon immediately after the return statement:

```
function sayHello(name){
  return;      // This return causes the function to exit
    "Hello " + name;
}
```

It is worth pointing out that having the item to be returned on the line after the return statement as it is above is pretty daft in the first place, although in functions that return a multi-part object there is a possible excuse for it. The "rules" for placing semicolons (not really rules, unless you count that I'll mark you down in any exams or project work where you miss them out) are simple enough. Place a semi-colon:

- At the end of any executable (i.e. non-control) statement
- At the end of a ***var*** statement (i.e. one that declares variables)
- Between the definition parts of a for() control structure (see above)
- To form an empty statement (i.e. one that does nothing)
- After a function definition that is also an assignment (see next section)

Normally, you should not put a semicolon:

- Immediately after an if(), while() or for() structure (unless you want it to act as an empty statement)
- After a closing brace (}), unless it is a statement that assigns function definition

Functions

The main programmatic unit in JS is the function. Functions operate as 'callable' blocks of code. They can accept parameters – i.e. values can be passed into a function for it to operate on. This makes them highly flexible. The general *design* principles for creating functions are:

- A function should do *one job* and do it well
- A function should have no side effects – i.e. apart from the specified job, it should leave no imprint at all. Essentially, this means that no variables outside the function should be modified by it, apart from where it is the functions job to do so
- A function can take zero, one or more *parameters* (its *inputs*) and should produce zero or a single result (its *output*)

For example, a function that results in the cube of the parameter passed to it can be written as:

```
function cube(value) {
  return value * value * value;
}
```

We can call this function from another statement like:

```
n = cube(4);      // assigns 64 to n.
```

Structurally:

- the function should have a name (but not always as we'll see later in the module)
- it should also have a return statement, which says what the output from the function will be (in this case, the input parameter multiplied by itself two times). In some cases, a return statement is not needed (if the function does not return an explicit value, or if the function is an *object constructor* [later] for example)
- input values (as many as you like) are defined by the identifiers in brackets after the function name. In fact, these brackets are necessary whether there are inputs or not
- a function can have one or many statements. For it to be a pure function (i.e. one that meets the *mathematical* definition of being a function), the last statement should be a return statement (there can only be one of these). Also any statements in the function should not access or modify variables outside the function. JS will not complain if you don't follow this rule, but lots of other programmers might

To make function definitions as easy to read and recognise as possible there are some stylistic conventions, which are not required to make the function work:

- the name should fit the purpose of the function. As the function above cubes the input value, it is called cube, but that is a stylistic convention – if we called it George, it would not change what it does
- to follow naming conventions, a function name should start with a lower-case letter. A function name that starts with a capital letter (e.g. Cube) would be assumed by many programmers to be a constructor function (i.e. one that created an object) – we'll see more about that next week

As an example of a pure function with more than just a return statement:

```
function factorial( value ) {
  var i, f = 1;
  for (i=1; i<=value; i++){
    f = f * i;
  }
  return f;
}
```

This function returns the Factorial of a number passed to it, which is the product of all of the integers from 1 to the number. For example, factorial(3) (which a mathematician would write as 3!)

is 1 x 2 x 3 = 6. factorial(4) is 1 x 2 x 3 x 4 = 24. The function has a few significant features:

- The first line in it declares ***local variables***. These variables exist while the function is executing, and are destroyed when the function is exited. This provides a guarantee that they can not interact with or affect any other part of the program
- There are two variable declarations – the first (i) has not immediately had a value assigned to it, so its value at that point is ***undefined*** (one of JSs falsey values). The second (f) has a value attached to it immediately, and this makes it a ***numeric*** type
- The last statement is the return statement, which passes the numeric value that has been accumulated in the f variable back to the point where the function was called from. If the return had not been the last line in the function, it would still have been the last line to be executed, and all other statements after it would have been ignored
- The way this function is defined makes it a global function – it can be called from any part of the JS program it is part of, even if it is in a different file from the statement that calls it. Since an HTML file (and therefore a browser document) can host any number of Javascript files, this makes it very easy to keep separate sections of a program in separate files and use the HTML document to integrate them. There are other ways that functions can be defined so that they are not global - we'll look into some of that later

Local and Global Variables

The code in a function can access variables that were created outside of it. In some cases, this will be a deliberate action that programmer makes; in others an accident that may have unfortunate side-effects. For example:

```
var PI = 3.14159268,
    x = 42,
    size = 6;
function area(radius) {
   return PI * radius * radius;
}
function getCube() {
   var x = 3;
   return Math.pow(size, x);
}
function getSquare() {
   x = 2;
   return Math.pow(size, x);
}
```

In the first of these functions (area()), we can assume that accessing the global variable PI was intentional since that value is needed to work out the area of a circle. In the second (getCube()), x has been declared inside the function. Therefore the variable x which is declared *outside* the function is invisible while the function is executing – the result is that the internal value of x will be used in the Math.pow() (raise a number to a power) function, so size will be raised to the power of 3 – cubed. The last function (getSquare()) assigns the value 2 to *the global variable* x. Almost certainly, this is an error since the purpose of the function is to square a value (so we need the value 2 as a power), but it also has the side effect of changing the global x from 42 to 2. Since the side effect has nothing to do with squaring a number, we can assume it was not deliberate

The problem is that JS cannot figure 3. out the programmer's intentions from the code – sometimes you need a function to alter a global variable, sometimes you don't. JS will not treat the 3rd function as an error, and the side effect of changing the global x could have serious repercussions later in the program.

The only way to prevent this type of problem is to be very careful when you write code. In situations where it is critical that programmers do not *inadvertently* change global variables in their code, a good procedure is to add a comment that indicates that the change in a global variable was deliberate. e.g.:

```
var currentInterestRate = 3.1;     // 3.1%

 /**
   * changeInterestRate function.
   * Purpose: Update the global currentInterestRate
   **/
function changeInterestRate(newRate){
    currentInterestRate = newRate;
}
```

In the function above, the comment block above it indicates that the function's purpose is to alter the global variable. Others who read this code will not assume that this is an accidental side-effect.

Function Definition Styles

One of the potentially annoying features of Javascript is that it often provides several ways to do the exact same thing. A programmer must then make a choice about which method to use; usually the only deciding issues are of convenience, programming style and conventions. The ***cube()*** function defined earlier will be the exact same function, callable in the exact same way, with any of these four styles of definition:

Style	Function Definition	Notes

Function declaration	`function cube(n) {`
	`  return n * n * n;`
	`}`
		The most common style.

Function expression	`var cube = function cube(n) {`
	`  return n * n * n;`
	`};`
		Significant features of this;
		1. The function is assigned to a variable – as an assignment expression, is should have a terminating semicolon (on the last line)
		2. The function name is given twice – these are its internal name (the name the function knows about itself, and the external name – the name of the variable it is assigned to. ***These can be different*** but that is not always a great idea

Anonymous function expression	`var cube = function(n) {`
	`  return n * n * n;`
	`};`
		Similar to a function expression, but the function has no internal name. There are reasons why this might sometimes

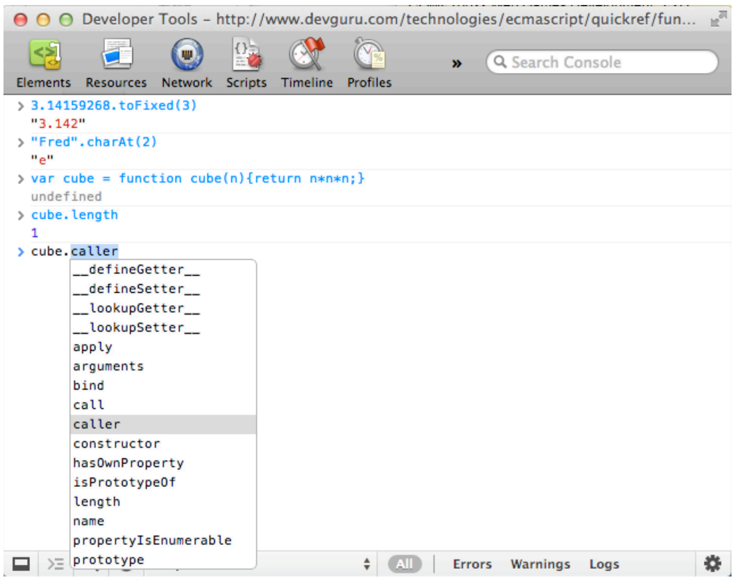
		be useful
Function constructor	var cube = new Function(“n”, “return n * n * n;”);	This uses the internal mechanisms in JS to build the function from strings. For various reasons we will look into later, this style is to be considered ***evil*** and you should never use it.

Table 3.2: Function definition styles

Most of the time you will find that the first version is the one that is easiest to use and the least ambiguous. Function expressions become useful when you start to use functions as part of objects. Anonymous functions are useful when you don’t want to introduce new names into a program. Function names are a finite resource, and every time you use one that name can’t be used elsewhere in the program. This may seem like an odd thing to guard against, but if you include a library of someone else’s code in a program, there is a growing chance that a name you have used will also be in the library, and therefor either your function or the one in the library will no longer work. “Namespace pollution” it the term used to define the situation where function and variable names are defined indiscriminately in a program. In Javascript, there are lots of situations where you might want to define a function and call it once, or pass it to an object to call it for you – in that case, its name is unnecessary, and the function would be better defined as an anonymous expression.

Objects

Javascript is best described as a prototypical object-oriented language. Everything in a JS program is an object, as can be easily demonstrated by entering a few statements in a browser’s JS console:



```
> 3.14159268.toFixed(3)
  "3.142"
> "Fred".charAt(2)
  "e"
> var cube = function cube(n){return n*n*n;}
  undefined
> cube.length
  1
> cube.caller
```

Figure 3.9: Objects in Chrome’s JS Console

Note that for all of the values shown in the console (each entry after the > prompt), I’ve inserted a method call (i.e. I’ve called one of the object’s functions). Any number has a range of function associated with it (including **toFixed()** as shown here), all string values can assess a wide range of functions (including **charAt()**, which extracts the character from the specified position, counting from 0), functions also have a range of functions associated with them (length is the number of arguments (parameter values) passed to the function – in this case there is just ***n*** so that is 1). The last entry in the screenshot in figure 3.9 is the most interesting, because you can see exactly how the JS console deals with objects.

If you type the name of an object followed by a period (.), it will pop up a list of all of the properties (member variables) and methods (member functions) known to be part of that object. You can see there is a long list of members of any JS function, all of which can be used either to query the object (what are its current settings) or manipulate it.

Literal syntax

JS provides a number of ways to define an object (much the same as the list of ways to define a function). One of these is very straightforward – literal object syntax is simply a way of creating an object in code directly and simply:

```
var aPerson = {
  first: “Freda”,
  last: “Bloggs”,
  age: 25,
  email: “freda@bloggo.com”;
  fullName: function() {
    return this.first + “ “ + this.last;
  };
};
```

This format effectively packages up several values (first, last, age, email) in such a way that they are the parts of an object. In this case a function is also added, which means that the internal object data can be accessed in a more useful way: if Freda Bloggs were to marry Joe Smith at some point, a simple statement:

```
aPerson.last = “Smith”;
```

would update the object so that Freda’s full name would be properly reported.

To access any part of the object (including the function), simply use it as an expression in a statement, using dot notation to indicate the part:

```
  document.writeln(aPerson.age);        // writes 25
  document.writeln(aPerson.email);      // writes
“freda@bloggo.com”
  document.writeln(aPerson.fullName()); // Note the brackets –
calls the function
```

As mentioned, object literal syntax is just one way to create an object in JS. There are other ways that can be more useful in some circumstances. We’ll look into this in detail in the next section

Javascript for jQuery Mobile

Of course Javascript code is still necessary in jQuery Mobile (JQM) applications to do many of the things it is used for in normal web apps. However, JQM provides a help in several areas – in a few cases removing the need for Javascript code, and mainly making it possible to write much less Javascript code to achieve the same ends.

The JQM framework takes over all of the cosmetic tasks that Javascript would often be used for – animated page transitions, menu management, URL manipulation and page changes for example. However, the main feature of jQuery Mobile from a coder's point of view is that it is based on a more general coding library called jQuery. jQuery is described as a small, fast, feature rich Javascript library. It was designed to simplify common tasks in HTML documents that Javascript would normally be used for; document manipulation, event handling and animation. The most significant feature of jQuery is that it evens out the differences between browsers so that the same website will (in most cases) work in the same way no matter what browser is used.

When working with jQuery Mobile, Javascript takes on all of the roles that a developer would normally use it for – data manipulation, calculations and business logic. However, the JQM libraries include functions that are specific to mobile apps, such as handling Touch-based interactions, changing screen orientation, changing app pages, handling "virtual mouse" events etc. For example, to make a button press in a jQuery Mobile app cause a switch to a different page, we could use the following event handler:

```
jQuery("#changeButton").bind('click', function() {
     jQuery.mobile.changePage("#newPage");
     // Do any other processing needed here…
}
```

Note that for this code to work, we are relying on there being a jQuery object and a jQuery.mobile object. Also, we have to provide code to 'bind' the new function to the button on the page whose id is "#changeButton". This code comes under the category of Object Oriented Programming, which we will look into in much more detail in the next chapter.

Exercises

1. Find out how to access the Console and the Developer Tools from all of the browsers you have access to in the labs: Internet Explorer V9, Firefox, Chrome and Safari should all be installed. Note that Firefox does not have built-in developer tools, but the Firebug plug-in should be installed in the lab machines
2. Open up a web page in a browser and investigate what all of the developer tools are for. For example, in Chrome (see figure 3. 7), check each of the main options on the developer tools window – Elements, Resources, Network, Scripts, Timeline and Profiles. You can get online help to find out more about each of these
3. Put together the short JS program seen in figure 3.s 3 & 4, and verify that it works in a browser. Once you have the page displaying properly in the browser, open the developer tools and find the script file in the Scripts view.
4. Open the Console view and verify that you can enter an expression (e.g. >2 * 3) and have its result printed out.

Questions

1. Give three reasons why it is good practice to put Javascript code into separate files instead of embedding the code directly into HTML documents
2. A Javascript code file that contains the code needed to validate numeric and date entries into forms on a web page has been named “validation.js”. The file is in the same place as the HTML document that will use it. Write the <script> statement that you would need to add to the HTML document.
3. Using a browser console, enter expressions that would result in the following being

displayed:

a. The number of seconds in a day

b. The number of seconds in a year

c. Given the statements ***first = "Joe"***, ***second = "Bloggs"*** have already been entered at the console, print out Joe's full name, including the space between the first and second names

d. The total price of an item which is £17.50 before VAT is added, given that the VAT rate is 20%

4. How would Javascript evaluate the expression: **127 + 0360 + 0x25**?
5. What type will JS say the expression **parseInt("99 red balloons")** has?
6. I wish to add together the values of Avagodro's constant (6.02e+23) and the charge on an electron (1.6e-19). Would this be sensible in JS?
7. The value ff in hexadecimal is 255 in decimal. How will JS evaluate the expression **64000/ff**?
8. How will JS print out the string expression **"Hello\nWorld!"**?
9. How will JS dislay **"17" * 2 + 11**?
10. How will JS display **"34" + 11**?
11. Javascript will try to evaluate any code as truthy or falsey if it needs to. If I enter an expression like **if(x = Math.sqrt(9)) y = "Yes it does";** in a program, what will be the result? (note: be careful here – look at the expression inside the brackets carefully)
12. Put appropriate semi-colons into the following code:

```
var x, y, i
x = 2
for(i=0; i<4; i++)
        y = y + y
document.write("y = " + y)
```

13. I need to execute some code if the following conditions hold:

    Either X is less than Y and greater than 20 and Y is more than 2 times Z

    Or the square root of Y + Z is zero.

    Write the JS if() control statement that will evaluate the conditions.
14. How does Javascript distinguish between code that assigns a function to a variable and code that calls it to get it to do its job?
15. If a function has been defined to accept two parameters (e.g. firstname and lastname), how are these distinguished in a call to the function (i.e. how does JS know which *argument* to apply to firstname, and which to apply to lastname?
16. If a function is defined with no return statement and then the function is called as part of an assignment (e.g. x = f()), what do you think will be assigned (to x, in this case)? Try this in the console to check your answer.
17. A function is needed to work out the area of a rectangular shape given its length and width. Write this function in Javascript